RapidIO™ Interconnect Specification Annex 1: Software/System Bring Up Specification

4.1, 6/2017

RapidIO.org

Revision History

Revision	Description	Date
1.0	First release	12/17/2003
1.3	Technical changes: the following errata showings: 04-09-00020.001, 04-09-00023.001 Converted to ISO-friendly templates Revision bumped to align with the rest of the specification stack	02/23/2005
2.0	Technical changes: errata showing 06-02-00001.005	06/14/2007
2.1	Technical changes: Errata showing 08-06-00000.000	07/09/2009
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

List of Figures

List of Figures

Blank page

Chapter 1 Overview

1.1 Introduction

This chapter provides an overview of the *RapidIO Annex 1: Software/System Bring Up Specification* document. This document assumes that the reader is familiar with the RapidIO specifications, conventions, and terminology.

1.2 Overview

The RapidIO Architectural specifications establish a framework that enables a wide variety of implementations. The *RapidIO Part 7: System and Device Inter-operability Specification* provides a standard set of device and system design solutions to support inter-operability. This document builds upon the inter-operability specification to define a standard set of software API functions for use in system bring up.

Each chapter addresses a different bring up topic. This revision of the *RapidIO Annex 1: Software/System Bring Up Specification* document covers the following issues:

Chapter 2, “Requirements for System Bring Up”

Chapter 3, “Hardware Abstraction Layer”

Chapter 4, “Standard Bring Up Functions”

Chapter 5, “Routing-Table Manipulation Functions”

Chapter 6, “Device Access Routine Interface”

Annex A, “System Bring Up Guidelines (Informative)”

1.3 Scope

Although RapidIO networks provide many features and capabilities, there are a few assumptions and restrictions that this specification relies on to simplify the bring up process and narrow the specification scope. These assumptions and restrictions are:

- Only two hosts may simultaneously enumerate a network. Two hosts may be needed on a network for fault tolerance purposes. System integrators must determine which hosts can perform this function.

- Only one host actually completes the network enumeration (this is referred to as the *winning host*). The second host must retreat and wait for the enumeration to complete or, assuming the winning host has failed, for enumeration to timeout. If a timeout occurs, the second host re-enumerates the network.
- After enumeration, other hosts in the system must passively discover the network to gather topology information such as routing tables and memory maps.

1.4 System Enumeration API

System enumeration API functions may be divided into two categories:

- Standard RapidIO functions that use hardware resources defined by the RapidIO specifications. These functions should rely on the support functions provided by the Hardware Abstraction Layer (HAL) to ensure portability between different platforms.
- Device-specific (vendor-specific) functions defined by a device manufacturer that use hardware resources outside of the scope of the RapidIO specifications. The main purpose of these functions is to provide Hardware Abstraction Layer (HAL) support to the standard RapidIO functions.

An important goal of this software API specification is to minimize the number of device-specific functions required for enumeration so that the portability of the API across hardware platforms is maximized.

1.5 Terminology

This document uses terms such as *local port, local configuration registers,* etc. to refer to hardware resources associated with a RapidIO end point device attached to (or combined with) the host processor that performs RapidIO system enumeration and initialization.

1.6 Software Conventions

To describe the software API functions, this document uses syntactic and notational conventions consistent with the C programming language. The conventions for naming functions and variables used by these APIs are outside of scope of this document.

Chapter 2 Requirements for System Bring Up

2.1 Introduction

This section describes basic requirements for system bring up and discovery. An overview of the system bring up process, including a system bring up example, is presented in Annex A, “System Bring Up Guidelines (Informative)”.

2.2 Boot Requirements

The following system state is required for proper system bring up:

After the system is powered on, the state necessary for system enumeration to occur using multiple host processors is automatically initialized as follows (These initial state requirements are specified in the *RapidIO Part 7: System and Device Inter-operability Specification*):

- System devices are initialized with the following Base Device IDs:
  - — Non-boot-code and non-host device IDs are set to 0xFF (0xFFFF for 16-bit deviceID systems).
  - — Boot code device IDs are set to 0xFE (0x00FE for 16-bit deviceID systems).
  - — Host device IDs are set to 0x00 (0x0000 for 16-bit deviceID systems).
- Physical layer link initialization of end points is complete.
- The default routing state of all switches between the boot code device and the host device is set to route all requests for device ID 0xFE (0x00FE for 16-bit deviceID systems) to the appropriate boot code device. All response packets are routed back to the host from the boot code device.
- Any host that participates in discovery must change its destination ID to a unique ID value before starting the system initialization process. This value is used by a device’s Host Base Device ID Lock CSR to ensure only one host can manipulate a device at a time. The allowed ID values for a discovering host are 0x00 (0x0000) and 0x01 (0x0001). A host with an ID of 0x00 (0x0000) has a lower priority than a host with an ID of 0x01 (0x0001). Host devices must be configured to accept maintenance packets with a destination ID of 0xFF (0xFFFF for 16-bit deviceID systems) as well as the unique host ID.

- All host devices have their Master Enable bit (Port General Control CSR) set to 1. Switch devices do not have a Master enable bit.
- All devices will accept requests with any sourceID or destinationID value

2.3 Enumeration Completion

One or two hosts can perform system enumeration in a RapidIO network. If two hosts are present, an algorithm is needed to determine which host has the priority to proceed with enumeration. The host with the higher priority is the *winning host* and the other host is the *losing host*. The enumeration algorithm suggested in Appendix A, "System Bring Up Guidelines (Informative)," on page 49 sets priority based on the value of the power-on device ID.

Enumeration is complete when the winning host releases the lock on the losing host. It is the losing host's responsibility to detect that it has been locked by the winning host and to later detect that the lock has been released by the winning host. The methods used to release locks on nodes other than the host nodes is outside the scope of this document.

2.4 Enumeration Time-Out

As mentioned in the previous section, two hosts can be used to enumerate the RapidIO network. The algorithm in Appendix A assumes the host with the higher power-on host device ID has priority over the other host. Because of this pre-defined priority, only one host (the one with higher priority) can win the enumeration task. In this case, the losing host enters a wait state.

If the winning host fails to enumerate the entire network, the losing host's wait state times out. When this occurs, the losing host attempts to enumerate the network. In an open 8-bit deviceID system, the losing host must wait 15 seconds before timing out and restarting the enumeration task. The length of the time-out period in a closed or a 16-bit deviceID system may differ from that of an open system.

To develop the 15 second time-out value, the following assumptions are made about the network maximal size:

NUMDEV = 256 devices

NUMSWITCHES = 256 switches

NUMFTE = 256 routing table entries per switch

It is assumed that a separate maintenance write packet is required to program each routing table entry for each switch. Since we need to establish a time base for operations, we assume:

CWTime = 100 microseconds per configuration write packet

Now we can estimate that the number of configuration writes it takes to program all of the switch routing table entries is (256 switches)*(256 routing table entries), or;

=> 256*256*CWTIME microsecs =

=> ~6.6 seconds.

Given these rough approximations, a 15 second time-out value is seen as appropriate and conservative for open systems. The chosen value must be such that if a time-out were to occur, it must be guaranteed that failure HAS occurred, and hence choosing a conservative value is necessary.

