Consortium Specification

RNIC Programming Interface (RNICPI)
Version 1.0

The Interconnect Software Consortium

in association with

Consortium Specification

RNIC Programming Interface (RNICPI) Version 1.0

ISBN: 1-931624-58-5

Document Number: C054

Published by The Open Group, September 2005.

Comments relating to the material contained in this document may be submitted to:

rnic-pi-comments@opengroup.org

Contents

List of Figures

List of Tables

Preface

The Interconnect Software Consortium

The purpose of the Interconnect Software Consortium is to develop and publish software specifications, guidelines, and compliance tests that enable the successful deployment of fast interconnects such as those defined by the InfiniBand specification.

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/testing.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Technical Standard for the RNIC Programming Interface (RNICPI). It has been developed and approved by The Interconnect Software Consortium in association with The Open Group.

The version of the specification has the format *m.n*, where *m* and *n* denote the major version number and minor version number, respectively. The present publication corresponds to Version 1.0 of the RNICPI.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- An unchanged major version number (*m*) and a new *minor version number* (*n*) indicate there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. The new version of the specification is source-code compatible with the previous one and *replaces* the previous specification.
- A new *major version number* (*m)* and minor version number *(n)* set to zero indicate there is substantive change to the definitive information contained in the previous version of the specification, and there may also be additions/extensions. The new version of the specification is not source-code compatible with the previous one. The specifications corresponding to both versions are maintained as current publications.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames and type names.
- *Italic* strings are used for emphasis. Italics in text also denote variable names, functions, and data structures.
- Normal font is used for the names of constants and literals.
- Syntax and code examples are shown in `fixed width` font.

Reference Page Conventions

The following editorial conventions are used on all reference pages of this document:

- The NAME section gives the name and title of the item.
- The SYNOPSIS section summarizes the syntax for an item.
- The DESCRIPTION section provides a summary description of an item.
- The RETURN VALUE section (for routines only) lists the returned values including immediate errors for API routines, along with a description for each.
- The CONSUMER USAGE section optionally provides information on the usage of the interface to the RNICPI Consumer.
- The IMPLEMENTATION NOTES section optionally provides information on implementation issues related to the current reference page.
- The SEE ALSO section refers to other reference pages with related content.

Trademarks

Boundaryless Information Flow™ is a trademark and UNIX® and The Open Group® are registered trademarks of The Open Group in the United States and other countries.

InfiniBand™ is a trademark of the InfiniBand™ Trade Association.

POSIX® is a registered trademark of the Institute of Electrical and Electronic Engineers, Inc.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Acknowledgements

The Interconnect Software Consortium gratefully acknowledges the contribution of the following in the development of the RNICPI, Version 1.0:

Caitlin Bestler
John Carrier
James Dykman
Eitan Eliahu
Uri Elzur
Carl Hensler
Venkata Jagana
Ted Kim
Martin Kirk
Mike Krause
Krishna Kumar
Bernard Metzler
Sriram Narasimhan
Fredy Neeser
Mike Penna
Kevin Reilly
Jordi Ros
Jay Rosser
Kanoj Sarcar
Muhammad Shafiq
Bob Sharp
Bill Taylor
Robert Teisberg
Chait Tumuluri
Ramesh VelurEunni

1 Introduction

1.1 Goals

The RDMA-Capable Network Interface Controller Programming Interface (RNICPI) defines programming interfaces for RDMA-capable Network Interface Controllers (RNICs), focusing on the services provided by the network interface driver. The RNICPI is intended for use by intermediate software and not directly by applications. The RNICPI takes into account existing Verb specifications that are guidance for the interface and services that should be provided by an RNIC. The value proposition of the RNICPI is to simplify and enable the incorporation of RNICs into various operating system environments.

The RNICPI interface specification aims to be mostly OS-agnostic. Support for any OS specifics is hidden by common interface syntax. OS-specific operations are supported by leaving the interpretation of some call parameters opaque to the RNICPI. Any OS-specific interpretation of call parameters is clearly stated in the specification.

It is the intention of the RNICPI to give access to different underlying RDMA transports. The present specification supports InfiniBand and iWARP transports. Dependent on the capabilities of the actual Interface Adapter, a certain set of functionalities will result; that is, RNICPI as an interface is exporting but not enriching functionality.

In more detail, RNICPI, Version 1.0 is giving access to the following RDMA transport services:

- Reliable Connection Services as part of the InfiniBand Architecture Specification 1.1 [IB-R1.1] and 1.2 [IB-R1.2]
- iWARP transports as defined in: [MPA-IETF] [MPA-RDMAC] [DDP-IETF] [DDP-RDMAC] [RDMAP-IETF] [RDMAP-RDMAC] [SCTP-RDMA]

This document defines interfaces for the support of both privileged and non-privileged RDMA Services Consumers. The specification is structured as follows:

- An introduction (this section)
- A glossary (see Section 2)
- A section on global behavior (see Section 4)
- A section on the assumed RDMA host stack implementation (see Section 3)
- Sections on memory management and connection management (see Section 5 and 6)
- Reference pages for RNICPI calls and their supporting data type definitions (see Section 7 and 8)
- Implementer’s guides (see Section 9)

- Sample header files (see Section 10)

The introduction and all Appendices, including implementer's guides and sample header files, are informative only; the remaining sections are the normative sections of the specification.

The key words "MUST", MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

1.2 RNICPI Architecture Model

Figure 1 shows the components and interfaces comprising the part of end systems architecture where the RNICPI gets employed. Note that the intent is here to describe the components at a functional level. Dependent on the host operating system environment and efficiency considerations, a typical end system implementation may deviate from this model by grouping together functionality and/or introducing other components to accompany the resulting communication stack with additional functionality. Furthermore, due to application needs, components and interfaces solely dedicated to the support of privileged or non-privileged Consumers may not be present at all. For example, a system supporting only privileged RDMA services Consumers will not implement the components serving non-privileged Consumers.

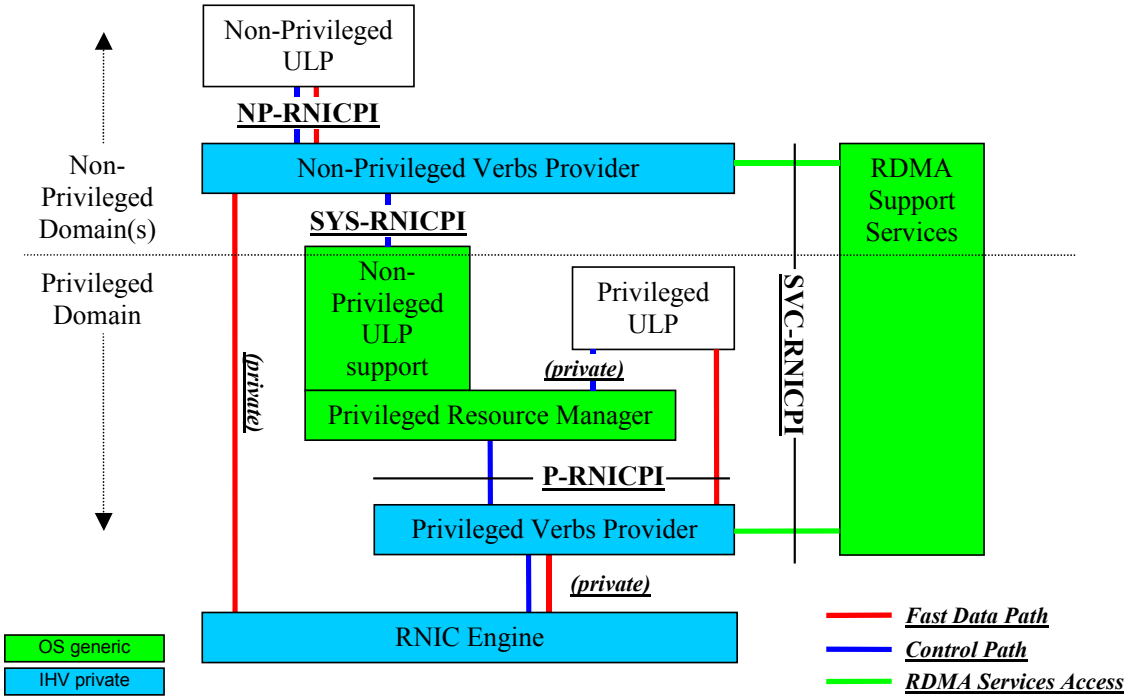

Figure 1: RNICPI Interfaces and Related Host Components

The components shown in Figure 1 are:

- RDMA Network Interface Controller (RNIC) Engine: The component that provides the RDMA Services by implementing the iWARP protocol suite (RDMAP/DDP and MPA/TCP and/or SCTP) and/or the HCA functionality of the InfiniBand stack or any other future RDMA Transport.

- Privileged Verbs Provider: The component implementing the privileged standard interface for accessing RDMA Services provided by the RNIC Engine.

- Privileged Resource Manager: The component responsible for managing resources associated with RDMA services. While not sending or receiving data, the component is policing all local resource reservations from both privileged and non-privileged ULPs. This component is further checking and enforcing the credentials of non-privileged ULPs to use only a subset of RDMA services. The granularity at which credentials can be granted to ULPs is dependent on the operational environment and outside the scope of the RNICPI.

- Privileged Upper Layer Protocol (ULP): The component representing the application or middleware using RDMA services directly via the Privileged Verbs Provider. A typical middleware would be the implementation of an RDMA-capable API for privileged Consumers such as kDAPL [kDAPL]. A Privileged ULP is trusted by the local system to not maliciously attack the operating environment, but is not trusted to deal fairly with allocation and usage of RDMA resources [RDMASEC].

- Non-Privileged ULP Support: The component implementing proxy services to provide the Non-Privileged ULP controlled access to privileged control operations of the RNICPI.

- Non-Privileged Verbs Provider: The component implementing the non-privileged standard interface for accessing RDMA services provided by the RNIC Engine.

- Non-Privileged Upper Layer Protocol (ULP): The component residing in the non-privileged domain that represents the application or middleware using RDMA services via the Non-Privileged Verbs Provider. A typical middleware would be the implementation of an RDMA-capable API for non-privileged Consumers such as uDAPL [uDAPL] or IT-API [IT-APIv2]. A Non-Privileged ULP is assumed to be untrusted in the respect of fair resource handling. Dependent on the credentials given to the ULP, several levels of trust with respect to the operational environment are possible. The Privileged Resource Manager may restrict the Non-Privileged ULP's credentials to use only a subset of the RDMA services.

- RDMA Support Services: The component implementing a set of utility functions provided for both the Privileged and Non-Privileged Verbs Provider. This specification assumes the provision of services including memory pinning and RNIC device management.

1.2.1 RNICPI Interfaces Specified in this Document

As shown in Figure 1, the architectural model assumes the existence of one privileged and optionally one or more non-privileged system domains. The non-privileged domain is hosting Non-Privileged ULPs; the privileged domain is hosting Privileged ULPs and the Resource Manager.

The existence of these different domains where ULPs may reside and the provision of additional RDMA Support Services by the host system led to the specification of the following four distinct RNICPI interfaces (see Figure 1):

1. Privileged RNICPI (P-RNICPI): This interface implements the full set of operations as specified in the RDMA Verbs document [VERBS-RDMAC] and the InfiniBand Specifications [IB-R1.1] and [IB-R1.2]. It further extends this set of functions as discussed in Section 1.2.1.3. The P-RNICPI is exported to the Resource Manager (control type operations) and to the Privileged ULP (fast path data type operations).

2. Non-Privileged RNICPI (NP-RNICPI): This interface implements most of the operations as specified in the RDMA Verbs document [VERBS-RDMAC] and the InfiniBand Specifications [IB-R1.1] and [IB-R1.2], including those specifically targeting user-level usage. It further extends this set of functions as discussed in Section 1.2.1.3. The NP-RNICPI is exported to the ULP by the Non-Privileged Verbs Provider. It is not part of the NP-RNICPI to check the credentials of its Consumer (NP-ULP) to be sufficient to request a certain RDMA service.

3. SYS-RNICPI: This interface implements the support for privileged RNICPI operations (control type operations) for the Non-Privileged Verbs Provider. While the ULP accesses all RDMA services via the NP-RNICPI, some operations (such as operations which will result in the manipulation of host or RNIC resources; e.g., QP creation or communication buffer memory registration) need the acknowledgement of the Resource Manager for further system support. The SYS-RNICPI is exported to the Non-Privileged Verbs Provider by the Non-Privileged ULP Support component.

4. RDMA Support Service Interface (SVC-RNICPI): This interface exports a set of utility functions to the Privileged and Non-Privileged Verbs Providers. While not part of the RDMA Verbs specification, the intention of the definition of this interface is to ease the development of Verbs Provider components for potentially different (OS-specific) operating environments.

1.2.1.1 Naming Conventions

For the definition of the interface calls (*fn_name*()) the following naming conventions apply:

1. Calls available at the P-RNICPI are named *ri_fn_name*().

2. Calls available at the NP-RNICPI are named *ri_fn_name*().

3. Calls available at the SYS-RNICPI are named *ri_sys_fn_name*().

4. Calls available at the SVC-RNICPI are named *ri_svc_fn_name*().

Section 7 defines syntax and semantics of the SYS-RNICPI and the SVC-RNICPI interface calls, and signatures and semantics of the P-RNICPI and NP-RNICPI interface calls. Since P-RNICPI and NP-RNICPI calls are exported to the Consumer in a provider-specific array of function pointers, the Verbs Providers is free to apply a private naming scheme when implementing these functions.

To ease the usage of the interface, Section 10.3 proposes an inlined calling scheme for P-RNICPI and NP-RNICPI calls. To abstract from explicitly dereferencing Verbs Provider-

specific function pointers, inline functions with the exact name as used on the reference pages are defined. This scheme makes some assumptions on the structure of Verbs Provider private resource handles which are not mandatory, but straightforward to implement.

1.2.1.2 *Provided Calls per Interface*

Table 1 lists all available RNICPI calls and the corresponding interfaces.

Table 1: Available RNICPI Function Calls

Function Call	P-RNICPI	NP-RNICPI	SYS-RNICPI	SVC-RNICPI
ri_rnic_open()	*x*			
ri_rnic_close()	*x*			
ri_async_set_event_handler()	*x*			
ri_rnic_get_hdl()	*x*	*x*	*x*	
ri_rnic_query()	*x*	*x*	*x*	
ri_rnic_modify()	*x*	*x*	*x*	
ri_rnic_free_hdl()	*x*	*x*	*x*	
ri_rnic_diag()	*x*	*x*	*x*	
ri_pd_alloc()	*x*	*x*	*x*	
ri_pd_dealloc()	*x*	*x*	*x*	
ri_cq_create()	*x*	*x*	*x*	
ri_cq_query()	*x*	*x*	*x*	
ri_cq_modify()	*x*	*x*	*x*	
ri_cq_destroy()	*x*	*x*	*x*	
ri_cq_set_event_handler()	*x*			
ri_cq_req_notification()	*x*	*x*		
ri_cq_poll()	*x*	*x*		
ri_qp_create()	*x*	*x*	*x*	
ri_qp_query()	*x*	*x*	*x*	
ri_qp_modify()	*x*	*x*	*x*	
ri_qp_destroy()	*x*	*x*	*x*	
ri_special_qp_get()	*x*	*x*	*x*	
ri_qp_postsq()	*x*	*x*		
ri_qp_post_send()	*x*	*x*		
ri_qp_post_rdma_read()	*x*	*x*		
ri_qp_post_rdma_write()	*x*	*x*		
ri_qp_post_bind()	*x*	*x*		

Function Call	P-RNICPI	NP-RNICPI	SYS-RNICPI	SVC-RNICPI
ri_qp_post_fmr()	*x*	*x*		
ri_qp_post_local_invalidate()	*x*	*x*		
ri_qp_post_atomic()	*x*	*x*		
ri_qp_post_datagram()	*x*	*x*		
ri_qp_postrq()	*x*	*x*		
ri_qp_post_recv()	*x*	*x*		
ri_srq_create()	*x*	*x*	*x*	
ri_srq_query()	*x*	*x*	*x*	
ri_srq_modify()	*x*	*x*	*x*	
ri_srq_destroy()	*x*	*x*	*x*	
ri_srq_post()	*x*	*x*		
ri_srq_post_recv()	*x*	*x*		
ri_ltag_alloc()	*x*	*x*	*x*	
ri_mr_reg()	*x*	*x*	*x*	
ri_mr_query()	*x*	*x*	*x*	
ri_mr_rereg()	*x*	*x*	*x*	
ri_mr_dereg()	*x*	*x*	*x*	
ri_mr_reg_phys()	*x*	*x*	*x*	
ri_mr_rereg_phys()	*x*	*x*	*x*	
ri_mr_reg_shared()	*x*	*x*	*x*	
ri_mw_alloc()	*x*	*x*	*x*	
ri_mw_query()	*x*	*x*	*x*	
ri_mw_dealloc()	*x*	*x*	*x*	
ri_mem_pin()	*x*	*x*	*x*	
ri_mem_unpin()	*x*	*x*	*x*	
ri_mem_map()	*x*	*x*	*x*	
ri_mem_unmap()	*x*	*x*	*x*	
ri_mem_sync_rread()	*x*	*x*	*x*	
ri_mem_sync_rwrite()	*x*	*x*	*x*	
ri_ah_create()	*x*	*x*	*x*	
ri_ah_query()	*x*	*x*	*x*	
ri_ah_modify()	*x*	*x*	*x*	
ri_ah_destroy()	*x*	*x*	*x*	

Function Call	P-RNICPI	NP-RNICPI	SYS-RNICPI	SVC-RNICPI
ri_mcast_attach()	*x*	*x*	*x*	
ri_mcast_detach()	*x*	*x*	*x*	
ri_process_sma_mad()	*x*	*x*	*x*	
ri_svc_attach()				*x*
ri_svc_detach()				*x*
ri_svc_lib_reg()				*x*
ri_svc_mem_pin()				*x*
ri_svc_mem_unpin()				*x*

1.2.1.3 Experimental Extensions to this Specification

Any experimental extension to the programming interface defined in this document should not be implemented by overloading specified call parameters with private semantic or introducing new defined values, but by additional experimental interface methods implementing the experimental extensions.

1.2.2 Relation to the RDMAC Verbs Document

The RNICPI follows the interface semantics defined in the RDMAC Verbs document [VERBS-RDMAC]. It extends this semantic in several ways, including:

- Specification of an NP-RNICPI supporting most of the privileged operations to the ULP
- Specification of the SYS-RNICPI to complement the NP-RNICPI with the necessary privileged system support
- Specification of the SVC-RNICPI to ease the development of Verbs Provider components for potentially different (OS-specific) operating environments
- Introducing additional opaque arguments to some calls to allow for OSV and IHV private state maintenance
- Extending the memory registration model by RNICPI-provided memory pinning and memory mapping functionality as well as the support of registering virtual memory
- Supporting the RDMAC/IETF interoperability extensions as proposed in [MPA-IETF]
- Some capabilities defined for transport neutrality are defined as optional features that iWARP IHVs may implement

1.2.3 MPA Support

The RNICPI assumes the MPA startup sequence to take place under control of the ULP. That is, the ULP is responsible for creating the MPA Request or MPA Reply frame, for initiating the sending of the frame, and the interpretation of the content of an incoming MPA startup frame.

Two ways exist to send an MPA Request/Reply frame:

1. In Streaming Mode as the last Streaming Mode message before calling *ri_qp_modify*() to enter RDMA Mode
2. During the *ri_qp_modify*() operation itself as the last Streaming Mode message argument

1.2.4 IETF/RDMAC Interoperability

The iWARP protocol stack consists of the MPA, DDP, and RDMAP protocols that exist in two versions developed by two independent standardization bodies, namely the RDMAC and the IETF. While the RDMAC versions of the protocols were developed first, the IETF versions aim at following the RDMAC specifications very closely. Though a few differences do exist, potentially causing non-interoperability between both versions of the stack:

- The version numbers of the DDP and the RDMAP protocol are the only differences for these protocols (RDMAC: Version 0, IETF: Version 1).
- The IETF version of the MPA protocol was extended by making marker and CRC usage negotiable during an additional startup sequence (MPA request/response). This startup sequence is not part of the RDMAC specification. Therefore, for the RDMAC version, marker and CRC usage is mandatory.

Implementations of the RDMAC version of the stack are already present in the market and its interoperability with upcoming RNIC products based on the IETF specification is unclear. While the IETF currently does not specify how interoperability with RDMAC devices could be achieved, a recent IETF draft suggests ways to solve the problem [MPA-IETF]. Basically, it assumes the existence of RDMAC RNICs, IETF RNICs, and RDMAC/IETF RNICs, where the latter RNIC implementation is assumed to negotiate MPA markers and CRCs like an IETF RNIC, but is additionally able to use both Version 0 and 1 of the RDMAP and DDP protocols. If such an RDMAC/IETF device is able to dynamically “downgrade” from protocol Version 1 to 0, then such a “permissive” device could be used to achieve interoperability with an RDMAC device after the MPA request/response procedure failed.

Without replicating the details of [MPA-IETF], this solution requires that the MPA protocol revision field (only part of the request/response sequence) can be used to signal the supported iWARP stack version. It further assumes, that the ULP or other supporting entities are sending the MPA Request/Reply Frames on behalf of RDMAC RNICs (which are unable to do so) and thus are able to connect with an IETF RNIC for negotiating the iWARP protocol versions “down” to RDMAC mode. When connecting to a permissive IETF/RDMAC RNIC, the RDMA connection could be established.

The RNICPI specification supports this proposal and most of its implications on the P-RNICPI. It introduces new RNIC attributes to signal RDMAC, IETF, or Permissive IETF versions of the protocol stack. In case of a permissive RNIC, DDP and RDMAP protocol versions as well as send markers, receive markers, and CRC enablement can be controlled by the ULP on a per-QP basis.

The current specification does not support Permissive IETF RNICs that do not allow permissiveness on a per-QP basis.

1.3 InfiniBand Support

The current version of the RNICPI provides a subset of the full InfiniBand specification. Reliable connection services, unreliable datagram services, and unreliable datagram multicast services are supported over IB through RNICPI.

1.4 Referenced Documents

The following documents are referenced in this document:

[DDP-IETF] H. Shah et al, Direct Data Placement over Reliable Transports, IETF Internet Draft, July 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-ddp-05.txt).

[DDP-RDMAC] H. Shah et al, Direct Data Placement over Reliable Transports (Version 1.0), RDMA Consortium, October 2002 (www.rdmaconsortium.org/home/draft-shah-iwarp-ddp-v1.0.pdf).

[IB-R1.1] InfiniBand Architecture Specification Volume 1, Release 1.1, InfiniBand Trade Association, November 2002.

[IB-R1.2] InfiniBand Architecture Specification Volume 1, Release 1.2, InfiniBand Trade Association, October 2004.

[IT-API-V1.0] Interconnect Transport API (IT-API) Issue 1.0, The Interconnect Software Consortium, February 2004.

[IT-API-V2.0] Interconnect Transport API (IT-API) Issue 2.0, The Interconnect Software Consortium, May 2005.

[kDAPL] DAT Collaborative, kDAPL: Kernel Direct Access Programming Library, Version 1.2. September 2004 (www.datcollaborative.org/kDAPL12_091504.zip).

[MPA-IETF] P. Culley et al, Marker PDU Aligned Framing for TCP Specification, IETF Internet Draft, February 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-mpa-02.pdf).

