# RapidIO™ Interconnect Specification Annex 2: Session Management Protocol Specification

4.1, 6/2017



RapidIO.org

# Revision History

| Revision | Description | Date |
|---|---|---|
| 2.0 | First release | 06/14/2007 |
| 2.0 | Public release | 03/06/2008 |
| 2.1 | No technical changes | 07/09/2009 |
| 2.1 | Removed confidentiality markings for public release | 08/13/2009 |
| 2.2 | No technical changes | 05/05/2011 |
| 3.0 | Changed RTA contact information. No technical changes | 10/11/2013 |
| 3.1 | No technical changes. | 09/18/2014 |
| 3.2 | No technical changes. | 01/28/2016 |
| 4.0 | No technical changes. | 06/15/2016 |
| 4.1 | No technical changes. | 06/30/2017 |



Questions regarding RapidIO.org, specifications, or membership should be forwarded to:

RapidIO.org
8650 Spicewood Springs #145-515
Austin, TX 78759
512-827-7680 Tel.

# Table of Contents

# Table of Contents



## Chapter 4 Message Format Descriptions

# 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

The Session Management Protocol permits system software to establish, manage, and remove *virtual streams* as defined in *RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification Rev. 2.1*. The data streaming protocol allows data of any format to be transported between two end points. The Session Management Protocol provides a method for two end points to negotiate the characteristics of the data stream and assign those characteristics so the receiving entity can use the appropriate software layers upon receiving the data stream.

## 1.2 Overview

A stream, is a unidirectional connection between two end points. Bidirectional traffic is carried over two streams, one in each direction.

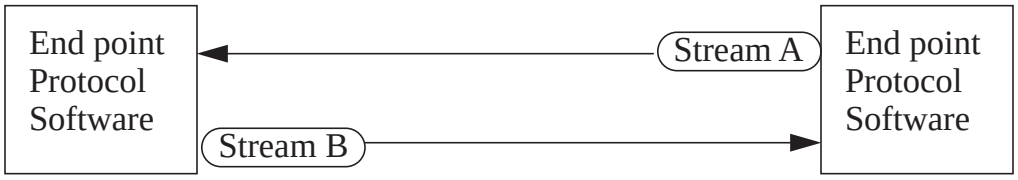


**Figure 1-1. Data Streaming**

The end points must have a common understanding of what the data within the stream is, and associate the stream with the right end point process. One of the principles of the data streaming protocol is the use of *virtual streams*. A virtual stream contains a generic tag called the *virtual stream ID (VSID)*. The VSID is assigned by the destination, based on the destination's method for decoding, and may be a software or a hardware feature.

VSIDs are unique between any pair of source and destinations. The Session Management Protocol allows a source and destination to discover what protocols the two can use to communicate, assign a VSID to carry that protocol, establishing an open stream. Once open, a stream can carry any number of protocol data units (PDUs) until the stream is not longer needed. The Session Management Protocol is then used to close the stream.

The Session Management Protocol is intended to manage streams of communication. When data is passed using Type 9 (Data-Streaming Class) as the conveyance, the streamID used for Session Management Protocol is the same as the streamID used for Type 9 data traffic. When the data are passed using Type 11 (Message Class) as the conveyance, the streamID used for Session Management Protocol can be encapsulated in the command header of the DATA or DATA1 commands, as defined in Section 4.3.

## 1.3 Features of the Session Management Protocol

The Session Management Protocol provides the following features:

- A method to contact the session management software running on any end point.
- A method to discover which protocols an end point is capable of receiving.
- A method to discover and assign streams.
- A method to manage the status of streams.
- A method to close active streams.
- A method to handle errors and to handle protocol violations.

## 1.4 Contents

Following are the contents of the *RapidIO Interconnect Specification Annex 2: Session Management Protocol Specification Rev. 2.1*:

- Chapter 1, “Overview,” is an overview of the Session Management Protocol specification.
- Chapter 2, “Managing Data Streams,” introduces system issues such as discovery and transport configurations.
- Chapter 3, “Session Management Operation,” describes the set of messages defined by the protocol and their use.
- Chapter 4, “Message Format Descriptions,” contains the packet format definitions for the session management messages.
- Chapter 5, “Registers,” describes the visible register set that allows an external processing element to discover and contact a session management process on another end point.
- Chapter 6, “Vendor-Defined Protocols,” describes how to specify vendor-defined protocol attributes.
- Chapter 7, “Ethernet Encapsulation,” contains the specific use case requirement for tunneling Ethernet using this protocol and data streaming, as well as requirements for vendor-specific attributes.

## 1.5 Terminology

The following terms are used in this document. The terms are consistent with their usage in *RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification*.

Refer to the Glossary at the back of this document for additional definitions.

**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.

The value of the cos field is not defined by this specification, but is implementation dependent. The requirements for this field are that every message with a given cos must be transmitted with equal priority.

**Conduit** - A bidirectional data path, consisting of one stream for data transfer in each direction.

**Conveyance** - The RapidIO logical layer protocol used to transmit and receive data within a stream. This specification defines the use of RapidIO Type 11 (Message Class) and Type 9 (Data-Streaming Class) conveyances.

**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.

**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.

**Process** - When a node communicates with a remote, some element of execution is responsible for managing the data communication. In this specification such element of execution is referred to as a process. No implication is intended regarding the internal structure of the operating system or other system organization.

**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. When using Type 9 (Data-Streaming Class) as the conveyance for data transfers, the VSID is encapsulated in the Type 9 protocol. When using Type 11 (Message Class) as the conveyance for data transfers, the VSID is encapsulated in fields in the DATA or DATA1 Session Management Protocol commands.

**StreamID** - a specific field in the data streaming protocol that is combined with the data stream'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.

**Suspect** - A node which, for some reason, is not behaving according to the specification. See "Section 3.7, Session Management Error Conditions and Recovery" on page 39 for more information.

## 1.6 Conventions

All fields and message formats are described using big endian format.

| | |
|---|---|
| \|\| | Concatenation, used to indicate that two fields are physically associated as consecutive bits |
| *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. |
| <variable> | Identifies a logical variable that may be a specific field of a register or packet or data structure. |

## 1.7 Useful References

*RapidIO Interconnect Specification Part 2: Message Passing Logical Specification*

*RapidIO Interconnect Specification Part 3: Common Transport Specification*

*RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification*

# Chapter 2 Managing Data Streams

## 2.1 Introduction

Data streaming provides a common layer in RapidIO that allows any protocol to simultaneously share the same physical transport with any and all other protocols. The system can be comprised of many end points, supporting multiple protocols, and utilizing a variety of hardware and software acceleration features. Using data streaming and this management protocol, any standard or proprietary protocol data format can be tunneled on a RapidIO fabric.

## 2.2 System Example

Figure 2-1 shows an example of a system using two methods to perform file transfers, one interworking with an Ethernet bridge and one tunneling FTP directly.

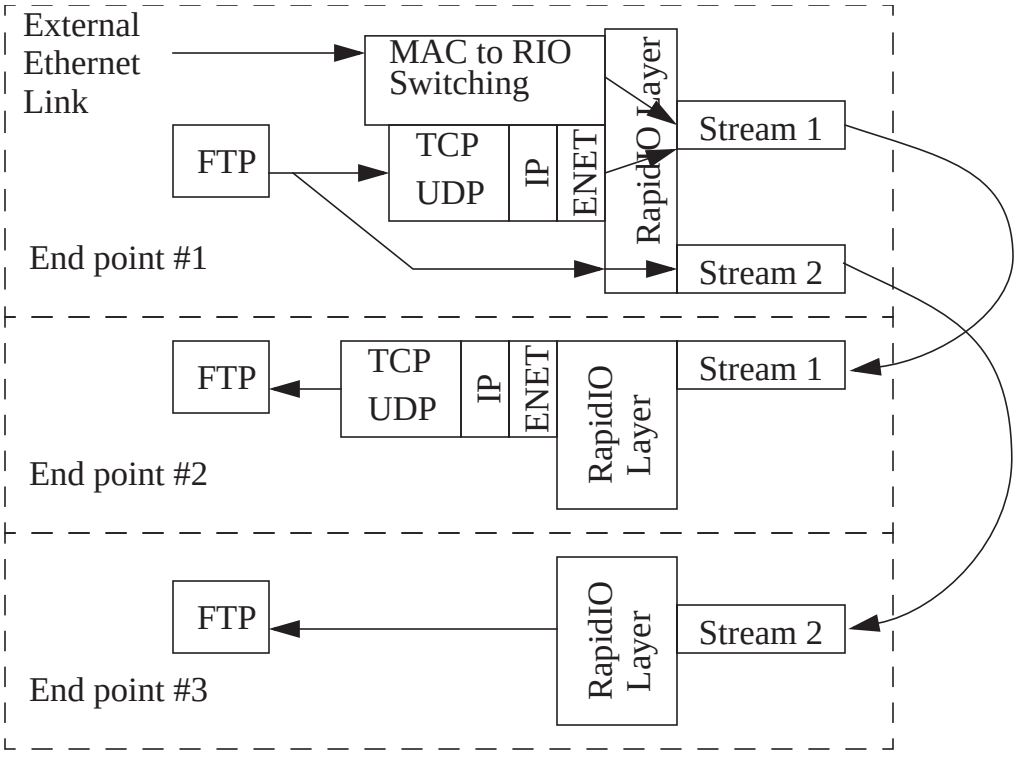


**Figure 2-1. Example of a RapidIO-Based Networking System**

## 2.3 Establishing Data Streams

The data streaming process is separated into two phases, with the management phase separated from the data phase. All the information about the stream is exchanged separately from the actual data transfer. Once established, a stream can support many data transfers. The data transfer only contains the information necessary to recover the original data. The received data must then be *associated* with an end process.

The method of association makes use of both the streamID and class of service (cos) to identify the stream. For example, the streamID can be linked directly to a given process which receives information for that stream. The cos may be used to determine the real time behavior necessary for the data, for example, guaranteed latency of *X* for responses.

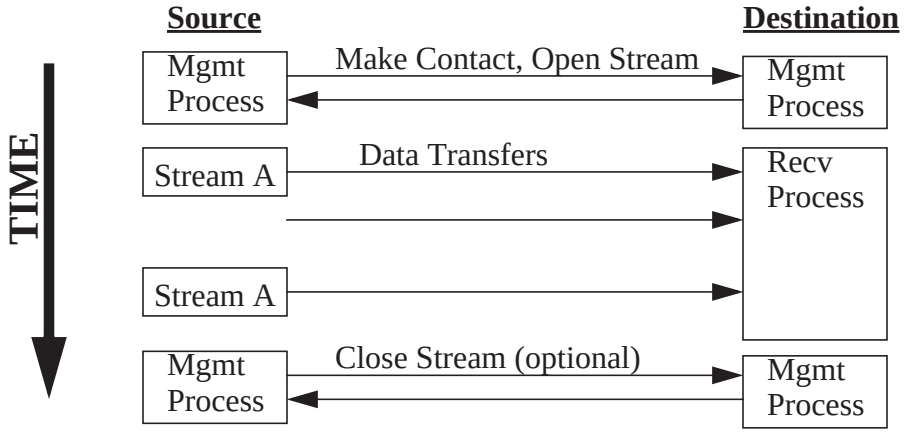


**Figure 2-2. Stream Process**

The Session Management Protocol begins with a discovery process. That discovery process detects which end points can perform session management negotiations. The discovery process is described in “Section 3.3, Contacting a Participating End point” on page 20.

Once two management end points are connected, they exchange a series of Session Management Protocol messages to discover the data streaming capabilities of the other. If both have the right capabilities, then one or more streams are opened for a given protocol.

With an open stream data can be transferred at any time. A stream may be persistent, existing even though there is no data to transfer at any given time.

Any stream may be closed by either end point, however most streams should remain open as long as the hardware and software might need to transfer data. If an end point is shutting down, or is terminating the process that handles data on that stream, it may close a stream.

As previously described, streams are unidirectional. However, many systems have

requirements that communication be bidirectional. A bidirectional data path is created out of two streams, one in each direction. Such a bidirectional data path is called a conduit. When there is a requirement for bidirectional data transfer, one system may initiate the connection. The recipient of the initial negotiation is expected to start negotiation for the connection in the other direction. The resulting pair of streams should be grouped together, so that whenever one stream of the conduit is opened or closed, the other stream is treated the same. For more information, see “Section 3.4, Establishing Conduits” on page 21.

## 2.4 Data Streaming System Configurations

*RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification* defines a logical layer for streaming data transfer. Hardware designed to this specification can transfer data using hardware or software resources to encapsulate data. The data streaming protocol uses a segmentation and reassembly protocol to manage variable sized PDUs.

Other methods may be used to transfer variable sized PDUs as long as they include the elements of segmentation and reassembly and contain a virtual stream ID. This specification also defines a method to encapsulate data using the *RapidIO Interconnect Specification Part 2: Message Passing Logical Layer Specification*.

The management protocol may also be run over a number of conveyances. The messages may be embedded in a predesignated stream in the data streaming logical layer, or it may be run over the message passing logical layer, even if the data is on the data streaming logical layer. Additional conveyances may be available as vendor-specific conveyances or in future versions of the RapidIO specifications. Table 2-1 shows the system configurations that are currently defined by this specification.