2.5 Function Return Codes

The following return codes and their constant values are defined for use by the system bring up functions.

```
typedef     unsigned int            STATUS;

#define     RIO_SUCCESS                             0x0     // Success status code
#define     RIO_WARN_INCONSISTENT                   0x1     // Used by
                                                            // rioRouteGetEntry—indicates
                                                            // that the routeportno returned is
                                                            // not the same for all ports
#define     RIO_ERR_SLAVE                           0x1001  // Another host has a higher
                                                            // priority
#define     RIO_ERR_INVALID_PARAMETER               0x1002  // One or more input parameters
                                                            // had an invalid value
#define     RIO_ERR_RIO                             0x1003  // The RapidIO fabric returned a
                                                            // Response Packet with ERROR
                                                            // status reported
#define     RIO_ERR_ACCESS                          0x1004  // A device-specific hardware
                                                            // interface was unable to generate
                                                            // a maintenance transaction and
                                                            // reported an error
#define     RIO_ERR_LOCK                            0x1005  // Another host already acquired
                                                            // the specified processor element
#define     RIO_ERR_NO_DEVICE_SUPPORT               0x1006  // Device Access Routine does not
                                                            // provide services for this device
#define     RIO_ERR_INSUFFICIENT_RESOURCES          0x1007  // Insufficient storage available in
                                                            // Device Access Routine private
                                                            // storage area
#define     RIO_ERR_ROUTE_ERROR                     0x1008  // Switch cannot support
                                                            // requested routing
#define     RIO_ERR_NO_SWITCH                       0x1009  // Target device is not a switch
#define     RIO_ERR_FEATURE_NOT_SUPPORTED           0x100A  // Target device is not capable of
                                                            // per-input-port routing
```

Blank page

Chapter 3 Hardware Abstraction Layer

3.1 Introduction

The Hardware Abstraction Layer (HAL) provides a standard software interface to the device-specific hardware resources needed to support RapidIO system configuration transactions. Configuration read and write operations are used by the HAL functions to access RapidIO device registers. The HAL functions are accessed by the RapidIO enumeration API during system bring up.

This section describes the HAL functions and how they can be used to access local and remote RapidIO device registers. These functions must be implemented by every new device-specific host-processing element to support RapidIO system enumeration and initialization. The HAL functions assume the following:

- All configuration read and write operations support only single word (4-byte) accesses.
- As required by the device, the size of the 8-bit or 16-bit deviceID field is considered by the device implementation (see section 2.4 of the *RapidIO Part 3: Common Transport Specification* for more information).
- An enumerating processor device may have more than one RapidIO end point (local port).

3.2 Device Addressing

One purpose of the HAL is to provide a unified software interface to configuration registers in both local and remote RapidIO processing elements. This is done using a universal device-addressing scheme. Such a scheme enables HAL functions to distinguish between accesses to local and remote RapidIO end points without requiring an additional parameter. The result is that only one set of HAL functions must be implemented to support local and remote configuration operations.

All HAL functions use the **destid** and **hopcount** parameters to address a RapidIO device. The HAL reserves **destid**=0xFFFFFFFF and hopcount of 0 for addressing configuration registers within the local RapidIO end point. A **destid**= 0xFFFFFFFF and hopcount of 0 value *must* be used to address the local processing end point regardless of the actual destination ID value. This reserved combination does not conflict with the address of other RapidIO devices. The **localport** parameter is used by the HAL functions to identify a specific local port within RapidIO devices containing multiple ports.

3.3 HAL Functions

The functions that form the RapidIO initialization HAL are described in the following sections.

3.3.1 Types and Definitions

/* The HOST_REGS value below is a destination ID used to specify that the registers of the processor/platform on which the code is running are to be accessed. */

#define HOST_REGS 0xFFFFFFFF

3.3.2 rioGetNumLocalPorts

Prototype:

```
INT32 rioGetNumLocalPorts (
        void
)
```

Arguments:

None

Return Value:

0	Error
n	Number of RapidIO ports supported

Synopsis:

rioGetNumLocalPorts() returns the total number of local RapidIO ports supported by the HAL functions. The number **n** returned by this function should be equal to or greater than 1. A returned value of 0 indicates an error.

3.3.3 rioConfigurationRead

Prototype:

```
STATUS rioConfigurationRead (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      offset,
        UINT32      *readdata
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the target device [IN]
hopcount	Hop count [IN]
offset	Word-aligned (four byte boundary) offset—in bytes—of the CAR or CSR [IN]
*readdata	Pointer to storage for received data [OUT]

Return Value:

RIO_SUCCESS	The read operation completed successfully and valid data was placed into the specified location.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioConfigurationRead() performs a configuration read transaction from CAR and/or CSR register(s) belonging to a local or remote RapidIO device. The function uses a device-specific hardware interface to generate maintenance transactions to remote devices. This hardware sends a configuration read request to the remote device (specified by **destid** and/or **hopcount**) and waits for a corresponding configuration read response. After the function receives a configuration read response it returns data and/or status to the caller. The method for accessing registers in a local device is device-specific.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

3.3.4 rioConfigurationWrite

Prototype:

```
STATUS rioConfigurationWrite (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          offset,
        UINT32          *writedata
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the target device [IN]
hopcount	Hop count [IN]
offset	Word-aligned (four byte boundary) offset—in bytes—of the CAR or CSR [IN]
*writedata	Pointer to storage for data to be written [IN]

Return Value:

RIO_SUCCESS	The write operation completed successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioConfigurationWrite() performs a configuration write transaction to CAR and/or CSR register(s) belonging to a local or remote RapidIO device. The function uses a device-specific hardware interface to generate maintenance transactions to remote devices. This hardware sends a configuration write request to the remote device (specified by **destid** and/or **hopcount**) and waits for a corresponding configuration write response. After the function receives a configuration write response it returns status to the caller. The method for accessing registers in a local device is device-specific.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Chapter 4 Standard Bring Up Functions

4.1 Introduction

This section describes the RapidIO functions that must be implemented to support system bring up. Functions are defined only for device registers used during the RapidIO enumeration and initialization process, not for all possible RapidIO device registers. These functions can be implemented using the HAL functions. Many of the functions can also be implemented as macros that specify predefined parameters for the HAL functions. The standard RapidIO bring up functions can be combined into a library if they are implemented as a set of subroutines.

4.2 Data Structures

```
typedef ADDR_MODE UINT32;

    #define ADDR_MODE_34BIT_SUPPORT 0x1

    #define ADDR_MODE_50_34BIT_SUPPORT 0x3

    #define ADDR_MODE_66_34BIT_SUPPORT 0x5

    #define ADDR_MODE_66_50_34BIT_SUPPORT 0x7
```

4.3 Bring Up Functions

4.3.1 rioInitLib

Prototype:

```
STATUS rioInitLib (
        void
)
```

Arguments:

None

Return Value:

RIO_SUCCESS	Initialization completed successfully.
RIO_ERROR	Generic error report. Unable to initialize library.

Synopsis:

rioInitLib() initializes the RapidIO API library. No routines defined in this chapter may be called unless and until rioInitLib has been invoked. If rioInitLib returns RIO_ERROR, no routines defined in this chapter may be called.

4.3.2 rioGetFeatures

Prototype:

```
STATUS rioGetFeatures (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *features
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*features	Pointer to storage containing the received features [OUT]

Return Value:

RIO_SUCCESS	The features were retrieved successfully and placed into the location specified by *features.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetFeatures() uses the HAL *rioConfigurationRead()* function to read from the Processing Element Features CAR of the specified processing element. Values read are placed into the location referenced by the ***features** pointer. Reported status is similar to *rioConfigurationRead()*

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.3 rioGetSwitchPortInfo

Prototype:

```
STATUS rioGetSwitchPortInfo (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      *portinfo
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*portinfo	Pointer to storage containing the received port information [OUT]

Return Value:

RIO_SUCCESS	The port information was retrieved successfully and placed into the location specified by *portinfo.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetSwitchPortInfo() uses the HAL *rioConfigurationRead()* function to read from the Switch Port Information CAR of the specified processing element. Values read are placed into the location referenced by the ***portinfo** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.4 rioGetExtFeaturesPtr

Prototype:

```
STATUS rioGetExtFeaturesPtr (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      *extfptr
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*extfptr	Pointer to storage containing the received extended feature information [OUT]

Return Value:

RIO_SUCCESS	The extended feature information was retrieved successfully and placed into the location specified by *extfptr.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetExtFeaturesPtr() uses the HAL *rioConfigurationRead()* function to read the pointer to the first entry in the extended features list from the Assembly Information CAR of the specified processing element. That pointer is placed into the location referenced by the ***extfptr** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Note that if the EF_PTR field of *extfptr is 0, no extended features are available.

4.3.5 rioGetNextExtFeaturesPtr

Prototype:

```
STATUS rioGetNextExtFeaturesPtr (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      currfptr,
        UINT32      *extfptr
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
currfptr	Pointer to the last reported extended feature [IN]
*extfptr	Pointer to storage containing the received extended feature information [OUT]

Return Value:

RIO_SUCCESS	The extended feature information was retrieved successfully and placed into the location specified by *extfptr.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetNextExtFeaturesPtr() uses the HAL *rioConfigurationRead()* function to read the pointer to the next entry in the extended features. That pointer is placed into the location referenced by the ***extfptr** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Note that if the EF_PTR field of *extfptr is 0, no further extended features are available. Invoking rioGetNextExtFeaturesPtr when currfptr has an EF_PTR field value of 0 will result in a return code of RIO_ERR_INVALID_PARAMETER.