[MPA-RDMAC] P. Culley et al, Marker PDU Aligned Framing for TCP Specification (Version 1.0), RDMA Consortium, October 2002 (www.rdmaconsortium.org/home/draft-culley-iwarp-mpa-v1.0.pdf).

[OpenRDMA] Work-in-progress (www.openrdma.org).

[RDMA-SEC] J. Pinkerton et al, DDP/RDMAP Security, April 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-security-07.txt).

[RDMAP-IETF] R. Recio et al, An RDMA Protocol Specification, IETF Internet Draft, July 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-05.txt).

[RDMAP-RDMAC] R. Recio et al, An RDMA Protocol Specification (Version 1.0), RDMA Consortium, October 2002 (www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf).

[RFC 2119] S. Bradner, Keywords for use in RFCs to Indicate Requirement Levels, RFC 2119, Internet Engineering Task Force, March 1997.

[RFC 2461] T. Narten, E. Nordmark, W. Simpson, Neighbor Discovery for IP Version 6 (IPv6), RFC 2461, Internet Engineering Task Force, December 1998.

[RFC 792] J. Postel, Internet Control Message Protocol: DARPA Internet Program Protocol Specification, RFC 792, Internet Engineering Task Force, September 1981.

[SCTP-RDMA] R. Steward et al, Stream Control Transmission Protocol (SCTP) Remote Direct Memory Access (RDMA) Direct Data Placement (DDP) Adaptation, IETF Internet Draft, September 2004 (www.ietf.org/internet-drafts/draft-ietf-rddp-sctp-01.txt).

[uDAPL] DAT Collaborative. uDAPL: User Direct Access Programming Library, Version 1.2, September 2004 (www.datcollaborative.org/udapl12_091504.zip).

[VERBS-RDMAC] J. Hilland et al, RDMA Protocol Verbs Specification (Version 1.0), RDMA Consortium, April 2003 (www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf).

2 Definitions

Access Rights

The Local and Remote Memory Access Rights assigned to an STag. This includes Local Read, Local Write, Remote Read, Remote Write, Remote Access Flag, and Bind.

Affiliated Asynchronous Event

This is an indication from the RNICPI to the Consumer that an event has occurred related to a specific identifiable RNIC Resource, such as a Completion Queue or Queue Pair.

Affiliated Error

An error that can be directly related back to a specific RNIC Resource, such as a QP, SRQ, or CQ, but that cannot be returned through a Work Completion.

ARP

Address Resolution Protocol. A network layer protocol used to convert an IP address into a physical address, such as a MAC address.

Asynchronous Error

An error that could not be reported through immediate or completion error-handling mechanisms at the local end. An asynchronous mechanism is necessary as a single point of error handling for errors which could not otherwise be reported through the normal mechanism since they are not associated directly with any single QP, SRQ, or CQ or the QP and/or CQ is in a state where an error cannot be reported. Asynchronous errors may be Unaffiliated or may be Affiliated with a specific QP, CQ, or SRQ.

Base Tagged Offset (Base TO)

The offset assigned to the first byte of a Memory Region or a Memory Window.

Bind, Binding, Bound

The act of associating an STag, TO, and Length within a previously registered Memory Region in order to define a Memory Window.

Block List

A list of physical addresses describing a set of memory blocks, which specifies the block size, list of physical addresses, and offset to the start of the Memory Region of the first block. Each block has the same length and that length can be any value in the range supported by the RNIC. Each block may start at a byte granularity address. The starting address for the entire list may be an offset into the first block and the entire list may have any length.

Complete (Completed, Completion, Completes)

When the Consumer can determine that a particular RDMA Operation has performed all functions specified for the RDMA Operation, including Placement and Delivery. This can be determined through a Work Completion for Signaled Work Requests. For Unsignaled Work Requests, this means that the Completion Rules have been met.

Completion Error

A Processing Error reported through the Completion Queue.

Completion Queue (CQ)

A sharable queue containing one or more entries that can contain Completion Queue Entries. A CQ is used to create a single point of completion notification for multiple Work Queues. The Work Queues associated with a Completion Queue may be from different QPs and of differing queue types (SQs or RQs).

Completion Queue Entry (CQE)

The RNIC Interface internal representation of a Work Completion.

Completion Status

The resultant status of a Work Request returned as part of a Work Completion.

Consumer, Verbs Consumer

A software process that communicates using RNICPI. The Consumer typically consists of an application program, or an operating system adaptation layer such as the kAL, which provides some API.

Device Driver Interface (DDI)

OS-specific standardized kernel interfaces used for writing device drivers. Its definition includes the calling and return syntax of the kernel functions that are used for communication between drivers and the devices they control, the calling and return syntax of the entry point routines and kernel utility functions that are used for communication between drivers and the kernel, and the contents of the data structures that are used to exchange information between drivers and the kernel.

Direct Data Placement Protocol (DDP)

A wire protocol that supports Direct Data Placement by associating explicit memory buffer placement information with the LLP payload units.

Data Delivery (Delivery, Delivered, Delivers)

Delivery is defined as the process of informing the ULP or Consumer that a particular Message is available for use. This is specifically different from Data Placement, which may generally occur in any order, while the order of Data Delivery is strictly defined.

Data Placement (Placement, Placed, Places)

A mechanism whereby ULP data contained within RDMA Transport Segments may be put directly into its final destination in memory without processing by the ULP. This may occur even when the Segments arrive out of order. Note that this differs from Data Delivery (see Data Delivery). From the RNICPI viewpoint, Data Placement is only confirmed upon Completion.

Event

An indication provided by the RNICPI to the ULP to indicate a Completion or other condition requiring immediate attention.

First Byte Offset (FBO)

The offset into the first Physical Buffer of a Memory Region. The value of the FBO cannot exceed the size of the Physical Buffer Entry Size associated with the Memory Region.

Handle

An opaque identifier used to reference an RNIC or an RNIC Resource. Whether this is an index, object, or some other construct is outside the scope of this specification.

ICMP

Internet Control Message Protocol. An extension to the Internet Protocol (IP). ICMP is defined by [RFC 792].

Immediate Error

An error discovered by the RNICPI provider during invocation and reported back without affecting the RNIC.

Invalidate STag (Invalidate, Invalidated, etc.)

A mechanism used to prevent the Remote Peer from re-using an Advertised STag, until the Local Peer transitions the STag to the Valid state.

Invalidate Local STag

A Work Request that takes an STag that is locally valid and performs an Invalidate STag operation.

iWARP

A suite of wire protocols comprised of [RDMAP-IETF] and [DDP-IETF]. The iWARP protocol suite may be layered above TCP and MPA [MPA-IETF], or it may be layered over SCTP [SCTP-RDMA] or other transport protocols.

Kernel Access Layer (kAL)

Kernel software module, which in a typical RDMA host stack implementation represents all OS generic RDMA support in the privileged domain. The kAL exports the P-RNICPI and the SVC-RNICPI.

Kernel Verbs Provider (kVP)

The Kernel Verbs Provider is an RNIC vendor private kernel software module that typically implements the vendor-specific code to access the RNIC hardware from the privileged domain. The kVP exports the P-RNICPI interface to the kAL.

LTag

An opaque reference to a Memory Region that can be used in Work Requests for local access. If the Memory Region is enabled for fast memory registration, then the least significant byte is defined as the “key” portion, and is under Consumer control. For iWARP, the key portion always exists. For InfiniBand, it is only guaranteed to exist if fast memory registration was enabled.

Local Access

The rights used to verify the RNIC's ability to access the Data Sink for incoming Untagged Messages, the Data Source for outgoing Untagged Messages, and the Data Source for outgoing RDMA Write Messages.

Lower Layer Protocol (LLP)

The protocol layer beneath the protocol layer currently being referenced. For example, for DDP, the LLP is SCTP, MPA, or other transport protocols. For RDMAP, the LLP is DDP.

LLP Closed (LLP Close)

When the LLP Stream can no longer be used for data transmission. If there is a single LLP Stream on an LLP Connection, it may also mean that the LLP Connection has been torn down. For example, for TCP this could include the states TIME_WAIT, CLOSING, LAST-ACK, and CLOSED.

632 **LLP Connection**

633 Corresponds to an LLP transport-level connection between the peer LLP layers on two nodes.

634 **LLP Reset**

635 The abnormal LLP closing mechanism, usually used to indicate that the LLP Stream (and
636 possibly Connection) was aborted mid-stream. An example of this would be a TCP connection
637 being closed due to the reception or transmission of a TCP RST on the connection.

638 **LLP Stream**

639 Corresponds to a single bi-directional LLP transport-level association between the peer LLP
640 layers on two Nodes. One or more LLP Streams may map to a single transport-level LLP
641 Connection. For transport protocols that support multiple Streams per connection (e.g., SCTP),
642 an LLP Stream corresponds to one transport-level Stream.

643 **MAC**

644 Media Access Control. Sublayer of the Data Link Layer responsible for moving data packets to
645 and from one Network Interface Card to another. A MAC Address is a physical address that
646 uniquely identifies a Network Interface Card within a network.

647 **Memory Region (MR)**

648 An area of memory that the Consumer wants the RNIC to be able to (locally or locally and
649 remotely) access directly in a logically contiguous fashion. A Memory Region is identified by an
650 STag, a Base TO, and a length. A Memory Region is associated with a Physical Buffer List
651 through the STag.

652 **Memory Registration (Registration, Register)**

653 The mechanism used to enable direct (local or local and remote) access by the RNIC of a
654 Consumer Memory Region. The memory registration operation associates a Physical Buffer List
655 to the Steering Tag (STag) returned.

656 **Memory Window (MW)**

657 A subset of a Memory Region, which can be remotely accessed in a logically contiguous
658 fashion. A Memory Window is identified by an STag, a Base TO, and a length, but also
659 references an underlying Memory Region and has Access Rights.

660 **Narrow Memory Window**

661 A Memory Window that is associated with a single Queue Pair.

ND

Neighbor Discovery. Protocol to discover relationships between neighboring nodes including discovery of physical network (MAC) addresses. ND is defined by [RFC 2461].

Non-Privileged Mode

An operating mode in which Consumers must rely on another agent, having a sufficiently high level of privilege, to manipulate OS data structures. In this mode, the Consumer is using RDMA services via the NP-RNICPI.

Non-Shared Memory Region

A Memory Region that solely owns the Physical Buffer List associated with the Memory Region. Specifically, the PBL is not shared, and has never been shared, with another Memory Region.

Outstanding

The state of a Work Request after it has been posted on a Work Queue, but before the retrieval of the Work Completion, or confirmation that the WR has been completed, by the Consumer.

Page List

A list of physical addresses describing a set of memory pages, which specifies the page size, list of physical addresses, and offset to the start of the Memory Region of the first page. The starting physical address of each page is aligned on power-of-two addresses and the size of the page is a power of two. Note that it is possible for the starting offset to be an offset into the first page and to be of a byte granularity and the entire list may have an arbitrary length.

Physical Address

A physical address is used by an RNIC to retrieve contents from the local host's memory. Physical addresses are determined via the translation of the STag and Tagged Offset by the use of the Memory Translation and Protection Table(s).

Physical Buffer

A set of physically contiguous memory locations that can be directly accessed by the RNIC through Physical Addresses. A Physical Buffer can either be a block buffer or a page buffer, depending on its use as part of a Page List or Buffer List.

Physical Buffer List (PBL)

A list of Physical Buffers. The Physical Buffer List can either be a Block List or a Page List.

692 **Physical Memory Addresses**

693 The addresses an RNIC uses when accessing host system memory.

694 **Pinning Memory**

695 A function supplied by the OS that forces the Memory Region to be resident in physical memory
696 and keeps the virtual-to-physical address translations constant from the RNIC's point of view.

697 **PMTU**

698 Path Maximum Transmission Unit. Maximum unit of data that can be transmitted over the
699 network between two endpoints without being fragmented. Exceeding PMTU will cause packet
700 fragmentation or packet dropping.

701 **Privileged Mode**

702 A mode in which Consumers operate where they have a privilege level sufficient to access OS
703 internal data structured directly, and that have the responsibility to control access to the
704 P-RNICPI.

705 **Protection Domain (PD)**

706 A mechanism for tracking the association of Queue Pairs, Memory Windows, and Memory
707 Regions. PDs are intended to provide protection of one process from accessing another's
708 memory through the use of the RNIC.

709 **Protection Domain Handle (PD Handle)**

710 The handle that represents a Protection Domain. It is passed in as an Input Modifier when
711 creating QPs, Memory Windows, and MRs. The value of PD Handles are compared during
712 processing of Work Requests.

713 **Queue Pair (QP)**

714 The pair of queues that allow the Consumer to interact with the RNIC Interface. The two queues
715 are the Send Queue and the Receive Queue. Each queue stores a Work Queue Element from the
716 time it is posted until the time it is completed.

717 **Queue Pair Identifier (QP ID)**

718 An identifier representing a Queue Pair.

719 **Receive Queue (RQ)**

720 One of the two Work Queues associated with a Queue Pair. The Receive Queue contains Work
721 Queue Elements that describe the buffers into which data from incoming Send Operation Types
722 is placed.

Remote Access

The Access Rights used to verify the RNIC's ability to access the Data Sink for incoming DDP Tagged Messages and the Data Source for RDMA Read Request Messages.

Remote Direct Memory Access (RDMA)

A method of accessing memory on a remote system in which the local system specifies the remote location of the data to be transferred. Employing an RNIC in the remote system allows the access to take place without interrupting the processing of the CPU(s) on the system.

RDMA Network Interface Controller (RNIC)

A network I/O adapter or embedded controller with iWARP and/or InfiniBand functionality.

RDMA Transport

A protocol suite implementing end-to-end RDMA Operations, such as InfiniBand or iWARP.

RDMA Operation

A sequence of RDMAP or IB Messages, including control Messages, to transfer data from a Data Source to a Data Sink.

RDMA Protocol (RDMAP)

A wire protocol that supports RDMA Operations to transfer ULP data between a Local Peer and the Remote Peer. See [RDMAP-IETF].

RTag

An opaque reference to a Memory Region or Memory Window that can be advertised to the remote peer to enable remote memory access. Additionally, it may be used to specify the local Data Sink for an RDMA Read Work Request.

Scatter/Gather Element (SGE)

An individual entry in a Scatter/Gather List. Each SGE consists of STag, Tagged Offset, and Length.

Scatter/Gather List (SGL)

A List of Scatter/Gather Elements. The list describes one or more ULP Buffers that will have their data gathered on transmission or scattered upon reception.

Send Queue (SQ)

One of the two Work Queues associated with a Queue Pair. The Send Queue contains PostSQ Work Queue Elements that have specific operation types, such as Send Type, RDMA Write, or RDMA Read Type Operations, as well as STag operations such as Bind and Invalidate.

Shared Memory Region

An MR that currently shares, or at one time shared, the Physical Buffer List associated with the Memory Region. Specifically, the PBL is currently shared or was previously shared with another Memory Region.

Shared Receive Queue (SRQ)

An optional mechanism which allows the Receive Queues from multiple QPs to retrieve Receive Queue Work Queue Elements from the same shared queue as needed.

Signaled

A WR that requires that the RNIC generate a Work Completion.

Solicited Event (SE)

A facility by which an RDMA Operation sender may cause an Event to be generated at the recipient, if the recipient is configured to generate such an Event, when a Send with Solicited Event or Send with Solicited Event and Invalidate Message is received.

Steering Tag (STag)

An iWARP-specific identifier of a Memory Window or Memory Region. STags are composed of two components: STag Index and STag Key. The Consumer forms the STag by combining the STag Index with the STag Key. See LTag and RTag for a more transport independent representation of an MW or MR identifier.

STag Key

The least significant 8-bit portion of an STag. This field of an STag can be set to any value by the Consumer when performing a Memory Registration operation, such as Bind Memory Window, Fast-Register Memory Region, and Register Memory Region.

STag Index

The most significant 24 bits of an STag. This field of the STag is managed by the Verbs Provider and is treated as an opaque object by the Consumer.

Tagged Buffer

A buffer that can be Advertised to a Remote Peer through exchange of an STag, Tagged Offset, and length.

Tagged Offset (TO)

The offset within a Tagged Buffer.

Terminate

An RDMA Message used by a Node to pass an error indication to the Remote Peer on an RDMA Stream.

Unaffiliated Asynchronous Event

This is an indication from the Verb layer to the Consumer that an event has occurred unrelated to any single identifiable RNIC Resource.

Unaffiliated Error

An error that cannot be directly related back to a specific RNIC Resource, such as a QP, SRQ, or CQ.

Unsignaled

A Work Request which only generates a Work Completion if it encounters an error during processing.

Upper Layer Protocol (ULP)

The component representing the application or middleware using RDMA services directly via the Privileged Verbs Provider.

User Access Layer (uAL)

Software module, which in a typical RDMA host stack implementation represents all OS generic RDMA support in the non-privileged domain. The uAL exports the NP-RNICPI, the SVC-RNICPI, and the SYS-RNICPI.

User Verbs Provider (uVP)

RNIC vendor private software module. It implements in a typical RDMA host stack implementation vendor private code to access the RNIC hardware from the non-privileged domain. The uVP exports the NP-RNICPI interface to the uAL.

807 **Verbs**

808 Abstract description of the functionality of an RNIC. The RNICPI defined in this document is
809 one implementation of the Verbs Interface.

810 **Virtual Address**

811 An address represented in the address space of a local process on a node. It is generally used to
812 present logically contiguous addressability for an underlying and possibly non-contiguous list of
813 physical pages.

814 **Virtual Address Based Tagged Offset (VA Based TO)**

815 The Base TO of an MR or MW that starts at a non-zero TO.

816 **Wide Memory Window**

817 A Memory Window which is associated with a Protection Domain and which can potentially be
818 shared between Queue Pairs residing in the same Protection Domain.

819 **Work Completion (WC)**

820 The output modifiers that the Consumer retrieves from a Completion Queue indicating the
821 results of a Work Request.

822 **Work Queue (WQ)**

823 One of either a Send Queue or Receive Queue.

824 **Work Queue Element (WQE)**

825 The RNIC Interface's internal representation of Work Request.

826 **Work Request (WR)**

827 An elementary object used by Consumers to enqueue a requested operation (WQE) onto the
828 Send and Receive Queues of a QP.

829 **Zero Based Tagged Offset (Zero Based TO)**

830 The Base TO of an MR or MW that starts at TO=0.

3 Assumed RDMA Host Stack Implementation

The RNICPI architecture model introduced in Section 1.2 takes a functional view on the stack. A real implementation will typically group components in a more efficient way. Figure 2 gives a high-level overview of a possible implementation of the architecture. Here, an operating system was assumed, which is comprised of a privileged operating system kernel and one or more non-privileged applications running in non-privileged domains. This example architecture closely follows the implementation architecture of the OpenRDMA project [OpenRDMA].

If not otherwise stated, the terms used to describe the RDMA stack implementation model introduced in this section will be used throughout the remaining document.

3.1 Host Stack Components and Interfaces

In the privileged domain, the Privileged Resource Manager, the RDMA Support Services, the Privileged ULP, and the Non-Privileged ULP Support components are combined into one kernel module. This kernel RDMA Services Access Layer (kAL) exports the SVC-RNICPI to Privileged (kernel) Verbs Providers (kVP) and calls the kVP via the exported P-RNICPI. Implementing privileged (kernel-level) ULP functionality, the kAL further exports RDMA-capable APIs (such as kDAPL) to kernel-level Consumers. Within a system, only one kAL entity may exist. It may attach to one or more kVP entities.

In the non-privileged domain, RDMA Support Services as well as Non-Privileged ULP Support and Non-Privileged ULP(s) are combined together into one module. This user-level RDMA Services Access Layer (uAL) exports the SVC-RNICPI and the SYS-RNICPI to Non-Privileged Verbs Providers (uVP). It calls the uVP via the exported NP-RNICPI. The uAL further exports RDMA-capable APIs (such as uDAPL or IT-API) to user-level Consumers. A uAL entity must be present for each application process that wants to use RDMA services. Dependent on the existence of RNICs in the system, the uAL may attach to one or more uVPs.

kAL and uAL(s) are sharing a private interface that logically interconnects the Non-Privileged ULP Support components of both kAL and uAL. Dependent on the implementation, it may also be used to transparently communicate state information between uVP and kVP (see Section 4.6).

The kVP receives generic OS Services Support via an OS private DDI interface.

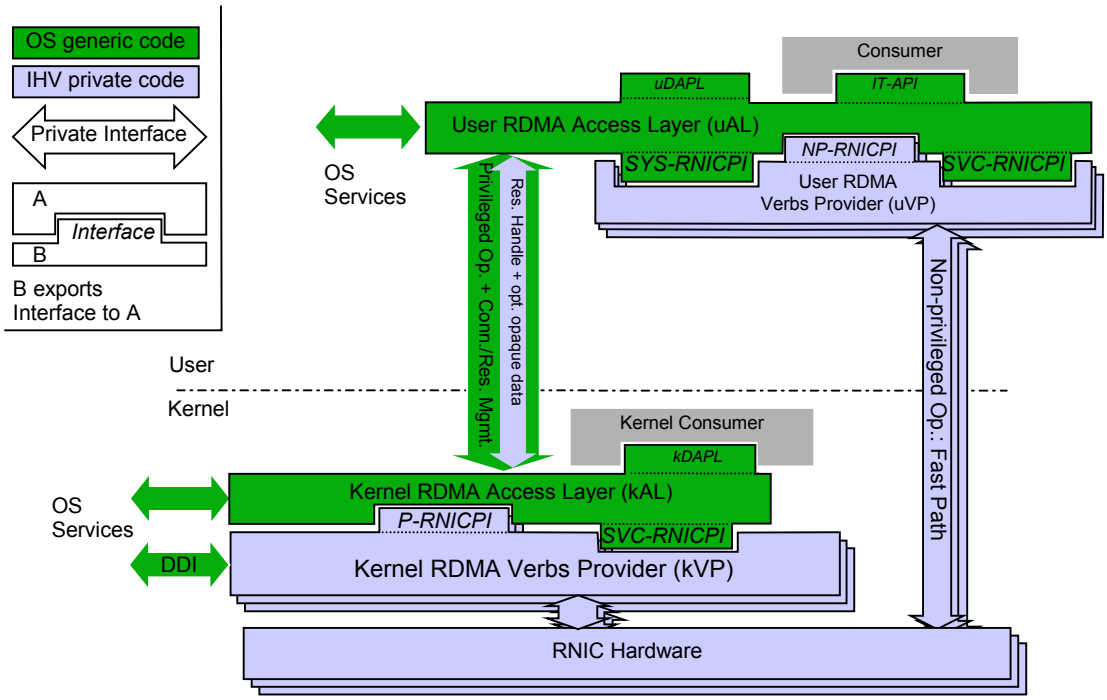

Figure 2: Typical RNICPI Host Environment

3.2 Responsibilities of a Kernel Access Layer

This document presumes the existence of a Kernel Access Layer (kAL) component as an embodiment of a generic RDMA services support infrastructure being part of the OS. RNICPI does not require that these services actually be provided by a single entity; merely its Consumers are never dependent on the structure of the implementation. The kVPs rely upon the kAL (or its equivalent components) to have performed critical security validations, and it is assumed that information encoded in the *ri_os_data_t* is understood by the service calls (*ri_svc_mem_pin*(), etc.) and that this information is sufficient to allow those services to be implemented. uVPs rely upon kAL functionality to communicate with their kVP counterparts.