**Table 2-1. Data Streaming System Configurations**

| Management | Data Transfers |
|---|---|
| Messaging (Type 11) | Messaging (Type 11) |
| Messaging (Type 11) | Data Streaming (Type 9) |
| Data Streaming (Type 9) | Messaging (Type 11) |
| Data Streaming (Type 9) | Data Streaming (Type 9) |

The protocol first establishes how to contact the management process. Once contacted, the management processes then exchange the necessary information to transfer the data.

When a single node supports Session Management Protocol on multiple conveyances, the initial information required to establish a Session Management Protocol connection on each conveyance must be put in a separate Session Management Protocol Extended Features Register Block, as described in “Section 5.2, Session Management Protocol Extended Features Register Block” on page 58.

Therefore, multiple Session Management Protocol Extended Features Register Blocks may be required.

# Chapter 3 Session Management Operation

## 3.1 Introduction

This chapter describes the RapidIO Session Management Protocol. The protocol consists of a sequence of messages to establish capabilities, open streams, manage streams, and close streams. The protocol includes methods for handling abnormal conditions.

## 3.2 Initialization of Session Management Advertisement CSRs

Before any management messages can be exchanged, an end point must first establish which end points support data streams, and discover how to contact their management process. The Session Management Protocol allows each end point to use its own resources as needed for the process of data streaming.

Participating end points place information in the *Session Management Advertisement CSR*, indicating that it supports the session management protocol, and identifies which conveyance to use to contact the management process. Legacy devices may also advertise participation in this protocol using the *Component TAG CSR* (see *RapidIO Interconnect Specification Part 3: Common Transport Specification* and “Section 5.3, Component Tag CSR Session Management Protocol Advertisement” on page 64).

On hardware power-up and on hardware reset, the Session Management Advertisement CSR and the Component Tag CSR must be initialized by hardware to indicate non-participation in Session Management Protocol. During software initialization, implementations conforming to this specification must indicate participation in the Session Management Protocol by modifying the Session Management Advertisement CSR, or optionally by modifying the Component Tag CSR if the Session Management Protocol Extended Features Register Block is not available.

The Session Management Advertisement CSRs, defined in “Section 5.2, Session Management Protocol Extended Features Register Block” on page 58, may be initialized by a processor which is part of the device implementing the Session Management Advertisement CSRs, by a processor remote from the device, or through a combination of the two. For example, protocol support related attributes could be initialized by the local processor, while system related attributes such as

message timeout values could be initialized by a remote processor.

The Session Management Advertisement CSRs include a number of registers to facilitate initialization by local and remote processors. These are:

- "Section 5.2.2, Session Management Protocol Register Write Enable CSR (Block Offset 0x4)" on page 59
- "Section 5.2.4, Session Management Attribute Range CSR (Block Offset 0xC)" on page 61

The Session Management Protocol Register Write Enable CSR implements mutual exclusion between processors attempting to write to the attribute registers. If the Session Management Protocol Register Write Enable CSR is locked, other processors must respect the lock and not attempt to modify the attribute registers.

The Session Management Attribute Range CSR indicates how many attributes have been initialized. It also supports encoding up to 16 stages of initialization, to allow sequencing of the attribute initialization process.

For more information, refer to the definitions of the named registers.

## 3.3 Contacting a Participating End point

The *Session Management Advertisement CSR* contains the following information:

<Conveyance> indicates which conveyance can be used for the Session Management Protocol

If the conveyance is Type 11 (messaging) then the CSR has the following information:

- <Mailbox ID> identifies a mailbox dedicated for the reception of management messages.

If the conveyance is Type 9 (streaming) then the CSR has the following information:

- <StreamID> identifies a stream dedicated for the reception of management messages.
- <COS> identifies the class of service dedicated for the reception of management messages.

The conveyance used to create and manage all streams is specified in the Session Management Advertisement CSR, or in the Component Tag CSR if the Session Management Protocol Extended Features Register Block is not available. Unless otherwise specified, the conveyance used to transmit DATA and FLOW_CONTROL messages for a stream is the same as the conveyance used to create and manage all streams. Specification of other conveyances for DATA and FLOW_CONTROL traffic is performed by including the *CONVEYANCE* attribute as described in Section 3.5.3.10.

A special case may be needed when a system is required to create and manage streams using both Type 11 (messaging) and Type 9 (streaming). This will occur on mixed systems, where some nodes provide support for only Type 11 conveyance and other nodes provide support for only Type 9 conveyance. When necessary, this can be accomplished by use of the Session Management Advertisement CSR to contain contact information for Type 9 management, and using the Component Tag CSR to contain contact information for Type 11 management. Other configurations may be possible, but are implementation specific.

End points have two options to determine whether a remote node participates in this protocol. First, they may choose an implementation-specific mechanism, such as a built-in table, to check only for specified remote systems. Second, they may scan management space for all remote nodes, collecting information on participating end points. In either case, the end point is then expected to contact remote end points as appropriate, asking for their capabilities. See Chapter 5, “Registers”, for the bit definitions.

## 3.4 Establishing Conduits

Conduits consist of a pair of unidirectional streams for transferring data for a single protocol between two end points. In every conduit, one stream transfers protocol data in one direction while the other stream transfers protocol data in the other direction.

Conduits are established using the same command set as unidirectional streams. The Session Management Protocol requires that the sender initiate the process of opening a stream. One consequence of these two facts is that for conduits to be established, both end points are required to send an OPEN command, and there must be a mechanism to link the two streams into a single conduit. Linking the two streams into a conduit is accomplished by use of the *CONDUIT_STREAM* attribute during OPEN/ACCEPT negotiation. The *CONDUIT_STREAM* attribute is a 32-bit attribute, as described in Section 3.5.3.8. The data associated with this attribute consists of two streamIDs involved in the conduit.

One difficulty arises, based on whether a known end point is required to establish the conduit based on some external criteria, or whether it is possible for either end point to initiate establishment of the conduit.

The simple case, where one end point initiates establishment of the conduit, is referred to as a master/slave configuration. In this case, the master may send an OPEN message at any time to initiate establishment of the conduit, but the slave may only send the corresponding OPEN message after receiving the OPEN message from the master.

The more general case is referred to as a peers configuration. In this case, either node may send an OPEN message at any time to initiate establishment of the conduit. The peers configuration is more complex, because there must be a mechanism to handle

the case when the two OPEN messages are in transmit at the same time.

It should be noted that an implementation capable of handling the peers configuration is capable of handling the master/slave configuration. For this reason, the algorithm for the peers configuration is presented in Section 3.4.3, but no algorithms for master/slave configuration are provided.

### 3.4.1 Master/Slave Configuration Conduit Establishment

In the master/slave configuration, one end point, the master, is always responsible for initiating conduit establishment. The other end point, the slave, completes conduit establishment in response to the establishment of the first conduit stream. The algorithm running on the master may not be identical to the algorithm running on the slave.

The master begins to establish the conduit by creating a local structure to contain information about the conduit. The first RapidIO transaction that the master makes is sending an OPEN message[1] containing the *CONDUIT_STREAM* attribute. The *CONDUIT_STREAM* attribute value is 0xFFFFFFFF in the first OPEN message when establishing a conduit, indicating that no previous negotiation has already been performed for this protocol between these two end points.

Upon receipt of an OPEN with *CONDUIT_STREAM* specified, the slave creates a local record containing information about the conduit. It responds to the OPEN with an ACCEPT message specifying a newly created streamID, which we refer to as 0xSSSS and *CONDUIT_STREAM* attribute value of 0xFFFFSSSS indicating that the slave, the sender of the ACCEPT message, will receive data on streamID 0xSSSS for this conduit.

To complete conduit creation, the slave then sends an OPEN message with the *CONDUIT_STREAM* attribute set to 0xFFFFSSSS to the master. The first field is the streamID that the slave will use to transmit data on, which has not yet been established. The second field is the streamID that the slave will use to receive data on, which has been assigned in the prior ACCEPT message.

The *CONDUIT_STREAM* attribute data contains the first field set to 0xFFFF and the second field set to a valid streamID. The fact that the *CONDUIT_STREAM* attribute is specified indicates that the OPEN message is related to a conduit. The value 0xFFFF in the first field indicates that the slave, the sender of the OPEN message, does not already know the streamID to use for transmitting data related to the conduit. The value of 0xSSSS in the second field indicates that the slave, the sender of the OPEN message, does already know the streamID to use for received data related to the conduit, and that the streamID is 0xSSSS.

The master, on receiving the ACCEPT message, reads the *CONDUIT_STREAM*

[1]REQUEST and ADVERTISE messages may have been previously exchanged. As these details add no relevant information to the discussion of establishing a conduit, they have been omitted.

attribute data, extracts the second field and finds 0xSSSS, and searches through its internal data for a conduit with matching streamID. The criteria for determining that the streamID is a match includes, but may not be limited to, the following tests. The matching streamID from the master's internal data must be part of a conduit. It must have the same value as specified, 0xSSSS. Finally, it must be the streamID that the master uses to transmit data on that conduit. If no such data record is found, the master may respond with REFUSE. However, when the master does find the data record associated with the conduit, it responds to the OPEN with ACCEPT, and updates its internal data structures. The ACCEPT message specifies a streamID on which the master will receive data, 0xMMMM, as well as the *CONDUIT_STREAM* attribute. The data contained in the *CONDUIT_STREAM* attribute consists of 0xSSSSMMMM.

The slave, on receiving the ACCEPT, searches through its internal data containing incomplete conduit records, finds a matching record, and updates its internal data structures.

### 3.4.2 Peers Configuration Conduit Establishment

In some distributed and/or reliable systems, it is necessary to allow each end of a conduit to attempt, simultaneously, to establish a conduit. The difference between the peers configuration and the master/slave configuration is that, in the peers configuration, an additional check is required before attempting to complete the conduit.

Assume that node M and node P are attempting to simultaneously create a conduit for a protocol. It is possible that node P receives node M's OPEN request before node P transmits is own OPEN request, and vice versa. When P receives the OPEN request, it checks to see if it has an outstanding OPEN request for establishing the same conduit. In this case, Node P does not, so the peers algorithm degenerates into the master/slave algorithm. Node P sends an ACCEPT response for node M's OPEN request, followed by node P's OPEN request to complete the creation of the conduit.

If node P receives node M's OPEN request after node P has transmitted its own OPEN request to node M, then when node P checks for outstanding OPEN requests for the creation of a conduit with the specified protocol with the other node, it will find one. Node P will respond with an ACCEPT message specifying the streamID that node P will receive data on using the *CONDUIT_STREAM* attribute, and will note that stream against its own attempt to establish the conduit. When node P receives the ACCEPT response for its own OPEN request for the conduit, both the transmit and receive stream IDs for the conduit will be known. Exactly the same procedure occurs on node M, so the conduit is established.

### 3.4.3 Conduit Establishment Algorithm

The following algorithm consists of three entry points, representing the procedure to call when processing incoming OPEN and ACCEPT messages, and the procedure for application code to initiate the procedure to establish a conduit. These are called process_incoming_open(), process_incoming_accept(), and create_conduit(), respectively. When working together, these three procedures create conduits.

There are several user-supplied procedures called from this algorithm. The function names should be self-explanatory, with the exception of find_partial_conduit(). This procedure searches through internal data structures for a conduit structure matching the specified remote nodeID and protocol, and with the streamIDs matching in the following manner. If the streamID specified in the call is 0xFFFF, then any streamID matches. If the streamID specified in the call is not 0xFFFF but the streamID in the local structure is 0xFFFF, then the streamID matches. If neither is 0xFFFF, then the values must be identical for them to match.

The algorithm specified here can be implemented for systems using a single execution thread and polled mode I/O. It does not indicate critical sections, which must be mutually exclusive. For multi-threaded OS implementations, the implementer must supply a locking mechanism to prevent concurrent access by other processes and/or interrupt service routines, and identify which portions of the algorithm need to be protected in their particular environment.

```
process_incoming_open(message)
        remote = get_sender(message)
        proto = get_protocol(message)
        rem_xmit = get_sender_transmit(message)
        rem_rcv = get_sender_receive(message)
        conduit = find_partial_conduit(remote, proto, rem_xmit, rem_rcv)
        if ( not found(conduit) )
                conduit = create_new_conduit(remote, proto)
                local_xmit = rem_rcv
                local_rcv = rem_xmit
                update_conduit_transmit(conduit, local_xmit)
        else
                local_xmit = get_conduit_transmit(conduit)
                local_rcv = get_conduit_receive(conduit)
        streamID = alloc_new_stream()
        update_conduit_receive(conduit, streamID)
        response = create_new_message()
        put_local_receive(response, streamID)
        put_local_transmit(response, local_xmit)
```

```
                send_accept(remote, response, streamID)
                if ( local_xmit == 0xFFFF )
                        response = create_new_message()
                        put_local_receive(response, streamID)
                        put_local_transmit(response, 0xFFFF)
                        update_conduit_flag(conduit, OPEN_SENT)
                        send_open(remote, response)

process_incoming_accept(message)
                remote = get_sender(message)
                proto = get_protocol(message)
                rem_xmit = get_sender_transmit(message)
                rem_rcv = get_sender_receive(message)
                local_xmit = rem_rcv
                conduit = find_partial_conduit(remote, proto, rem_xmit, rem_rcv)
                if ( not found(conduit) )
                        conduit = create_new_conduit(remote, proto)
                        local_rcv = rem_xmit
                        update_conduit_receive(conduit, local_rcv)
                else
                        local_rcv = get_conduit_receive(conduit)
                        if ( local_rcv == 0xFFFF )
                                local_rcv = rem_xmit
                                update_conduit_receive(conduit, local_rcv)
                update_conduit_transmit(conduit, local_xmit)
                flag = get_conduit_flag(conduti)
                if ( not flagIsSet(flag, OPEN_SENT) )
                        response = create_new_message()
                        put_local_receive(response, local_rcv)
                        put_local_transmit(response, 0xFFFF)
                        update_conduit_flag(conduit, OPEN_SENT)
                        send_open(remote, response)

create_conduit(remote, proto)
                conduit = find_partial_conduit(remote, proto, 0xFFFF, 0xFFFF)
                if ( not found(conduit) )
                        conduit = create_new_conduit(remote, proto)
```

```
update_conduit_receive(conduit, 0xFFFF)
update_conduit_transmit(conduit, 0xFFFF)
message = create_new_message()
put_local_transmit(message, 0xFFFF)
put_local_receive(message, 0xFFFF)
update_conduit_flag(conduit, OPEN_SENT)
send_open(remote, message)
```

## 3.5 Management Messages

The Session Management protocol consists of the following messages. The formats of the messages are defined in Chapter 4, “Message Format Descriptions,” on page 45.