4.3.6 rioGetSourceOps

Prototype:

```
STATUS rioGetSourceOps (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *srcops
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*srcops	Pointer to storage containing the received source operation information [OUT]

Return Value:

RIO_SUCCESS	The source operation information was retrieved successfully and placed into the location specified by *srcops.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetSourceOps() uses the HAL *rioConfigurationRead()* function to read from the Source Operations CAR of the specified processing element. Values read are placed into the location referenced by the ***srcops** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.7 rioGetDestOps

Prototype:

```
STATUS rioGetDestOps (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *dstops
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*dstops	Pointer to storage containing the received destination operation information [OUT]

Return Value:

RIO_SUCCESS	The destination operation information was retrieved successfully and placed into the location specified by *dstops.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetDestOps() uses the HAL *rioConfigurationRead()* function to read from the Destination Operations CAR of the specified processing element. Values read are placed into the location referenced by the ***dstops** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.8 rioGetAddressMode

Prototype:

```
STATUS rioGetAddressMode (
        UINT8                   localport,
        UINT32                  destid,
        UINT8                   hopcount,
        ADDR_MODE               *amode
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*amode	Pointer to storage containing the received address mode (34-bit, 50-bit, or 66-bit address) information [OUT]

Return Value:

RIO_SUCCESS	The address mode information was retrieved successfully and placed into the location specified by *amode.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetAddressMode() uses the HAL *rioConfigurationRead()* function to read from the PE Logical Layer CSR of the specified processing element. The number of address bits generated by the PE (as the source of an operation) and processed by the PE (as the target of an operation) are placed into the location referenced by the ***amode** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.9 rioGetBaseDeviceId

Prototype:

```
STATUS rioGetBaseDeviceId (
        UINT8           localport,
        UINT32          *deviceid
)
```

Arguments:

localport	Local port number [IN]
*deviceid	Pointer to storage containing the base device ID [OUT]

Return Value:

RIO_SUCCESS	The base device ID information was retrieved successfully and placed into the location specified by *deviceid.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetBaseDeviceId() uses the HAL *rioConfigurationRead()* function to read from the Base Device ID CSR of the local processing element (the **destid** and **hopcount** parameters used by *rioConfigurationRead()* must be set to HOST_REGS and zero, respectively). Values read are placed into the location referenced by the ***deviceid** pointer. Reported status is similar to *rioConfigurationRead()*. This function is useful only for local end-point devices.

4.3.10 rioSetBaseDeviceId

Prototype:

```
STATUS rioSetBaseDeviceId (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT32      newdeviceid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
newdeviceid	New base device ID to be set [IN]

Return Value:

RIO_SUCCESS	The base device ID was updated successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioSetBaseDeviceId() uses the HAL *rioConfigurationWrite()* function to write the base device ID in the Base Device ID CSR of the specified processing element (end point devices only). Reported status is similar to *rioConfigurationWrite()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.11 rioAcquireDeviceLock

Prototype:

```
STATUS rioAcquireDeviceLock (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT16          hostdeviceid,
        UINT16          *hostlockid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
hostdeviceid	Host base device ID for the local processing element [IN]
*hostlockid	Device ID of the host holding the lock if ERR_LOCK is returned [OUT]

Return Value:

RIO_SUCCESS	The device lock was acquired successfully.
RIO_ERR_LOCK	Another host already acquired the specified processor element. ID of the device holding the lock is contained in the location referenced by the *hostlockid parameter.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioAcquireDeviceLock() tries to acquire the hardware device lock for the specified processing element on behalf of the requesting host. The function uses the HAL *rioConfigurationWrite()* function to write the requesting host device ID into the Host Base Lock Device ID CSR of the specified processing element. After the write completes, this function uses the HAL *rioConfigurationRead()* function to read the value back from the Host Base Lock Device ID CSR. The written and read values are compared. If they are equal, the lock was acquired successfully. Otherwise, another host acquired this lock and the device ID for that host is reported.

This function assumes unique host-based device identifiers are assigned to discovering hosts. For more details, refer to Annex A, "System Bring Up Guidelines (Informative)".

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.12 rioReleaseDeviceLock

Prototype:

```
STATUS rioReleaseDeviceLock (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT16      hostdeviceid,
        UINT16      *hostlockid
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
hostdeviceid	Host base device ID for the local processing element [IN]
*hostlockid	Device ID of the host holding the lock if ERR_LOCK is returned [OUT]

Return Value:

RIO_SUCCESS	The device lock was released successfully.
RIO_ERR_LOCK	Another host already acquired the specified processor element.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioReleaseDeviceLock() tries to release the hardware device lock for the specified processing element on behalf of the requesting host. The function uses the HAL *rioConfigurationWrite()* function to write the requesting host device ID into the Host Base Lock Device ID CSR of the specified processing element. After the write completes, this function uses the HAL *rioConfigurationRead()* function to read the value back from the Host Base Lock Device ID CSR. If the Device ID that is read back from the Host Base Device ID register is 0xFFFF then the lock has been released successfully.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.13 rioGetComponentTag

Prototype:

```
STATUS rioGetComponentTag (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          *componenttag
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
*componenttag	Pointer to storage containing the received component tag information [OUT]

Return Value:

RIO_SUCCESS	The component tag information was retrieved successfully and placed into the location specified by *componenttag.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetComponentTag() uses the HAL *rioConfigurationRead()* function to read from the Component Tag CSR of the specified processing element. Values read are placed into the location referenced by the ***componenttag** pointer. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.14 rioSetComponentTag

Prototype:

```
STATUS rioSetComponentTag (
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT32          componenttag
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
componenttag	Component tag value to be set [IN]

Return Value:

RIO_SUCCESS	The component tag was updated successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioSetComponentTag() uses the HAL *rioConfigurationWrite()* function to write the component tag into the Component Tag CSR of the specified processing element. Reported status is similar to *rioConfigurationWrite()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

4.3.15 rioGetPortErrStatus

Prototype:

```
STATUS rioGetPortErrStatus (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT16      extfoffset,
        UINT8       portnum,
        UINT32      *porterrorstatus
)
```

Arguments:

localport	Local port number [IN]
destid	Destination ID of the processing element [IN]
hopcount	Hop count [IN]
extfoffset	Offset from the previously reported extended features pointer [IN]
portnum	Port number to be accessed [IN]
*porterrorstatus	Pointer to storage for the returned value [OUT]

Return Value:

RIO_SUCCESS	The read completed successfully and valid data was placed into the location specified by *porterrorstatus.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.

Synopsis:

rioGetPortErrStatus() uses the HAL *rioConfigurationRead()* function to read the contents of the Port *n* Error and Status CSR of the specified processing element. Reported status is similar to *rioConfigurationRead()*.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Chapter 5 Routing-Table Manipulation Functions

5.1 Introduction

This section describes the RapidIO functions that must be provided to support routing tables used within the switch fabric. The RapidIO common transport specification requires implementing device-identifier-based packet routing. The detailed implementation of routing tables is beyond the scope of this specification.

The routing-table manipulation functions assume the following:

- The destination ID of the device that receives a packet routed by the switch is the *route destination ID*.
- The specific port at the route destination ID that receives a packet routed by the switch is the *route port number*.
- The software paradigm used for routing tables is a linear routing table indexed by the route destination ID.
- Switches may implement a global routing table, “per port” routing tables, or a combination of both.

5.2 Routing Table Functions

The functions defined for RapidIO routing-table manipulation are described in the following sections.