The kAL (or its equivalents) is responsible for providing the following:

- A **proxy service** that allows the uVP to invoke the matching method in its partner kVP
- A **registry service** to attach kVP modules with the RDMA subsystem
- An **interface mapping service** that enables applications to correlate RNIC devices with non-RDMA Ethernet/IP interfaces; this will frequently be related to a **Connection Management** service, although RNICPI does not require there to be an explicit Connection Management service
- An **event handling service** for relaying event notifications to its Consumers
- A **memory pinning service** to offer OS-independent memory pinning

- A **cleanup service** to clean up resources allocated to a Consumer which is not gracefully terminating RDMA services usage

3.2.1 Proxy Service

The Proxy Service must enable the uVP to make authenticated calls to their matching kVP component. The uVP typically uses this service implicitly when calling the uAL-provided Syscall Service.

When non-null *vp_data* is supplied, the kVP may assume that the call was proxied from user-mode by the kAL. It is the kVP's responsibility to use normal OS methods to access the uVP's address space from the kVP's address space (such as *copy_from_user*()). The kAL MUST prevent the uVP from passing a null *vp_data* and thereby impersonating a kernel client. To fulfill this requirement in all circumstances it is up to the verbs provider implementation to assign a default value for *vp_data*, if no actual information would have to be transferred between uVP and kVP.

All input parameters that are *rnic* handles MUST be validated to ensure that they are valid handles previously delivered to the uVP instance. Any *rnic* handle issued to another uVP instance MUST NOT be forwarded to the kVP. The kAL SHOULD act as a proxy for the kVP and return the appropriate Invalid Handle error.

The kAL MAY include or invoke a Resource Manager to track and approve usage of RDMA resources in a device-independent manner. For example, the kAL may limit the number of MRs any single uVP instance is allowed to create, or it may simply desire to log MR creation in a device-independent manner. The kAL MAY reject resource allocation requests with the appropriate error without forwarding them to the kVP.

The kAL MAY modify the *os_data* struct as required to support the pinning service. For example, it could record the process ID of the caller in an *os_data* field, or otherwise reference the caller's virtual memory space.

The kAL MUST NOT forward any privileged calls on behalf of any client that does not have full authority over all referenced resources. For example, the kAL MAY translate a user virtual memory map to a page list on its own, but MUST NOT forward that page list without validating that the caller has the requested access rights to each of the pages listed.

The kAL MUST validate that the Consumer has the appropriate credentials to use the requested RDMA service. For example, on some systems a Consumer with superuser privileges may be allowed to register physical memory, while a Consumer with normal user privileges may be not allowed to do so.

The kAL MUST validate that the Consumer has write privileges to any portion of its virtual memory space that it requests be writable even when passing the request through to the kVP via *ri_mr_reg*().

Note that in almost all cases the RI_RET_NO_PERMISSION error code is actually generated by the kAL. The kVP is *never* obligated to check OS permissions of the caller. The kVP SHOULD assume that all such checking has already been done by the kAL.

3.2.2 Registry Service

The kAL MUST accept *ri_svc_attach*() and *ri_svc_detach*() calls from the kVP.

The kAL MUST communicate with the uAL the set of loaded or loadable kVPs. The kAL MAY provide a service where an application can request that the loadable library or module for a specific RNIC device be loaded.

3.2.3 Interface Mapping Service

The kAL MUST be able to correlate any Ethernet or IP layer interface to at most one RNIC device that provides RDMA service for connections routed through that Ethernet/IP interface.

The kAL MUST be able to enumerate one or more Ethernet or IP layer interfaces for each RNIC device registered.

The kAL SHOULD be able to specify to an application which RNIC device the application should open if it intends to provide RDMA services with a given IP address.

The kAL SHOULD be able to specify to an application which RNIC device(s) the application could open if it intends to establish an RDMA connection with a given remote IP Address. This query MAY be qualified with a requested “class of service”.

3.2.4 Event Handling Service

The kAL MUST support the mapping of event notification mechanisms available in the Consumer’s execution environment onto kVP-provided event callback functionality. This includes both the registration of an event notification mechanism and the passing of notification during event processing. The actual implementation of the event notification passing mechanism may include unblocking a waiting worker thread of the uAL, updating a registered file descriptor, or using signaling mechanisms and is thus implementation-dependent. A more detailed discussion of Event Handling can be found in Section 4.7.

3.2.5 Pinning Service

The kAL MUST implement *ri_svc_mem_pin*() and *ri_svc_mem_unpin*().

The kAL MAY use fields in *os_data* to assist in identification of the client’s virtual address space.

The kAL MUST NOT allow the pinning services to be used in any fashion that circumvents Operating System control over memory access.

3.2.6 Cleanup Service

The kAL MUST maintain an active connection with each uAL instance. In the event that the user process is terminated, or the uAL unloaded, the kAL must promptly close all resources opened on behalf of that uAL.

The Cleanup Service MUST be prompt enough to guarantee that no OS managed resource will be assigned to a different process before the RNIC resource referencing it has been properly

953 closed. For example, all MRs MUST be deregistered before the referenced pages might be
954 assigned to a different process.

955 3.3 Responsibilities of a User-Level Access Layer

956 Currently, RNICPI does not define any normative responsibilities for the User-Level Access
957 Layer.

4 Global Behavior of the RNICPI

This section describes certain general aspects of the behavior of the RNICPI.

4.1 Synchronous versus Asynchronous Operations

The RNICPI distinguishes between synchronous and asynchronous operations. A synchronous operation comprises a single call that may return success of the operation or reason of failure immediately.

An asynchronous operation is comprised of an asynchronous request call to initiate the operation and a notification mechanism. The return value of the request call indicates only if the request was properly placed or placing of the request failed. In case of failure, the return value indicates the reason for the failure. The ULP can be notified subsequently through an Event whether the operation completed successfully or in error. During Event Notification, the ULP will typically reap the completion status of the request via the *ri_cq_poll*() interface. The calls for posting Work Requests to a Working Queue form one particularly important set of asynchronous request calls.

4.2 Interrupt Safety

Fast path calls such as posting Work Requests to a Queue Pair (e.g., *ri_qp_postsq*()), or reaping Completion Notifications (e.g., *ri_cq_poll*()) are expected to be safe to call in an interrupt context. Some calls such as *ri_qp_modify*() may not be safe for interrupt context as they could block the caller's execution waiting for a locally or remotely generated event or condition. A call is assumed to be *not* safe for interrupt context unless individual reference pages say otherwise. Usage based on specific VP implementations that extend support to more calls in interrupt context outside of RNICPI is inherently non-portable.

4.3 Thread Safety

Thread safety refers to the mechanisms provided by the RNICPI in order to ensure mutually exclusive access to certain resources shared by multiple threads executing in parallel. The RNICPI refers to those resources that require mutually exclusive access as *critical objects*.

RNICPI formally defines the thread safety of a function call as follows. A function call is said to be thread-safe if the result of its execution is as defined by this document even when invoked in a multi-threaded environment. Likewise, a function call is said to not be thread-safe if the result of its execution can possibly be not well-defined if the call is invoked in a multi-threaded environment.

For instance, this definition of thread-safe function calls allows simultaneous execution of the same function call so long they access different critical objects (since in this case the result is always well-defined).

If multiple callers try to access a critical object, then the result is as if all the calls were serialized in arbitrary order. RNICPI will not guarantee any specific order, so a well-behaved application should only invoke multiple calls on critical objects if such calls are commutative (i.e., the result of such calls is invariant with respect to the order of their execution).

4.3.1 Supported Thread Safety Models

Ensuring mutual exclusion may affect the performance and so is only desirable when needed. Furthermore, different implementation strategies (such as thread-affinity to single objects), different locking methods (semaphores, spinlocks, blocking waits), and execution environments (single processor, SMP) may prefer to serialize access to critical objects at different points within the code path, mainly before or after crossing the RNICPI. The RNICPI thread safety model tries to take into account this potential heterogeneity by providing two locking strategies:

- Verbs Consumer controlled serialization of critical object access: The verbs provider will not implement any thread safety mechanisms. The Access Layer and/or the Consumer are responsible for serializing those calls accessing critical objects. If the Access Layer is responsible for locking, then it may hold a spinlock across non-allocating RNICPI calls (i.e., all fast path operations and many modify operations). If invoked while spinlocked, the IHV routine MUST NOT do anything that would cause a context switch or block indefinitely.

- Verbs provider controlled serialization of critical object access: Only the verbs provider will implement thread safety mechanisms. The Access Layer and Consumer are expected not to apply any locking mechanisms which might interfere with verbs provider locking. In particular the Access Layer MUST NOT be holding a spinlock when it invokes any RNICPI routine.

Verbs Consumer controlled serialization is defined as the default behavior. Verbs provider controlled serialization is defined as optionally available. The verbs Consumer may choose from available behaviors during registering with the RNIC. See the description of the *ri_svc_attach*() and *ri_svc_reg_lib* calls for more details on how to choose the desired behavior.

4.4 Error Handling

The RNICPI defines three different mechanisms to provide to Consumers' Error Notification: as immediate error return values to interface calls, as completion status returned for an earlier request, and as asynchronous Events that are either associated with a particular object (affiliated errors) or not (non-affiliated errors). Immediate Error Notification is available for all calls of the RNICPI. Completion errors are polled to retrieve the results of previously posted Work Requests on the fast data path of the P-RNICPI and NP-RNICPI. Asynchronous errors are delivered by upcall to privileged Consumers in kernel mode. How to relay this information to user space Consumers is outside the scope of RNICPI.

Immediate Error Notification generally happens when a call argument is invalid or incompatible with a condition relevant to the request. However, some errors of this type may be determined by the RNIC Engine below the RNICPI, and in this case the Consumer is notified of call parameter-related errors through an Event.

Where there is a choice between immediate or completion error types to signal an error condition for a posting function, the kVP implementation chooses which single error type is used to report the error condition. The uVP for the same RNIC may make a different choice than the kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as noted in Section 4.5.2.

Since the RNICPI is providing access to multiple RDMA Transports, some error values are defined only in the context of a specific RDMA Transport. See the reference pages for defined return values.

4.5 Object Handle Management

An Object Handle is an abstract representation of an RNICPI resource. It gets provided by the creator of this resource, typically an IHV component, during the resource creation call. The Consumer of the resource uses this handle to reference the resource in subsequent calls, but is typically not interpreting its value.

4.5.1 Scope of Object Handles

The scope of an RNIC handle is bound to the address domain where it was created. All resource handles created under an RNIC handle are only valid in the same domain and its lifetime is limited to the lifetime of the corresponding RNIC handle.

It further gets defined that a child process is not authorized to use handles granted to the parent. Any unauthorized attempt MUST NOT interfere with the authorized holder's activities. The implementation should treat such an unauthorized activity like any other use of an invalid handle by an unauthorized process.

4.5.2 Validation of User-Level Object Handles

Resource handles are not expected to be strongly validated by the Verbs Provider code. Therefore, it is expected that the OS infrastructure (uAL/kAL) will ensure handle validity before passing any user-provided handle to the kVP. However, handles passed to the uVP only (as in post operations) may not be validated by the OS infrastructure. In this case, the uVP is allowed to terminate the application or incur an address fault if invalid handles are passed to it.

4.6 Opaque RNICPI Call Arguments

The RNICPI introduces the concept of Opaque Call Arguments. Present on some interface calls, an Opaque Call Argument passes so-called Opaque Data (OD). The RNICPI does not make any assumptions on the format of the Opaque Call Argument and on the format and content of the referenced OD. OD is introduced to help in the OSV and IHV-specific synchronization of

RNICPI components and in the synchronization of RNICPI components with the OS-specific DDI.

Typically, the caller will pass a pointer to a data structure, which contains some context information not defined in the RNICPI. From the perspective of the recipient of the OD, each OD argument can be further qualified by the definition of the following three orthogonal characteristics:

1. Persistence: The RNICPI always specifies if the recipient must keep the handle referencing OD beyond the context of the actual call. Persistence is typically defined if the handle must be kept for reference in successive calls on the RNICPI or the DDI. If persistence is not explicitly defined by the RNICPI and the recipient must keep the OD information for local reference, it may have to copy the content.
2. Interpretation: The RNICPI always specifies if the recipient may interpret the received OD.
3. Passing: The RNICPI always specifies if the recipient must pass the OD to other RNICPI calls and lists those calls.

Regarding the caller of a function passing OD, the RNICPI only specifies if it is the source of the data or is just passing along the OD received before by another RNICPI interface call.

The RNICPI defines the following three different types of OD: Operating System Private Data, Verbs Provider Private Data, and Opaque Error Information. The following subsections define each of them in more detail.

4.6.1 Operating System Private Data

Operating System Private Data is always created in the OS generic part of the RDMA stack (uAL or kAL in Figure 2). Their typical usage is to synchronize OS components of the RDMA stack implementation across IHV private components. Specific to the call, this information may be transient to an intermediate IHV component (such as the uVP, which simply passes down OS Private Data received at the NP-RNICPI to the kAL via the SYS-RNICPI). The caller may have to keep this information for reference in further calls (such as the kVP which must present the data during a P-RNICPI event notification upcall), and the data may have to be interpreted by the recipient (such as the kAL, when the communication is meant to be uAL to kAL).

In the reference pages, OS private OD is represented by the *os_data* argument.

4.6.2 Verbs Provider Private Data

Verbs Provider Private Data is always created in the IHV private part of the RDMA stack (uVP or kVP in Figure 2). Their typical usage is to synchronize IHV components of the RDMA stack (uVP and kVP) implementation across OS generic components. Verbs Provider Private Data is always handled transient in the OS components and is never interpreted during passing.

A typical example for the usage of Verbs Provider Private Data is the creation of RDMA endpoint resources for a non-privileged ULP (e.g., creation of a QP): After requesting resource creation at the NP-RNICPI via *ri_qp_create*(), the uVP may establish some initial QP data structures. For completing the creation sequence, it will call for privileged support at the SYS-RNICPI. During this *ri_qp_create*() call, the uVP may want to pass down to the kVP some

context information regarding the local QP structure (to enable RNIC Engines access to Work Request structures, etc.). Carried in the Verbs Provider Private Data, the kAL will hand over this information to the kVP in the following *ri_qp_create*() P-RNICPI call without any interpretation.

In the reference pages, IHV private OD is represented by the *vp_data* argument.

4.6.3 OS/IHV Private Error Information

The RNCIPI defines an additional OUT parameter to each interface call. This argument optionally transfers back context-specific error information in case of call failure. This extends the standard synchronous failure reporting mechanism (via the calls return value) by an OSV/IHV private method. This additional error reporting mechanism allows for the thread-safe handback of error information, which cannot always be achieved with an environment variable (such as *errno*, which in the current version of this specification never gets set or interpreted by the RNICPI). Furthermore, it supports the association of error information with the call instance rather than the object instance (e.g., multiple errors occurred on a single object or multiple objects are erroneous). The definition of valid error information and its interpretation is completely outside the scope of RNICPI.

In the reference pages, OSV/IHV Private Error Information is represented by the *err_data* argument.

4.7 Event Handling

The IB and RDMAC verbs specification define two types of events: one type is related to previously issued Work Requests which could complete in success or error (Work Completion Events), the other type is not related to a certain Work Request, but to a QP (affiliated error) or to another RNIC resource (non-affiliated error).

4.7.1 Event Handler Registration

In a typical implementation, the kAL is responsible for all Event Handler registration with the RNIC(s) available in the system. The kAL registers the single Asynchronous Event Handler (AEH) supported per RNIC as well as one or more Completion Event Handlers (CEHs). The maximum number of CEHs supported by the RNIC can be determined through *ri_rnic_query*().

The kAL registers its Event Handlers using the *ri_cq_set_event_handler*() resp. *ri_async_set_event_handler*() call. A successful call to *ri_cq_set_event_handler*() will return a Completion Event Handler ID. Since only one Asynchronous Event Handler can exist, each call to *ri_async_set_event_handler*() will replace the current handler.

User-level event notification mechanisms are outside the scope of this specification. If user-level Consumer event mechanisms are used, it is envisioned that these mechanisms would have the uAL/kAL act as a proxy for the RNIC. The kAL/uAL will keep state to establish the appropriate mapping between the user registered mechanism and kVP registered Event Handler and related resource (CQ and/or RNIC Handle).

A privileged Consumer could register Event Handler directly with the kVP. Since the number of supported Event Handlers is typically a scarce RNIC resource, it is expected that an Event Handler mapping would also apply.

4.7.2 Event Notification

Event processing typically needs the support of a privileged intermediate responsible for scheduling of the Consumer for event notification. The Event Notification path will naturally start with the event information transfer between RNIC and the kVP. The kVP will typically install an ISR which gets called after the RNIC placed event records within the event queue located in the context of the kVP. The ISR is responsible for transferring control and information to its Consumer, which in this model is always the kAL. Thus, the Event Handlers registered with the RNIC are always representing callbacks from the kVP into the kAL.

If the kAL gets control during event passing (callback), it is able to de-multiplex events to individual Consumers based on the information passed with the call:

- Asynchronous Events are described by an Event Record which contains the type of the corresponding resource (QP, CQ, RNIC, SRQ), a resource identifier (e.g., QP ID), and the Event Identifier which indicates the reason for the event. This information is sufficient to retrieve the Consumer or group of Consumers.

- A CEH is called with the RNIC Handle and the CQ Handle as arguments. The Consumer can be directly derived from this information.

For kernel-level Consumers, the kAL will trigger the execution of the Consumer's registered Event Handler directly. User-level Consumers get control via the kAL/uAL Interface.

The Consumer can request completion notification by invoking *ri_cq_req_notification*() on a CQ that is enabled to receive notifications. A user-level Consumer may wait for notifications in an implementation-dependent manner (an uAL thread sleeping on a kAL device *read*() call, a *select*() on a file descriptor, etc.). When a completion event is generated by the RNIC, the kAL's CEH is invoked, upon which the kAL wakes up the waiting thread/updates the registered file descriptor. Alternatively, the Consumer can poll for completions using *ri_cq_poll*().

5 Memory Management

5.1 Overview

To enable read and write access to Consumer's communication buffers, the Consumer must register the corresponding memory with the RNIC Engine via the RNICPI. The result of any memory registration is the provision of a set of handles by the Verbs Provider:

- A Memory Region Handle to be used in subsequent RNICPI calls
- A Local Tag (LTag) that can be used along with a Tagged Offset (TO) and a length to reference portion of the Memory Region; LTags are used to specify the local source and/or destination in Work Requests
- An optional Remote Tag (RTag) that can be used to enable remote access to the Memory Region; an RTag may also be used on some RNICs to specify the local Data Sink for an RDMA Read Work Request

All memory registration is further qualified by the requested setting of remote and or local memory access rights. For iWARP, the LTag and RTag will always be the same value, and the non-key portion may be the Memory Region handle. For IB, these will be distinct values.

A Memory Window (MW) can be bound to an MR to allow for fine grained access to a portion (window) of a Memory Region. Memory Windows are suitable for refining access permissions on a per-operation basis.

During registration of an MR, a virtually contiguous block of Consumer memory (Virtual Memory, VM) gets pinned in physical memory, translated into a Physical Buffer List (PBL), and I/O-mapped for access by the RNIC Engine. Binding an MW to an already existent MR does not involve memory pinning or translation.

Memory registration can be requested with a synchronously completing interface call (Register Memory; e.g., via *ri_mr_reg*()) or asynchronously via posting a Work Request onto the Send Queue (Fast Register). In case of Fast Register, a Steering Tag must be pre-registered with the RNIC. Fast Registration is available for privileged Consumers only.

The usage of a special Steering Tag (STag with value 0 for iWARP) represents the single exception to the otherwise mandatory explicit memory registration. Using the special STag, the Consumer is able to reference an already pinned and translated physical buffer in a Work Request without its previous registration. The usage of this mechanism is restricted to referencing memory for local access and it is only available to privileged (kernel) Consumers.

5.2 Extensions to the RDMAC Verbs

The [VERBS-RDMAC] document describes one generic way to register memory with the RNIC. Independent of the registration method (Register Memory or Fast Register), the interface call or Work Request always presents already pinned and translated memory as a PBL. Pinning and translation of the memory is outside the scope of [VERBS-RDMAC].

Due to the diversity of target operating systems, the RNICPI deviates from this simplistic approach and provides a set of additional interface calls to aid in the development of a standardized memory management. Main OS-specific differences for the management of communication buffers include:

- Translation between kernel physical memory representation and RNIC drivers memory addressing mode (bus or I/O addresses). On some systems, a kernel created PBL could not always be synchronously mapped into addresses reachable for the RNIC Engine. For those systems, the PBL must be created under control of the driver.
- Pinning and translation are not necessarily combined together within a single OS service.
- System support for pinning of non-privileged Consumer memory may be available only as a kernel function or only outside the kernel (e.g., via a system call).
- Privileged Consumer memory may always be pinned or may have to be explicitly pinned by the kAL.

Therefore, besides providing RDMAC semantics (*ri_mr_reg_phys*(), *ri_mr_rereg_phys*()), RNICPI is extending the memory management by the following functionalities:

- Support for the registration of virtual memory (VM) (*ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_qp_postsq*()).
- Support for the mapping of memory for RNIC access. The *ri_mem_map*() function maps VM into an RNIC accessible PBL.
- Memory pinning support from the SVC-RNICPI. The *ri_svc_mem_pin*() routine provided by the OS is pinning virtual memory ranges provided by the Verbs Provider.
- Memory pinning downcall *ri_mem_pin*() to allow for RNIC driver controlled pinning of Consumer memory.

5.3 Memory Registration Procedure

5.3.1 Usage of Opaque Call Arguments

Independent of the actual type of memory registration, all memory registration operations are accompanied by related calls provided by the P-RNICPI, SVC-RNICPI, and DDI. The opaque argument concept introduced in Section 4.6 will be used to keep and transfer Access Layer and Verbs Provider-specific context information during the resulting call sequences. See the appropriate reference pages for further details.

5.3.2 Registering Virtual Memory

During virtual memory registration, the Consumer calls the appropriate registration routine (*ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*()), and the Verbs Provider controls mapping and pinning of the memory. While the mapping is done under local responsibility of the Verbs Provider with typically some interaction via the DDI interface, the pinning is done by calling back the *ri_svc_mem_pin*() routine provided by the Access Layer. It is possible that the Verbs Provider calls the pinning service multiple times to allow for efficient mapping. This call completes synchronously and provides a handle back to the registered MR. The PBL type representation of the registered memory does not have to be provided to the Consumer.

5.3.3 Registering Physical Memory

Dependent on the OS architecture, either the Consumer pins the memory on its own responsibility (outside the RNICPI) or is using the *ri_mem_pin*() P-RNICPI downcall to the Verbs Provider. In a second step, the Consumer establishes the mapping of the pinned memory via *ri_mem_map*(). This call provides a PBL which now can be presented to the Verbs Provider via *ri_mr_reg_phys*() or *ri_mr_rereg_phys*().

5.3.4 Fast-Register Memory

Fast Registration of memory is a Post operation on the Send Queue (*ri_qp_postsq*()). It expects a PBL as input which gets created by the very same procedure as described in Section 5.3.3.

5.3.5 Usage of Special STag

With this mechanism, a privileged Consumer may reference already pinned and translated memory in a Work Request without any previous registration with the RNICPI. The memory must be pinned and mapped using the same procedure as described in Section 5.3.3.

6 Connection Management

6.1 iWARP Connection Management

An RDMA Stream is always bound to an established bi-directional LLP transport-level connection (TCP plus MPA, SCTP, or another Transport). Before conversion into RDMA Mode, an established LLP connection is represented by an operating system-specific LLP Handle which typically gets created by the operating system. After converting the LLP connection into RDMA mode, the LLP connection is owned by the RNICPI. The LLP handle may change during RDMA mode conversion.