Message names in this document use only capital letters to avoid confusion with non-message related use of the terms for message names.

Attribute names in this document are italicized and in uppercase, with individual words separated by underscore, to avoid confusion with non-attribute related use of the terms for attribute names.

### 3.5.1 Session Management Message Types

The following subsections list the messages used in the Session Management Protocol.

#### 3.5.1.1 REQUEST

A REQUEST message is used to request information related to the protocols supported by the remote. The REQUEST can be for a list of supported protocols, or a list of attributes associated with a particular protocol.

For more information, refer to “Section 4.2.1, REQUEST” on page 45.

#### 3.5.1.2 ADVERTISE

An ADVERTISE message is the response to a REQUEST message. If the REQUEST message specified that a list of protocols supported should be returned, the ADVERTISE message contains only a list of protocols.

If the REQUEST message specified a particular protocol, and the recipient of the REQUEST supported the protocol, the ADVERTISE message contains all required and optional attributes related to the protocol.

Optional attributes and vendor defined attributes may be negotiated. See Section 3.5.3 for discussion on this topic.

For more information, refer to “Section 4.2.2, ADVERTISE” on page 46.

#### 3.5.1.3 OPEN

An OPEN message is sent to attempt to open a session on a target. The attributes specified in the OPEN message may or may not be acceptable by the target.

For more information, refer to “Section 4.2.3, OPEN” on page 47.

#### 3.5.1.4 ACCEPT

An ACCEPT message is sent if the OPEN message is successful. The ACCEPT message specifies the stream to be used to identify the session, as well as the other attribute values which will govern the session.

OPEN messages may be sent to attempt to open multiple sessions with the same target and attributes. A target may optionally support only a single session for a given protocol and source, in which case attempts to OPEN multiple sessions will result in ACCEPT responses which all specify the same stream.

For more information, refer to “Section 4.2.4, ACCEPT” on page 48.

#### 3.5.1.5 REFUSE

A REFUSE message is sent if the OPEN message is unsuccessful. The REFUSE message contains all the attributes in the OPEN request, in order to allow the OPEN sender to differentiate which OPEN request was refused.

For more information, refer to “Section 4.2.5, REFUSE” on page 49.

#### 3.5.1.6 FLOW-CONTROL

FLOW-CONTROL messages start or stop transmission of DATA messages. It is not necessary for any implementation to send FLOW-CONTROL messages, however FLOW-CONTROL messages must always be supported when received.

For more information, refer to “Section 4.2.6, FLOW_CONTROL” on page 49.

#### 3.5.1.7 DATA

DATA messages are used to transfer data using the message passing logical layer. There are several DATA messages, each with a different header format for use in different hardware and/or software environments.

For more information, refer to “Section 4.3, Data Formats” on page 53.

#### 3.5.1.8 CLOSE

CLOSE messages are used to terminate the existence of a stream between a source and destination.

**NOTE:**

The stream may or may not exist at either the source or destination. After reception of a CLOSE message, the stream specified must no

longer be used. After sending a CLOSE message, the system must be able to receive, without error, messages in transit at the time the CLOSE was sent. Such messages may be dropped.

When the stream is part of a conduit, the conduit should be closed. The means of determining that a stream is part of a conduit is implementation specific.

For more information, refer to “Section 4.2.7, CLOSE” on page 50.

#### 3.5.1.9 STATUS

STATUS messages can be used to query status information related to a specified stream. In this form of the STATUS command, a streamID is included in the message, and other parts of the command indicate that the purpose is to query the status of the stream.

STATUS message are also used to provide status information, related either to a specified stream or to a specified command. A STATUS message is sent in three situations: as a response to a STATUS message querying status for a specific stream, indicating the current status of the stream; as a response to a CLOSE message, indicating that the CLOSE was successful, and as a response to an illegal, unknown, or malformed command, indicating that the command was not understood.

Note that the STATUS response to a STATUS query can include a query of the same stream.

For more information, refer to “Section 4.2.8, STATUS” on page 51.

### 3.5.2 Message Header Fields

All Session Management Protocol messages begin with a *command* header field, followed by one or more additional fields. The header fields and arrangement of the header fields are fixed for each message type.

#### 3.5.2.1 Command Header Field: <CMD><VER>

The command header field consists of two octets, a command value denoted <CMD>, and a version value denoted <VER>. Each command value corresponds to one of the messages laid out in Section 3.5.1 on page 26.

<VER> for all commands described in this version of the specification, except the DATA1 command, must be set to a numeric value of 0x01. Receivers of commands must check <VER>. If a receiver receives an unrecognized message or a message other than STATUS with unknown <VER>, then the receiver must respond with a STATUS command with the Command_Unknown bit set. Recipients must not respond to STATUS messages with unknown <VER>.

#### 3.5.2.2 SourceID and DestID

SourceID, denoted <SourceID>, is the RapidIO destination ID of the device which

transmitted the message. DestID, denoted <DestID>, is the RapidIO destination ID for the device which should receive this message.

Both <SourceID> and <DestID> are 2 octets in size.

#### 3.5.2.3 Protocol Identifier: <ProtoID>

The protocol identifier is used to specify what encapsulated protocol is being referenced by the Session Management message. The <ProtoID> is always 16 bits in size.

Ethernet encapsulation, as described below, uses protocol ID 0x0102.

Proprietary protocols use 0x0101 for all proprietary protocols. Different protocols are distinguished by protocol attributes, as described in Chapter 6, “Vendor-Defined Protocols”.

Protocol ID 0xFFFF is reserved for special usage within messages. Refer to the definition of individual message types for the use of this special value, if any.

#### 3.5.2.4 Class of Service: <COS>

The Class of Service field can be used to specify the priority of a given stream or virtual stream. The value of the COS field is not defined by this spec, but can be implementation dependent. The requirements for this field are that every message with a given COS value must be transmitted with equal priority.

Class of Service, or <COS>, is always one octet in size.

#### 3.5.2.5 Stream Identifier: <StreamID>

The stream identifier is the value used to identify the particular session for a given protocol between a <SourceID> and <DestID>. The combination of <SourceID><DestID><ProtoID><StreamID> must always be unique in the system.

The <StreamID> is always two octets in size.

StreamIDs are assigned by the recipient of an OPEN message. The value of the StreamID must conform to the criteria shown in Table 3-1.

**Table 3-1. StreamID Assignments**

| StreamID value | Usage |
|---|---|
| 0x0000 - 0xDFFF | Available for applications |
| 0xE000 - 0xEFFF | Vendor specific |
| 0xF000 - 0xFFFE | Reserved |
| 0xFFFF | Invalid |

StreamIDs marked “available for applications” in Table 3-1 must be opened with an OPEN message. StreamIDs with vendor specific values and reserved values are pre-defined and may be used without explicitly opening a stream. Definitions of

reserved StreamIDs are defined in protocol specific chapters.

### 3.5.3 Session Management Protocol Attributes

Protocol attributes are relevant to individual virtual streams, and not to the session management protocol. Protocol attributes specify information about the stream, about the data transferred within the stream, or about how the data is to be represented on egress.

Each session management protocol attribute is encoded and made available to the remote system. Each attribute is encoded into an attribute field. Regardless of the data being identified, the attribute field always consists of a 64-bit (8-octet) value, consisting of two fields. The first field, Attribute ID, contains an identifier of the attribute. The second field, Attribute Value, contains the value associated with the attribute on a particular system. The fixed size allows for consistent parsing of attributes between multiple different protocols. It also allows attributes to be ignored if they are not understood by a given implementation. Another reason to have fixed size attributes is that it will simplify the implementation of hardware support for these attributes.

Attribute IDs can have one of three sizes: 8-bits, 16-bits, and 32-bits. The size of each attribute can be determined based on the first octet of the Attribute ID. The remaining bits in the attribute encoding are available for the Attribute Value. Attribute ID sizes are pre-defined, independent of protocol. Table 3-3 lists the sizes of each Attribute ID for all protocols.

**Table 3-2. System Management Protocol Attribute Sizes**

| Attribute ID size | Attribute ID value |
|---|---|
| 8 bits | 0x00 - 0x7F |
| 16 bits | 0x8000 - 0xEFFF |
| 32 bits | 0xF0000000 - 0xFFFFFFFF |

Attribute IDs marked as vendor specific are available for use by vendors for their own purposes. Table 3-3 shows the attribute ID ranges available for vendor use. All attribute ID ranges not explicitly assigned as general attributes, protocol-specific attributes, or vendor-specific attributes, are reserved and must not be used.

**Table 3-3. Vendor-Specific Attribute Ranges**

| Attribute ID size | Vendor-Specific Range |
|---|---|
| 8-bits | 0x78-0x7F |
| 16-bits | 0xEF00 - 0xEFFF |
| 32-bits | 0xFE000000 - 0xFFFFFFFF |

There are minimal ordering requirements for protocol attributes. All required attributes must occur ahead of all optional or vendor specific attributes. The order of

any attributes required by a protocol may be specified by the protocol. Optional attributes may occur in any order. All optional attributes must occur after all required attributes, and before any vendor specific attributes. If vendor specific attributes are used, the first vendor specific attribute must begin with attribute 0x00, which identifies the vendor associated with the vendor specific attributes. Any further ordering requirements of vendor specific attributes may be defined by the vendor.

All implementations of the Session Management Protocol must include complete support for all standard attributes for every protocol supported. In the event that an optional or vendor specific attribute is not understood by an implementation, the session must not be opened.

### 3.5.3.1 *VENDOR* Attribute

The *VENDOR* attribute is an 8-bit attribute ID (0x00) with a 56-bit value. Only one *VENDOR* attribute may be specified in any single command.

The attribute value for the *VENDOR* attribute must conform to one of two formats:

OUI format: The three-octet OUI for the vendor defining the protocol may be assigned to the second, third, and fourth octets of the attribute, and the remaining four octets set to zero.

**Table 3-4. System Management Protocol Attribute Sizes**

| AttributeID | OUI #1 | OUI #2 | OUI #3 | Zero | Zero | Zero | Zero |
|---|---|---|---|---|---|---|---|
| 0x01 | 0x00 | 0xA0 | 0x1E | 0x00 | 0x00 | 0x00 | 0x00 |

NAME format: A seven-octet value, where every octet must contain a non-zero value. This format may be assigned to a seven-character representation of the ASCII value of the vendor company name.

**Table 3-5. System Management Protocol Attribute Sizes**

| AttributeID | Char #1 | Char #2 | Char #3 | Char #4 | Char #5 | Char #6 | Char #7 |
|---|---|---|---|---|---|---|---|
| 0x01 | 0x52 | 0x61 | 0x70 | 0x69 | 0x64 | 0x49 | 0x4F |

Vendors should use the OUI format if an OUI is available. If NAME format is used, the vendor is responsible for insuring that the choice of NAME does not conflict with any existing *VENDOR* attribute value.

**NOTE:**

The values specified above are for example only, and should not be used.

### 3.5.3.2 *DATA_OFFSET_VENDOR* Attribute

The *DATA_OFFSET_VENDOR* attribute is an 8-bit attribute ID (0x02) with a 56-bit value. The value must conform to the formats specified for the *VENDOR* attribute.

The *DATA_OFFSET_VENDOR* attribute identifies the vendor associated with the contents of the *DATA_OFFSET* Attribute.

The *DATA_OFFSET_VENDOR* attribute is optional, however, if the *DATA_OFFSET_VENDOR* attribute is specified, then the *DATA_OFFSET* attribute must follow it.

#### 3.5.3.3 *DATA_OFFSET* Attribute

The *DATA_OFFSET* attribute is specified with the 16-bit attribute ID 0x8003, leaving six octets of data.

The *DATA_OFFSET* attribute specifies the number of octets of data that will be appended to the header of a Data message for a given stream. These octets are known as the Offset octets. The Offset octets can be used to convey information such as TCP/IP offload information, packet classification, and other vendor specific features. The remaining five octets in the *DATA_OFFSET* attribute are available for vendor specific information, labelled <VendorN>. The format of the *DATA_OFFSET* Attribute is as follows:

**Table 3-6. *DATA_OFFSET* Attribute Format**

| attributeID | Offset | Vendor0 | Vendor1 | Vendor2 | Vendor3 | Vendor4 |
|---|---|---|---|---|---|---|
| 16-bits | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits |
| 0x8003 | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF | 0xFF |

*DATA_OFFSET* must be present if the *DATA_OFFSET_VENDOR* is specified.