5.2.1 rioRouteAddEntry

Prototype:

```
STATUS rioRouteAddEntry (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT8       tableidx,
        UINT16      routedestid,
        UINT8       routeportno
)
```

Arguments:

localport	Local port number (RapidIO switch) [IN]
destid	Destination ID of the processing element (RapidIO switch) [IN]
hopcount	Hop count [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
routeportno	Route port number—value written to the selected routing table entry [IN]

Return Value:

RIO_SUCCESS	The routing table entry was added successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.
RIO_WARN_INCONSISTENT	Used by rioRouteGetEntry—indicates that the routeportno returned is not the same for all ports.

Synopsis:

rioRouteAddEntry() adds an entry to a routing table for the RapidIO switch specified by the **destid** and **hopcount** parameters. The **tableidx** parameter is used to select a specific routing table in the case of implementations with "per port" routing tables. A value of **tableidx**=0xFFFFFFFF specifies a global routing table for the RapidIO switch. The **routeportno** parameter is written to the routing table entry selected by the **routedestid** parameter.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

5.2.2 rioRouteGetEntry

Prototype:

```
STATUS rioRouteGetEntry (
        UINT8       localport,
        UINT32      destid,
        UINT8       hopcount,
        UINT8       tableidx,
        UINT16      routedestid,
        UINT8       *routeportno
)
```

Arguments:

localport	Local port number (RapidIO switch) [IN]
destid	Destination ID of the processing element (RapidIO switch) [IN]
hopcount	Hop count [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
*routeportno	Route port number—pointer to value read from the selected routing table entry [OUT]

Return Value:

RIO_SUCCESS	The routing table entry was added successfully.
RIO_ERR_INVALID_PARAMETER	One or more input parameters had an invalid value.
RIO_ERR_RIO	The RapidIO fabric returned a Response Packet with ERROR status reported. Error status returned by this function may contain additional information from the Response Packet.
RIO_ERR_ACCESS	A device-specific hardware interface was unable to generate a maintenance transaction and reported an error.
RIO_WARN_INCONSISTENT	Used by rioRouteGetEntry—indicates that the routeportno returned is not the same for all ports.

Synopsis:

rioRouteGetEntry() reads an entry from a routing table for the RapidIO switch specified by the **destid** and **hopcount** parameters. The **tableidx** parameter is used to select a specific routing table in the case of implementations with "per port" routing tables. A value of **tableidx**=0xFF specifies a global routing table for the RapidIO switch. The value in the routing table entry selected by the **routedestid** parameter is read from the table and placed into the location referenced by the ***routeportno** pointer.

Reads from the global routing table may be undefined in the case where per-port routing tables exist.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Blank page

Chapter 6 Device Access Routine Interface

6.1 Introduction

This section defines the device access routine (DAR) interface that must be provided for RapidIO device configuration. The client for this interface is the boot loader responsible for RapidIO network enumeration and initialization. By using a standard DAR interface, the firmware does not need to include knowledge of device-specific configuration operations. Thus, enumeration and initialization firmware can operate transparently with devices from many component vendors.

6.2 DAR Packaging

For each processor type supported by a DAR provider, linkable object files for DARs shall be supplied using ELF format. Device-specific configuration DARs shall be supplied using C-language source code format.

6.3 Execution Environment

The functions provided by device-specific configuration DARs must be able to link and execute within a minimal execution context (e.g., a system-boot monitor or firmware). In general, configuration DARs should not call an external function that is not implemented by the DAR, unless the external function is passed to the configuration DAR by the initialization function. Also, configuration DAR functions may not call standard C-language I/O functions (e.g., *printf*) or standard C-language library functions that might manipulate the execution environment (e.g., *malloc* or *exit*).

6.4 Type Definitions

The following type definitions are to be used by the DAR functions in Section 6.5.

```
typedef struct RDCDAR_PLAT_OPS_STRUCT {
        UINT32 specversion;
        UINT32 (*rioConfigurationRead)          (       UINT8           localport,
                                                        UINT16          destid,
                                                        UINT8           hopcount,
                                                        UINT32          offset,
                                                        UINT32          *readdata);
        UINT32 (*rioConfigurationWrite)         (       UINT8           localport,
                                                        UINT16          destid,
                                                        UINT8           hopcount,
                                                        UINT32          offset,
                                                        UINT32          *writedata);
} RDCDAR_PLAT_OPS;

typedef struct RDCDAR_OPS_STRUCT {
        UINT32 specversion;
        UINT32 (*rioDarInitialize)                      (...);
        UINT32 (*rioDarTerminate)                       (...);
        UINT32 (*rioDarTestMatch)                       (...);
        UINT32 (*rioDarRegister)                        (...);
        UINT32 (*rioDarGetSwitchInfo)                   (...);
        UINT32 (*rioDarSetPortRoute)                    (...);
        UINT32 (*rioDarGetPortRoute)                    (...);
        UINT32 (*rioDarGetMemorySize)                   (...);
} RDCDAR_OPS

typedef struct RDCDAR_DATA_STRUCT {
        UINT32 databytesallocated;
        CHAR   *data;
} RDCDAR_DATA

typedef struct RDCDAR_SWITCH_INFO_STRUCT {
        BOOL   useslutmodel;
        BOOL   separatelutperinputport;
        UINT32 maxlutentries;
} RDCDAR_SWITCH_INFO
```

6.5 DAR Functions

The functions that must be provided for a RapidIO device-specific configuration DAR are described in the following sections. For the *rioDar_name*GetFunctionTable functions, the *rioDar_name* portion of the function name shall be replaced by an appropriate name for the implemented driver.

6.5.1 *rioDar_name*GetFunctionTable

Prototype:

```
UINT32 rioDar_nameGetFunctionTable(
        UINT32                  specversion,
        RDCDAR_OPS_STRUCT       *darops,
        UINT32                  maxdevices,
        UINT32                  *darspecificdatabytes
)
```

Arguments:

specversion	Version number of the DAR interface specification indicating the caller's implementation of the type definition structures [IN]
*darops	Pointer to a structure of DAR functions that are allocated by the caller and filled in by the called function (see Section 6.4) [OUT]
maxdevices	Maximum expected number of RapidIO devices that must be serviced by this configuration DAR [IN]
*darspecificdatabytes	Number of bytes needed by the DAR for the DAR private data storage area [OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDar_name*GetFunctionTable()* is called by a client to obtain the list of functions implemented by a RapidIO device-specific configuration DAR module. It shall be called once before enumerating the RapidIO network.

The **specversion** parameter is the version number defined by the revision level of the specification from which the DAR type definition structures are taken (see Section 6.4).

The **maxdevices** parameter is an estimate of the maximum number of RapidIO devices in the network that this DAR must service. The DAR uses this estimate to determine the size required for the DAR private data storage area. The storage size is returned to the location referenced by the ***darspecificdatabytes** pointer. After the client calls this function, the client shall allocate a DAR private data storage area of a size no less than that indicated by ***darspecificdatabytes**. The client shall provide that private data storage area to *rioDarInitialize()*.

6.5.2 rioDarInitialize

Prototype:

```
UINT32 rioDarInitialize (
        UINT32                  specversion,
        UINT32                  maxdevices,
        RDCDAR_PLAT_OPS         *platops,
        RDCDAR_DATA             *privdata
)
```

Arguments:

specversion	Version number of the DAR interface specification indicating the caller's implementation of the type definition structures [IN]
maxdevices	Maximum expected number of RapidIO devices that must be serviced by this configuration DAR [IN]
*platops	Pointer to a structure of platform functions for use by the DAR (see Section 6.4) [IN]
*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDarInitialize() is called by a client to initialize a RapidIO device-specific configuration DAR module. This function shall be called once after calling the rioDar_name*GetFunctionTable()* functions and before enumerating the RapidIO network.

The **specversion** parameter is the version number defined by the revision level of the specification from which the DAR type definition structures are taken (see Section 6.4).

The **maxdevices** parameter is an estimate of the maximum number of RapidIO devices in the network that this DAR must service. The **maxdevices** value must be equal to the value used in the corresponding rioDar_name*GetFunctionTable()* function call. The client is responsible for allocating the structure referenced by ***privdata**. The client is also responsible for allocating a DAR private data storage area at least as large as that specified by the rioDar_name*GetFunctionTable()* call. The client must initialize the structure referenced by ***privdata** with the number of bytes allocated to the DAR private data storage area and with the pointer to the storage area. After calling *rioDarInitialize()*, the client may not deallocate the DAR private data storage area until after the *rioDarTerminate()* function has been called.