6.1.1 Network Addresses and RNIC Selection

RNICPI assumes that an IP address associated with the RNIC device is consistent with the IP address administration of the underlying Ethernet device and host routing tables.

How an RNICPI Consumer selects a given RNIC for communication is outside the scope of RNICPI. It is however expected that the Consumer select a given RNIC by consulting host routing tables. Once a given RNIC has been selected and an RDMA endpoint established, the route associated with the underlying LLP can dynamically change at any point in time (e.g., ICMP PMTU update, ICMP destination unreachable, etc.), possibly resulting in an LLP connection loss (which in turn will generate an affiliated asynchronous LLP connection lost error on the QP). Similarly, the MAC address associated with a given source or next hop IP address may change. It is even possible for the IP addresses associated with an LLP connection to change for a certain class of LLPs (e.g., SCTP). In all cases, it is expected that the RNIC remain consistent with the host routing tables, ARP, and ND mechanisms.

The list of IP addresses associated with the IP interface configured on the underlying Ethernet device is expected to be returned through *ri_rnic_query*(). The information provided includes both the list of all physical ports associated with the RNIC and the list of IP addresses associated with each of the ports. Note that a subsequent query MAY return a different list of IP addresses, to reflect dynamic IP address additions, deletions, and updates.

6.1.2 Packet Filtering

Any packet filtering rules that are associated with the IP address/LLP of a given RDMA endpoint MAY be applied only during the setup of the RDMA endpoint and will typically not remain effective if the endpoint entered RDMA mode. The RNICPI Consumer MUST NOT assume that all sophisticated filtering rules found in existing tools – such as *ipfilter*, *iptables*, etc. – can be applied on RDMA endpoints. Further, how the packet filtering rules are associated with a given RDMA endpoint is outside the scope of RNICPI.

6.1.3 LLP Connection Management before RDMA Mode Transition

LLP Stream Management before RDMA Mode Transition is completely outside the scope of RNICPI. The RNICPI does not interfere with the end systems establishment of the LLP stream. With the one exception of optional MPA signaling as last streaming mode message (see [RDMAC-VERBS]) during RDMA mode conversion, it does not support any data exchange over the established LLP Connection. Explicitly, the RNICPI does not support access to any TOE mode services, which might be available on an already offloaded LLP connection.

6.1.4 LLP Connection Conversion

The only reference of RNICPI to the LLP is the LLP handle provided by the ULP during RDMA mode conversion of the connection via the *ri_qp_modify*() interface. When provided by the ULP, it is assumed that it references an established transport connection in streaming mode. On many systems, this handle will equal with a socket file descriptor, referencing a fully established bi-directional transport connection. Both a non-privileged and a privileged ULP may provide an LLP handle to the RNICPI.

As already introduced in Section 1.2.3 and 1.2.4, the RNICPI supports both RDMAC-style (negotiation of RDMA parameters at ULP level before conversion) and IETF-style RDMA mode initialization (negotiation of RDMA parameters during initial MPA request/reply exchange sequence). In case of the IETF style, the ULP may send the MPA request/reply header in streaming mode before requesting the conversion or as an argument to the *ri_qp_modify*() call itself. Typically, the MPA request frame will get issued in streaming mode and the reply frame as an argument of *ri_qp_modify*().

Since the RNICPI treats an LLP connection as completely opaque, it may also support the RDMA mode conversion of an already offloaded LLP connection. The only requirement is that this connection is completely referenced by the LLP handle and the handle is valid in the context of the kVP.

6.1.5 LLP Connection Management after RDMA Mode Transition

After RDMA mode conversion, the LLP handle may change. This reflects the situation that the ownership of the LLP got transferred from the ULP/socket layer to the RDMA service provider. In fact, the DDP/RDMAP protocols became the new ULP of the LLP. The Consumer may query the new value of the handle by issuing an *ri_qp_query*().

It is understood that some ULPs may want to control the behavior of the LLP connection even after transition into RDMA mode. One example is the control of a transport protocols keepalive feature by the ULP. While this option is directly available at the LLP API if still in streaming mode (e.g., setting the SO_KEEPALIVE option on a TCP socket), after RDMA mode transition the result of direct operations on the potentially changed LLP handler is undefined. The RNICPI defines optional support to control the keepalive state of the LLP via *ri_qp_query*() and *ri_qp_modify*(). No other direct LLP control gets defined in the RNICPI. The Consumer may query keepalive support of the RNIC via the *ri_rnic_query*() call.

6.1.6 RDMA Association Termination

The RNICPI supports all five connection termination modes as defined in [VERBS-RDMAC] (normal close, ULP initiated termination, ULP initiated abortive teardown, remote termination, local termination, and local or remote abortive teardown).

6.2 Non-TCP Connection Management

RNICPI Connection Management interfaces were designed to transition an MPA/TCP connection into RDMA mode. However, the definition of the LLP Handle passed in is flexible enough to accommodate other Connection Management requirements.

For SCTP, the LLP Handle would not reference a TCP connection, but rather an established SCTP association and the Stream ID to be used for this QP.

For IB, the concept of LLP Handle does not apply.

What all transports share is that the LLP Handle supplies all data required to initiate an RDMA layer connection with a remote QP. For MPA/TCP, that is typically a socket handle. RNICPI does not define this type to be a socket handle, however, because that would interfere with using the same API for SCTP, IB, and other RDMA transports.

7 API Reference Pages

ri_rnic_open()

NAME

ri_rnic_open – open an RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_rnic_open(
    IN      ri_vp_handle_t  vp_hdl,
    IN      ri_pbl_mode_t   pbl_mode,
    IN OUT  ri_os_data_t    os_data,
    OUT     ri_err_data_t  *err_data
);

typedef enum ri_pbl_mode ri_pbl_mode_t;
enum ri_pbl_mode {
    RI_PHYS_PAGE_LIST_MODE,
    RI_PHYS_BLOCK_LIST_MODE
};
```

DESCRIPTION

vp_hdl Verb provider handle to the RNIC as passed by the verb provider in the *ri_svc_attach*() call.

pbl_mode Desired physical buffer list mode.

os_data Opaque data owned by OS modules. IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged via "syscall". Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_rnic_open*() function allows the caller to open a given RNIC in page or block mode. The *ri_rnic_open*() operation is expected to prepare the RNIC for regular use by applications. If the RNIC does not support the desired physical buffer list mode, an error is returned. Note that at least "page" mode must always be supported by the hardware.

This call can be invoked only in privileged mode in the kernel. While technically there is a user-level entry point and a "syscall" version for this call, they are never expected to be called. The user-level version of this call is simply a stub which returns the RI_RET_NO_PERMISSION error.

Before the RNIC can be opened, verb providers are expected to call *ri_svc_attach*() to attach the RNIC to the RDMA access layer. Once the RNIC is successfully opened using *ri_rnic_open*(), it cannot be opened again until it is closed using *ri_rnic_close*(). Kernel verb providers shall be able to handle this call anytime after *ri_svc_attach*() is called including from within *ri_svc_attach*().

The definitions for *ri_vp_handle_t, ri_os_data_t,* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_VP_INVALID_HANDLE	Verbs provider handle passed is invalid.
RI_RET_INVALID_PARAMETER	*pbl_mode* parameter contains an undefined value.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_INVALID_PBL_MODE	Desired physical buffer list mode is not supported.
RI_RET_RNIC_INUSE	*ri_rnic_open*() has already been called for this RNIC.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_close*(), *ri_rnic_free_hdl*(), *ri_rnic_get_hdl*(), *ri_svc_attach*(), *ri_svc_detach*()

ri_rnic_close()

NAME

ri_rnic_close – close an RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_rnic_close(
    IN   ri_vp_handle_t  vp_hdl,
    OUT  ri_err_data_t  *err_data
);
```

DESCRIPTION

vp_hdl Verb provider handle of the RNIC to close. This handle is the same one passed in the *ri_rnic_open*() function.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API allows the caller to close a given RNIC. This call is responsible for deallocating all resources that were allocated by *ri_rnic_open*(). The verb provider handle specified is the same one that was used as an input parameter for the *ri_rnic_open*() call. After this call completes, there will be no upcalls (i.e., asyncs) in progress for this RNIC. Note that an RNIC must be closed before an *ri_svc_detach*() call can occur.

This call can be invoked only in privileged mode in the kernel. While technically there is a user-level entry point and a "syscall" version for this call, they are never expected to be called. The user-level version of this call is simply a stub which returns the RI_RET_NO_PERMISSION error.

If the verb provider handle has already been closed (via *ri_rnic_close*()) or if the specified verb provider handle does not match one that was passed to the *ri_rnic_open*() call, this call will fail with a return value of RI_RET_VP_INVALID_HANDLE. Consumers are expected to have freed all RNIC handles via *ri_rnic_free_hdl*() calls before invoking this entry point. Otherwise, RI_RET_RNIC_INUSE is returned.

The definitions for *ri_vp_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The specified RNIC is successfully closed.
RI_RET_VP_INVALID_HANDLE	The verb provider handle passed is invalid.
RI_RET_RNIC_INUSE	RNIC handles are still associated with the RNIC.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_free_hdl*(), *ri_rnic_get_hdl*(), *ri_rnic_open*(), *ri_svc_attach*(), *ri_svc_detach*()

ri_async_set_event_handler()

NAME

ri_async_set_event_handler – register an event handler for asynchronous events from the specified RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_async_set_event_handler (
    IN   ri_vp_handle_t             vp_hdl,
    IN   ri_async_event_callback_t  async_event_callback,
    OUT  ri_err_data_t             *err_data
);

typedef struct ri_ib_port_event ri_ib_port_event_t;
struct ri_ib_port_event {
/* IB verbs do not specify that port number has to be returned in a
port event. Port number 0 indicates that there is no knowledge of which
port this event is associated with. */
    uint8_t             port_number;
    ri_ib_port_state_t  port_state;
};

typedef union ri_event_data ri_event_data_t;
union ri_event_data {
    ri_ib_port_event_t  port_event;
    ri_os_data_t        resource_os_data;
};

typedef struct ri_event_record ri_event_record_t;
struct ri_event_record {
    ri_async_event_type_t  event_type;
    ri_event_data_t        event_data;
    ri_err_data_t          err_data;
};

typedef void (*ri_async_event_callback_t)(
    IN OUT  ri_os_data_t        vp_hdl_os_data,
    IN      ri_event_record_t  *event
);

typedef enum ri_async_event_type ri_async_event_type_t;
enum ri_async_event_type {

    /* iWARP-only events */
    RI_EVENT_QP_LLP_CLOSE_COMPLETE,
    RI_EVENT_QP_TERMINATE_MESSAGE_RECEIVED,
    RI_EVENT_QP_LLP_CONNECTION_RESET,
    RI_EVENT_QP_LLP_CONNECTION_LOST,
    RI_EVENT_QP_LLP_INTEGRITY_ERROR_SIZE,
    RI_EVENT_QP_LLP_INTEGRITY_ERROR_CRC,
    RI_EVENT_QP_LLP_INTEGRITY_ERROR_BAD_FPDU,
    RI_EVENT_QP_REMOTE_OP_ERROR_DDP_VERSION,
    RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_VERSION,
```

```
1494                RI_EVENT_QP_REMOTE_OP_ERROR_OPCODE,
1495                RI_EVENT_QP_REMOTE_OP_ERROR_DDP_QN,
1496                RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_RD_DISABLED,
1497                RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_WR_RD_RESP_DISABLED,
1498                RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_RD_INVALID,
1499                RI_EVENT_QP_REMOTE_OP_ERROR_NO_L_BIT,
1500                RI_EVENT_QP_PROTECTION_ERROR_STAG_QP_MISMATCH,
1501                RI_EVENT_QP_PROTECTION_ERROR_BOUNDS_VIOLATION,
1502                RI_EVENT_QP_PROTECTION_ERROR_ACCESS_VIOLATION,
1503                RI_EVENT_QP_PROTECTION_ERROR_INVALID_PD
1504                RI_EVENT_QP_PROTECTION_ERROR_WRAP_ERROR,
1505                RI_EVENT_QP_BAD_CLOSE,
1506                RI_EVENT_QP_BAD_LLP_CLOSE,
1507                RI_EVENT_QP_RQ_PROTECTION_ERROR_INVALID_MSN,
1508                RI_EVENT_QP_RQ_PROTECTION_ERROR_MSN_GAP,
1509                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_TOO_MANY_RDMA_RDs,
1510                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_MSN_GAP,
1511                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_MSN,
1512                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_STAG,
1513                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_BOUNDS_VIOLATION,
1514                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_ACCESS_VIOLATION,
1515                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_PD,
1516                RI_EVENT_QP_IRRQ_PROTECTION_ERROR_WRAP_ERROR,
1517                RI_EVENT_QP_CQ_SQ_ERROR,
1518                RI_EVENT_QP_CQ_RQ_ERROR,
1519                RI_EVENT_CQ_OVERFLOW_DETECTED, /*optional for iWARP */
1520                RI_EVENT_CQ_OP_ERROR,
1521                RI_EVENT_QP_SRQ_ERROR,
1522                RI_EVENT_QP_LOCAL_CATASTROPHIC_ERROR,
1523                RI_EVENT_QP_RQ_LIMIT_REACHED,
1524
1525                /* common events */
1526                RI_EVENT_SRQ_LIMIT_REACHED,
1527                RI_EVENT_SRQ_CATASTROPHIC_ERROR,
1528                RI_EVENT_LOCAL_CATASTROPHIC_ERROR,
1529                RI_EVENT_RNIC_LOCAL_CATASTROPHIC_ERROR =
1530                RI_EVENT_LOCAL_CATASTROPHIC_ERROR,
1531                RI_EVENT_QP_LAST_WQE_REACHED, /* optional for iWARP */
1532
1533                /* IB-specific events */
1534                RI_EVENT_QP_PATH_MIGRATED,
1535                RI_EVENT_QP_COMMUNICATION_ESTABLISHED,
1536                RI_EVENT_QP_SEND_QUEUE_DRAINED,
1537                RI_EVENT_CQ_ERROR,
1538                RI_EVENT_QP_LOCAL_WQ_CATASTROPHIC_ERROR,
1539                RI_EVENT_QP_INVALID_REQ_LOCAL_WQ_ERROR,
1540                RI_EVENT_QP_LOCAL_ACCESS_VIOLATION_WQ_ERROR,
1541                RI_EVENT_QP_PATH_MIG_REQ_ERROR,
1542                RI_EVENT_PORT_EVENT,
1543                RI_EVENT_CLIENT_REG_EVENT
1544          };
```

DESCRIPTION

vp_hdl Handle to the RNIC to get events from. This must be same as the handle used in *ri_rnic_open*().

async_event_callback
A handler that is called when there is an asynchronous event from the RNIC. See *ri_async_event_callback_t* for arguments passed back in the callback.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_async_set_event_handler() allows privileged Consumers to register an asynchronous event handler for the specified RNIC. Only one asynchronous event handler can be registered per RNIC. Subsequent calls to *ri_async_set_event_handler*() with a given RNIC handle will overwrite the asynchronous event handler routine to be called. If *async_event_callback* is NULL, no callback function is invoked upon any asynchronous events or errors. Any asynchronous event that occurs prior to the registration of the event handler will be lost.

The *os_data* passed by the Consumer at the time of *ri_rnic_open*() is provided as the *vp_hdl_os_data* parameter to the callback function. The event record *event* contains the appropriate event type and additional information on the resource affiliated to the event. *resource_os_data* is set to the *os_data* of the resource set by the Consumer at the time of resource creation. Any fatal error detected in the RNIC not specifically related to a QP, CQ, SRQ, or port results in an RI_EVENT_RNIC_LOCAL_CATASTROPHIC_ERROR event and *resource_os_data* is set to the *os_data* passed by the Consumer at the time of *ri_rnic_open*(). If the *event_type* field in the event record indicates that it is a CQ-related event (i.e., events with RI_EVENT_CQ prefix), *resource_os_data* is the *os_data* of the CQ set by the Consumer during *ri_cq_create*(). If the event type indicates it is an SRQ-related event (i.e., events with the RI_EVENT_SRQ prefix), *resource_os_data* is the *os_data* of the SRQ set by the Consumer during *ri_srq_create*(). For events related to QP (i.e., events with RI_EVENT_QP prefix), *resource_os_data* is the *os_data* set by the Consumer at the time of *ri_qp_create*().

IB devices generate an unaffiliated RI_EVENT_PORT_EVENT event if the link state of a port changes. *port_state* indicates if the link is active or has become unavailable. A *port_number* of 0 indicates that no additional port-specific information is provided in the event.

Most of the errors are transport-specific. See the related verb specifications for more details on the cause of the events or errors.

ri_async_set_event_handler() can be invoked only in the kernel. The user mode entry point of this call is simply a stub that returns RI_RET_NO_PERMISSION.

RETURN VALUE

RI_RET_SUCCESS	The specified asynchronous RNIC event handler has been successfully registered.
RI_RET_VP_INVALID_HANDLE	The verb provider handle passed is invalid.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_set_event_handler*(), *ri_qp_create*() *ri_rnic_open*(), *ri_srq_create*()

ri_rnic_get_hdl()

NAME

ri_rnic_get_hdl – obtain an RNIC handle usable by applications

SYNOPSIS

```
ri_api_return_t ri_rnic_get_hdl(
    IN      ri_vp_handle_t        vp_hdl,
    IN      ri_consumer_domain_t  location,
    OUT     ri_rnic_handle_t     *rnic,
    IN OUT  ri_vp_data_t          vp_data,
    IN OUT  ri_os_data_t          os_data,
    OUT     ri_err_data_t        *err_data
);

typedef struct ri_consumer_domain ri_consumer_domain_t;
struct ri_consumer_domain {
    RI_CONSUMER_LOCATION_KERNEL,
    RI_CONSUMER_LOCATION_USER
};
```

DESCRIPTION

vp_hdl Verb provider handle to the RNIC originally passed by the verb provider in *ri_svc_attach*().

location Consumer's domain (user/kernel). This argument is ignored in the user-level version of this call (the location is inherently user-level).

rnic Returned handle to the RNIC for regular operations.

vp_data Opaque data owned by IHV modules, OS modules should not try to interpret it. This information gets communicated from uVP through a "syscall" which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules, IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged via "syscall". Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This function allows the caller to create a handle to the RNIC indicated by the verb provider handle. The RNIC handle returned is specified as an input parameter to most subsequent API calls (there are a few exceptions such as *ri_rnic_close*()). Further, the returned RNIC handle is used to set the context of these subsequent operations. This call may be made more than once to get different handles for the same RNIC.

The verb provider handle specified as the input parameter to this call must be opened with *ri_rnic_open*() prior to invoking *ri_rnic_get_hdl*(). If it is called on an RNIC that is not currently open, RI_RET_RNIC_NOT_OPEN is returned.

The definitions for *ri_rnic_handle_t*, *ri_vp_handle_t*, *ri_vp_data_t*, *ri_os_data_t*, and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid RNIC handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_VP_INVALID_HANDLE	The verb provider handle passed is invalid.
RI_RET_INVALID_PARAMETER	*location* has an undefined value or *rnic* is NULL.
RI_RET_RNIC_NOT_OPEN	RNIC is not open.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

How the uAL finds out about available RNICs to call *ri_rnic_get_hdl*() is outside of the scope of RNICPI. However, typically it is done through private communication (e.g., syscalls) between uAL and kAL.

The *location* parameter specified will be used by the implementation to make choices about type of memory copy, kind of protection, need to do memory pinning, etc. One common usage of this function is to create a new RNIC handle per user process (i.e., different address space). Some implementations may be able to share resources between objects created through the same RNIC handle.

The number of handles that can be supported by this function is implementation-dependent. On the other hand, if an implementation does not really support the concept of different RNIC handles, it might simply return the same handle to all.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_close*(), *ri_rnic_free_hdl*(), *ri_rnic_open*(), *ri_svc_attach*(), *ri_svc_detach*()

ri_rnic_query()

NAME

ri_rnic_query – return the attributes of an RNIC

SYNOPSIS

```
ri_api_return_t ri_rnic_query(
    IN      ri_rnic_handle_t       rnic,
    IN      ri_rnic_query_flags_t  flags,
    OUT     ri_rnic_attr_t        *attrs,
    IN OUT  uint_t                *attr_size,
    OUT     ri_err_data_t         *err_data
);

typedef enum ri_rnic_query_flags ri_rnic_query_flags_t;
enum ri_rnic_query_flags {
    RI_QUERY_RNIC_PORT_INFO = 1 << 0
};
```

DESCRIPTION

rnic Handle to the RNIC to query obtained from *ri_rnic_get_hdl*().

flags Flag to indicate whether port information is also desired.

attrs Consumer allocated buffer for attributes of the RNIC.

attr_size The input value is the size of the buffer. The output value is the actual size of the requested attributes.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This function returns the attributes of the specified RNIC in *attrs*. The format of the attribute information is described in the *ri_rnic_attr_t* reference page and the related reference pages for each transport-specific attribute. The *flags* argument specifies whether the information queried will also include information about the ports of the RNIC. Undefined flags cause the error RI_RET_INVALID_PARAMETER to be returned. The size of the *attrs* buffer is specified in *attr_size*. The *attr_size* parameter should be large enough to contain all requested information, including the main *ri_rnic_attr_t* struct, any transport-specific struct, and – if port information has been requested – the associated port-related structs (which may include lists of address information). When the function returns, *attr_size* contains the size of the queried data.

If *attr_size* is not large enough to fit the requested attributes, then the error RI_RET_BUFFER_TOO_SMALL is returned and no attributes are returned. Note that *attr_size* is still updated in that case to provide the correct size. An initial size for attributes including port data is returned in the *query_size* field of *ri_svc_attach*(). Note that some transports allow the size of the attributes to vary dynamically, so *attr_size* may only indicate the size of the attributes at the moment that *ri_svc_attach*() was called.

The definitions of *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. *attrs* contains the RNIC attributes.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	Undefined *flags* parameter or *attrs* and/or *attr_size* parameter is NULL.
RI_RET_BUFFER_TOO_SMALL	Input *attr_size* was not large enough to contain the requested attributes.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_modify(), *ri_rnic_attr_t*

ri_rnic_modify()

NAME

ri_rnic_modify – modify the attributes of an RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_rnic_modify(
    IN   ri_rnic_handle_t      rnic,
    IN   ri_rnic_attr_mask_t   attr_mask,
    IN   ri_rnic_modify_req_t *attrs,
    OUT  ri_err_data_t        *err_data
);

typedef enum ri_rnic_attr_mask ri_rnic_attr_mask_t;
enum ri_rnic_attr_mask {
    RI_IB_RNIC_ATTR_SI_GUID = 1 << 0
};

typedef enum ri_port_attr_mask ri_port_attr_mask_t;
enum ri_port_attr_mask{
    RI_IB_PORT_ATTR_SHUTDOWN = 1 << 0,
    RI_IB_PORT_ATTR_INIT_TYPE = 1 << 1,
    RI_IB_PORT_ATTR_QKEY_RESET = 1 << 2,
    RI_IB_PORT_ATTR_CAP_SET = 1 << 3,  /* set FLAGs below */
    RI_IB_PORT_ATTR_CAP_FLAG_SM = 1 << 4,
    RI_IB_PORT_ATTR_CAP_FLAG_SNMP = 1 << 5,
    RI_IB_PORT_ATTR_CAP_FLAG_DEVMGT = 1 << 6,
    RI_IB_PORT_ATTR_CAP_FLAG_VENDOR = 1 << 7
};

#define RI_IB_ALL_PORTS (0)

typedef struct ri_port_modify_req ri_port_modify_req_t;
struct ri_port_modify_req {
    ri_port_attr_mask_t  port_attr_mask;
    uint8_t              port_number;
    uint8_t              port_init_type;  /* 4 bits */
};

typedef struct ri_rnic_modify_req ri_rnic_modify_req_t;
struct ri_rnic_modify_req {
    uint64_t               si_guid;
    ri_port_modify_req_t  *port_attrs;
    uint8_t                port_list_size;
};
```

DESCRIPTION

rnic RNIC handle of the RNIC to modify.

attr_mask Mask to select attributes to be modified.

attrs Attributes structure which contains the requested values for the attributes corresponding to the mask bits chosen.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This function allows the caller to modify the attributes of the RNIC associated with the specified handle. The attributes to be modified are selected using *attr_mask* which the caller can construct by inclusive OR-ing the required values of *ri_rnic_attr_mask_t*. Any bits outside the definition of *ri_rnic_attr_mask_t* will cause the error RI_RET_INVALID_PARAMETER to be returned. Upon successful completion of the call, a subsequent successful *ri_rnic_query*() call will see the new values reflected in the attributes returned. Because relatively few attributes of an RNIC can be modified, this function uses a different version of the attribute structure which is not shared by *ri_rnic_query*().