If the *DATA_OFFSET_VENDOR* and *DATA_OFFSET* attributes are not included in an Open request, the default number of Offset octets is 0.

If the *DATA_OFFSET_VENDOR* and *DATA_OFFSET* attributes are present, the *DATA_OFFSET* attribute must follow the *DATA_OFFSET_VENDOR* attribute.

#### 3.5.3.4 *REQUEST_RETRY_PERIOD* Attribute

The *REQUEST_RETRY_PERIOD* attribute is specified with the 32 bit attribute ID 0xF0000000.

The *REQUEST_RETRY_PERIOD* attribute indicates the time period, in microseconds, after which an REQUEST, OPEN, CLOSE, or STATUS request which has not received a response should be sent again.

The *REQUEST_RETRY_PERIOD* attribute default value is 250 milliseconds.

#### 3.5.3.5 *REQUEST_TIMEOUT_PERIOD* Attribute

The *REQUEST_TIMEOUT_PERIOD* attribute is specified with the 32 bit attribute ID 0xF0000001.

The *REQUEST_TIMEOUT_PERIOD* attribute indicates the time period, in

microseconds, after which an REQUEST, OPEN, CLOSE, or STATUS request which has not received a response should be judged to have failed.

The *REQUEST_TIMEOUT_PERIOD* attribute default value is 1 second.

#### 3.5.3.6 *FLOW_CONTROL_XON_TIMEOUT_PERIOD* Attribute

The FLOW_CONTROL_XON TIMEOUT attribute is specified with the 32 bit attribute ID 0xF0000002.

The FLOW_CONTROL_XON TIMEOUT attribute value indicates the time period, in microseconds, from the time a FLOW CONTROL XOFF request is received until another FLOW CONTROL XOFF/XON message must be received. In the event that a FLOW CONTROL XOFF/XON message is not received in the timeout interval, transmission of the XOFF'ed stream should resume. This behavior is designed to detect the loss of FLOW CONTROL XON messages.

The FLOW_CONTROL_XON TIMEOUT attribute default value is 1 second.

#### 3.5.3.7 *OPEN_MESSAGE_NUMBER* Attribute

The *OPEN_MESSAGE_NUMBER* attribute is specified with the 32 bit attribute ID 0xF0000003.

The *OPEN_MESSAGE_NUMBER* attribute is used by the originator of an OPEN message to identify the response to the OPEN request. This allows the originator to have multiple parallel OPEN requests in flight, and to be able to match responses to the requests.

The *OPEN_MESSAGE_NUMBER* attribute value is used in an implementation specific manner.

#### 3.5.3.8 *CONDUIT_STREAM* Attribute

The *CONDUIT_STREAM* attribute is specified with the 32 bit attribute ID 0xF0000004.

The *CONDUIT_STREAM* attribute is used by both sides, in OPEN and ACCEPT messages, when creating a bidirectional conduit. The 32 bit value associated with the *CONDUIT_STREAM* attribute consists of the two stream IDs associated with the conduit. The first sixteen bits are used to indicate the stream ID of the stream which the sender of the OPEN or ACCEPT will use to transmit data on. The last sixteen bits are used to indicate the stream ID of the stream which the sender of the OPEN or ACCEPT will use to receive data on. The value of 0xFFFF for either field indicates that the specified stream ID was not known at the time the OPEN or ACCEPT message was sent.

See "Section 3.4, Establishing Conduits" on page 21 for more information, and examples of usage.

### 3.5.3.9 *DATA_HEADER_FORMAT* Attribute

The *DATA_HEADER_FORMAT* attribute is specified with the 32 bit attribute ID 0xF0000005.

The *DATA_HEADER_FORMAT* attribute is used in an ACCEPT message to indicate which DATA header should be used for DATA messages. The value is the numeric value of the DATA command, that is 0x06 indicates DATA Message Format, MAILBOX, as described in "Section 4.3.1, DATA Message Format, MAILBOX" on page 53, 0x09 indicates DATA1 message format for large PDUs, and so on.

Note that a special case exists for the DATA2 command. The implementation-specific value in the DATA2 header precedes the DATA2 command in the attribute value field, as shown in Table 3-7.

**Table 3-7. *DATA_HEADER_FORMAT* Attribute Values**

| | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits | 8-bits |
|---|---|---|---|---|---|---|---|---|
| DATA | 0xF0 | 0x00 | 0x00 | 0x05 | Reserved | Reserved | 0x00 | 0x06 |
| DATA1 | 0xF0 | 0x00 | 0x00 | 0x05 | Reserved | Reserved | Implementation-Specific | 0x09 |
| DATA2 | 0xF0 | 0x00 | 0x00 | 0x05 | Reserved | Reserved | 0x00 | 0x0A |
| DATA3 | 0xF0 | 0x00 | 0x00 | 0x05 | Reserved | Reserved | 0x00 | 0x0B |

When the *DATA_HEADER_FORMAT* attribute is not specified, the DATA command (0x06) must be used.

### 3.5.3.10 *CONVEYANCE* Attribute

The transmission channel is assigned with the 16-bit attribute ID 0x8001, leaving six octets for channel information. The channel is encoded in the first sixteen bits, leaving 32 bits for channel-specific information. The values for the conveyance are shown in Table 3-8.

Note that if a vendor-specific channel is used, the *VENDOR* attribute must be specified.

**Table 3-8. Ethernet Encapsulation Conveyance**

| Channel (16 bits) | Channel-specific information (32 bits) |
|---|---|
| 0x0000 = reserved | N/A |
| 0x0001 = message | 0x0000_00nn; nn indicates a mailbox |
| 0x0002 - 0xFEFF = reserved | N/A |
| 0xFF00 - 0xFFFF = vendor-specific | Vendor-defined |

### 3.5.3.11 Other Attributes

For all protocols, system vendors may choose to define additional vendor-specific attributes not defined by the protocol. If additional protocol-specific attributes are

used, the list of attributes must include the *VENDOR* attribute.

## 3.6 Message Sequence Examples

This section presents flow diagrams, to illustrate some typical uses of the Session Management command set.

### 3.6.1 Stream Initiation

Figure 3-1 shows the message sequence for initiation of a stream.

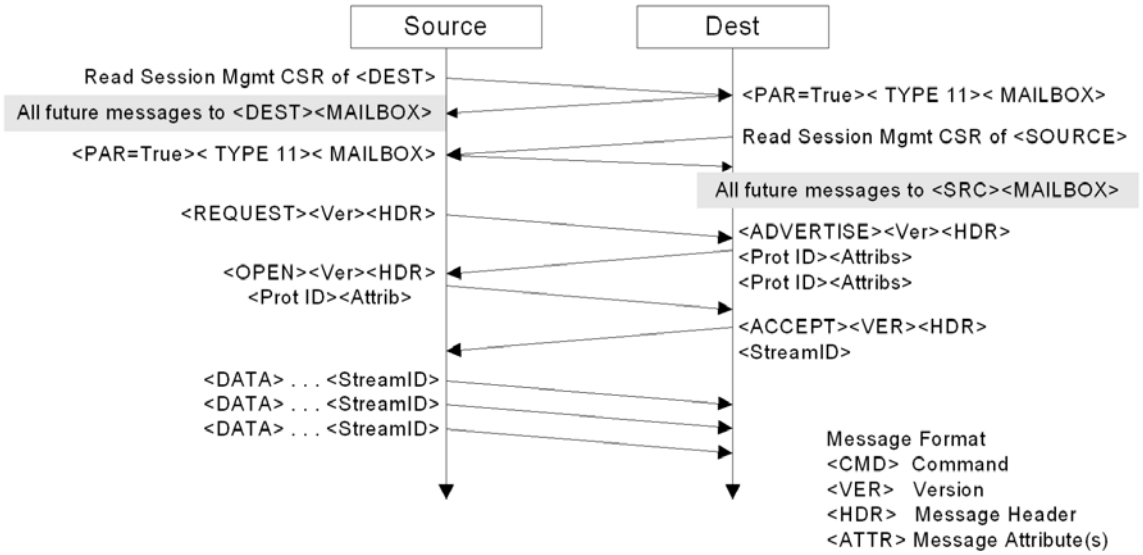


**Figure 3-1. Normal Stream Initiation**

### 3.6.2 Refusal to Initiate a Stream

Figure 3-2 shows the usage of a REFUSE command, when the recipient of an OPEN command does not allow the stream to be created.

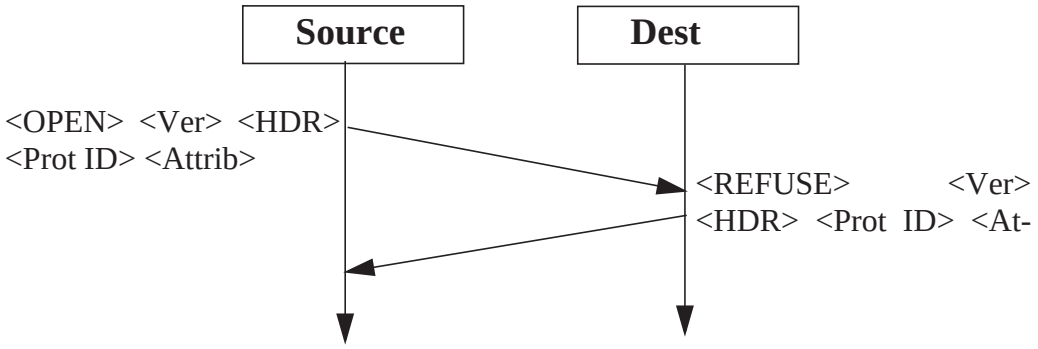


**Figure 3-2. Use of REFUSE Command**

### 3.6.3 Stream Shutdown

Figure 3-3 shows the sequence of commands for use during stream shutdown, when the initiator sends the CLOSE command. The receiver may also initiate the shutdown procedure.

Note that after the CLOSE message has been received, subsequent data must be discarded.

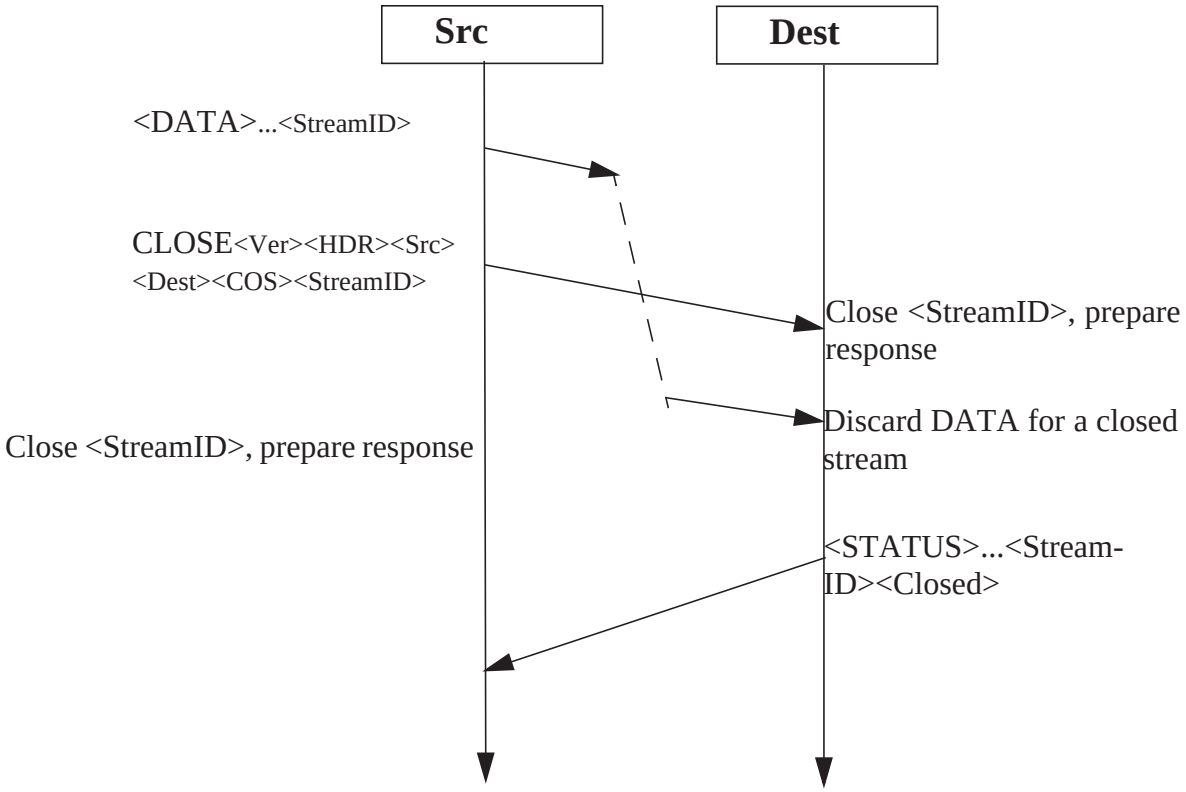


**Figure 3-3. Normal Stream Shutdown**

### 3.6.4 Uses of the STATUS command

Figure 3-4 demonstrates several uses of the STATUS command.