6.5.3 rioDarTerminate

Prototype:

```
UINT32 rioDarTerminate (
        RDCDAR_DATA             *privdata
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]

Return value:

RIO_SUCCESS	On successful completion

Synopsis:

rioDarTerminate() is invoked by a client to terminate a RapidIO device-specific configuration DAR module. This function shall be called once after all use of the DAR services is completed. After calling this function, the client may deallocate the DAR private data storage area in the structure referenced by ***privdata**.

6.5.4 rioDarTestMatch

Prototype:

```
UINT32 rioDarTestMatch (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]

Return value:

RIO_SUCCESS	Device DAR does provide services for this device
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR does not provide services for this device.

Synopsis:

rioDarTestMatch() is invoked by a client to determine whether or not a RapidIO device-specific configuration DAR module provides services for the device specified by **destid**. The DAR interrogates the device (using the platform functions supplied during DAR initialization), examines the device identity and any necessary device registers, and determines whether or not the device is handled by the DAR.

The DAR does not assume that a positive match (return value of 0) means the DAR will actually provide services for the device. The client must explicitly register the device with *rioDARregister()* if the client will be requesting services.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.5 rioDarRegister

Prototype:

```
UINT32 rioDarRegister (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]

Return value:

RIO_SUCCESS	Device DAR successfully registered this device.
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR does not provide services for this device.
RIO_ERR_INSUFFICIENT_RESOURCES	Insufficient storage available in DAR private storage area

Synopsis:

rioDarRegister() is invoked by a client to register a target device with a RapidIO device-specific configuration DAR. The client must call this function once for each device serviced by the DAR. The client should first use the *rioDarTestMatch()* function to verify that the DAR is capable of providing services to the device.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.6 rioDarGetMemorySize

Prototype:

```
UINT32 rioDarGetMemorySize (
        RDCDAR_DATA         *privdata,
        UINT8               localport,
        UINT32              destid,
        UINT8               hopcount,
        UINT32              regionix,
        UINT32              *nregions,
        UINT32              *regbytes[2],
        UINT32              *startoffset[2]
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number used to access the network [IN]
destid	Destination device ID for the target device [IN]
hopcount	Number of switch hops needed to reach the target device [IN]
regionix	Index of the memory region being queried (0, 1, 2, 3, ...) [IN]
*nregions	Number of memory regions provided by the target device [OUT]
*regbytes	Size (in bytes) of the queried memory region [OUT]
*startoffset	Starting address offset for the queried memory region [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully returned memory size information for the target device.
RIO_ERR_NO_DEVICE_SUPPORT	Device DAR could not determine memory size information for the target device.

Synopsis:

rioDarGetMemorySize() is invoked by a client to determine the number of, the sizes of, and the offsets for the memory regions supported by a RapidIO target device. The function is intended to support the mapping of PCI or other address windows to RapidIO devices. If the **regionix** parameter is greater than the number of regions provided by the device (***nregions**), the DAR should return a value of zero for the ***regbytes** and ***startoffset** parameters, and indicate a "successful" (0) return code.

rioDarGetMemorySize always returns at least one region. The first index, index 0, always refers to the region controlled by the Local Configuration Space Base Address Registers.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.7 rioDarGetSwitchInfo

Prototype:

```
UINT32 rioDarGetSwitchInfo (
        RDCDAR_DATA             *privdata,
        UINT8                   localport,
        UINT32                  destid,
        UINT8                   hopcount,
        RDCDAR_SWITCH_INFO      *info
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
*info	Pointer to switch information data structure (see Section 6.4) [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully retrieved the information for RDCDAR_PLAT_OPS_STRUCT.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_NO_SWITCH	Target device is not a switch.

Synopsis:

rioDarGetSwitchInfo() is invoked by a client to retrieve the data necessary to initialize the RDCDAR_SWITCH_INFO structure.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.8 rioDarSetPortRoute

Prototype:

```
UINT32 rioDarSetPortRoute (
        RDCDAR_DATA         *privdata,
        UINT8               localport,
        UINT32              destid,
        UINT8               hopcount,
        UINT8               tableidx,
        UINT16              routedestid,
        UINT8               routeportno
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
inport	Target switch device input port [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
routeportno	Route port number—value written to the selected routing table entry [IN]

Return value:

RIO_SUCCESS	Device DAR successfully modified the packet routing configuration for the target switch device.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_ROUTE_ERROR	Switch cannot support requested routing.
RIO_ERR_NO_SWITCH	Target device is not a switch.
RIO_ERR_FEATURE_NOT_SUPPORTED	Target device is not capable of per-input-port routing.

Synopsis:

rioDarSetPortRoute() is invoked by a client to modify the packet routing configuration for a RapidIO target switch device.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

6.5.9 rioDarGetPortRoute

Prototype:

```
UINT32 rioDarGetPortRoute (
        RDCDAR_DATA     *privdata,
        UINT8           localport,
        UINT32          destid,
        UINT8           hopcount,
        UINT8           tableidx,
        UINT16          routedestid,
        UINT8           *routeportno
)
```

Arguments:

*privdata	Pointer to structure containing DAR private data area (see Section 6.4) [IN/OUT]
localport	Local port number to be used to access network [IN]
destid	Destination device ID to reach target switch device [IN]
hopcount	Number of switch hops to reach target switch device [IN]
tableidx	Routing table index for per-port switch implementations [IN]
routedestid	Route destination ID—used to select an entry into the specified routing table [IN]
*routeportno	Route port number—pointer to value read from the selected routing table entry [OUT]

Return value:

RIO_SUCCESS	Device DAR successfully modified the packet routing configuration for the target switch device.
RIO_ERR_NO_DEVICE_SUPPORT	Insufficient switch routing resources available.
RIO_ERR_ROUTE_ERROR	Switch cannot support requested routing.
RIO_ERR_NO_SWITCH	Target device is not a switch.

Synopsis:

rioDarGetPortRoute() is invoked by a client to read the packet routing configuration for a RapidIO target switch device.

The client must register the target device with the RapidIO device-specific configuration DAR before calling this function.

A destid value of HOST_REGS and hopcount of 0 results in accesses to the local hosts RapidIO registers.

Blank page

Annex A System Bring Up Guidelines (Informative)

A.1 Introduction

The *RapidIO Annex 1: Software/System Bring Up Specification* defines a standard set of software API functions for use in system enumeration and initialization. These API functions enable up to two RapidIO hosts to cooperatively enumerate and configure a RapidIO network.

This appendix is provided as a reference model for the system bring up process. An algorithm is presented that enables up to two cooperating host processors in a Rapid IO system to enumerate the entire network, set up a route to every system node, and enable the booting software to start the next boot-process phase. The actual implementation of the algorithm used to bring up a RapidIO network can vary greatly from this model in both capability and complexity.

A.2 Overview of the System Bring Up Process

This section presents a high-level overview of the system bring up process.

1. The system is powered on. Refer to Chapter 2, “Requirements for System Bring Up” for the system power-on requirements.
2. The host processor fetches the initial boot code (if necessary). If two processors are present, both can fetch the initial boot code.
3. The system exploration and enumeration algorithm is started. The algorithm for this process is outlined in Section A.3 on page 50.
4. All devices have been enumerated and stored in the device database, and routes have been set up between the host device and all end point devices. The enumeration process may optionally choose to do the following:

    a) Compute and configure optimal routes between the host device and end point devices, and between different end point devices.

    b) Configure the switch devices with the optimal route information.

    c) Store the optimal route and alternate route information in the device database.
5. The address space is mapped.