This call applies only to privileged callers. Callers without sufficient privilege will return the RI_RET_NO_PERMISSION error.

Currently, only the InfiniBand transport supports modifying RNIC attributes. Calling this function on RNICs using other transports will return RI_RET_XPORT_UNSUPPORTED. The *attrs* argument contains the requested changes to be made to the RNIC. If the RI_IB_RNIC_ATTR_SI_GUID flag is set, the new system image GUID is specified in the *si_guid* field of the *attrs* argument. The *port_attrs* field is a pointer to an array of port modification requests. The length of the array is specified in the *port_list_size* field. If *port_list_size* is zero, no port attributes will be changed and the *port_attrs* field is ignored.

Each *ri_port_modify_req_t* structure contains a *port_attr_mask* field which specifies the port attributes to be changed. The mask can be constructed by inclusive OR-ing the required values of *ri_port_attr_mask_t.* Any bits outside the definition of *ri_port_mask_attr_t* will cause the error RI_RET_INVALID_PARAMETER to be returned. The *port_number* field specifies the port to be modified. If RI_IB_ALL_PORTS is specified, then the change is made to all ports on the RNIC. If a non-existent port is specified, then RI_RET_INVALID_PARAMETER is returned.

If RI_IB_PORT_ATTR_CAP_SET is specified, then the RI_IB_PORT_ATTR_CAP_FLAG_ fields specified are set and those not specified are cleared. If RI_IB_PORT_ATTR_CAP_SET is not specified, then the RI_IB_PORT_ATTR_CAP_FLAG_ items will trigger the RI_RET_INVALID_PARAMETER error. If RI_IB_PORT_ATTR_INIT_TYPE is set, then the *port_init_type* field specified the new *InitType* value.

The system image GUID, port shutdown capability, *InitType*, and *Q_Key* violation counter are all optional features. If an attempt is made to use an unsupported optional item, RI_RET_OP_UNSUPPORTED is returned.

The definitions of *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	*attrs* passed is NULL or unknown flag in *attr_mask* specified.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.

RI_RET_OP_UNSUPPORTED	Specified operation not supported (e.g., setting an unsupported optional item).
RI_RET_XPORT_UNSUPPORTED	The operation was called on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_query()

ri_rnic_free_hdl()

NAME

ri_rnic_free_hdl – invalidate an application RNIC handle

SYNOPSIS

```
ri_api_return_t ri_rnic_free_hdl(
    IN   vp_handle_t       vp_hdl,
    IN   ri_rnic_handle_t  rnic,
    OUT  ri_err_data_t     *err_data
);
```

DESCRIPTION

vp_hdl Verb provider handle to the RNIC, same as the one passed in *ri_rnic_get_hdl*().

rnic RNIC handle to free, originally returned through *ri_rnic_get_hdl*().

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This function allows the caller to destroy association of the specified handle (from *ri_rnic_get_hdl*()) with its RNIC. Further calls using the specified RNIC handle may fail with a return value of RI_RET_RNIC_INVALID_HANDLE. Note that it is necessary to free all handles from *ri_rnic_get_hdl*() belonging to an RNIC, before *ri_rnic_close*() can be called on that RNIC.

The Consumer is expected to free, deallocate, or deregister all resources obtained through this RNIC handle (for example, Protection Domains, Queue Pairs, Completion Queues, Memory Registrations, etc.) before invoking this call. As each object is released through the appropriate function call, no further upcalls (i.e., CQ completion events and asyncs) will be in progress that can reference the released object. So by the time *ri_rnic_free_hdl*() completes, there should be no upcalls in progress for any of the objects that were obtained through the released RNIC handle. If *ri_rnic_free_hdl*() is called and there are outstanding RNIC resources associated with the RNIC handle specified, RI_RET_RNIC_INUSE is returned.

The definitions of *ri_vp_handle_t*, *ri_rnic_handle_t,* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The specified RNIC handle is successfully destroyed.
RI_RET_VP_INVALID_HANDLE	Verb provider handle to the RNIC is invalid.
RI_RET_RNIC_INVALID_HANDLE	RNIC handle passed is invalid.
RI_RET_RNIC_INUSE	There are outstanding resources – like Protection Domains, Queue Pairs, Memory Registrations, etc. – still associated with the RNIC handle.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_close*(), *ri_rnic_get_hdl*(), *ri_rnic_open*(), *ri_svc_attach*(), *ri_svc_detach*()

ri_rnic_diag()

NAME

ri_rnic_diag – perform privileged diagnostic operations on the RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_rnic_diag (
    IN      ri_rnic_handle_t   rnic,
    IN      ri_diag_op_type_t  op,
    IN      void              *arg,
    IN OUT  unsigned char     *opval,
    IN OUT  ri_length_t       *oplen,
    OUT     ri_err_data_t     *err_data
);
```

DESCRIPTION

rnic Handle to RNIC to perform diagnostic operation on.

op Opcode of the diagnostic operation.

arg Operation-dependent argument.

opval Pointer to location to hold kVP-specific information.

oplen Value result parameter indicating size of *opval*.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

op	*arg*	*opval*	*oplen*
RI_DIAG_RNIC_ DETAILS	Ignored	Upon successful return, *opval* contains kVP-dependent information pertaining to RNIC state, global diagnostic/error counters, etc. The kVP may also include detailed error information on any prior calls related to the RNIC. The information that is returned is ascii printable and may be logged by a diagnostic tool. If *opval* is NULL or is not big enough to hold the entire information as reflected by *oplen*, an error is returned and the correct size is returned in *oplen*.	IN: size of *opval* OUT: expected/ actual size of information returned in *opval*

op	*arg*	*opval*	*oplen*
RI_DIAG_QP_ DETAILS	QP handle	Upon successful return, *opval* contains kVP-dependent information pertaining to QP internal state, dynamic counters associated with QP, etc. The kVP can also include detailed error information on any prior calls related to the QP. The information that is returned is ascii printable and may be logged by a diagnostic tool. If *opval* is NULL or is not big enough to hold the entire information as reflected by *oplen*, an error is returned and the correct size is returned in *oplen*.	IN: size of *opval* OUT: expected/ actual size of information returned in *opval*
RI_DIAG_RNIC_ DUMP	Ignored	Upon successful return, *opval* contains kVP-dependent dump of RNIC internal memory, contexts, traces, etc. This information may be logged by a diagnostic tool. If *opval* is NULL or is not big enough to hold the entire information as reflected by *oplen*, an error is returned and the correct size is returned in *oplen*.	IN: size of *opval* OUT: expected/ actual size of information returned in *opval*
RI_DIAG_RDMA_ RESET	Ignored	Ignored. Upon successful return, the RDMA interface of the RNIC is reset. Any RNIC Consumers at the time of reset may be killed or shut down. Reset of the RDMA interface is not expected to interfere with any underlying interfaces (e.g., TOE, Ethernet) that may be exported by the RNIC.	Ignored

This programming interface can be used to perform diagnostic operations on the RNIC in a kVP-independent manner. Upon successful completion, the kVP-dependent information is returned in *opval*. The definition of this information is outside the scope of RNICPI.

ri_rnic_diag() is expected to be invoked by privileged Consumers only. The responsibility of determining whether a Consumer is privileged or not, is outside the scope of RNICPI. If the Consumer is not privileged, RI_RET_NO_PERMISSION is returned.

It is optional for the kVP to support any of these diagnostic operations. If not supported, an RI_RET_OP_UNSUPPORTED error is returned by the kVP.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_INVALID_PARAMETER	Invalid *arg*.

RI_RET_BUFFER_TOO_SMALL	The size of *opval* as specified in *oplen* is insufficient to hold the kVP-dependent information, or *opval* is NULL.
RI_RET_OP_UNSUPPORTED	The specified *opcode* is not supported by the kVP.
RI_RET_NO_PERMISSION	The caller is not privileged.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

Here is a sample usage:

```
ri_length_t       oplen;
unsigned char    *opval;
ri_rnic_handle_t  rnic;
ri_api_return_t   ret;

/* Determine the amount of memory required for the operation */
oplen = 0;
if (ri_rnic_diag(rnic, RI_DIAG_RNIC_DETAILS, NULL, NULL, &oplen)
    == RI_RET_BUFFER_TOO_SMALL)) {

    /* Allocate memory for opval */
    ret = ri_rnic_diag(rnic, RI_DIAG_RNIC_DETAILS,
        NULL, opval, &oplen);
}
```

IMPLEMENTATION NOTES

The kAL, or its equivalent, is responsible for ensuring that the caller has the required permissions to perform diagnostic operations on the RNIC.

SEE ALSO

ri_rnic_open(), *ri_rnic_get_hdl*()

ri_pd_alloc(), ri_pd_dealloc()

NAME

ri_pd_alloc – allocate a Protection Domain

ri_pd_dealloc – deallocate a Protection Domain

SYNOPSIS

```
ri_api_return_t ri_pd_alloc (
    IN      ri_rnic_handle_t  rnic,
    OUT     ri_pd_handle_t   *pd,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

ri_api_return_t ri_pd_dealloc (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_pd_handle_t    pd,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC on which to allocate/deallocate a Protection Domain.

pd Handle to the Protection Domain.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP through a “syscall” which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged via “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_pd_alloc() allocates a Protection Domain within the context of the RNIC specified by *rnic*. *ri_pd_dealloc*() deallocates a Protection Domain that was previously allocated by *ri_pd_alloc*() within the context of the RNIC specified by *rnic.*

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid handle to the allocated Protection Domain is returned.
RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.

RI_RET_PD_INUSE	The Protection Domain indicated by *pd* is still being used by some other resource.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle indicated by *rnic* is invalid.
RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_get_hdl(), *ri_rnic_free_hdl*()

ri_cq_create()

NAME

ri_cq_create – create a Completion Queue (CQ) on the specified RNIC

SYNOPSIS

```
ri_api_return_t ri_cq_create (
    IN      ri_rnic_handle_t  rnic,
    IN OUT  ri_cq_attr_t     *cq_attr,
    OUT     ri_cq_handle_t   *cq,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef uint_t ri_cq_event_handler_id_t;
typedef struct ri_cq_attr ri_cq_attr_t;
struct ri_cq_attr {
    uint32_t                  num_cqe;
    ri_cq_event_handler_id_t  handler_id;
};
```

DESCRIPTION

rnic Handle to the RNIC on which to create a CQ.

cq_attr Pointer to CQ attribute structure.

cq Handle to the newly created CQ.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_create() allows the caller to create a Completion Queue on the specified RNIC. The Consumer fills in the desired number of CQ entries and completion handler ID in *cq_attr*. Upon successful completion, the actual number of allocated CQ entries is returned in the *num_cqe* field of *cq_attr*, and a handle to the newly created CQ is returned in *cq*. The number of CQ entries returned can be greater than or equal to the one requested. If the number of CQ entries requested is greater than the maximum number of CQ entries supported by the RNIC, RI_RET_CQ_CAPACITY_EXCEEDED is returned and the CQ is not created.

A privileged RNICPI Consumer such as kAL can specify the completion handler that is associated with the CQ in the *handler_id* field of *cq_attr*. Privileged Consumers can obtain the

completion handler through the *ri_cq_set_event_handler*() call. If the *handler_id* specified in *cq_attr* is NULL, no completion handler is associated with the given CQ and the Consumer cannot enable notifications on the CQ. If an invalid *handler_id* is specified, *ri_cq_create*() fails with an RI_RET_CQ_INVALID_EVENT_HANDLER error. It is not possible to change the *handler_id* associated with the CQ once the CQ is created. However, privileged Consumers can change the completion handler that is associated with a *handler_id* through the *ri_cq_set_event_handler*() call.

Non-privileged RNICPI Consumers such as uAL cannot specify completion handlers for CQ. uAL must set *handler_id* in *cq_attr* to zero and uVP must ignore the *handler_id* field passed.

If a completion handler is registered for a given CQ event handler ID and the CQ is enabled for notification, the kVP shall call the completion handler with the *os_data* passed in during *ri_cq_create*() as the CQ handle.

RETURN VALUE

RI_RET_SUCCESS	The call completed successfully and a valid CQ handle is returned.
RI_RET_RESOURCE_SHORTAGE	Inadequate resources to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	*cq* and/or *cq_attr* is NULL.
RI_RET_CQ_CAPACITY_EXCEEDED	Number of CQ entries requested exceeds RNIC capability.
RI_RET_CQ_INVALID_EVENT_HANDLER	Invalid CQ Event handler.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_destroy(), *ri_cq_modify*(), *ri_cq_query*(), *ri_cq_set_event_handler*()

ri_cq_query()

NAME

ri_cq_query – query the attributes associated with a given CQ

SYNOPSIS

```
ri_api_return_t ri_cq_query (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_cq_handle_t    cq,
    OUT  ri_cq_attr_t     *cq_attr,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC the CQ belongs to.

cq Handle to CQ.

cq_attr Pointer to CQ attribute information.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_query() returns the CQ event handler and the number of entries in the specified CQ, upon successful completion. The number of CQ entries shall be less than or equal to the maximum number of CQ entries supported by the RNIC. Non-privileged RNICPI Consumers and privileged Consumers that did not associate a completion handler with the CQ will see NULL returned in the *handler_id* field of *cq_attr*.

If the CQ has encountered an internal error or an overrun, RI_RET_CQ_ERROR is returned. All QPs associated with this CQ will no longer function properly and will need to be shut down using *ri_qp_destroy*() or moved to idle state using *ri_qp_modify*().

RETURN VALUE

RI_RET_SUCCESS	CQ attributes are returned in *cq_attr* upon successful completion.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
RI_RET_INVALID_PARAMETER	*cq_attr* is NULL.
RI_RET_CQ_ERROR	CQ has an overrun or has become inaccessible.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_destroy*(), *ri_cq_modify*(), *ri_cq_set_event_handler*()

ri_cq_modify()

NAME

ri_cq_modify – modify some attributes of a CQ

SYNOPSIS

```
ri_api_return_t ri_cq_modify (
    IN      ri_rnic_handle_t     rnic,
    IN      ri_cq_handle_t       cq,
    IN      ri_cq_modify_mask_t  mask,
    IN OUT  ri_cq_attr_t        *cq_attr,
    OUT     ri_err_data_t       *err_data
);

typedef enum ri_cq_modify_mask ri_cq_modify_mask_t;
enum ri_cq_modify_mask {
    RI_CQ_MODIFY_MASK_NUM_CQE = 0x1
};
```

DESCRIPTION

rnic	Handle to the RNIC the CQ belongs to.
cq	Handle to CQ.
cq_attr	Pointer to CQ attribute information.
mask	Mask for the CQ modify operation.
err_data	Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_modify() allows the caller to modify some attributes of a given CQ. The only supported CQ modify operation is changing the number of CQ entries. A CQ can be resized even if there are outstanding Work Completions or outstanding Work Requests on the queues associated with the specified CQ. Completions will not be lost as a result of a resize. Upon successful completion, the actual number of CQ entries supported by the RNIC is returned in the *num_cqe* field of *cq_attr*. The resize operation may adversely affect performance while the CQ is being resized. However, the act of resizing will not directly generate completion or asynchronous errors.

If the CQ has encountered an internal error or an overrun, RI_RET_CQ_ERROR is returned. All QPs associated with this CQ will no longer function properly and will need to be shut down using *ri_qp_destroy*() or moved to idle state using *ri_qp_modify*().

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Inadequate resources to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.

RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
RI_RET_INVALID_PARAMETER	*cq_attr* is NULL or *num_cqe* is 0 or invalid *mask* specified.
RI_RET_CQ_CAPACITY_EXCEEDED	Number of CQ entries requested exceeds RNIC capability.
RI_RET_CQ_WOULD_OVERFLOW	More outstanding entries on CQ than number of CQ entries specified. This can happen if caller attempts to decrease the number of CQ entries.
RI_RET_CQ_ERROR	CQ has an overrun or has become inaccessible.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_destroy*(), *ri_cq_modify*(), *ri_cq_query*()

ri_cq_destroy()

NAME

ri_cq_destroy – destroy the specified CQ

SYNOPSIS

```
ri_api_return_t ri_cq_destroy (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_cq_handle_t    cq,
    OUT  ri_err_data_t     *err_data
);
```

DESCRIPTION

rnic Handle of the RNIC the CQ belongs to.

cq Handle to the CQ to be destroyed.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_destroy() destroys the specified CQ. Resources allocated by *ri_cq_create*() and subsequently by *ri_cq_modify*() will be deallocated by this call. If there is a QP still associated with the CQ, an RI_RET_CQ_INUSE error is returned. Upon successful return, the VP shall not invoke the CQ event handler or post any further CQ event to this CQ. Also, no completion event handling for this CQ will be in progress upon successful return. Any completion that has not been retrieved from the CQ is discarded. Upon successful return, no async error/event for this CQ will be in progress, and no further ones will be invoked for this CQ.

RETURN VALUE

RI_RET_SUCCESS	The specified CQ has been destroyed successfully.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
RI_RET_CQ_INUSE	One or more Work Queues are still associated with the CQ.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_modify*()

ri_cq_set_event_handler()

NAME

ri_cq_set_event_handler – register a handler to be called when the next Work Completion becomes available on the specified RNIC

SYNOPSIS: KERNEL

```
ri_api_return_t ri_cq_set_event_handler (
    IN      ri_vp_handle_t            vp_hdl,
    IN OUT  ri_cq_event_handler_id_t *handler_id,
    IN      ri_cq_event_callback_t    cq_callback,
    OUT     ri_err_data_t            *err_data
);

/* Completion callback function pointer type */
typedef void (*ri_cq_event_callback_t)(
    IN OUT  ri_os_data_t  vp_hdl_os_data,
    IN OUT  ri_os_data_t  cq_os_data
);
```

DESCRIPTION

vp_hdl Handle to RNIC to register completion handler on. This must be the same as the handle used in *ri_rnic_open*().

handler_id Pointer to CQ event handler identifier.

cq_callback A callback function to be called when the next Work Completion becomes available on any CQ that is associated with the specified/new *handler_id* and that has requested notification through *ri_cq_req_notification*(). See *ri_cq_event_callback_t* for a description of arguments passed to the callback function.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_set_event_handler() registers a completion event handler. Only one completion event handler can be registered per CQ event handler ID on an RNIC. However, an RNIC may support more than one CQ event handler ID.

If *handler_id* points to zero, a new CQ event handler ID is returned in *handler_id* and *cq_callback* is registered on the newly generated handler ID. Subsequent calls to *ri_cq_set_event_handler*() with a valid CQ event handler ID will overwrite the CQ event handler routine to be called. The CQ event handler ID is associated with a CQ at the time of *ri_cq_create*(). This call does not automatically request a notification on a completion event. The caller must call *ri_cq_req_notification*() in order to request notification. Specifying a NULL *cq_callback* for a given CQ event handler ID clears the completion handler associated with that handler ID. No callback function will be invoked upon completion on CQs associated with that handler ID even if *ri_cq_req_notification*() has been called on the CQ.

ri_cq_set_event_handler() can be invoked only in the kernel. The user mode entry point of this call is simply a stub that returns RI_RET_NO_PERMISSION.

The *vp_hdl_os_data* parameter that is passed to *cq_callback* is identical to *os_data* passed by the Consumer at the time of *ri_rnic_open*(). The *cq_os_data* parameter that is passed to *cq_callback* is identical to *os_data* passed by the Consumer at the time of *ri_cq_create*().

RETURN VALUE

RI_RET_SUCCESS	The completion event handler is successfully registered on the specified RNIC.
RI_RET_RESOURCE_SHORTAGE	Inadequate resources to complete the call. RNIC may have run out of handler IDs.
RI_RET_VP_INVALID_HANDLE	The VP handle passed is invalid.
RI_RET_INVALID_PARAMETER	*handler_id* is NULL.
RI_RET_INVALID_HANDLER_ID	Invalid *handler_id* passed.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_req_notification*(), *ri_rnic_open*()

ri_cq_req_notification()

NAME

ri_cq_req_notification – cause the completion handler to be called when the next Work Completion becomes available on the specified CQ

SYNOPSIS

```
ri_api_return_t ri_cq_req_notification (
    IN   ri_rnic_handle_t        rnic,
    IN   ri_cq_handle_t          cq,
    IN   ri_cmplt_notify_type_t  type,
    OUT  ri_err_data_t          *err_data
);

typedef enum ri_cmplt_notify_type ri_cmplt_notify_type_t;
enum ri_cmplt_notify_type {
    RI_CMPLT_NOTIFY_SOLICITED,
    RI_CMPLT_NOTIFY_ANY
};
```

DESCRIPTION

rnic Handle to RNIC the specified CQ belongs to.

cq Handle to CQ to enable notification on.

type Completion notification type. See *ri_cmplt_notify_type_t*.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_req_notification() requests the CQ event handler to be called when the next completion entry of the specified type is added to the specified CQ. The CQ event handler is registered using the *ri_cq_set_event_handler*() call. The handler is called at most once per *ri_cq_req_notification*() call for a particular CQ. Any CQ entries that existed before notification is enabled will not result in a call to the handler.

Completion Events are one of two types: solicited or unsolicited. A solicited completion event occurs when an incoming RDMA Write with Immediate Data (valid only for IB) or Send message, with the solicited event header bit set causes a successful receive Work Completion to be added to a CQ, or when a Work Request that was posted with RI_WR_NOTIFY_FLAG enabled causes a successful completion to be added to a CQ, or when an unsuccessful Work Completion is added to a CQ. An unsolicited completion event occurs when any other successful receive Work Completion, or any successful send Work Completion, is added to a CQ.

If an *ri_cq_req_notification*() is pending, subsequent calls to *ri_cq_req_notification*() for the same CQ prior to the completion event affect only when the notification occurs. A *ri_cq_req_notification*() for any completion event takes precedence over an *ri_cq_req_notification*() for a solicited event completion for the same CQ. If multiple calls to *ri_cq_req_notification*() have been made for the same CQ and at least one of the requests set the notification type to the next completion, the CQ event handler will be called when the next completion is added to that CQ. The CQ event handler will be called only once, even though

multiple CQ notification requests were made prior to the completion event for the specified CQ. Once the CQ event handler is called, another completion event handler must be enabled before it gets called again. It is the responsibility of the caller to call *ri_cq_poll*() to retrieve a Work Completion.

If *ri_cq_req_notification*() is invoked on a CQ that is not associated with an event handler (i.e., the CQ was created with a zero *handler_id*), RI_RET_CQ_NO_EVENT_HANDLER is returned. If there is no completion callback function registered with the handler ID that is associated with the CQ, no notification will be generated, even if *ri_cq_req_notification*() is invoked.

This call can be invoked from an interrupt context.

RETURN VALUE

RI_RET_SUCCESS	The completion event handler will be called on the next completion event of the specified type.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
RI_RET_INVALID_PARAMETER	The completion notification type passed is invalid.
RI_RET_CQ_NO_EVENT_HANDLER	The CQ was not associated with any event handler ID during CQ creation and hence cannot be enabled for notifications.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_poll*(), *ri_cq_set_event_handler*()

ri_cq_poll()

NAME

ri_cq_poll – poll the specified CQ for one or more new Work Completions

SYNOPSIS