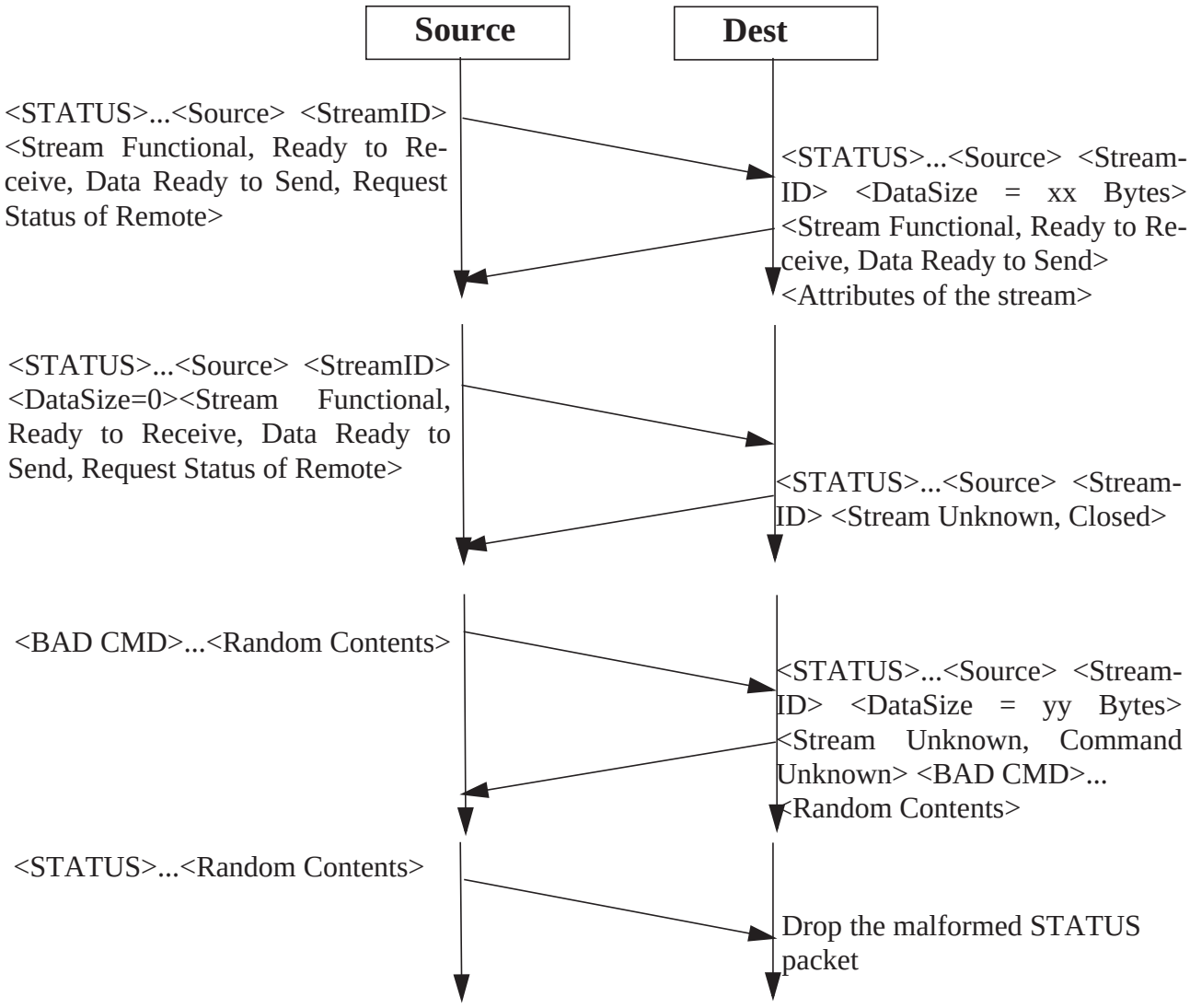


**Figure 3-4. Use of the STATUS Command**

There are four examples of the use of a Status query and response contained in Figure 3-4. The first message sent is a well formed STATUS request for information on a known VSID. The response to this message includes the state of the VSID, as well as all of the attributes used to OPEN the VSID. The second example is similar to the first, but for a VSID which is not open. The third example shows the use of a STATUS message to respond to a malformed packet. The last example shows the discard of a malformed STATUS command.

### 3.6.5 Use of the FLOW_CONTROL Command

Figure 3-5 shows sample usage of the FLOW_CONTROL command.

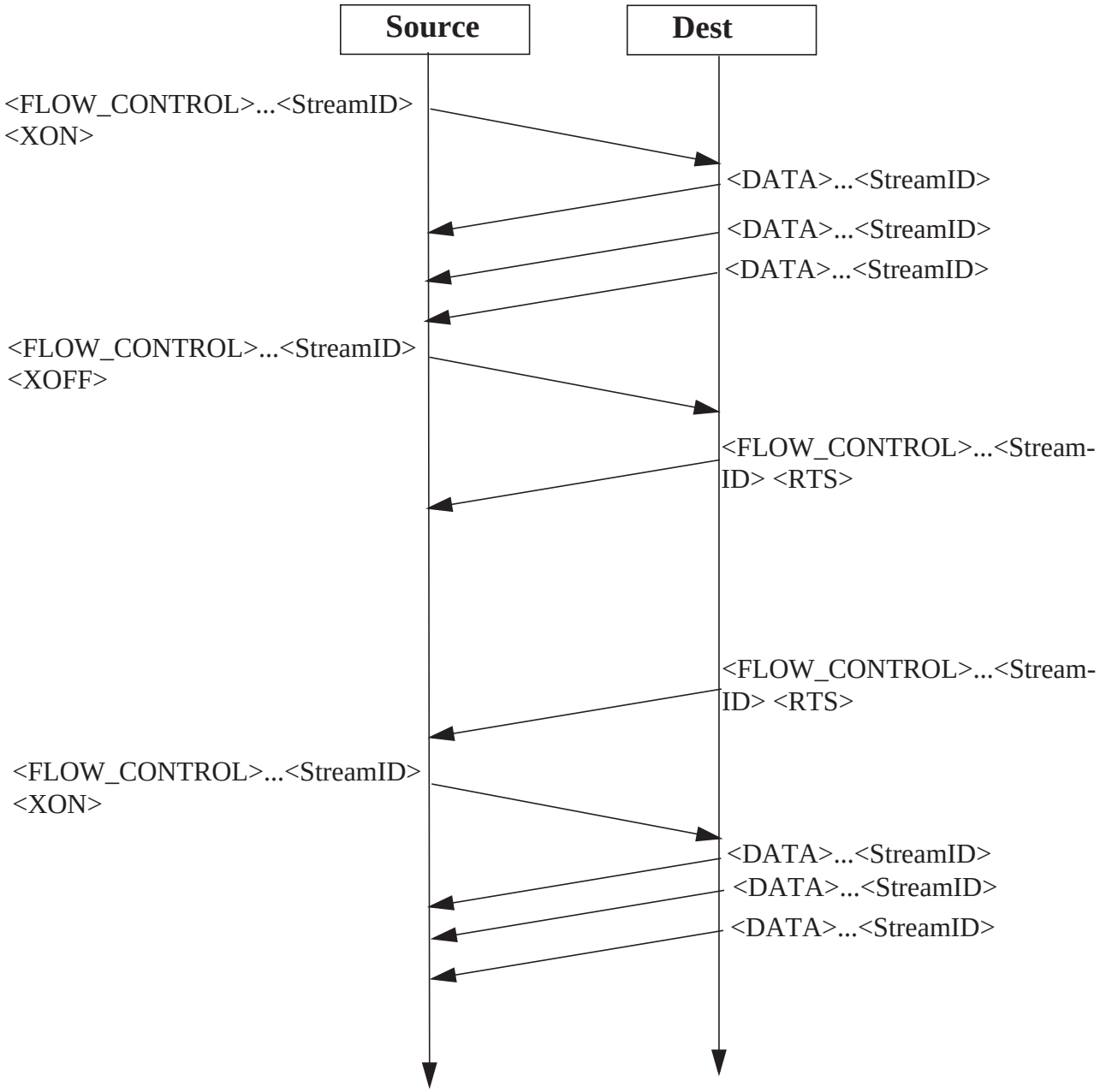


**Figure 3-5. Use of the FLOW_CONTROL Command**

FLOW_CONTROL packets are used to start and stop the transmission of data for a VSID, and to inform receivers that data is available to be transmitted. The above example shows a transmission being started (<XON>), stopped (<XOFF>), and the receiver being informed that there is data to be sent. The VSID is then started (<XON>), and data transmission begins anew.

## 3.7 Session Management Error Conditions and Recovery

Error conditions which the Session Management Protocol is designed to handle are:

- Message Loss
- Message Congestion at Source or Destination
- Session Management Protocol Non-compliance

Under some situations, some operating systems may need to drop messages. It is this source of message loss that the Session Management Protocol is designed to deal with. Message loss is not likely to occur in RapidIO fabrics and hardware when reliable communication channels are used. Session Management Protocol must not be used over unreliable channels.

RapidIO fabrics may experience congestion. Many mechanisms are available to manage congestion within RapidIO systems, including those designed in the Session Management Protocol.

The Session Management Protocol is designed to allow deployment of reliable systems in the face of software defects. The scope of defects is generally limited to non-responsiveness or poorly formed responses. Pathological software defects and malicious intent may still result in Session Management Protocol failure.

Non-compliance is tested in several ways. First, whenever illegal values are specified in individual commands, the system receiving the command must respond with an indication of error. Second, compliant software systems must be able to be configured into a validation mode, in which fields are tested for validity whenever possible and all reserved fields are tested to insure that the contents are zero.

### 3.7.1 Message Loss

Message loss for the Session Management Protocol REQUEST, OPEN, CLOSE and STATUS request messages is detectable through timeouts on ADVERTISE, ACCEPT, REFUSE, and STATUS responses.

Session Management control messages may be dropped due to lack of resources by the receiver. In this case, the transmitter can detect the fact that the message has been lost through a response timeout, which will trigger retransmission of the original request. A timeout period limits the number of retries which can be attempted.

Both the interval between retries, and the overall timeout period, are negotiated using attributes in the OPEN command. These attributes are the *REQUEST_RETRY_PERIOD* attribute, and the *REQUEST_TIMEOUT_PERIOD* attribute, respectively. By default, a REQUEST, OPEN, CLOSE or STATUS request should be retried once every 250 msec, with an overall timeout period of 1 second.

FLOW_CONTROL XOFF message loss is detected through continued transmission of DATA on the stream which was turned off. In this case, the implementation may

send additional FLOW_CONTROL XOFF messages, or if the congestion becomes severe, send a CLOSE request for the stream or conduit.

FLOW_CONTROL XON message loss is detected through timeouts on reception of a FLOW_CONTROL XON after the reception of a FLOW_CONTROL XOFF. The timeout period is set using the FLOW_CONTROL_XON TIMEOUT attribute in the OPEN message for the stream. To support a wide variation in timeout periods and data patterns, it may be necessary to handle to receive repeated FLOW_CONTROL XOFF messages for a stream.

If DATA message loss is allowed to occur in a system, it may be handled by the application or by the Session Management Protocol implementation.

Once message transmission has been timed out, the target of the message and all intervening nodes should be deemed suspect by the system.

In the event that no response is received in reply to a STATUS command within the reply delay period, the streamID specified in the STATUS command must be silently closed locally, and the remote node must be considered suspect.

For information on dealing with suspect nodes, refer to Chapter 3.7.3, “Session Management Protocol Non-Compliance,” on page 40.

## 3.7.2 Session Management Protocol Congestion Management

The Session Management Protocol is designed to avoid congestion conditions.

It is strongly recommended that messaging hardware implement a mechanism allowing ACCEPT, CLOSE, FLOW_CONTROL and STATUS messages to be sent and received with higher priority than messages containing commands OPEN, REQUEST, ADVERTISE, REFUSE, DATA and USERDEFINED. This allows the mechanisms for avoiding and managing congestion to operate in the presence of congestion.

The relative priority within the set of ACCEPT, CLOSE, FLOW_CONTROL and STATUS is implementation specific, with equality being the norm. Similarly, the relative priority within the set of OPEN, REQUEST, ADVERTISE, REFUSE, DATA, and USERDEFINED messages is implementation specific, with equality being the norm.

## 3.7.3 Session Management Protocol Non-Compliance

Session Management Protocol Non-Compliance is the term used to indicate one of two conditions. It can indicate that a target node is not operating in strict compliance with the Session Management Protocol and/or the timeout values used by the transmitter. It can also indicate that a target node is making illegal or non-specified use of reserved fields. In both cases, the target node is judged to be suspect (unreliable) by the Session Management Protocol implementation.

When a target node is judged to be suspect, the local system should follow the procedures, if any, defined for abnormal behavior for each command as described in Chapter 4, “Message Format Descriptions,” on page 45. System recovery actions may be initiated. The system recovery actions are outside the scope of this specification.

## 3.8 Rules for Session Management

This section describes restrictions and conditions during use of the Session Management Protocol.

### 3.8.1 Optional Features

This document describes a fully functional model, in which RapidIO end points can probe whether remote end points participate, send queries to discover what protocols each end point supports, and establish conduits and/or Virtual Streams with which to communicate. This functionality is designed for inter-operability of software, independent of the choice of OS.

In closed systems, the system designer may design the system so that each end point uses hard-coded information about all the remote end points with which it needs to communicate. In this case, the full functionality described in this document may not be necessary. Within such a closed system the use of the Component Tag CSR should be considered optional; however, a Session Management Protocol Register Extension Block, if available in the hardware, is not optional. The commands REQUEST and ADVERTISE are also optional. Designers of such systems should keep in mind that the rules below related to Attribute order may still place restrictions on the required order of Attributes to the OPEN command, and design the system accordingly.