The host may access the network across a host-RapidIO bridge or host-PCI bridge. The address-space mapping across this bridge must be done when devices are enumerated and stored in the device database. This allows the address of a found device to be retrieved later and presented to the device access routines during operating system (OS) initialization. The pseudocode for this process is as follows:

```
1   ACQUIRE the host bridge address-space requirement
2   MAP the address space into a host address partition X
3   FOR every device in the database
4       IF the component is a RapidIO device
5           ACQUIRE the device's address-space requirement
6           MAP the address space into a new host address partition
7           EXPAND the partition X window to cover the new partition
8           UPDATE the device database with the new host address
9       ELSE IF the component is a PCI bridge
10          ACQUIRE the bridge's PCI bus ID
11          ACQUIRE the bridge's address-space requirement
12          // All devices that appear behind this PCI bridge must have their address spaces
            // mapped within the region specified for this bridge.
13          MAP the address space into a new host address partition
14          EXPAND the partition X window to cover the new partition
15          UPDATE the device database with the new host address
16      ENDIF
17  ENDFOR
```

After discovery has been concluded, it is expected that the majority of systems will then attempt to load in a software image from a boot device.

A.3 System Enumeration Algorithm

The system enumeration algorithm is designed for use by one or two host processors. The outline of the algorithm is as follows:

1. Access the RapidIO network. This step may involve generating special transaction cycles to ensure that the RapidIO network is accessible.
2. Discover the host and assign a device ID to it.
3. Discover the neighbor, if present.
4. If necessary, repeat the previous step recursively to discover additional devices.
5. Clear up.

When a host begins exploring, it must acquire the Host Base Device ID Lock before it can proceed. Once acquired, it can set its device ID and discover its neighbor (if necessary).

If two hosts are used, both can execute the enumeration algorithm. However, only one host (the one with higher priority) can win the enumeration task. The losing host enters a wait state. The guidelines for prioritizing hosts to enumerate the network and restarting enumeration should the winning host fail to complete the task are described in Chapter 2, “Requirements for System Bring Up,” on page 9.

The enumeration algorithm described below sets priority based on the value of the power-on device ID. The winning host is the device with the higher power-on host device ID. The losing host has the lower power-on host device ID. The losing host enters a wait state until the winning host completes enumeration or until the wait

state times out.

The prioritization mechanism never results in a deadlock if the priorities of both host processors are unique. The enumeration process is initially performed in parallel by both hosts until they meet at a device. When a meeting occurs, prioritization guarantees one winning host—the other host retreats (enters a wait state).

The enumeration algorithm described below uses a recursive, depth-first graph traversal to discover the network. It may be possible to improve the algorithm using non-recursive or breadth-first graph traversal. However, those improvements and optimizations are implementation dependent and beyond the scope of this document.

A.3.1 Data Structures, Constants, and Global Variables

This section outlines the data structures, constants, and global variables used by the system enumeration algorithm pseudocode.

The example system is composed of only 8 bit capable devices.

Data Structures

```
struct rioRouteTable {

        // The switch routing table is implemented as a linear routing table for destination IDs. The table is
        // indexed using the destination ID and the table index range is equal to the maximum destination ID
        // value. The value of a table entry indicates the output port number used to route messages for the
        // destination ID. The table entry default value is implementation dependent. Table entries must be
        // initialized to support FLASH memory accesses. The algorithm pseudocode described in this
        // document assumes the device ID is equal to the RapidIO protocols destination ID. This assignment
        // is not a general requirement.

        UINT8   LFT[MAX_DEVICEID];
}

struct rioSwitch {

        …

        UINT16 SwitchIdentity;                  // Switch Identity
        UINT16 hopCount;                        // Hop Count to reach this switch
        UINT16 DeviceID;                        // Associated Device ID in the path to this switch
        struct rioRouteTable RouteTable;        // Switch Routing Table

        …

}
```

Constants

```
        RIO_GEN_DFLT_DID 0x00FFFFFF     // RIO_GEN_DFLT_DID is the general default device
                                        // ID assigned to non-host and non-boot code end
                                        // points
        RIO_BOOT_DFLT_DID 0x0000FFFE    // RIO_BOOT_DFLT_DID is the default device ID
                                        // assigned to boot code devices
        RIO_HOST_DFLT_DID 0x00000000    // RIO_HOST_DFLT_DID is the default device ID
                                        // assigned to host devices
```

Global Variables

```
        UINT16 DeviceID = 0;            // Currently available Device ID to be assigned to the
```

```
                                        // end point device
    UINT16 SwitchID = 0;                // Currently available Switch ID. This is used
                                        // internally by the to index
                                        // switches that have been discovered.

                                        // The following global arrays are used to store device
                                        // information
                                        // collected from rioGetFeatures and
                                        // rioGetSwitchPortInfo. They are
                                        // also used to store the hopCount and DeviceID
                                        // assigned to switches.

    struct rioSwitch Switches[MAX_SWITCHES];
```