```
ri_api_return_t ri_cq_poll (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_cq_handle_t    cq,
    OUT     ri_wc_t          *wcs,
    IN OUT  uint_t           *num_wcs,
    OUT     ri_err_data_t    *err_data
);

/* Performance Note: An attempt to align the WC types with IT API's
event types has been done for the first five types. It is highly
desirable that the definitions do not change with future versions of IT
API */

typedef enum ri_wc_type ri_wc_type_t;
enum ri_wc_type {
    RI_WC_TYPE_SEND = 0x1,
    RI_WC_TYPE_DATA_RECEIVED = 0x2,
    RI_WC_TYPE_RDMA_WRITE = 0x4,
    RI_WC_TYPE_RDMA_READ = 0x5,
    RI_WC_TYPE_BIND = 0x6,
    RI_WC_TYPE_FAST_REGISTER,
    RI_WC_TYPE_LOCAL_INVALIDATE,
    RI_WC_TYPE_RDMA_READ_LOCAL_INVALIDATE,
    RI_WC_TYPE_COMPARE_AND_SWAP,
    RI_WC_TYPE_FETCH_AND_ADD,
    RI_WC_TYPE_RDMA_WRITE_IMMEDIATE_RECEIVED
};

typedef enum ri_wc_flags ri_wc_flags_t;
enum ri_wc_flags {
    RI_WC_FLAG_SOLICITED_EVENT = 1 << 0,
    RI_WC_FLAG_LOCAL_SOLICITED = 1 << 1,
    RI_WC_FLAG_RTAG_INVALIDATE = 1 << 2,
    RI_WC_FLAG_IMMEDIATE_DATA = 1 << 3,
    /* Pass through if there is no local solicited notification */
    RI_WC_FLAG_CONSUMER2 = RI_WC_FLAG_LOCAL_SOLICITED,
    RI_WC_FLAG_CONSUMER1 = 1 << 29,
    RI_WC_FLAG_CONSUMER0 = 1 << 30
};

/* Performance Note: An attempt to align the completion status with IT
API's dto_status_t has been done for the first five types. It is highly
desirable that the definitions do not change with future versions of IT
API */

typedef enum ri_wc_status ri_wc_status_t;
enum ri_wc_status {
```

```
        RI_WC_STATUS_SUCCESS = 0,
        RI_WC_STATUS_WR_FLUSHED_ERROR = 1,
        RI_WC_STATUS_LOCAL_QP_CATASTROPHIC =2,
        RI_WC_STATUS_LOCAL_QP_OP_ERROR = 2,
            /* same as LOCAL_QP_CATASTROPHIC */

        /* iWARP-specific status */
        RI_WC_STATUS_WQE_FORMAT_ERROR,
        RI_WC_STATUS_REMOTE_TERMINATION_ERROR,
        RI_WC_STATUS_INVALID_STAG_ERROR,
        RI_WC_STATUS_BASE_BOUNDS_VIOLATION_ERROR,
        RI_WC_STATUS_ACCESS_VIOLATION_ERROR,
        RI_WC_STATUS_INVALID_PD_ERROR,
        RI_WC_STATUS_ADDRESS_WRAP_ERROR,
        RI_WC_STATUS_STAG_INVALIDATE_ERROR,
        RI_WC_STATUS_ZERO_ORD_ERROR,
        RI_WC_STATUS_FMR_QP_NOT_PRIVILEGED_ERROR,
        RI_WC_STATUS_STAG_NOT_IN_INVALID_STATE_ERROR,
        RI_WC_STATUS_INVALID_FMR_PAGE_SIZE_ERROR,
        RI_WC_STATUS_INVALID_FMR_BLOCK_SIZE_ERROR,
        RI_WC_STATUS_INVALID_FMR_PBL_ENTRY_ERROR,
        RI_WC_STATUS_INVALID_FMR_FBO_ERROR,
        RI_WC_STATUS_INVALID_FMR_LENGTH_ERROR,
        RI_WC_STATUS_INVALID_FMR_PERMS_ERROR,
        RI_WC_STATUS_FMR_PBL_TOO_LONG_ERROR,
        RI_WC_STATUS_INVALID_FMR_VA_ERROR,
        RI_WC_STATUS_INVALID_BIND_MR_ERROR,
        RI_WC_STATUS_INVALID_BIND_MW_ERROR,
        RI_WC_STATUS_HUGE_PAYLOAD_ERROR,

        /* IB-specific status */
        RI_WC_STATUS_LOCAL_LENGTH_ERROR,
        RI_WC_STATUS_LOCAL_PROTECTION_ERROR,
        RI_WC_STATUS_MEMORY_MGMT_ERROR,
        RI_WC_STATUS_BAD_RESPONSE_ERROR,
        RI_WC_STATUS_LOCAL_ACCESS_ERROR,
        RI_WC_STATUS_REMOTE_INVALID_REQ_ERROR,
        RI_WC_STATUS_REMOTE_ACCESS_ERROR,
        RI_WC_STATUS_REMOTE_OP_ERROR,
        RI_WC_STATUS_TRANSPORT_RETRY_EXCEEDED,
        RI_WC_STATUS_RNR_RETRY_EXCEEDED,
        RI_WC_STATUS_REMOTE_ABORTED_ERROR,
    };

    /* Performance Note: An attempt to align RC (non-immediate data) Work
    Completion structure with IT API's dto_cmpl_event_t structure has been
    made. It is highly desirable that the definitions do not change with
    future versions of IT API */

    typedef struct ri_rc_wc ri_rc_wc_t;
    struct ri_rc_wc {
        /* op_type should be the first field in the structure */
        ri_wc_type_t  op_type;
```

```
        /* Placeholder for it_evd_handle_t in IT API. OSVs may need to define
        flags and invalidated_rtag to take the size of it_evd_handle_t on their
        platforms. */

            ri_os_data_t   cq_os_data;
            ri_wc_flags_t  flags;
            ri_rtag_t      invalidated_rtag;

        /* Placeholder for it_ep_handle_t in IT API. OSVs may need to define
        ri_os_data_t identical to it_ep_handle_t on their platforms. */

            ri_os_data_t  qp_os_data;

            ri_wr_id_t      id;
            ri_wc_status_t  op_status;
            uint32_t        bytes_transferred;  /* valid only for receive */
            uint32_t        immediate_data;  /* valid only if immediate data is
                                                received along with incoming
                                                Send or RDMA Write (IB only) */
            ri_err_data_t   err_data;
        };

        typedef struct ri_gen_wc ri_gen_wc_t;
        struct ri_gen_wc {
            /* op_type should be the first field in the structure */
            ri_wc_type_t  op_type;

        /* Placeholder for it_evd_handle_t in IT API. OSVs may need to pad this
        appropriately to take the size of it_evd_handle_t on their platforms.*/

            ri_os_data_t   cq_os_data;
            ri_wc_flags_t  flags;
            uint32_t       qpn;

        /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
        ri_handle_t identical to it_ep_handle_t on their platforms. */
            ri_os_data_t    qp_os_data;
            ri_wr_id_t      id;
            ri_wc_status_t  op_status;
            uint32_t        bytes_transferred;
            uint32_t        immediate_data;

            ri_lid_t        remote_lid;
            uint32_t        remote_qpn;
            uint8_t         remote_sl;
            uint8_t         local_lid_pathbits;
            uint16_t        ether_type;
            uint32_t        grh_present;
            uint32_t        pkey_index;
            uint32_t        freed_resource_count;
            ri_err_data_t   err_data;
        };

        typedef union ri_wc ri_wc_t;
```

```
union ri_wc {
    ri_wc_type_t  type_wc;
    ri_rc_wc_t    rc_wc;
    ri_gen_wc_t   ud_wc;
};
```

DESCRIPTION

rnic Handle to the RNIC the specified CQ belongs to.

cq Handle to the CQ to poll for Work Completion.

wcs Work Completion(s) returned containing information related to the completed Work Request from a QP the specified CQ is associated with.

num_wcs Number of Work Completions desired by the Consumer on input. On output, *num_wcs* returns the actual number of Work Completions returned by the Verbs Provider.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_cq_poll() polls the specified CQ for one or more Work Completions. The Consumer specifies the number of Work Completions desired in *num_wcs* upon input. If a Work Request has completed, a corresponding Work Completion is returned in *wcs*. The actual number of Work Completions returned by the Verbs Provider will be less than or equal to the desired number of Work Completions and will be returned in *num_wcs*. If no Work Completions have been found, then the call returns RI_RET_CQ_EMPTY. The Consumer must not assume that all Work Completions have been reaped if the actual number of Work Completions returned is less than the desired number of Work Completions. A Verbs Provider may return any number of completions less than *num_wcs* even if there are more completions to be reaped. The Consumer is guaranteed that all Work Completions have been reaped only when the call returns RI_RET_CQ_EMPTY. The Consumer is advised to pass an array of *ri_wc_t* even if the Consumer knows that the CQ is only associated with RC QP; e.g., a Consumer of only iWARP devices should not pass an array of *ri_rc_wc_t* to *ri_cq_poll*().

This call can be invoked from an interrupt context.

Field Name	Description
op_type	Type of the Work Request that completed. Most types returned in completion have one-one correspondence with *ri_wr_t*. Work Requests of type RI_WR_TYPE_SEND, RI_WR_TYPE_SEND_INVALIDATE with or without RI_WR_FLAG_SOLICITED_EVENT complete as RI_WC_TYPE_SEND and flags indicate further details on the send. Receive requests can complete with either RI_WC_TYPE_DATA_RECEIVED or RI_WC_TYPE_RDMA_IMMEDIATE_RECEIVED.
flags	Indicates if the send that completed was send with or without invalidate operation and/or if the send had RI_WR_FLAG_SOLICITED_EVENT turned on. For RI_WC_TYPE_DATA_RECEIVED type, the RI_WC_FLAG_RTAG_INVALIDATED indicates if the *invalidated_rtag* field is valid suggesting the message received also invalidated an *Rtag*. For Sends or RDMA Write with immediate data (IB only), RI_WC_FLAG_IMMEDIATE_DATA indicates the presence of immediate data.

Field Name	Description
invalidated_rtag	The *Rtag* that was invalidated due to an incoming Send with Invalidate. This field is valid only if the RI_WC_FLAG_RTAG_INVALIDATED flag is set.
qp_os_data	The opaque *os_data* parameter passed by the Consumer at the time of *ri_qp_create*() of the QP. This Work Completion relates to one of the Work Requests posted on the QP or the SRQ associated with the QP.
id	The 64-bit Work Request ID set by the Consumer in the associated Work Request. This is always valid, regardless of the status of the operation.
op_status	Status of the operation. This is always valid regardless of *op_type*.
bytes_transferred	The number of bytes transferred. Valid for all send, RDMA read/write, atomic, and receive Work Completion types. This does not include the length of any immediate data. In the case of IB-Raw IPv6 and IB-UD QPs, the number of bytes transferred is the payload of the message plus the 40 bytes reserved for the GRH. The 40 bytes is always included, whether or not the GRH is present.
immediate_data	4-byte immediate data. This is valid only if RI_WC_FLAG_IMMEDIATE_DATA is enabled in flags.
grh_present	GRH Present indicator, for Raw IPv6 and UD QPs only. If this indicator is set, the first 40 bytes of the buffer(s) referred to by the Scatter/Gather list will contain the GRH of the incoming message. If it is not set, the contents of the first 40 bytes of the buffer(s) will be undefined. Contents of the payload of the message will begin after the first 40 bytes.
remote_lid *remote_sl* *remote_qpn* *local_lid_pathbits* *ether_type*	Remote node address. Returned only for Datagram services.
freed_resource_ count	This is always valid, regardless of the status of the operation.
pkey_index	P_Key index, for GSI only.
qpn	QP number of the local QP that caused the completion.
err_data	Opaque information in each Work Completion that can be used for OS/IHV private communication. The contents and usage of this opaque information are outside the scope of RNICPI.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A new Work Completion is returned in *wc*.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
RI_RET_CQ_EMPTY	The CQ does not have any completions.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_req_notification*(), *ri_mr_reg*(), *ri_qp_create*(), *ri_qp_postrq*(), *ri_qp_postsq*(), *ri_rnic_open*()

ri_qp_create()

NAME

ri_qp_create – create a Queue Pair (QP) on the specified RNIC

SYNOPSIS

```
ri_api_return_t ri_qp_create(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  ri_qp_attr_t     *attrs,
    OUT     ri_qp_handle_t   *qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic — Handle to RNIC.

attrs — Attributes of the QP. This contains the desired values when the caller makes this call. Upon successful completion of the call, this is updated with the number of Work Requests and scatter/gather elements indicating the actual values the RNIC supports. See *ri_qp_attr_t* for a description of QP attributes.

qp — Handle to the newly created QP.

vp_data — Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data — Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data — Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_qp_create() allows the caller to create a QP for the specified RNIC. A set of initial QP attributes must be specified by the caller. See *ri_qp_attr_t* for which attributes are required to be specified based on the transport. If any of the required initial attributes are illegal, an appropriate error is returned and a QP will not be created.

Upon successful completion, *attrs* will indicate the actual values the RNIC assigned to the following fields of *ri_qp_attr_t*:

qp_id
sq_size
rq_size
sq_max_sges
sq_max_sges_rdmaw
rq_max_sges

max_rdma_read_outgoing
max_rdma_read_incoming

The *qp_id* field is set to the newly assigned QP ID or number. The values of the remaining fields are guaranteed to be greater than or equal to the values requested. The QP is created in RI_QP_STATE_IDLE state.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid QP handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_INVALID_PARAMETER	*attrs* passed is NULL.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_CQ_INVALID_HANDLE	*scq* or *rcq* in *ri_qp_attr_t* is invalid.
RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements requested exceeds RNIC capability.
RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
RI_RET_QP_INVALID_SERVICE_TYPE	Invalid Transport Service Type for this QP.
RI_RET_SRQ_LIMIT_OUT_OF_RANGE	RQ limit specified by *rq_hiwat* exceeds the maximum supported by the RNIC.
RI_RET_SRQ_UNSUPPORTED	SRQs not supported in the RNIC.
RI_RET_EXT_BMM_UNSUPPORTED	Base Memory management extensions not supported.
RI_RET_MPA_ATTR_UNSUPPORTED	RNIC does not support the specified values of some of the MPA attributes. Reasons include the RNIC is RDMAC-type and hence disabling markers is not supported, or it is a non-permissive IETF RNIC that cannot adjust its receive marker setting to interoperate with an RDMAC RNIC.
RI_RET_INVALID_DDP_RDMAP_VERSION	Specified DDP and RDMAP version number not supported.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_qp_query(), *ri_qp_destroy*(), *ri_qp_modify*()

ri_qp_query()

NAME

ri_qp_query – query the attributes of the specified QP

SYNOPSIS

```
ri_api_return_t ri_qp_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    OUT  ri_qp_attr_t     *attrs,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to an RNIC to which the QP belongs.

qp Handle to the QP to query.

attrs Attributes of the QP filled in as a result of the query operation.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_qp_query() returns the current values of the attributes of the specified QP. See *ri_qp_attr_t* for a description of the QP attributes. Either all the attributes are returned or none is returned in case of error.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. *attrs* contains the QP attributes.
RI_RET_INVALID_PARAMETER	*attrs* parameter passed is NULL.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_QP_INVALID_HANDLE	Invalid QP handle passed.
RI_RET_INVALID_PARAMETER	*attrs* is NULL or invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_qp_create(), *ri_qp_destroy*(), *ri_qp_modify*()

ri_qp_modify()

NAME

ri_qp_modify – modify selected attributes of the specified QP

SYNOPSIS

```
ri_api_return_t ri_qp_modify(
    IN      ri_rnic_handle_t   rnic,
    IN      ri_qp_handle_t     qp,
    IN      ri_qp_attr_mask_t  qp_mask,
    IN OUT  ri_qp_attr_t      *attrs,
    OUT     ri_err_data_t     *err_data
);

enum ri_qp_attr_mask {
    RI_QP_ATTR_NEXT_STATE                  = 1 << 0,
    RI_QP_ATTR_SQ_SIZE                     = 1 << 1,
    RI_QP_ATTR_RQ_SIZE                     = 1 << 2,
    RI_QP_ATTR_MAX_RDMA_READ_OUTGOING      = 1 << 3,
    RI_QP_ATTR_MAX_RDMA_READ_INCOMING      = 1 << 4,

    /* iWARP-specific masks */
    RI_QP_ATTR_RQ_HIWAT                    = 1 << 5,
    RI_QP_ATTR_RQ_ARMED                    = 1 << 6,
    RI_QP_ATTR_LLP_STREAM_HANDLE           = 1 << 7,
    RI_QP_ATTR_STREAM_MSG_BUFFER           = 1 << 8,
    RI_QP_ATTR_MPA_ATTR                    = 1 << 9,

    /* IB-specific masks */
    RI_QP_ATTR_ENABLE_RDMA_READ            = 1 << 5,
    RI_QP_ATTR_ENABLE_RDMA_WRITE           = 1 << 6,
    RI_QP_ATTR_ENABLE_ATOMICS              = 1 << 7,
    RI_QP_ATTR_DEST_QPID                   = 1 << 8,
    RI_QP_ATTR_ENABLE_ASYNC_NOTIFY_FOR_SQD = 1 << 9,
    RI_QP_ATTR_QKEY                        = 1 << 10,
    RI_QP_ATTR_SEND_PSN                    = 1 << 11,
    RI_QP_ATTR_RECV_PSN                    = 1 << 12,
    RI_QP_ATTR_MIN_RNR_TIMER               = 1 << 13,
    RI_QP_ATTR_PRIMARY_PATH                = 1 << 14,
    RI_QP_ATTR_ALTERNATE_PATH              = 1 << 15,
    RI_QP_ATTR_MIGSTATE                    = 1 << 16,
    RI_QP_ATTR_PRIMARY_PKEY                = 1 << 17,
    RI_QP_ATTR_PORT_NUM                    = 1 << 18,

    /* cannot go beyond << 31 */
};

typedef enum ri_qp_attr_mask ri_qp_attr_mask_t;
```

DESCRIPTION

rnic Handle to an RNIC to which the QP belongs.

qp	Handle to the QP whose attributes are to be modified.
qp_mask	Bitwise OR of *ri_qp_attr_mask_t* to select attributes to be modified.
attrs	Attributes structure which contains the requested values for the attributes corresponding to the mask bits chosen in *qp_mask* by the caller. Upon successful completion of the call, this contains the actual values of some of the attributes.

ri_qp_modify() allows the caller to modify the attributes of the QP identified by the specified handle. The attributes to be modified are identified by *qp_mask.* This call will cause the QP to transition to the specified state if RI_QP_ATTR_NEXT_STATE is part of *qp_mask.* Only a subset of the QP attributes can be modified in each of the QP states. If any of the QP attributes to be modified are invalid for the current state of the QP or the requested state transition is illegal for the underlying transport, none of the QP attributes will be modified, the QP state will remain unchanged, and an appropriate error will be returned. Some QP attributes can be modified with outstanding Work Requests. Any outstanding Work Request on a Work Queue may not execute as expected if the QP modifiers are changed. For instance, if RDMA Reads, which were successfully posted, are outstanding when the QP is modified to no longer allow RDMA Reads, some outstanding in-flight RDMA Reads may complete while pending WRs may fail.[1]

Upon successful completion, *attrs* will indicate actual values the RNIC supports if any of the following fields of *ri_qp_attr_t* are being modified:

sq_size
rq_size
max_rdma_read_outgoing
max_rdma_read_incoming

The values are guaranteed to be greater than or equal to the values requested. *sq_size* and *rq_size* can be modified only if the RNIC supports resizing of the Work Queues. If the QP is associated with an SRQ and the Consumer attempts to modify *rq_size* or if the QP is not associated with an SRQ and the Consumer attempts to modify either *rq_hiwat* or *rq_armed*, then RI_RET_QP_ATTRIBUTE_CANT_CHANGE is returned. The error is also returned when the RNIC does not support resizing of Work Queues and the Consumer attempts to modify either *sq_size* or *rq_size*.

A value of zero for *rq_hiwat* in *ri_qp_modify*() means the Consumer wants to get an asynchronous notification for the first message since arming the Receive Queue of the QP for high water mark notification. If the QP's Receive Queue is not armed or has not been armed since last notification, the QP can, however, consume more than one receive Work Request from its SRQ. The zero value for *rq_hiwat* will not be interpreted similarly if this is specified in an *ri_qp_create*() call. It merely leaves the newly created QP disarmed for high water mark notification.

If RI_QP_ATTR_LLP_STREAM_HANDLE is set in *qp_mask*, the *llp_stream_handle* field of *attrs* indicates the LLP stream handle that is to be associated with the QP. This could point to an opaque data structure that is interpreted alike by the RNICPI Consumer and the RNIC driver. For example, it may point to an LLP context that is being handed off from the operating system's host stack to the RNIC, or it may be a handle to an already offloaded LLP connection on the RNIC. Which mechanism is used and how the information is made known to the

[1] For a description of permitted QP state transitions, refer to Table 91 of [IB-R1.2] or Figure 6 of [VERBS-RDMAC].

Consumer and the driver are outside the scope of RNICPI. Keep-alive on the LLP could be turned by passing vendor-dependent data in *llp_stream_handle* if RNIC supports it.

If RI_QP_ATTR_STREAM_MSG_BUFFER is set in *qp_mask, stream_msg_buf* points to the last streaming mode message that needs to be sent on the LLP before the QP goes into RDMA mode and *stream_msg_buf_length* indicates the length of the message.

The mask RI_QP_ATTR_MPA_ATTR can be used to modify the MPA-specific attributes to be used for the IETF MPA startup framing protocol. *recv_marker_enabled* declares a receiver's requirement for markers. Setting this field will cause the RNIC to look for and accept markers in FPDUs received on the LLP connection. If the field is not set, the RNIC will assume markers will not be present in the received FPDUs on the connection. Similarly, setting *xmit_marker_enabled* declares that markers must be enabled in FPDUs that are transmitted by the RNIC and not setting it will cause markers not to be inserted on the stream. *crc_enabled* declares preferred CRC usage. Setting this field will cause the RNIC to generate and check CRCs on the connection. Not setting the field would mean CRCs must not be checked and need not be generated on the connection. Values of all the fields of *ri_mpa_attr_t* must be specified by the Consumer if RI_QP_ATTR_MPA_ATTR is set in *qp_mask*. It is the responsibility of the caller to modify the MPA attributes as desired before the QP goes into the RDMA mode. The MPA attributes can be set either in a call prior to or in the same call as the one that modifies the LLP stream handle.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_QP_INVALID_HANDLE	Invalid QP handle passed.
RI_RET_INVALID_PARAMETER	The *attrs* parameter passed is NULL or *qp_mask* is invalid.
RI_RET_RNIC_INVALID_PKEY_INDEX	PKey index is out of range.
RI_RET_RNIC_INVALID_PKEY_ENTRY	PKey index specifies invalid entry in PKey table.
RI_RET_QP_INVALID_STATE	Invalid state transition.
RI_RET_RNIC_INVALID_MTU	MTU of RNIC port exceeded.
RI_RET_RNIC_INVALID_PORT	Invalid port specified.
RI_RET_MIGRATION_STATE_INVALID	Invalid path migration state.
RI_RET_RNIC_INVALID_RNR_VALUE	Invalid RNR NAK Timer Field Value.
RI_RET_QP_INVALID_SERVICE_TYPE	Invalid Transport Service Type for this QP.
RI_RET_INVALID_LLP_HANDLE	Invalid LLP Handle (iWarp only).

RI_RET_QP_ATTRIBUTE_CANT_CHANGE	Requested attribute modification is not valid for the state transition asked for.
RI_RET_QP_RESIZE_UNSUPPORTED	RNIC does not support resizing QP.
RI_RET_RNIC_ATOMICS_NOT_SUPPORTED	Cannot enable incoming atomic operations as RNIC does not support atomic operations.
RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
RI_RET_RNIC_IRD_ORD_EXCEEDED	*max_rdma_read_outgoing* or *max_rdma_read_outgoing* exceeds RNIC capability.
RI_RET_QP_CANNOT_SHRINK	More outstanding entries on WQ than size specified.
RI_RET_WINDOWS_BOUND	Type 2A Memory Windows still bound to QP.
RI_RET_STILL_FLUSHING_WQES	QP is in error state and transition to Idle cannot happen as RI is busy flushing WQEs.
RI_RET_SRQ_LIMIT_OUT_OF_RANGE	RQ limit specified by *rq_hiwat* exceeds the maximum supported by the RNIC.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_MPA_ATTR_UNSUPPORTED	RNIC does not support the specified values of some of the MPA attributes. Reasons include the RNIC is RDMAC-type and hence disabling markers is not supported, or it is a non-permissive IETF RNIC that cannot adjust its receive marker setting to interoperate with an RDMAC RNIC.
RI_RET_INVALID_DDP_RDMAP_VERSION	Specified DDP and RDMAP version number not supported.
RI_RET_LLP_BAD_STATE	Invalid state for modifying MPA attributes. LLP has already moved to RDMA mode and may be exchanging FPDUs.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

iWARP Consumers that wish to move a QP to Error state are advised to follow it up with potentially repeated attempts to *ri_qp_modify*() to Idle if the error RI_RET_STILL_FLUSHING_WQES is returned. When the operation succeeds, they should reap all the expected completions for the QP. If the QP is associated with an SRQ, iWARP Consumers of RNICs that support generating an RI_EVENT_QP_LAST_WQE_REACHED asynchronous event may simply wait for this event in order to see *ri_qp_modify*() to Idle stop getting the RI_RET_STILL_FLUSHING_WQES error. If the QP is associated with an SRQ, IB

Consumers should wait for an RI_EVENT_QP_LAST_WQE_REACHED asynchronous event. As a subsequent step, either IB or iWARP Consumers should perform *ri_poll* on the associated CQs until they return RI_RET_CQ_EMPTY or a full scan through the CQs is complete. At this point, Consumers may safely reuse the QP as though it is newly created or go ahead and call *ri_qp_destroy*() on the QP. Not taking the steps suggested here may lead to loss of or corruption of the completions.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_qp_create(), *ri_qp_destroy*(), *ri_qp_query*()

ri_qp_destroy()

NAME

ri_qp_destroy – destroy the specified QP

SYNOPSIS