Even in systems not intended as closed, the REQUEST and ADVERTISE commands are optional. No system may refuse to establish a connection based solely on the fact that no REQUEST command had been previously received and a corresponding ADVERTISE command sent in response. Systems which do not implement support for REQUEST and ADVERTISE will respond to REQUEST with STATUS, indicating Command_Unknown. When implementations do include support for REQEUEST and ADVERTISE, this also allows the system to attempt faster startup, trying OPEN first with the preferred protocol and attributes, and only fall back on the REQUEST / ADVERTISE mechanism in case of failure.

### 3.8.2 Attribute Related Rules

Implementations may choose to view Attributes as ordered lists. Therefore, any given implementation may refuse to open a Virtual Stream if the Attributes are not in the same order as presented in the ADVERTISE command. To ensure inter-operability, OPEN commands should maintain the order of Attributes that was

used in the ADVERTISE command.

When advertising attributes, there may be cases when multiple values for a specific Attribute are provided for a single protocol block, resulting in duplicate copies of the Attribute. In this case, the initiator may remove the duplicate copies in order to select a specific value, or it may leave all values in place. However, by the time the ACCEPT command is sent, all duplicate copies must be removed. Therefore, if an OPEN command is received, containing duplicate copies of any attribute, the recipient must remove the duplicates and determine a specific value for every required Attribute.

In some cases, duplicate copies of Attributes may indicate that the receiver can receive a range of values. In this case, there will always be exactly two copies of the Attribute, indicating the minimum and maximum values. A system receiving an ADVERTISE command with this condition may attempt to OPEN a Virtual Stream with an intermediate value. If the receiver responds with REFUSE, then the sender must not attempt any other intermediate value for any such attribute, but restrict itself to the values specified. A single attempt at intermediate values may be attempted. It is not necessary for implementations to exhaustively check attribute ranges.

In all systems, vendor-specific VSIDs may be handled in vendor-specific manner. In this case, the participants are not required to go through the establishment protocol with OPEN, ACCEPT, and REFUSE, but may be defined by the vendor to be available for immediate DATA messages.

### 3.8.3 Rules Related to Virtual Stream Status

If an OPEN message is received for a protocol and remote end point, where an existing Virtual Stream is already open, the recipient may choose to either re-send the existing VSID, or to create a new VSID for a second instance of the protocol connection.

In the case that the recipient of an OPEN message responds with an ACCEPT command containing the existing VSID, the ACCEPT should be followed by a STATUS message indicating that the stream is functional and indicating other status information as appropriate. If any DATA or FLOW_CONTROL command is received, which specifies a VSID which the recipient does not understand, the recipient must respond with a STATUS command indicating that the VSID is unknown. Upon receipt of such a STATUS command, the node should, at a time deemed appropriate by the system designer, check all other open VSIDs to verify that they are functional. This is intended to handle the condition where a node is rebooted and the Virtual Streams need to be closed and/or reestablished.

### 3.8.4 Rules Related to Vendor-Specific Commands

If any USERDEFINED command is received, which the recipient does not

understand, the receiver must respond with a STATUS command indicating that the USERDEFINED command is unknown.

### 3.8.5 Rules Related to Reserved Fields

Several fields in the structures described by this specification are marked reserved. On transmission, these fields must be filled with zeros.

Implementations of Session Management Protocol conforming to this specification must be able to be configured into a validation mode. When configured in this mode, the receiver must test all reserved fields for zero-filled content, and reject the received command if not zero-filled. The sender of such a message is thereafter to be treated as suspect (unreliable). Normal error handling is used in such a case, or if no other error handling is specified, a STATUS message must be returned, indicating Command_Unknown.

For higher performance, implementations of Session Management Protocol conforming to this specification may be able to be configured into a non-validation mode, in which reserved fields are ignored. Other error handling must not be disabled when the implementation is configured in non-validation mode.

## 3.9 Notes on Optional Features and Inter-Operability

For full inter-operability, an implementation must support all the features of this specification, including the optional features. There are conditions in which lack of optional features may restrict functionality and inter-operability. This section lists a sample of some potential consequences of not implementing all defined features, though this list is neither complete nor comprehensive. It is included as a warning of some consequences of design decisions during implementation.

### 3.9.1 Optional Attributes

Every implementation may choose to implement vendor specific attributes. If any vendor specific attribute is used, it should be optional. If not, then it is unlikely that other systems, which may not understand the vendor-specific attribute, will be able to inter-operate with the implementation.

For example, if the *DATA_OFFSET_VENDOR* and *DATA_OFFSET* attributes are required, then the system will not be able to communicate with other implementations. However, if the *DATA_OFFSET_VENDOR* and *DATA_OFFSET* attributes are used but not required, then the implementation may operate more efficiently with other systems using the feature, but will continue to inter-operate with systems not implementing that feature.

### 3.9.2 REQUEST and ADVERTISE

The REQUEST and ADVERTISE message types are listed as optional. This is true in two senses. First, no system is required to use REQUEST or ADVERTISE when attempting to open a connection. Second, no system is required to recognize incoming REQUEST and ADVERTISE messages, but may respond to them with STATUS messages indicating Command_Unknown.

However, any arbitrary node, which wants to establish a connection with a node that does not recognize incoming REQUEST messages, may not be able to determine the appropriate attributes to use for the desired protocol. Without a listing of the available attributes from an ADVERTISE message, it is not possible reliably to make a connection. Such systems would not inter-operate.

Furthermore, if an implementation that does not support REQUEST and ADVERTISE messages also chooses to view attributes as an ordered list, then even in the case that some external agent provided the attributes and values to use, then the two systems may still not be inter-operable due to ordering restrictions.

# Chapter 4 Message Format Descriptions

## 4.1 Introduction

This chapter contains the definition of the data streaming packet format.

## 4.2 Control Message Formats

All message formats are given in big endian format - the most significant octet is on the left of each field.

### 4.2.1 REQUEST

A REQUEST message is used to request information related to protocols supported by the remote. There are two primary variants to the REQUEST message: first, to request a list of protocols supported by the remote; second, to request attributes of a specified protocol. These two variants are distinguished by the contents of the ProtoID field. A value containing all ones, 0xffff, indicates a request for a list of protocols without attributes. Protocol specific values of ProtoID are described in “Section 3.5.2.3, Protocol Identifier: <ProtoID>” on page 29.

**Table 4-1. REQUEST Message Format**

| | Octet 0 | Octet 1 | Octet 2 | Octet 3 |
|---|---|---|---|---|
| Word 0 | <CMD=0x01> | <VER> | <SourceID> | <SourceID> |
| Word 1 | <DestID> | <DestID> | <COS> | Reserved |
| Word 2 | <ProtoID> | <ProtoID> | <NumAttrib> | <NumAttrib> |
| Word 3 | Reserved | Reserved | Reserved | Reserved |

The form of REQUEST, which is used to request the attributes for a specified protocol, may limit the request by including a list of required attributes. The <NumAttrib> field contains the number of attributes listed, or zero if no attributes are listed. Upon receipt of a REQUEST message, the receiver should respond only with attributes that match the attributes included in the REQUEST message.

All the header fields in a REQUEST message other than <NumAttrib> are described in Chapter 3.5.2, “Message Header Fields,” on page 28.

### 4.2.2 ADVERTISE

An ADVERTISE message is sent in response to a REQUEST message, to identify supported protocols or attributes of a specified protocol, depending on the contents of the REQUEST message. The two variants are distinguished by the value of the <A> bit in the message header.

**Table 4-2. ADVERTISE Message Format - Protocol Attributes**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x02> | <VER> | <SourceID> | <SourceID> |
| <DestID> | <DestID> | <S+A+Count> | <Count=M> |
| <ProtoID_1> | <ProtoID_1> | <numAttributes_1=N1> | <numAttributes_1=N1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_N1> | <Attribute_N1> | <Attribute_N1> | <Attribute_N1> |
| <Attribute_N1> | <Attribute_N1> | <Attribute_N1> | <Attribute_N1> |
| <ProtoID_2> | <ProtoID_2> | <numAttributes_2=N2> | <numAttributes_2=N2> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_N2> | <Attribute_N2> | <Attribute_N2> | <Attribute_N2> |
| <Attribute_N2> | <Attribute_N2> | <Attribute_N2> | <Attribute_N2> |
| ..... | | | |
| <ProtoID_M> | <ProtoID_M> | <numAttributes_2=NM> | <numAttributes_2=NM> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_NM> | <Attribute_NM> | <Attribute_NM> | <Attribute_NM> |
| <Attribute_NM> | <Attribute_NM> | <Attribute_NM> | <Attribute_NM> |

**Table 4-3. ADVERTISE Message Format - Protocol List**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x02> | <VER> | <SourceID> | <SourceID> |
| <DestID> | <DestID> | <1+0+Count> | <Count> |
| <ProtoID_1> | <ProtoID_1> | <ProtoID_2> | <ProtoID_2> |
| ..... | | | |
| <ProtoID_Count> | <ProtoID_Count> | Reserved | Reserved |
| Reserved | Reserved | Reserved | Reserved |

<S>: (1 bit) indicates whether or not the requested protocol is supported. If the value of <S> is 1, then the protocol is supported. If the value of <S> is 0, then the protocol is not supported, and the <A> bit must be set to zero (0).

<A>: (1 bit) indicates whether or not the ADVERTISE message includes attributes or whether it is a simple list of protocols. Note that if <S> is set, then the <A> bit is invalid and must contain the value zero (0). <A>=1 indicates that the format follows Table 4-2. <A>=0 indicates that the format follows Table 4-3.

If <A>=0, then the message contains a list of <ProtoID> values, as shown in Table 4-3. Note that the final Reserved fields shown in the table indicate padding to a multiple of 8 octets, and that there may be zero to seven such Reserved octets, and not exactly the six (6) octets shown in the Table 4-3. The Reserved octets must contain zero (0) values.

The valid values of S+A are 0b00, 0b10, and 0b11. The value of 0b01 is reserved.

If <S>=0, then <Count> is invalid and must be set to zero (0). If <S>=1, then <Count> is valid. <Count> is a 14 bit value the number of <ProtoID>'s included in the ADVERTISE message. Note that this field does not indicate octet length, but rather the number of protocols.

If <A>=0, then the message contains a list of <ProtoID> values, padded out to a multiple of 8 octets.

If <A>=1, then the message contains a list of Attributes associated with the <ProtoID>, padded out to the nearest multiple of 8 octets. The number of Attributes in the message is contained in the <numAttributes> field. Note that a value of 0, meaning that no Attributes are supported for this <ProtoID>, is valid.

### 4.2.3 OPEN

An OPEN message is used to request that the remote system create a stream or virtual stream suitable for the remote system to receive data on. The OPEN message specifies the protocol which is to be carried on the stream or virtual stream, as well as any protocol attributes that need to be used.

**Table 4-4. OPEN Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x03> | <VER> | <SourceID> | <SourceID> |
| <ProtoID> | <ProtoID> | <numAttributes=N> | <numAttributes=N> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |

Note that the packet format specifies the SourceID, but no DestID. When the OPEN message is received, the receiver uses it's own nodeID as the recipient of traffic, and the SourceID specified in the message is the information that the recipient needs to have in order to pass control traffic.

The OPEN request must have the *OPEN_MESSAGE_NUMBER* attribute as the first attribute in its attribute list.

### 4.2.4 ACCEPT

An ACCEPT message is used to inform a requestor that a stream or virtual stream is now open and that the requestor can send data traffic using the StreamID specified in the ACCEPT message.

**Table 4-5. ACCEPT Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x04> | <VER> | <DestID> | <DestID> |
| <ack-type> | <COS> | <StreamID> | <StreamID> |
| <ProtoID> | <ProtoID> | <numAttributes=N> | <numAttributes=N> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |

"ack type" values:

- 0: normal ACK
- 1-255: reserved

For ACCEPT and REFUSE messages, the DestID is the nodeID of the receiver, which is the system sending the ACCEPT or REFUSE message.

The only attribute required in the ACCEPT message is the *OPEN_MESSAGE_NUMBER* attribute from the OPEN. Other attributes from the REQUEST message may optionally be copied to the ACCEPT message.

### 4.2.5 REFUSE

A REFUSE message is sent in response to an OPEN request, if the requestee cannot create a stream or virtual stream with the protocol and attributes specified in the OPEN request.

**Table 4-6. REFUSE Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x05> | <VER> | <DestID> | <DestID> |
| <nack-type> | 0xFF | 0xFF | 0xFF |
| <ProtoID> | <ProtoID> | <numAttributes=N> | <numAttributes=N> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_1> | <Attribute_1> | <Attribute_1> | <Attribute_1> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| <Attribute_2> | <Attribute_2> | <Attribute_2> | <Attribute_2> |
| ..... | | | |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |
| <Attribute_N> | <Attribute_N> | <Attribute_N> | <Attribute_N> |

“nack type” values:

- 0: normal NACK
- 1-255: reserved

For ACCEPT and REFUSE messages, the <DestID> is the nodeID of the receiver, which is the system sending the ACCEPT or REFUSE message.

The only attribute required in the REFUSE message is the *OPEN_MESSAGE_NUMBER* attribute from the OPEN. Other attributes from the REQUEST message may optionally be copied to the REFUSE message.