A.3.2 Pseudocode

This section outlines the detailed pseudocode for the system enumeration algorithm.

```
1   //************************************************************************
2   // System enumeration and initialization using the power-on device ID as the hostDeviceID
3   // —Discover the host first
4   // —Discover the host's neighbor recursively
5
6   STATUS rioSystemEnumerate (hostDeviceID)
7   {
8       // Discover the host first.
9       status = rioEnumerateHost (hostDeviceID);
10
11      if (status == ERR_SLAVE) {
12          rioClearUp (hostDeviceID);
13          return ERR_SLAVE;
14      }
15
16      // Discover the host neighbor
17      status = rioEnumerateNeighbor (hostDeviceID, hopCount = 1);
18
19      if (status == ERR_SLAVE) {
20          rioClearUp (hostDeviceID);
21          return ERR_SLAVE;
22      }
23
24      // If the code advances to this point successfully, the host must acquire the
25      // HostBaseDeviceIdLock for all devices in the system. When this is done, the Discovered bit
26      // Master Enable bit, etc. can be set for all devices.
27
28  }   // end rioSystemEnumerate
29
30  //************************************************************************
31  // System Delay
32  // —Wait for other host to release the lock
33
34  rioDelay () {
35  }   // end rioDelay
36
37  //************************************************************************
38  // Host enumeration and initialization
39
40  STATUS rioEnumerateHost (hostDeviceID)
```

```
41  {
42      // Try to acquire the lock
43      rioAcquireDeviceLock (0, hostDeviceID, 0, hostDeviceID);
44
45      while (HostBaseDeviceIdLockCSR.HostBaseDeviceID < hostDeviceID) {
46          // Delay for a while
47          rioDelay ();
48
49          // Retry lock acquisition
50          rioAcquireDeviceLock (0, hostDeviceID, 0, hostDeviceID, &lockingHost);
51      }
52
53      // Check to see if there is a master with a larger host device ID
54      if (HostBaseDeviceIdLock.HostBaseDeviceID > hostDeviceID) {
55          // Release the current lock
56          rioReleaseDeviceLock (0, hostDeviceID, 0, hostDeviceID);
57
58          return ERR_SLAVE;
59      }
60
61      // Lock has been acquired so enumeration can begin
62
63      // Assign the default host ID to the host
64      rioSetBaseDeviceId (0, hostDeviceID, hostDeviceID);
65
66      // Increment the available device ID
67      if (DeviceID == hostDeviceID) {
68          DeviceID ++;
69      }
70
71      return RIO_SUCCESS;
72  }  // end rioEnumerateHost
73
74  //************************************************************************
75  // Neighbor enumeration
76
77  STATUS rioEnumerateNeighbor (hostDeviceID, hopCount)
78  {
79      // The host has already discovered this node if it currently owns the lock
80      rioGetCurHostLock (0, 0, 0, &owner_device_id);
81      if (owner_device_id == hostDeviceID) {
82          return RIO_SUCCESS;
83      }
84
85      // Try to acquire the lock
86      rioAcquireDeviceLock (0, RIO_GEN_DFLT_DID, hopCount, hostDeviceID, &lockingHost);
87
88      while (HostBaseDeviceIdLockCSR.HostBaseDeviceID < hostDeviceID) {
89          // Delay for a while
90          rioDelay ();
91
92          // Retry lock acquisition
93          rioAcquireDeviceLock(0, RIO_GEN_DFLT_DID, hopCount, hostDeviceID,
                &lockingHost);
94      }
95
```

```
96      // Check to see if there is a master with a larger host device ID
97      if (HostBaseDeviceIdLock.HostBaseDeviceID > hostDeviceID) {
98          return ERR_SLAVE;
99      }
100
101     // Lock has been acquired so enumeration can begin
102
103     // Check Source Operation CAR and Destination Operation CAR to see if a Device ID can be
104     // assigned
105
106     rioGetSourceOps (0, RIO_GEN_DFLT_DID, hopCount, &SourceOperationCAR);
107     rioGetDestOps (0, RIO_GEN_DFLT_DID, hopCount, &DestinationOperationCAR);
108
109     if ( (SourceOperationCAR.Read || Write || Atomic) &&
110         (DestinationOperationCAR.Read || Write || Atomic)) {
111
112         // Set the device ID
113         rioSetBaseDeviceId (0, RIO_GEN_DFLT_DID, DeviceID);
114
115         // Increment the available device ID
116         DeviceID ++;
117         if (DeviceID == hostDeviceID) {
118             DeviceID ++;
119         }
120     }
121
122     // Check to see if the device is a switch
123     rioGetFeatures (0, RIO_GEN_DFLT_DID, hopCount, &ProcessingElementFeatureCAR);
124     if (ProcessingElementFeatureCAR.Switch == TRUE) {
125
126         // Read the switch information
127         rioGetSwitchPortInfo (0, RIO_GEN_DFLT_DID, hopCount,
                &SwitchPortInformationCAR);
128
129         // Record the switch device identity
130         Switches[SwitchID].SwitchIdentity = DeviceIdentityCAR.DeviceIdentity;
131
132         // Bookkeeping for the current switch ID
133         curSwitchID = SwitchID;
134
135         // Increment the available switch ID
136         SwitchID ++;
137
138         // Initialize the current switch routing table to add entries for all previously discovered
139         // devices so that they are routed correctly. Start with the host device ID (0x00) and end with
140         // DeviceID-1.
141         for (each deviceID in [0..DeviceID-1]) {
142             rioRouteAddEntry (0, RIO_GEN_DFLT_DID, hopCount, RIO_GEN_DFLT_DID,
                        deviceID,
143                     SwitchPortInformationCAR.PortNumber, NULL);
144         }
145
146         // Synchronize the current switch routing table with the global table
147         for (each deviceID in [0.. DeviceID-1]) {
148             Switches[curSwitchID].RouteTable.LFT[deviceID] =
149                 SwitchPortInformationCAR.PortNumber;
```

```
150         }
151
152         // Update the hopCount to reach the current switch
153         Switches[curSwitchID].HopCount = hopCount;
154
155         for (each portNum in SwitchPortInformationCAR.PortTotal) {
156             if (SwitchPortInformationCAR.PortNumber == portNum) {
157                 continue;
158             }
159
160             // Bookkeeping for the current available device ID
161             curDeviceID = DeviceID;
162
163             rioGetPortErrStatus (0, RIO_GEN_DFLT_DID, hopCount,
                    &PortErrorStatusCSR[portNum]);
164
165             // Check if it is possible to have a neighbor
166             if (PortErrorStatusCSR[portNum].PortUninitialized == TRUE) {
167                 continue;
168             }
169
170             else if (PortErrorStatusCSR[portNum].PortOK == TRUE) {
171
172                 // Check if it is an enumeration boundary port
173                 if (PortControlCSR[portNum].PortEnumerationBoundary == TRUE) {
174                     continue;
175                 }
176                 rioRouteAddEntry(0, RIO_GEN_DFLT_DID, hopCount, RIO_GEN_DFLT_DID, 0,
                        portNumber, NULL);
177
178                 // Discover the neighbor recursively
179                 if (status = rioEnumerateNeighbor(hopCount + 1) != RIO_SUCCESS) {
180                     return status;
181                 }
182
183                 // If more than one end point device was found, update the current switch routing table
184                 // entries beginning with the curDeviceID entry and ending with the DeviceID-1
185                 // entry.
186                 if (DeviceID > curDeviceID) {
187                     for (each deviceID in [curDeviceID..DeviceID-1]) {
188                         rioRouteAddEntry(0, RIO_GEN_DFLT_DID, hopCount, deviceID,
                                portNumber);
189                     }
190
191                     // Synchronize the current switch routing table with the global table
192                     for (each deviceID in [curDeviceID..DeviceID-1]) {
193                         Switches[curSwitchID].RouteTable.LFT[deviceID] = portNumber;
194                     }
195
196                     // Update the associated Device ID in the path.
197                     Switches[curSwitchID].DeviceID = curDeviceID;
198                 }   // end if
199             }   // end else if
200         }   // end for
201     } // end if (ProcessingElementFeatureCAR.Switch == TRUE)
202
```

```
203     return RIO_SUCCESS;
204
205 }   // end rioEnumerateNeighbor
206
207 // ***************************************************************************
208 // System clear up
209 // —Reset the previously acquired lock because a master exists elsewhere. Use hostDeviceID to
210 // reset the lock
211
212 STATUS rioClearUp (hostDeviceID) {
213
214     // Clear the host lock
215     if (hostDeviceID > DeviceID –1) {
216         rioReleaseDeviceLock (0, hostDeviceID, 0, hostDeviceID);
217     }
218
219     // Clear the discovered end point device lock
220     while (DeviceID >= 1) {
221         rioReleaseDeviceLock (0, DeviceID-1, 0, hostDeviceID);
222         DeviceID --;
223     }
224
225     // Clear the discovered switch device lock
226     while (SwitchID >= 1) {
227         rioReleaseDeviceLock (0, Switches[SwitchID–1].DeviceID,
228                               Switches[SwitchID-1].hopCount, hostDeviceID);
229         SwitchID --;
230     }
231
232     return RIO_SUCCESS;
233 }   // end rioClearUp
```

A.4 System Bring Up Example

This section walks-through a system bring up example. The system described in this example is shown in Figure A-1.

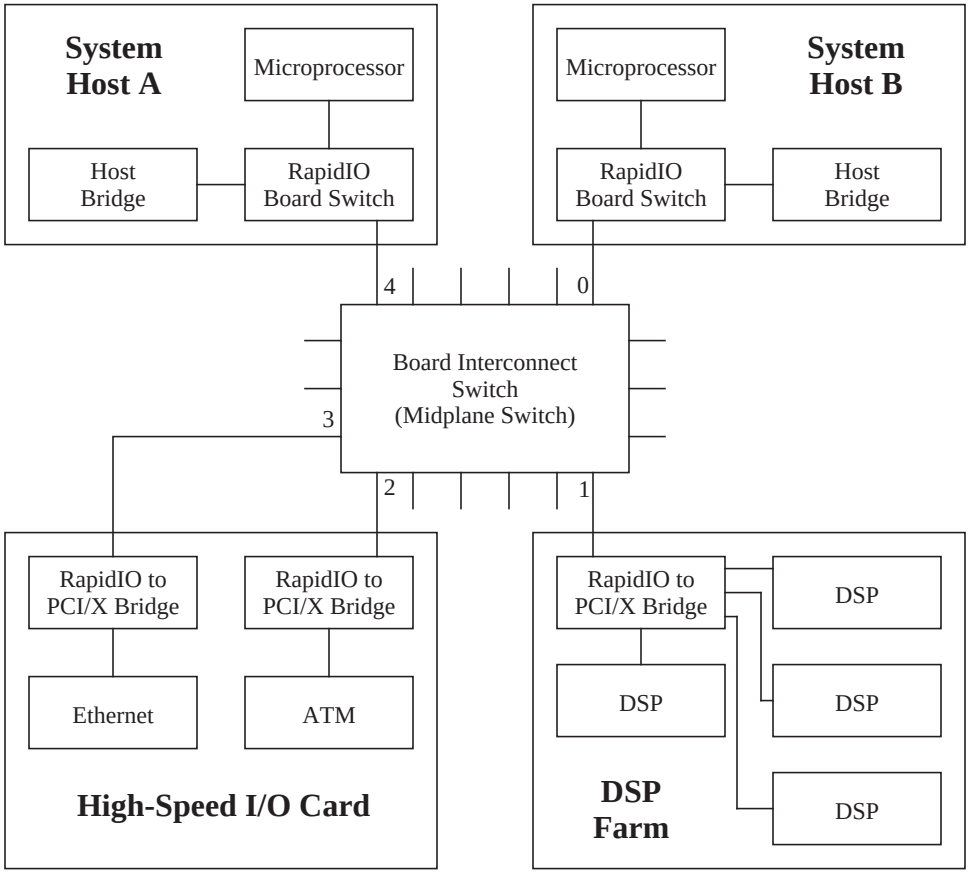
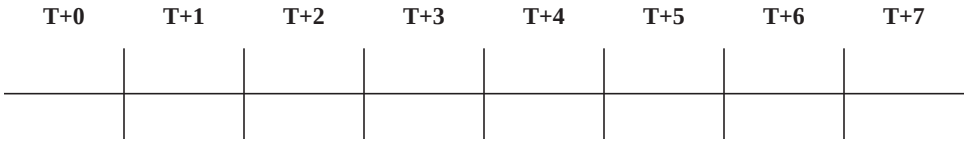