```
ri_api_return_t ri_qp_destroy(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to an RNIC to which the QP belongs.

qp Handle to the QP that is to be destroyed.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API allows the caller to destroy the specified QP and deallocate all the resources allocated to process Work Requests. A QP cannot be destroyed if it has a Type 2A Memory Window bound to it, in which case, this call will fail with RI_RET_WINDOWS_BOUND. If the QP is bound to only Type 2B Memory Windows, this call will not fail with this error. However, a QP instance is allowed to have Work Requests outstanding when an *ri_qp_destroy*() call is made. When a QP is destroyed, all outstanding Work Requests are implicitly cancelled. It is the responsibility of the caller to clean up resources associated with such Work Requests. No outstanding Work Requests will be completed after this call returns. Incoming operations destined for a QP that has been destroyed are discarded. Any completions sitting on the CQs associated with this QP may or may not be returned to the Consumer. The Consumer may also have the risk of overflowing the CQs if they have not successfully polled the CQs at least up to the number of expected completions. Upon successful return of this call, there will no async errors/events for this QP in progress and no further ones invoked.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The specified QP handle is successfully destroyed.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_QP_INVALID_HANDLE	Invalid QP handle passed.
RI_RET_WINDOWS_BOUND	Type 2A Memory Windows still bound to QP.
RI_RET_MULTICAST_ATTACHED	The QP is still attached to one or more IB multicast groups.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_qp_destroy(), *ri_qp_modify*(), *ri_qp_query*()

ri_special_qp_get()

NAME

ri_special_qp_get –return a handle to an IB special QP of specified type

SYNOPSIS

```
ri_api_return_t ri_special_qp_get(
    IN      ri_rnic_handle_t  rnic,
    IN      uint8_t           port,
    IN      ri_sqp_type_t     qp_type,
    IN OUT  ri_qp_attr_t     *attrs,
    OUT     ri_qp_handle_t   *qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef enum
{
    RI_SQP_TYPE_SMI,
    RI_SQP_TYPE_GSI,
} ri_sqp_type_t;
```

DESCRIPTION

rnic RNIC handle on which a special QP needs to be allocated.

port RNIC port number.

qp_type QP type requested. The allowed types are: SMI QP (QP0) and GSI QP (QP1).

attrs Attributes of the QP filled in as a result of the query operation. Upon successful completion of the call, this becomes a copy of *attrs* with the number of Work Requests and scatter/gather elements indicating the actual values the RNIC is capable of.

qp Handle to the newly created QP.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the “syscall” version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_special_qp_get() returns the handle for the specified QP type for the specified RNIC port. The special QP types include: SMI QP (QP0) and GSI QP (QP1). Handles associated with the SMI QP and the GSI QP will only be given out once for each QP per RNIC port. Subsequent

invocations of this call, without an intervening *ri_qp_destroy*(), will return an error. SMI/GSI QPs cannot share a completion queue with any non-SMI/GSI QP. An attempt to do so will result in an RI_RET_CQ_HANDLE_INVALID error. This operation cannot be performed on a device that does not support InfiniBand transport.

A set of initial QP attributes must be specified by the caller. If any of the required initial attributes are illegal, an appropriate error is returned and a QP will not be created. The following fields of *ri_qp_attr_t* are required to be specified in *attrs* with this call:

scq
rcq
sq_size
rq_size
sq_max_sges
rq_max_sges
signal_completions
pd
trans_type

Upon successful completion, *attrs* will indicate the actual values the RNIC is capable of supporting for the following fields of *ri_qp_attr_t*:

sq_size
rq_size
sq_max_sges
rq_max_sges

These are guaranteed to be greater than or equal to the values requested.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid QP handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_INVALID_PARAMETER	The *attrs* or *qp* parameter passed is NULL.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_QP_INVALID_SPECIAL_TYPE	Invalid Special QP type.
RI_RET_QP_INUSE	QP already in use (applies only to SMI and GSI QPs).
RI_RET_RNIC_INVALID_PORT	Invalid Port number specified.
RI_RET_CQ_INVALID_HANDLE	*scq* or *rcq* in *ri_qp_attr_t* is invalid.
RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements requested exceeds RNIC capability.

RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_qp_postsq(), ri_qp_post_send(), ri_qp_post_rdma_read(), ri_qp_post_rdma_write(), ri_qp_post_bind(), ri_qp_post_fmr(), ri_qp_post_local_invalidate(), ri_qp_post_atomic(), ri_qp_post_datagram()

NAME

ri_qp_postsq – post operations on the Send Queue of the specified QP

ri_qp_post_send – post a Send operation on the Send Queue of the specified QP

ri_qp_post_rdma_read – post RDMA Read operation on the Send Queue of the specified QP

ri_qp_post_rdma_write – post RDMA Write operation on the Send Queue of the specified QP

ri_qp_post_bind – post Memory Window Bind operation on the Send Queue of the specified QP

ri_qp_post_fmr – post fast Memory (Region) registration operation on the Send Queue of the specified QP

ri_qp_post_local_invalidate – post local (Memory Region) invalidation operation on the Send Queue of the specified QP

ri_qp_post_atomic – post atomic operation on the Send Queue of the specified QP

ri_qp_post_datagram – post datagram operation on the Send Queue of the specified QP

SYNOPSIS

```
ri_api_return_t ri_qp_postsq (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_qp_handle_t    qp,
    IN      ri_wr_t          *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data
);

/* See ri_wr_t for a description of the parameters in
   the following send methods */
ri_api_return_t ri_qp_post_send(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   ri_rtag_t         rtag,
    IN   uint_t            num_sgle,
    IN   uint32_t          imm_data,
    OUT  ri_err_data_t    *err_data
);

ri_api_return_t ri_qp_post_rdma_read (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
```

```
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint64_t          rdma_addr,
    IN   uint_t            num_sgle,
    IN   ri_rtag_t         rtag,
    OUT  ri_err_data_t    *err_data
);

ri_api_return_t ri_qp_post_rdma_write(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint64_t          rdma_addr,
    IN   ri_rtag_t         rtag,
    IN   uint_t            num_sgle,
    IN   uint32_t          imm_data,
    OUT  ri_err_data_t    *err_data
);

ri_api_return_t ri_qp_post_bind (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_mw_handle_t    mw_handle,
    IN   ri_rtag_t         rtag,
    IN   ri_mr_handle_t    mr_handle,
    IN   ri_ltag_t         ltag,
    IN   ri_addr_t         vaddr,
    IN   ri_length_t       length,
    IN   ri_mem_attr_t     mem_attr,
    OUT  ri_err_data_t    *err_data
);

ri_api_return_t ri_qp_post_fmr (
    IN   ri_rnic_handle_t       rnic,
    IN   ri_qp_handle_t         qp,
    IN   ri_wr_type_t           type,
    IN   ri_wr_flags_t          flags,
    IN   ri_wr_id_t             id,
    IN   ri_mr_reg_phys_attr_t  attr;
    IN   ri_ltag_t              ltag,
    IN   ri_rtag_t              rtag,
    IN   ri_mr_handle_t         mr_handle,
    OUT  ri_err_data_t         *err_data
);

ri_api_return_t ri_qp_post_local_invalidate(
    IN   ri_rnic_handle_t  rnic,
```

```
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_ltag_t         ltag,
        IN   ri_rtag_t         rtag,
        IN   ri_mrw_handle_t   mrw_handle,
        OUT  ri_err_data_t    *err_data
    );

    ri_api_return_t ri_qp_post_atomic (
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_rtag_t         rtag,
        IN   uint64_t          atomic_operand1,
        IN   uint64_t          atomic_operand2,
        IN   ri_data_seg_t     atomic_result,
        OUT  ri_err_data_t    *err_data
    );

    ri_api_return_t ri_qp_post_datagram (
        IN   ri_rnic_handle_t     rnic,
        IN   ri_qp_handle_t       qp,
        IN   ri_wr_type_t         type,
        IN   ri_wr_flags_t        flags,
        IN   ri_wr_id_t           id,
        IN   ri_data_seg_t       *sg_list;
        IN   uint_t               num_sgle;
        IN   uint32_t             imm_data;
        IN   ri_address_handle_t  remote_node_address,
        IN   uint32_t             dest_qp,
        IN   uint32_t             dest_qkey,
        IN   uint8_t              max_static_rate,
        OUT  ri_err_data_t       *err_data
    );
```

DESCRIPTION

rnic Handle to the RNIC to send a message out on.

qp Handle to the *qp* on which to post the send.

wr Pointer to an array of Work Requests containing the information required to perform the send. The fields that must be specified in *ri_wr_t* are dependent on the operation type specified.

num_wrs Number of Work Requests submitted. Upon return, it will indicate the number of Work Requests processed.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_qp_postsq() builds one or more WQEs from the information contained in the Work Requests and posts to the Send Queue of the specified QP. If the QP is enabled for selectable completion notification, the caller must specify whether a successful completion of this Work Request results in a completion entry on the CQ through RI_WR_SIGNALED_FLAG in the *flags* field of *wr*. When the call returns, *wr* is in the scope of the caller and can no longer be accessed by the implementation. *wr* points to an array of *ri_wr_t* structures filled in by the Consumer. *wr* can contain a mixture of Work Request types. There is no difference, semantically, between posting the different Work Requests in the same or different *ri_qp_postsq*() calls.

If the call returns successfully, all the Work Requests in *wr* will be posted to the Send Queue of the QP. If the call returns with an error, *num_wr* contains the number of Work Requests successfully posted to the Send Queue and the error corresponds to *num_wr*+1st Work Request in *wr*. Each successfully posted Work Request may generate a completion on the CQ associated with the Send Queue depending on *flags*. The Work Requests in *wr* are posted in the same order as they appear in the array. If RNIC supports enabling notification on a per-Work Request basis, setting RI_WR_LOCAL_SOLICITED_FLAG will cause the completion handler of the CQ to be invoked.

If RI_WR_SIGNALED_FLAG is not set, the implementation must ignore the setting of RI_WR_LOCAL_SOLICITED_FLAG and behave as though the flag is not set.

In general, the Consumer must use the union structure *ri_wr_t*. Depending on the operation type, an appropriate sub-structure needs to be filled in. For example, the Consumer must not pass a pointer to an array of *ri_wr_send_t* typecast to a pointer to *ri_wr_t* unless *num_wr* is 1. See *ri_wr_t* for a description of the different types of Work Requests that can be posted using *ri_qp_postsq*(). Consumers should not assume all the contents of the Work Requests are validated until Work Completion is received for the Work Request on the associated CQ.

If RI_WR_TYPE_FAST_REGISTER type is specified, the implementation may fail with RI_RET_NO_PERMISSION if called by a non-privileged Consumer.

An attempt to post an RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE type Work Request will fail with RI_RET_WR_SG_INVALID_FORMAT if *num_sgle* is greater than 1.

More than one type of Work Request can be outstanding on the same QP at the same time. The ordering and fencing considerations for atomic operations are the same as for RDMA Read.

The following allow Consumers to post one Work Request of a specific type at a time:

ri_qp_post_atomic
ri_qp_post_bind
ri_qp_post_datagram
ri_qp_post_fmr
ri_qp_post_local_invalidate
ri_qp_post_rdma_read
ri_qp_post_rdma_write
ri_qp_post_send

All that is said in the context of *ri_qp_postsq*() is also valid for these calls unless it is about posting more than one Work Request in the same call.

Where there is a choice between immediate or completion error types to signal an error condition for a posting function, the kVP implementation chooses which single error type is used to report the error condition. The uVP for the same RNIC may make a different choice than the

kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as noted in Section 4.5.2.

The calls described in this reference page can all be invoked from an interrupt context.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The send is successfully posted on the specified QP.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_QP_INVALID_HANDLE	The QP handle passed is invalid.
RI_RET_QP_CAPACITY_EXCEEDED	Too many Work Requests posted.
RI_RET_WR_INVALID_TYPE	Invalid operation type.
RI_RET_QP_INVALID_STATE	Invalid QP state. This error may be returned on iWARP transport if the QP is not in RTS state and it cannot complete the Work Request with flushed status. On IB transport, this error is returned only when the QP is in the Idle, Init, or RTR states. It is not returned when the QP is in the Error state due to race conditions that could result in indeterminate behavior. Work Requests posted to the Send Queue while the QP is in the Error state, are completed with a flush error.
RI_RET_WR_INVALID_FLAGS	Invalid flags specified.
RI_RET_WR_SG_INVALID_FORMAT	Invalid Scatter/Gather list format. Note that for Raw IPv6 QPs, the first 40 bytes of the buffer(s) referred to by the Scatter/Gather list must contain the IPv6 header of the outgoing message.
RI_ADDRESS_HANDLE_INVALID	Address handle specified in the Work Request is invalid.
RI_RET_INVALID_WINDOW_TYPE	Consumer attempted to perform either: an Invalidate operation on a Type 1 Memory Window; or a zero-length Bind operation on a Type 2 Memory Window.
RI_RET_MW_INVALID_HANDLE	The Memory Window handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The MR handle passed is invalid.
RI_MEM_INVALID_RTAG	RTag passed is invalid.
RI_MEM_INVALID_LTAG	LTag passed is invalid.
RI_RET_MEM_INVALID_ADDRESS	Invalid virtual address passed.
RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.

RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
RI_RET_RNIC_SG_LIMIT_EXCEEDED	Invalid Scatter/Gather list length.
RI_RET_RNIC_ATOMICS_NOT_SUPPORTED	Cannot enable incoming atomic operations as RNIC does not support atomic operations.
RI_RET_BASE_MEM_EXTN_UNSUPPORTED	Base Memory management extensions not supported
RI_RET_EXT_LIF_UNSUPPORTED	RNIC does not support Local Invalidate Fencing.
RI_RET_NOT_IN_BLOCK_MODE	RNIC does not support Block Type Physical Buffers or was not opened in this mode.
RI_RET_NOT_IN_PAGE_MODE	RNIC was not opened in Page mode.
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support Zero-Based Virtual Address (ZBVA).
RI_RET_NO_PERMISSION	Only privileged users are allowed to fast registrations.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

As a VP must implement both the WR list model (*ri_qp_postsq*()) and the parameterized model (example: *ri_qp_post_send*()), the following is provided as a simple implementation aid.

If the underlying implementation is the parameterized version, the Work Request list model can be implemented trivially as follows:

```
ri_qp_postsq( ri_work_request_t *wr, ... ) {

    For each work_request in wr
    /* Call the appropriate type-specific function passing the fields
       of wr as parameters. Clear RI_WR_COALESCE_FLAG for the last Work
       Request in the array. */
    ri_qp_post_xxxx();
}
```

If the underlying implementation is a Work Request list, parameterized versions can be supported as follows:

```
#define ASSIGN_WR_FIELD( wr, field_name ) \
    wr.field_name = field_name;

ri_qp_post_xxxx( rnic, qp, parameters,... ) {

    ri_work_request_xxxx_t  wr;
    For each parameter
    ASSIGN_WR_FIELD(wr, parameter) = parameter;
```

```
        ri_qp_postsq(rnic, qp, &wr, 1, err_data );
    }
```

If some vendor wants to implement without wrappers by writing optimized code for each model, they are free to do so.

SEE ALSO

ri_cq_create(), *ri_cq_poll*(), *ri_cq_req_notification*(), *ri_mr_reg*(), *ri_qp_create*(), *ri_rnic_open*(), *ri_qp_postrq*(), *ri_wr_t*

ri_qp_postrq(), ri_qp_post_recv(), ri_srq_post(), ri_srq_post_recv()

NAME

ri_qp_postrq – post list of receive Work Requests on the specified QP

ri_qp_post_recv – post buffers for a receive operation on the specified QP

ri_srq_post – post list of receive Work Requests on the specified SRQ

ri_srq_post_recv – post buffers for a receive operation on the specified SRQ

SYNOPSIS

```
ri_api_return_t ri_qp_postrq (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_qp_handle_t    qp_handle,
    IN      ri_wr_recv_t     *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data
);

ri_api_return_t ri_qp_post_recv (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp_handle,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint_t            num_sgle,
    OUT  ri_err_data_t    *err_data
);

ri_api_return_t ri_srq_post (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_srq_handle_t   srq_handle,
    IN      ri_wr_recv_t     *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data
);

ri_api_return_t ri_srq_post_recv (
    IN   ri_rnic_handle_t  rnic,
    IN   ri_srq_handle_t   srq_handle,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint_t            num_sgle,
    OUT  ri_err_data_t    *err_data
);

struct ri_wr_recv {
    ri_wr_flags_t  flags;
    ri_wr_id_t     id;
    ri_data_seg_t *sg_list;
    uint_t         num_sgle;
};
```

```
typedef struct ri_wr_recv ri_wr_recv_t;
```

DESCRIPTION

rnic	Handle to the RNIC to receive data on.
qp_handle	Handle to the QP on which to post buffers for receive operation.
srq_handle	Handle to the SRQ on which to post buffers for receive operation.
wr	Pointer to an array of Work Requests that contain the information required to perform the buffer posting.
flags	Flags for the receive operation. See *ri_wr_flags_t.*
id	Work Request ID.
sg_list	See *ri_wr_t*.
num_sgle	Number of valid scatter/gather elements in *sg_list.*
num_wrs	Number of Work Requests submitted. Upon return, it will indicate the number of Work Requests processed.
err_data	Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_qp_postrq() builds one or more WQEs from the information contained in the Work Requests and posts to the Receive Queue of the specified QP. *ri_srq_post*() builds one or more WQEs from the information contained in the Work Requests and posts to the Receive Queue of the specified SRQ. When the call returns, the Work Request is in the scope of the caller and will no longer be accessed by the implementation. *wr* points to an array of *ri_wr_recv_t* structures filled in by the Consumer. Each Work Request can contain a gather list of receive buffers. Each incoming message on the Receive Queue of the QP satisfies only one of the Work Requests posted.

If the call returns successfully, all the Work Requests in *wr* will be posted to the Receive Queue of the QP. If the call returns with error, *num_wr* contains the number of Work Requests successfully posted to the Receive Queue and the error corresponds to *num_wr*+1st Work Request in *wr*. Each successfully posted Work Request will generate a completion on the CQ associated with the Receive Queue. The Work Requests in *wr* are posted in the same order as they appear in the array. Flags other than RI_WR_LOCAL_SOLICITED_FLAG and RI_WR_COALESCE_FLAG are not valid in *ri_qp_postrq*()/*ri_srq_post*(). If RNIC supports enabling notification on a per-Work Request basis, setting RI_WR_LOCAL_SOLICITED_FLAG will cause the completion handler of the CQ to be invoked. Consumers should not assume all the contents of the Work Requests are validated until Work Completion is received for the Work Request on the associated CQ.

ri_qp_post_recv() and *ri_srq_post_recv*() let Consumers post one Work Request at a time rather than a list of Work Requests as permitted by *ri_qp_postrq*() and *ri_srq_post*(). All that is said in the context of *ri_qp_postrq*()/*ri_srq_post*() is also valid for *ri_qp_post_recv*() and *ri_srq_post_recv*() unless it is about posting more than one Work Request in the same call.

Note that for IB UD QPs, the first 40 bytes of the buffer(s) referred to by the Scatter/Gather list will contain the GRH of the incoming message. If no GRH is present, the contents of the first 40 bytes of the buffer(s) will be undefined. The presence of the GRH will be indicated by a bit in the Work Completion.

Where there is a choice between immediate or completion error types to signal an error condition for a posting function, the kVP implementation chooses which single error type is used to report the error condition. The uVP for the same RNIC may make a different choice than the kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as noted in Section 4.5.2.