### 4.2.6 FLOW_CONTROL

A FLOW_CONTROL message is used to regulate traffic and manage traffic congestion. There are three types of flow control behavior which can be specified. First, use the XON command to enable data traffic. Second, use the XOFF command to suspend data traffic. In the case that data traffic is suspended, RTS can be used to inform the stream owner (data receiver) that data is available to be sent.

**Table 4-7. FLOW_CONTROL Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x07> | <VER> | <COS> | <Flow_Control> |
| <SourceID> | <SourceID> | <StreamID> | <StreamID> |
| <ProtoID> | <ProtoID> | Reserved | Reserved |
| Reserved | Reserved | Reserved | Reserved |

The default state when a stream is opened is for traffic to be enabled. No explicit FLOW_CONTROL command XON needs to be sent in the default case.

Once an RTS message is sent, additional RTS messages must not be sent to indicate that additional data is available. A single RTS message is sufficient. Because no explicit response to RTS is required, the sender of a FLOW_CONTROL command RTS may retry the message at 250 msec intervals, until 1 second has elapsed.

Flow_Control values:

- XON: 0x01
- XOFF: 0x00
- RTS: 0xff

No node is required to initiate FLOW-CONTROL. However, all RapidIO nodes must accept and handle FLOW-CONTROL commands coming from the remote. Note that it is possible to receive a FLOW-CONTROL command with XON flow when no previous FLOW-CONTROL command with XOFF flow has been received, since the FLOW-CONTROL XOFF command could have been lost.

### 4.2.7 CLOSE

A CLOSE message is used to terminate a stream or virtual stream. Either endpoint may initiate the CLOSE behavior. In the event that the stream or virtual stream is part of a conduit, both streams or virtual streams must be closed at the same time. The means of determining that a stream is part of a conduit is implementation specific.

**Table 4-8. CLOSE Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x08> | <VER> | <SourceID> | <SourceID> |
| <DestID> | <DestID> | <COS> | Reserved |
| <StreamID> | <StreamID> | Reserved | Reserved |
| Reserved | Reserved | Reserved | Reserved |

NOTE: both SourceID and DestID must be specified in order to allow both endpoints to initiate a shutdown. Regardless of which endpoint initiates the CLOSE, the SourceID indicates the node sending data.

When receiving a CLOSE message, the receiver of the CLOSE must reply with a STATUS message indicating that the StreamID has been closed.

### 4.2.8 STATUS

A STATUS message is used to request the status of a stream or virtual stream, to report the status of a stream or virtual stream, and to indicate certain error conditions such as illegal commands. The status bits indicate the status of the stream and/or the reason for sending the STATUS message.

**Table 4-9. STATUS Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x10> | <VER> | <COS> | <DataSize> |
| <SourceID> | <SourceID> | <StreamID> | <StreamID> |
| <Mailbox> | <Reserved> | <CmdID> | <CmdVersion> |
| <Status> | <Status> | <Status> | <Status> |
| <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> |
| <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> |
| ..... | | | |
| <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> |
| <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> | <ContextSpecificData> |

**Table 4-10. Status Bit Values**

| Status Bit | Meaning |
|---|---|
| 0x00000001 | Stream Unknown |
| 0x00000002 | Stream Functional |
| 0x00000004 | Ready to Receive |
| 0x00000008 | Data Ready to Send |
| 0x0ffffff0 | Reserved |
| 0x10000000 | Error |
| 0x20000000 | Closed |
| 0x40000000 | Command Unknown |
| 0x80000000 | Request Status of Remote |

A STATUS message consists of two parts. The initial header fields, as shown in Table 4-9, and context specific data. The contents of the context specific data depends on the reason for sending the STATUS message. In all cases, the <DataSize> field is used to indicate he number of 8-octet words contained in the <ContextSpecificData> fields. That is, the number of octets in the <ContextSpecificData> field is eight times the value of <DataSize>.

If the STATUS message is sent in response to an invalid, unknown, or malformed command, then the original command is sent in the <ContextSpecificData> fields

and the Command_Unknown flag bit is set. If the STATUS message is sent in response to a STATUS message with the Request_Status_of_Remote bit set, then the <ContextSpecificData> fields should be set to a copy of the ProtoID and attributes used to create the StreamID or virtual StreamID. If the STATUS message is sent in order to request the status of a StreamID, the <ContextSpecificData> fields are not used, and <DataSize> must be set to zero.

In the case that the Stream_Unknown bit is set in the status field, the only two valid flag bits are Stream_Unknown and Closed. Other flags are implicitly Reserved, and must be set to zero by the sender.

When configured in validation mode, the receiver must test the value of the Reserved bits of the Status field, including the bits explicitly marked as Reserved and the bits which are implicitly Reserved if the Stream_Unknown bit is set.

The receiver must not respond to an illegally formed STATUS command with another STATUS command. If not well-formed, the STATUS command must be ignored. Such as system should indicate the reception of an illegal STATUS message with a message printed to the console or other implementation specific error reporting mechanism, but the method and format to indicate this error is beyond the scope of this specification.

When a STATUS message is received with the ’Request_Status_of_Remote’ bit set, a STATUS message must be sent back to the originator for the streamID requested. The STATUS message which is sent back must not have the ’Request_Status_of_Remote’ bit set.

The STATUS message will only include <ContextSpecificData> fields when a STATUS message is sent as a response for an unrecognized/unsupported/malformed message or in response to a STATUS message.

### 4.2.9 User Defined

A number of message IDs are reserved for application specific commands.

If a system receives a USERDEFINED command which is not understood, it must respond with a STATUS message, with the Command Unknown bit set to indicate that it does not understand the command.

**Table 4-11. USERDEFINED Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0xF0 to 0xFF> | <VER> | <COS> | Reserved |
| <SourceID> | <SourceID> | <StreamID> | <StreamID> |
| <user-defined data> | <user-defined data> | <user-defined data> | <user-defined data> |
| <user-defined data> | <user-defined data> | <user-defined data> | <user-defined data> |

# 4.3 Data Formats

## 4.3.1 DATA Message Format, MAILBOX

DATA message format is used to send virtual Type 9 data (streams, see *RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification*) when the only conveyance provided by the hardware is Type 11 (messages). The <Mailbox> field is set to the mailbox ID used by the receiver of the data. <COS> is defined in Table 3.5.2.4, “Class of Service: <COS>,” on page 29.

**Table 4-12. DATA Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x06> | <VER> | <Mailbox> | <COS> |
| <Reserved> | <Reserved> | <SourceID> | <SourceID> |
| <S+E+00 +Length(4bits)> | <Length> | <StreamID/PDU-Length> | <StreamID/PDU-Length> |
| Payload Data octet | Payload Data octet | Payload Data octet | Payload Data octet |
| ..... | | | |
| Payload Data octet | Payload Data octet | Payload Data octet | Payload Data octet |

In the case that data is received for a StreamID unknown to the receiver, then the receiver must respond with a STATUS message with the Closed bit set to indicate that the stream is no longer open. It may also send STATUS messages to all other StreamIDs for the same remote system, whether the StreamID is for transmitted traffic or for received traffic.

The S and E bits are used to segment transfers larger than the maximum PDU length over multiple messages. The S bit indicates that this is the first segment in a transfer. The E bit indicates that this is the last segment in a transfer. If neither the S or the E bit is set, then this is one of the middle segments in a multi-segment transfer. If a transfer fits within a single segment, then both the S and E bits are set. PDUs in a multi-segment transfer must be sent in order: Start segment, middle segments, End segment.

The length field contains the size of the data payload included in this message excluding the encapsulation header. The StreamID/PDU-Length field contains either the StreamID of the message or the length of the data payload excluding the encapsulation header. If either the S or E bit is set, then the StreamID/PDU-Length field contains the StreamID. If neither S nor E is set, then the StreamID/PDU-Length field contains PDU-Length, which includes the total size of all RapidIO messages making up this packet, exclusive of encapsulation headers.

### 4.3.2 DATA1 Message Format, Large PDU

DATA1 message format is used in situations where large amounts of data need to be transferred as a block. All fields have the same meaning as in the DATA header.

**Table 4-13. DATA Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x09> | <VER> | <Mailbox> | <COS> |
| <Reserved> | <Reserved> | <SourceID> | <SourceID> |
| <S+E+Length(6bits)> | <Length> | <Length> | <Length> |
| <StreamID/PDU-Length> | <StreamID/PDU-Length> | <Reserved/PDU-Length> | <Reserved/PDU-Length> |
| Payload Data octet | Payload Data octet | Payload Data octet | Payload Data octet |
| ..... | | | |
| Payload Data octet | Payload Data octet | Payload Data octet | Payload Data octet |

All fields in this header format are used in the same manner as the DATA command. However, the capacity of this format allows transfers of up to 4 gigabytes of data.

Underlying hardware may place limits on the size of messages, such as only allowing data to be transferred in multiples of eight octets. In this case, padding must be used where necessary to fill the size constraints. If padding is used, the data must contain zeros, and the pad octets are not included in the <S+E+00+Length>, <Length>, <StreamID/PDU-Length>, and <Reserved/PDU-Length> fields.

### 4.3.3 DATA2 Message Format

The DATA2 command is available for use when the full information contained in the DATA and DATA1 headers is not required. Note that use of this format restricts the packet size to the size limited by hardware, so that the maximum PDU size is 16380 octets, but may be smaller due to hardware constraints.

**Table 4-14. DATA Message Format**

| 3 | 2 | 1 | 0 |
|---|---|---|---|
| <CMD=0x0A> | <Implementation-Specific> | <S+E+Length (6 bits)> | <Length> |

The Implementation-Specific field is available for use by implementers. Note that the DATA2 message format is only used when the DATA Header attribute is used during stream creation. The value to be used in this field is the seventh octet of the DATA Header attribute, preceding the DATA command value.

### 4.3.4 DATA3 Zero-length DATA header

When using some conveyances, it may be possible to fully segregate traffic based on streamID or other information. In this case, no header is required. For the

purposes of determining the DATA header format, this can be considered as CMD=0x0B, though the actual command value of 0x0B must never be transmitted.

This header format is used for hardware assisted data streaming, as defined in *RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification*.

### 4.3.5 Data Streaming

When the data is carried by the Data Streaming Protocol is uses a logical layer packet format defined in *RapidIO Interconnect Specification Part 10: Data Streaming Logical Specification*. In this case, no additional header needs to be used.

It is strongly recommended that the RapidIO specific information necessary for software to compose and respond to Data messages is made accessible to software through implementation specific means by hardware that supports type 9 packets.

Blank page

# Chapter 5 Registers

## 5.1 Introduction

Before sending session management protocol messages, it is necessary to know if the target node supports session management protocol, and the messaging method used by the target node. The target node advertises this information using registers.

A Session Management Protocol register extension block may be used to advertise a target nodes session management protocol parameters. This method is the recommended approach.

Devices which are capable of accepting session management protocol messages may advertise this fact in the Component Tag CSR. The Component Tag CSR may be used to advertise session management protocol parameters only if a Session Management Protocol register extension block is not present in a device. Only devices whose Destination Operations CAR and Source Operations CAR indicate support for Data Message and/or Data Streaming transactions may advertise support for the Session Management Protocol using the Component Tag CSR.

The use of the Component Tag CSR is deprecated for new devices.

## 5.2 Session Management Protocol Extended Features Register Block

Where Reserved fields are used in the following structures, the value must be set to zero. Implementations configured in validation mode should check these fields when first reading the Session Management Protocol Extended Features Register Block, and indicate the presence of an illegal Session Management Protocol Extended Features Register with a message to the console or other implementation specific error reporting mechanism, but the method and format to indicate this error is beyond the scope of this specification.

Note that there should be an instance of the Session Management Protocol Extended Features Register Block for every conveyance supported by the endpoint.

### 5.2.1 Session Management Protocol Register Block Header (Block Offset 0x0)

The Session Management Protocol Register Block Header contains the EF_PTR to the next extended features block and the EF_ID that identifies this as the Session Management Protocol Register Block Header.

**Table 5-1. Bit Settings for Session Management Protocol 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 | 0x000C | Hard wired Extended Features ID |

### 5.2.2 Session Management Protocol Register Write Enable CSR (Block Offset 0x4)

The Session Management Protocol Advertisement register is used allow an external RapidIO entity write access to the Session Management Protocol registers, which otherwise are read only.

The operation of this register is identical to the Host Base Device ID CSR specified in Part 2 Common Transport Specification:

- When the Lock_Val is 0xFFFF, all other registers in this block are read only.
- Writing to this register when Lock_Val is 0xFFFF sets the Lock_Val field to the value written.
- When the Lock_Val field is not 0xFFFF, all registers in this block are writable. Implementation specific checking may be done on the write transactions to this block.
- When the Lock_Val field is not 0xFFFF, writing the value of the Lock_Val field to this register resets the Lock_Val field to 0xFFFF.
- When the Lock_Val field is not 0xFFFF, writing a value different from the Lock_Val field to this register does not affect the value of the Lock_Val field.
- Writing 0xFFFF to the Lock_Val field when the Lock_Val field is 0xFFFF has no effect.

**Table 5-2. Bit Settings for Session Management Protocol Register Write Enable Register**

| Bit | Name | Reset Value | Description |
|---|---|---|---|
| 0-15 | Reserved | 0x0000 | Not Used |
| 16-31 | Lock_Val | 0xFFFF | See description above |

### 5.2.3 Session Management Advertisement CSR (Block Offset 0x8)

The Session Management Protocol Advertisement register is used to indicate whether or not the node supports the session management protocol and, if so, how to send session management protocol messages to the node.