Figure A-1. Example System

Referring to Figure A-1, system Host A is preloaded with device ID 0x00 and system Host B is preloaded with device ID 0x01. Host A is configured to accept maintenance packets with destination IDs of 0x00 and 0xFF. Host B is configured to accept maintenance packets with destination IDs of 0x01 and 0xFF. System Bring Up advances through time slots along the following timeline:



The time slots shown above are defined as follows:

- **T+0:** Host A begins RapidIO enumeration.
- **T+1:** Host B begins RapidIO enumeration and Host A continues RapidIO enumeration.
- **T+2:** Host B discovers another host in the system (Host A) and waits.
- **T+3:** Host A discovers a higher priority host in the system (Host B) and retreats.

- **T+4:** Host B assumes sole enumeration of the system.
- **T+5:** Host B enumerates the PE on switch port 1.
- **T+6:** Host B enumerates the PEs on switch ports 2, 3 and 4.
- **T+7:** System enumeration is complete.

The following describes the actions taken during each time slot in more detail:

Time T+0

Host A attempts to acquire the lock from its Host Base Device ID Lock CSR by writing 0x00 to the CSR. Host A confirms it has acquired the lock when it reads the value of 0x00 (the host device ID) from the Lock CSR. Host A continues by reading the Processing Element Features CAR and adding the information from the CAR to its RapidIO device database. Host A updates its Base Device ID CSR with the host device ID (0x00).

Time T+1

Host B attempts to acquire the lock from its Host Base Device ID Lock CSR by writing 0x01 to the CSR. Host B confirms it has acquired the lock when it reads the value of 0x01 (the host device ID) from the Lock CSR. Host B continues by reading the Processing Element Features CAR and adding the information from the CAR to its RapidIO device database. Host B updates its Base Device ID CSR with the host device ID (0x01).

Host A begins neighbor enumeration. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x00), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. Host A confirms it has acquired the lock when it reads the value of 0x00 (the host device ID) from the Lock CSR.

Time T+2

Host B begins neighbor enumeration. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. However, after Host B issues a maintenance read from the Lock CSR it finds that the device was already locked by host device ID 0x00. Because Host B has a higher priority than the current lock holder (0x01 is greater than 0x00), Host B spins in a delay loop and repeatedly attempts to acquire the lock.

Time T+3

Host A continues neighbor enumeration. It issues a maintenance read cycle to the Device Identity CAR of the Board Interconnect Switch and looks for a matching entry in the device database. Device configuration continues because no match is found (Host A has not enumerated the device). Host A reads the Source Operations and Destination Operations CARs for the device. It is determined that the device

does not support read/write/atomic operations and does not require a device ID. Host A reads the Processing Element Feature CAR for the device and determines that it is a switch element.

Because the device is a switch, Host A reads the Switch Port Information CAR and records the device identity in the switch database. Next, Host A adds a set of entries to the switch's routing table. For each previously discovered device ID, an entry is created containing a target ID (0xFF), hop count (0), and the route port number (from the Switch Port Information CAR). The switch database is updated with the same routing information. Host A reads the Port Error Status CSR for switch port 0, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch’s routing table containing target ID (0xFF), hop count (0), and the route port number (0).

Host A continues neighbor enumeration using a hop count of 1. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the neighbor PE on port 0. A maintenance write of the host device ID (0x00), the destination device ID (0xFF), and the hop count (1) is issued for the Lock CSR. However, after Host B issues a maintenance read from the Lock CSR it finds that the device was already locked by host device ID 0x01. Because Host A has a lower priority than the current lock holder (0x00 is less than 0x01), Host A retreats. It begins the process of backing out all enumeration and configuration changes it has made.

Host A checks its device and switch databases to find all host locks it obtained within the system (System Host A and the Board Interconnect Switch). It issues a maintenance write transaction to their Host Base Device ID Lock CSRs to release the locks.

Time T+4

As Host B spins in its delay loop, it attempts to acquire the lock from the Host Base Device ID Lock CSR of the Board Interconnect Switch. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (0) is issued for the Lock CSR. Because Host A released the lock, Host B is able to confirm it has acquired the lock when it reads the value of 0x01 from the Lock CSR.

Host B continues neighbor enumeration. It issues a maintenance read cycle to the Device Identity CAR of the Board Interconnect Switch and looks for a matching entry in the device database. Device configuration continues because no match is found (Host B has not enumerated the device). Host B reads the Source Operations and Destination Operations CARs for the device. It is determined that the device does not support read/write/atomic operations and does not require a device ID. Host B reads the Processing Element Feature CAR for the device and determines that it is a switch element.

Because the device is a switch, Host B reads the Switch Port Information CAR and records the device identity in the switch database. Next, Host B adds a set of entries to the switch's routing table. For each previously discovered device ID, an entry is

created containing a target ID (0xFF), hop count (0), and the route port number (from the Switch Port Information CAR). The switch database is updated with the same routing information. Host B reads the Port Error Status CSR for switch port 0, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch's routing table containing target ID (0xFF), hop count (0), and the route port number (0). Host B detects that it is attached to port 0. Because Host B has already been enumerated, neighbor enumeration continues on the next port.

Time T+5

Host B reads the Port Error Status CSR for switch port 1, verifying that it is possible for the port to have a neighbor PE. An entry is created in the switch's routing table containing target ID (0xFF), hop count (0), and the route port number (1).

Host B continues neighbor enumeration using a hop count of 1. It attempts to acquire the lock from the Host Base Device ID Lock CSR of the neighbor PE on port 1. A maintenance write of the host device ID (0x01), the destination device ID (0xFF), and the hop count (1) is issued for the Lock CSR. Host B confirms it has acquired the lock when it reads the value of 0x01 from the Lock CSR.

Host B issues a maintenance read cycle to the Device Identity CAR of the DSP Farm and looks for a matching entry in the device database. Device configuration continues because no match is found (Host B has not enumerated the device). Host B reads the Source Operations and Destination Operations CARs for the device. It is determined that the device supports read/write/atomic operations. A maintenance write is used to update the Base Device ID CSR with the value of 0x00 (the first available device ID). DeviceID is incremented and compared with the Host B device ID. Because they are equal, deviceID is assigned the next available device ID.

Time T+6

The process described in the previous step (Time T+5) is repeated on switch ports 2–4. Device IDs 0x02, 0x03, and 0x04 are assigned to the PEs on switch ports 2, 3 and 4, respectively.

Time T+7

Host A detects that its Host Base Device Lock CSR has been acquired by another host device, indicating it has been enumerated. Host A can initiate passive discovery to build a local system database.

Glossary of Terms and Abbreviations

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

A

Application programming interface (API.). A standard software interface that promotes portability of application programs across multiple devices.

C

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

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

D

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

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

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

Discovery. The passive exploration of a RapidIO network fabric. This process involves walking an already enumerated RapidIO fabric to determine network topology and resource allocations.

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

E

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

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

Enumeration. The active exploration of a RapidIO network fabric. This process involves configuring device identifiers and maintaining proper host locking.

H

Hardware abstraction layer (HAL). A a standard software interface to device-specific hardware resources.

I

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

O

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

P

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

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

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

S

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

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

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

T

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

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

W

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

Write port. Hardware within a processing element that is the target of a port-write operation.