All of the fields of this register are read-only using RapidIO maintenance transactions, but may be written by the local processing element.

**Table 5-3. Bit Settings for Session Management Protocol Advertisement Register**

| Bit | Name | Reset Value | Description |
|---|---|---|---|
| 0-3 | Conveyance | Impl. Spec. | Identifies which conveyance this Session Management Protocol register block applies to:<br>0x0 - Messaging (Type 11)<br>0x1 - Data Streaming (Type 9)<br>0x2-0xE - Reserved<br>0xF - Not Supported<br><br>Write Protected by Session Management Protocol Register Write Enable register. |
| 4-31 | Conveyance_Info | Impl. Dep. | Conveyance Info<br>This field communicates conveyance specific information for reception of Session Management Protocol messages.<br><br>If Conveyance = type 11, the field format is 0x00000nn<br>where nn represents the mailbox ID, formatted according to the definition in Part 2.<br><br>If Conveyance = type 9, the field format is 0x0ccssss<br>where:<br>cc - Class of Service (COS) value to be used<br>ssss - StreamID to be used<br><br>Write Protected by Session Management Protocol Register Write Enable register. |

### 5.2.4 Session Management Attribute Range CSR (Block Offset 0xC)

The Session Management Protocol Advertisement register is used to indicate whether or not the node supports the session management protocol and, if so, how to send session management protocol messages to the node.

All of the fields of this register are read-only using RapidIO maintenance transactions.

**Table 5-4. Bit Settings for Session Management Attribute Range Register**

| Bit | Name | Reset Value | Description |
|---|---|---|---|
| 0-7 | — | 0x00 | reserved |
| 8-15 | Sess_Mgmt_Max_Attr | Impl. Dep. | This field contains the maximum number of attributes. Each attribute takes up 8 octets.<br><br>0 - No Session Management Protocol Attribute Registers<br>1 - 2 Session Management Protocol Attribute Registers<br>2 - 4 Session Management Protocol Attribute Registers<br>3 - 6 Session Management Protocol Attribute Registers<br>...<br>n - 2n Session Management Protocol Attribute Registers<br>...<br>0xFE - 508 Session Management Protocol Attribute Registers<br>0xFF - Reserved<br><br>This field is Read Only from the RapidIO interface, but may be writable by the local Processing Element. |
| 16-19 | — | 0x0 | reserved |

## Table 5-4. Bit Settings for Session Management Attribute Range Register

| Bit | Name | Reset Value | Description |
|---|---|---|---|
| 20-23 | Sess_Mgmt_Init_Stage | Impl. Dep. | This field indicates progress through the register initialization sequence, with a standard value indicating that initialization is complete.<br><br>0x0 - Initialization of registers complete<br>0x1-0xF - Initialization of registers incomplete, initialization stage X in progress.<br><br>Write Protected by Session Management Protocol Register Write Enable register. |
| 24-31 | Sess_Mgmt_Num_Attr | Impl. Dep. | This field contains the number of valid attributes following this register. Each attribute takes up 8 octets.<br><br>0 - No Session Management Protocol Attribute Registers<br>1 - 2 Session Management Protocol Attribute Registers<br>2 - 4 Session Management Protocol Attribute Registers<br>3 - 6 Session Management Protocol Attribute Registers<br>...<br>n - 2n Session Management Protocol Attribute Registers<br>...<br>0xFE - 508 Session Management Protocol Attribute Registers<br>0xFF-0xFFFE - Reserved<br>0xFFFF - Session Management Protocol Attribute Registers not initialized<br><br>Write Protected by Session Management Protocol Register Write Enable register. |

### 5.2.5 Session Management Protocol Attributes 0-508 CSRs (Block Offset 0x10-0x7F8)

The number of valid Session Management Protocol Attribute registers is indicated by the Sess_Mgmt_Num_Attr field of the Session Management Protocol Advertisement register.

The number of Session Management Protocol Attributes registers is implementation specific. Up to 508 Session Management Protocol Attributes registers can exist.

It is recommended that at least 8 Session Management Protocol Attributes registers be implemented.

**Table 5-5. Bit Settings for Session Management Protocol Attributes 0-508 Registers**

| Bit | Name | Reset Value | Description |
|---|---|---|---|
| 0-63 | Attribute Data | Impl Dep | Attribute specification data, formatted as per the attributes defined in Section 3.5.3 and in the protocol-specific chapters. |

## 5.3 Component Tag CSR Session Management Protocol Advertisement

If a Session Management Protocol Register Block Header does not exist within the registers for a device, the device may advertise support for the Session Management Protocol using the Component Tag CSR, as described in Table 5-6.

Only devices whose Destination Operations CAR and Source Operations CAR indicate support for Data Message and/or Data Streaming transactions may advertise support for the Session Management Protocol using the Component Tag CSR.

If Bit 0 of the Component Tag CSR contains a value of 0, then the node does not conform to this specification. In this case, this specification puts no requirements on the remaining bits of the Component Tag CSR.

If Bit 0 contains a value of 1, then the node must not use the first 16 bits of this register for any purpose other than advertising support for Session Management Protocol messages.

**Table 5-6. Component Tag CSR Bit Usage**

| Bit | Field Name | Description |
|---|---|---|
| 0 | Sess_Mgmt_Sup | Session Management Protocol Support<br>This bit indicates whether or not a processing element supports the Session Management Protocol.<br><br>0b0 - This processing element does not support the Session Management Protocol<br>0b1 - This processing element does support the Session Management Protocol |

**Table 5-6. Component Tag CSR Bit Usage**

| Bit | Field Name | Description |
|---|---|---|
| 1-2 | Sess_Mgmt_Chan | Session Management Protocol Channel<br>This bit indicates what logical layer to use, and how to interpret the Channel Information below:<br><br>0b00 = Use Message Passing (type 11)<br>0b01 = Use Data Streaming (type 9)<br>0b10 = Reserved<br>0b11 = Expansion<br><br>This field is only valid if Sess_Mgmt_Sup is 0b1. |
| 2-15 | Sess_Mgmt_Chan_Info | Session Management Protocol Channel Info<br>This field communicates channel specific information for reception of Session Management Protocol messages.<br><br>If Session Management Channel = type 11:<br>bits 2-15 = 0b00000nnnnnnnn<br>where the low 8 bits are the mailbox ID (formatted according to the definition in Part 2)<br><br>If Session Management Channel = type 9:<br>bits 2-5 = 0bx0000nnnnnnnn where:<br>x = 1 = StreamID MSB = 0xFF<br>x = 0 = StreamID MSB = 0x00<br>n = StreamID LSB<br>The StreamID is the Stream to use to embed the management messages. The COS field in the VSID should be the highest priority COS supported by the interface.<br>ex: 0b100001111000 = StreamID = 0xFFF0<br><br>This field is only valid if Sess_Mgmt_Sup is 1. |

The Sess_Mgmt_Chan bits indicate the conveyance to use for Session Management Protocol traffic. The value of 0b11, Expansion, is used to indicate that a reserved, expanded Sess_Mgmt_Chan field, larger than two bits, is to be used. In this case, Sess_Mgmt_Chan_Info will consist of fewer bits.

Blank page

# Chapter 6 Vendor-Defined Protocols

## 6.1 ProtoID

The ProtoID value for vendor-defined protocols is 0x0101.

## 6.2 Attributes

Two protocol attributes are required for vendor-defined protocols. These two attributes distinguish a vendor-defined protocol from all other vendor-defined protocols. The attributes are the *VENDOR* attribute, and the *PROTOCOL-NAME* attribute.

### 6.2.1 *VENDOR* attribute

The VENDOR attribute, described in “Section 3.5.3.1, VENDOR Attribute” on page 31, is required. It must be the first attribute listed.

### 6.2.2 *PROTOCOL_NAME* attribute

The *PROTOCOL-NAME* attribute is an 8-bit attribute ID (0x01) with a 56-bit value. The *PROTOCOL-NAME* attribute is required for all vendor-defined protocols, and must immediately follow the *VENDOR* attribute. The format of the attribute value is defined by the vendor. The sole restriction is that the value defined by the vendor must uniquely identify the protocol in that it differentiates the protocol from all other protocols defined by the vendor. It may consist of an ASCII string or a numeric value, at the discretion of the vendor defining the protocol.

### 6.2.3 Other attributes

Other attributes may be defined by the vendor.

## 6.3 Other Requirements for Vendor-Defined Protocols

Vendors wishing to make their protocols available may choose to create an RFC describing attributes and other issues related to their protocol.

Blank page

# Chapter 7 Ethernet Encapsulation

## 7.1 ProtoID

The protocol ID value for Ethernet encapsulation is 0x0102.

## 7.2 Attributes

Several protocol attributes are required: *MTU*, *CONVEYANCE*, and *MAC_ADDRESS*. No ordering restrictions are placed on these attributes by the specification, though implementations may impose ordering restrictions. Additional optional protocol attributes exist, depending on the system configuration.

### *7.2.1 MTU* Attribute

The MTU is assigned with the 16-bit attributed ID 0x8002, leaving six octets of data for the MTU value. Only the lowest two octets should be used, allowing a maximum MTU of 64 KBytes.

### 7.2.2 *CONVEYANCE* Attribute

The *CONVEYANCE* attribute, as described in “Section 3.5.3.10, CONVEYANCE Attribute” on page 34, is required for Ethernet encapsulation.

### 7.2.3 *MAC_ADDRESS* Attribute

The MAC address is specified with the 16-bit attribute ID 0x8003, leaving six octets of data for the MAC address.

This protocol is intended for use where RapidIO nodes and Ethernet nodes may co-exist on the same virtual Ethernet segment. Therefore, MAC addresses are required to conform to industry standards. This includes the requirement that the first three octets of the MAC address should be a valid OUI (Organizationally Unique Identifier) assigned by IEEE.

## 7.3 Other Requirements of Ethernet Encapsulation

There are a number of additional requirements for Ethernet encapsulation. These requirements are discussed in the following sections.

### 7.3.1 Dropped Messages

In Ethernet, it is appropriate for a packet to be dropped if there are difficulties sending it, since higher level protocols handle retransmission. Although this differs from the normal goal of RapidIO, that is, to have reliable transmission, there are some conditions in which it is appropriate to drop Ethernet encapsulation messages.

If a node has received an XOFF from the remote system using Ethernet encapsulation at the time a new message is to be sent, the node must retain at least one pending message for transmission, however it may choose to drop additional messages. The pending message(s) should be the most recent message(s), and older messages should be dropped.

### 7.3.2 Broadcast

#### 7.3.2.1 Broadcast With Multicast Extensions

If multicast capabilities are available in the RapidIO switches, they should be used for broadcast messages. The value of the destination ID for ethernet encapsulation should be 0xFE for 8-bit IDs or 0xFFFE for 16-bit IDs.

#### 7.3.2.2 Broadcast Without Multicast Extensions

If multicast capabilities are not available, broadcast messages must be sent by unicast transmission. The reserved StreamID value 0xF000 may be used for transmission of broadcast traffic. Note that no OPEN message is required for this StreamID, so no dedicated resources are required to be permanently allocated in order to receive broadcast traffic. Receivers of broadcast traffic know that it is a broadcast message, by reading the StreamID of incoming messages.

#### 7.3.2.3 Vendor defined Broadcast Server

Vendors may wish to define a vendor-specific protocol for use by a broadcast server, thereby eliminating the requirement for multiple unicast message transmission by other nodes. Such broadcast servers must not re-transmit incoming broadcast messages received on the broadcast StreamID 0xF000.

### 7.3.3 Ingress/Egress Nodes

Ingress/Egress nodes may be configured as switches or as routers. If configured as routers, ingress and egress traffic are handled at a higher level protocol, and not the subject of this protocol.

If ingress/egress nodes are configured as switches, the node must forward ingress and egress traffic. This is handled as a layer 2 switch, the behavior of which is well understood and not defined in this protocol.

When Ethernet-over-RapidIO traffic is transmitted outside the RapidIO fabric, the

egress system must be able to be configured to forward broadcast packets to the external network and forward external broadcast packets into the RapidIO fabric. This can be done by forwarding RapidIO messages received on StreamID 0xF000 to ethernet interfaces. Incoming broadcast packets coming from Ethernet interfaces must be sent using the RapidIO broadcast mechanism.

Blank page

# 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.

**Conduit**. A bidirectional data transfer mechanism consisting of two streams or virtual streams, one for communication in each direction.

**Conveyance**. A communication channel, e.g. mailbox, stream, shared memory mechanism, etc.

**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.

**Double-word**. An eight octet or 64 bit quantity, aligned on eight octet 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 octet or 16 bit quantity, aligned on two octet 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.

**Protocol Attributes.** Information relevant to communication using a particular protocol, or to data transferred within the context of a connection using that protocol.

## 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*.

**Session Management Protocol.** The protocol specified in this document, used for negotiation of communication sessions and optionally for data transfers.

**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.

**Suspect.** A communication partner, which may not fully conform to the session management protocol.

**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, used to identify individual data streams. When using Type 9 (streaming) as the conveyance for data transfers, the VSID is encapsulated in the Type 9 protocol. When using Type 11 (messaging) as the conveyance for data transfers, the VSID is encapsulated in fields in the DATA or DATA1 Session Management Protocol commands.

**W**

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