The calls described in this reference page can all be invoked from an interrupt context

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The list of Work Requests is successfully posted on the specified QP.
RI_RET_QP_INVALID_HANDLE	The QP handle passed is invalid.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_WR_SG_INVALID_FORMAT	Invalid Scatter/Gather list format.
RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
RI_RET_WR_INVALID_FLAGS	Invalid flags specified.
RI_RET_QP_INVALID_STATE	Invalid QP state. Note that this error is returned only when the QP is in the Idle or Init state. It is not returned when the QP is in the Error state due to race conditions that could result in indeterminate behavior. Work Requests that are posted to the Receive Queue while the QP is in the Error state, are completed with a flush error.
RI_RET_QP_CAPACITY_EXCEEDED	Too many Work Requests posted.
RI_RET_RNIC_SG_LIMIT_EXCEEDED	Invalid Scatter/Gather list length.
RI_RET_QP_TIED_TO_SRQ	Attempt to post to a QP that is tied to an SRQ.
RI_RET_SRQ_UNSUPPORTED	SRQs not supported in the RNIC.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_cq_create(), *ri_cq_poll*(), *ri_cq_req_notification*(), *ri_mr_reg*(), *ri_qp_create*(), *ri_rnic_open*(), *ri_qp_postsq*()

ri_srq_create()

NAME

ri_srq_create – create a Shared Receive Queue object

SYNOPSIS

```
ri_api_return_t ri_srq_create(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  ri_srq_attr_t    *srq_attrs,
    OUT     ri_srq_handle_t  *srq,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

srq_attrs Desired SRQ attributes on input. Actual SRQ attributes on output.

srq Returned SRQ handle of the newly created SRQ.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_srq_create() creates a Shared Receive Queue for the specified RNIC. A set of initial SRQ attributes listed as IN parameters in *ri_srq_attr_t* must be specified by the Consumer. If the RNIC supports SRQ and any of the required initial attributes are illegal or missing, an error shall be returned and the SRQ shall not be created. On success, a handle to the newly created SRQ is returned.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_RNIC_INVALID_PARAMETER	NULL supplied for mandatory OUT parameter.

RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests exceeds RNIC's capability.
RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements exceeds RNIC's capability.
RI_RET_PD_INVALID	Invalid PD handle.
RI_RET_SRQ_LIMIT_OUT_OF_RANGE	*lowat* supplied in *srq_attrs* exceeds RNIC's capability.
RI_RET_SRQ_UNSUPPORTED	SRQ not supported in the RNIC.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_srq_destroy(), *ri_srq_modify*(), *ri_srq_query*()

ri_srq_query()

NAME

ri_srq_query – query a Shared Receive Queue object

SYNOPSIS

```
ri_api_return_t ri_srq_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_srq_handle_t   srq,
    OUT  ri_srq_attr_t    *srq_attrs,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

srq SRQ handle of the queried SRQ.

srq_attrs SRQ attributes of *srq*.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_srq_query() returns the attributes of the specified SRQ.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
RI_RET_INVALID_PARAMETER	NULL supplied for mandatory OUT parameter.
RI_RET_SRQ_UNSUPPORTED	SRQ not supported in the RNIC.
RI_RET_SRQ_IN_ERROR	SRQ is in error state.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_srq_create(), *ri_srq_destroy*(), *ri_srq_modify*()

ri_srq_modify()

NAME

ri_srq_modify – modify the attributes of a Shared Receive Queue

SYNOPSIS

```
ri_api_return_t ri_srq_modify(
    IN   ri_rnic_handle_t    rnic,
    IN   ri_srq_handle_t     srq,
    IN   ri_srq_attr_mask_t  srqmask,
    OUT  ri_srq_attr_t      *attrs,
    OUT  ri_err_data_t      *err_data
);

enum ri_srq_attr_mask {
    RI_SRQ_ATTR_MAX_WRS = 1 << 0,
    /* max_sgl attribute is NOT modifiable */
    RI_SRQ_ATTR_LOWAT = 1 << 1,
    /* pd is NOT modifiable *
     * modifying lowat implicitly arms the alarm.*/
};
typedef enum ri_srq_attr_mask ri_srq_attr_mask_t;
```

DESCRIPTION

rnic RNIC handle.

srq SRQ handle of the queried SRQ.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_srq_modify() modifies the attributes of an SRQ for the specified RNIC. If any of the modify attributes are invalid, none of the attributes shall be modified. The SRQ attributes that can be modified after the SRQ has been created are:

max_wrs If resizing of the SRQ is supported by the RNIC.

lowat If the SRQ Limit is greater than zero, then it shall be armed upon returning from this verb.

The actual number of outstanding Work Requests supported on the Shared Receive Queue is returned in *max_wrs*. If an error is not returned, this is guaranteed to be greater than or equal to the number requested. (This may require the Consumer to increase the size of the CQ.)

Note that there is no mechanism to modify an SRQ to disarm the limit alarm. However, the threshold can be changed even when the alarm is already enabled.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.

RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ Handle.
RI_RET_RNIC_INVALID_PARAMETER	NULL supplied for mandatory OUT parameter or *srqmask* is invalid.
RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
RI_RET_SRQ_IN_ERROR	SRQ is in the Error State.
RI_RET_SRQ_RESIZE_UNSUPPORTED	RNIC does not support resizing SRQ.
RI_RET_SRQ_LIMIT_OUT_OF_RANGE	SRQ Limit exceeds maximum number of Work Requests allowed on the SRQ.
RI_RET_SRQ_CANNOT_SHRINK	More outstanding entries on WQ than size specified.
RI_RET_SRQ_UNSUPPORTED	RNIC does not support SRQ.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_srq_create(), *ri_srq_destroy*(), *ri_srq_query*()

ri_srq_destroy()

NAME

ri_srq_destroy – destroy a Shared Receive Queue object

SYNOPSIS

```
ri_api_return_t ri_srq_destroy(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_srq_handle_t   srq,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

srq SRQ handle.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_srq_destroy() destroys the specified SRQ. An SRQ can be destroyed only after all the QPs associated with the SRQ have been destroyed. However, an SRQ can be destroyed even if there are Work Requests posted to the SRQ. It is the Consumer's responsibility to know what Work Requests were still in the SRQ when it was destroyed.

The only way to obtain a list of Work Requests still in an SRQ is to have a list of all buffers ever posted to the SRQ and then remove those that have completed. The Work Requests in the SRQ will not be flushed to any CQ when the SRQ is destroyed.

This limitation is not as severe as it might appear at first. The Consumer may know the entire set of buffers posted to the SRQ because they all come from a single resource that is about to be freed. Certainly that applies when SRQs are only destroyed just before the application exits. If the Consumer posts buffers from a variety of sources to an SRQ, and will attempt to continue operating after destroying that SRQ, they are strongly advised to maintain their own list of posted but not yet returned buffers.

Upon successful return from this call, there will be no async errors/events in progress for this SRQ and no further async errors/events invoked for this SRQ.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
RI_RET_SRQ_INUSE	SRQ still has QPs associated with it.
RI_RET_SRQ_UNSUPPORTED	RNIC does not support SRQ.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_srq_create(), *ri_srq_modify*(), *ri_srq_query*()

ri_ltag_alloc()

NAME

ri_ltag_alloc – allocate a Memory Region Stag

SYNOPSIS

```
ri_api_return_t ri_ltag_alloc(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  uint_t           *address_list_length,
    IN      ri_pd_handle_t    pd,
    IN      ri_mem_attr_t     attr,
    OUT     ri_mr_handle_t   *mr,
    IN OUT  ri_ltag_t        *ltag,
    OUT     ri_rtag_t        *rtag,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic — *rnic* handle. Must support fast register.

address_list_length
: Non-null pointer to length of the address list. On input: minimum length desired. On output: if successful, the actual length allocated (which is greater than or equal to the length requested).

pd — Handle to the Protection Domain.

attr — Set of permissions ever to be enabled for the created Memory Region. The ability to later grant local permissions is assumed. The ability to later grant remote permissions is only granted if at least one remote permission is specified here. Flags not related to access permission are not relevant for this call.

mr — Non-null pointer to mem handle to be created.

ltag — Non-null pointer to output location for STag to be created for local use. The key portion of the LTag will be taken from the input value.

rtag — Pointer to output location for STag to be created for remote use. It must be non-NULL if remote permissions are requested. See ri_rtag_t.

vp_data — Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the “syscall” version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data — Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API allocates a Memory Region STag without specifying the memory mapped. This will allocate or reserve physical buffer list resources for use by this Memory Region. This call either allocates the entire capacity requested or returns an error without modifying the state of the *rnic*. The actual size of the PBL resources allocated is returned in *address_list_length*. If the call is successful, the actual size allocated will be equal to or greater than the size requested. The allocated memory registration handle is released by invoking *ri_mr_dereg*().

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_PD_INVALID	Invalid PD handle.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_EXT_BMM_UNSUPPORTED	Base memory extensions not supported on the RNIC.
RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

The returned *ltag*, *rtag*, and memory handle may be interchangeable on many RNICs, particularly iWARP RNICs. To maintain portability the Consumer should take care to avoid confusing these returns, as that some RNICs will not detect using the "wrong" value in later calls while others will rely upon these values being distinct.

IMPLEMENTATION NOTES

The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform physical registrations. This should not be the default.

SEE ALSO

ri_mr_dereg(), *ri_qp_postsq*(), *ri_ltag_t*, *ri_mem_attr_t*

ri_mr_reg()

NAME

ri_mr_reg – register a Memory Region for direct access by the RNIC

SYNOPSIS

```
ri_api_return_t ri_mr_reg(
    IN        ri_rnic_handle_t  rnic,
    IN const  ri_mr_reg_attr_t *mr_reg_attr,
    OUT       ri_mr_handle_t   *mr,
    IN OUT    ri_ltag_t        *ltag,
    OUT       ri_rtag_t        *rtag,
    IN OUT    ri_vp_data_t      vp_data,
    IN OUT    ri_os_data_t      os_data,
    OUT       ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC on which to register the region.

mr_reg_attr Attributes of the Memory Region to be created.

mr Handle to the Memory Region returned.

ltag On output, *Ltag* for local access. See *ri_ltag_t*. On input, the "key" portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified (and the RNIC indicates support for "extended" key splitting through the RI_RNIC_EXT_INDEX_KEY_SPLIT flag). Otherwise, this argument is ignored on input.

rtag *Rtag* for remote access. This should be non-NULL and points to a valid value only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is requested by the caller.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API allows the caller to prepare a Memory Region for use by the specified RNIC. The memory registered is specified as a virtual address range. This call uses OS-specific methods to pin memory pages and create the virtual to physical translations that represent the Memory Region. The RI_MEM_FLAGS_USER_VA in *mr_reg_attr* (in the *mem_attr* field) indicates whether the virtual address supplied should be interpreted in a user process address space or in the kernel virtual address space.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid region handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_MEM_INVALID_ADDRESS	Invalid virtual address passed.
RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.
RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

To allow later use of *ri_mr_rereg_phys*() or the fast register Work Request, set the RI_MEM_CONSUMER_OWNS_KEY *mem_attr*.

Translation of virtual addresses to RNIC usable addresses must be performed using *ri_mem_map*().

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mem_map(), *ri_mr_dereg*(), *ri_mr_reg_phys*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_ltag_t*, *ri_mr_reg_attr_t*

ri_mr_query()

NAME

ri_mr_query –retrieve information about the specified Memory Region

SYNOPSIS

```
ri_api_return_t ri_mr_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mr_handle_t    mr,
    OUT  ri_mr_info_t     *mr_info,
    OUT  ri_err_data_t    *err_data
);

typedef struct ri_mr_info ri_mr_info_t;
struct ri_mr_info {
    ri_mr_reg_attr_t attr;
    ri_ltag_t        ltag;
    ri_rtag_t        rtag;
    ri_boolean_t     is_shared_region;
    ri_addr_t        local_lb;
    ri_addr_t        local_ub;
    ri_addr_t        remote_lb;
    ri_addr_t        remote_ub;
    ri_stag_state_t  stag_state;
};
```

DESCRIPTION

rnic Handle to the RNIC to which the region belongs.

mr Handle to the Memory Region to query.

mr_info Holding buffer for Memory Region information.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_mr_info_t fields:

attr Memory Region attributes also used for create/modify.

ltag *Ltag* issued at the time of registration.

rtag *Rtag* issued at the time of registration.

is_shared_region
Flag indicating whether the region is currently a shared region. If false, the Memory Region does not share page table resources with any other Memory Region. Invalidates and fast memory registers can only be performed on non-shared regions.

local_lb,local_ub
Actual local protection bounds enforced for the region.

remote_lb,remote_ub
Actual remote protection bounds enforced for the region.

stag_state Stag state (valid/invalid).

This API retrieves the keys, protection domain and bounds, and access control settings for the specified region. The remote protection bounds *remote_lb* and *remote_ub* contain valid values only if the region is enabled with RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ access.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The region handle passed is invalid.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

InfiniBand 1.1 adapters may enable a larger region than requested. Consumers should use *ri_mr_query*() to determine the actual region enabled if adjacent memory structures must be protected from RNIC access. The boundaries returned are in the same address domain as those originally specified. The “actual” boundaries only differ from the input boundaries because of “rounding out” to page boundaries. They are not “actual” in the sense that they have been translated to physical addresses or from zero-based addresses.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mr_dereg(), *ri_mr_reg*(), *ri_mr_rereg*(), *ri_ltag_t*, *ri_mem_attr_t*, *ri_stag_state_t*

ri_mr_rereg()

NAME

ri_mr_rereg –modify the attributes of an existing Memory Region

SYNOPSIS

```
ri_api_return_t ri_mr_rereg(
    IN        ri_rnic_handle_t  rnic,
    IN        ri_mr_handle_t    in_mr,
    IN        ri_mr_rereg_t     change_mask,
    IN const  ri_mr_reg_attr_t *mr_attr,
    OUT       ri_mr_handle_t   *out_mr,
    IN OUT    ri_ltag_t        *ltag,
    OUT       ri_rtag_t        *rtag,
    IN OUT    ri_vp_data_t      vp_data,
    IN OUT    ri_os_data_t      os_data,
    OUT       ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC the region belongs to.

in_mr Handle to the Memory Region originally registered.

change_mask Set of Memory Region Attributes to be changed: address translation, protection domain, and/or access rights.

mr_attr Attributes of the Memory Region to be created.

out_mr Handle to the modified Memory Region. This may or may not be the same as mem passed.

ltag On output, the new *Ltag* assigned for local access. See *ri_ltag_t*. On input, the "key" portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified (and the RNIC indicates support for "extended" key splitting through the RI_RNIC_EXT_INDEX_KEY_SPLIT flag). Otherwise, this argument is ignored on input.

rtag New *Rtag* assigned for remote access. This should be non-NULL and points to a valid value only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is requested by the caller.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API modifies the attributes of an existing Memory Region. Any existing region owned by the caller can be modified, regardless of whether *ri_mr_reg*(), *ri_mr_reg_shared*(), or *ri_mr_rereg*() has been previously called on the region. The *ltag* returned is a handle for the Memory Region suitable for use in Work Requests to describe locally accessible memory. When perms requested contains RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ, and the region is not already enabled with the access control, a remote handle (*rtag*) for the Memory Region suitable for use by inbound RDMA and/or atomic operations (*Rtag*) is returned. The RI_MEM_FLAGS_USER_VA in *mr_attr* (in the *mem_attr* field) indicates whether the virtual address supplied should be interpreted in a user process address space or in the kernel virtual address space.

This call conceptually performs the functions *ri_mr_dereg*() followed by *ri_mr_reg*(). If the caller attempts to reregister a Memory Region while the region still has any Memory Windows bound to it, an RI_RET_WINDOWS_BOUND error is returned. If the original region is sharing the translation tables in the RNIC with other region(s) through *ri_mr_reg_shared*(), this call will not invalidate those registrations. No change will be made to the existing registration if the following errors are returned:

RI_RET_WINDOWS_BOUND
RI_RET_RNIC_INVALID_HANDLE
RI_RET_MR_INVALID_HANDLE
RI_RET_INVALID_PARAMETER

If the call returns any other error, both “old” and “new” registrations are invalidated and any associated resources are freed.

If a remote agent is accessing a Memory Region while it is in the process of being reregistered, the behavior of a deregistration operation followed by a separate registration operation would be exhibited.

To allow later use of *ri_mr_rereg_phys*() or the fast register Work Request, set the RI_MEM_CONSUMER_OWNS_KEY *mem_attr*.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid region handle is returned in *out_mem*.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to reregister the Memory Region.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The region handle passed is invalid.
RI_RET_INVALID_PARAMETER	NULL supplied for mandatory OUT parameter.
RI_RET_MEM_INVALID_ADDRESS	Invalid virtual address passed.
RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.
RI_RET_PD_INVALID	Invalid Protection Domain.
RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.

RI_RET_WINDOWS_BOUND	Operation Denied; region still has bound Memory Window(s).
RI_RET_EXT_BMM_UNSUPPORTED	Base Memory management extensions not supported.
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

Where possible, resources are expected to be reused instead of deallocated and reallocated. Resources should not be available for other callers to claim between "deallocation" and "reallocation".

SEE ALSO

ri_mr_dereg(), *ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_mr_rereg_phys*(), *ri_ltag_t*, *ri_mr_reg_attr_t*

ri_mr_dereg()

NAME

ri_mr_dereg – deregister the specified Memory Region

SYNOPSIS

```
ri_api_return_t ri_mr_dereg(
    IN  ri_rnic_handle_t  rnic,
    IN   ri_mr_handle_t   mr,
    OUT  ri_err_data_t   *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC on which to register the region.

mr Handle to a registered Memory Region.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API removes a Memory Region that was previously allocated by *ri_ltag_alloc*(), *ri_mr_reg_phys*(), or *ri_mr_reg*() from the RNIC translation table. The region is unpinned by using OSV services if it was previously pinned in the associated registration call. If the caller attempts to deregister a Memory Region while it still has any Memory Windows bound to it, an RI_RET_WINDOWS_BOUND error is returned.

Work Requests or remote operation requests that reference the deregistered Memory Region will fail with a completion status of RI_COMPLT_STATUS_LOCAL_PROTECTION_ERROR. On successful return from this call, all access to local memory enabled by this Memory Region will have been completed or terminated.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The specified Memory Region is successfully destroyed.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The Memory Region handle passed is invalid.
RI_RET_WINDOWS_BOUND	Operation denied; region still has bound Memory Window(s).
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

This function deallocates a Memory Region. Use *ri_mw_dealloc*() to deallocate a Memory Window.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_ltag_alloc(), *ri_mr_reg*(), *ri_mr_reg_phys*(), *ri_mr_reg_shared*()

ri_mr_reg_phys()

NAME

ri_mr_reg_phys – register a set of physically addressed buffers as a single Memory Region for direct access by RNIC

SYNOPSIS

```
ri_api_return_t ri_mr_reg_phys(
    IN      ri_rnic_handle_t       rnic,
    IN OUT  ri_mr_reg_phys_attr_t *mr_phys_attr,
    OUT     ri_mr_handle_t        *mr,
    IN OUT  ri_ltag_t             *ltag,
    OUT     ri_rtag_t             *rtag,
    IN OUT  ri_vp_data_t           vp_data,
    IN OUT  ri_os_data_t           os_data,
    OUT     ri_err_data_t         *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC on which to register the region.

mr_phys_attr
Attributes requested for the Memory Region to be created on input. Actual attributes on output.

mr Handle to the Memory Region returned.

ltag On output, *Ltag* for local access. See *ri_ltag_t*. On input, the "key" portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified. Otherwise, this argument is ignored on input.

rtag *Rtag* for remote access. This should be non-NULL and point to a valid value only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is requested by the caller.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API prepares a Memory Region for use by an RNIC based upon a physical addresses. This call differs from *ri_mr_reg*() primarily in the type of memory range that is registered. The additional difference is that *ri_mr_reg_phys*() also outputs the actual Memory Region attributes,

which are typically unchanged from a successful request. When the call completes, the region handle returned can be used just as a handle obtained from the *ri_mr_reg*() call.

The virtual address (in *mr_phys_attr->common.va*) actually assigned may differ from that specified on input for InfiniBand adapters. The virtual address field is updated to reflect the actual value used in these cases.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid region handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter..
RI_RET_MEM_INVALID_ADDRESS	At least one of the physically addressed buffers is specified incorrectly.
RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.
RI_RET_MEM_INVALID_PBLE_LENGTH	Invalid PBLE length passed.
RI_RET_MEM_INVALID_PBLE	Invalid PBLE entry supplied.
RI_RET_MEM_INVALID_OFFSET	The offset passed is invalid.
RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
RI_RET_NO_PERMISSION	Caller does not have permission to register physical pages.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform physical registrations. This should not be the default.

SEE ALSO

ri_mr_dereg(), *ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_mr_rereg_phys*(), *ri_ltag_t*, *ri_mr_reg_phys_attr_t*

ri_mr_rereg_phys()

NAME

ri_mr_rereg_phys – modify the attributes of an existing non-shared Memory Region with the translation changes, if any, specified as a list of physically addressed buffers

SYNOPSIS

```
ri_api_return_t ri_mr_rereg_phys(
    IN      ri_rnic_handle_t         rnic,
    IN      ri_mr_handle_t           in_mr,
    IN      ri_mr_rereg_t            change_mask,
    IN OUT  ri_mr_reg_phys_attr_t   *mr_phys_attr,
    OUT     ri_mr_handle_t          *out_mr,
    IN OUT  ri_ltag_t               *ltag,
    IN OUT  ri_rtag_t               *rtag,
    IN OUT  ri_vp_data_t             vp_data,
    IN OUT  ri_os_data_t             os_data,
    OUT     ri_err_data_t           *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC the region belongs to.

in_mem Handle to the non-shared Memory Region originally registered.

change_mask
Set of Memory Region Attributes to be changed: address translation, protection domain, and/or access rights.

mr_phys_attr
Attributes desired for the Memory Region. This is an IN OUT parameter; some attributes may be modified by call completion to indicate actual resources/addresses granted/assigned.

out_mem Handle to the modified Memory Region. This may or may not be same as *in_mem*.

ltag On output, the new *Ltag* assigned for local access. See *ri_ltag_t*. On input, the "key" portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified. Otherwise, this argument is ignored on input.

rtag New *Rtag* assigned for remote access. This should be non-NULL and point to a valid value only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is requested by the caller in perms.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API modifies the attributes of an existing Memory Region. Any non-shared existing region owned by the caller can be modified, regardless of whether it was created using *ri_mr_reg*(), or *ri_mr_reg_phys*() has been previously called on the region. A description of the Memory Region suitable for use in Work Requests to describe locally accessible memory locations (*Ltag*) is returned. When the access permissions requested contains RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ, and the region is not already enabled with the access control, a description of the Memory Region suitable for use by inbound RDMA and/or atomic operations (*Rtag*) is returned.

The virtual address (in *mr_phys_attr->common.va*) actually assigned may differ from that specified on input for InfiniBand adapters. The virtual address field is updated to reflect the actual value used in these cases.

This call conceptually performs the functions *ri_mr_dereg*() followed by *ri_mr_reg_phys*(). If the caller attempts to reregister a Memory Region while the region still has any Memory Windows bound to it, an RI_RET_WINDOWS_BOUND error is returned. If the original region is sharing the translation tables in the RNIC with other region(s) through *ri_mr_reg_shared*(), this call will not invalidate those registrations. No change will be made to the existing registration if the following errors are returned:

RI_RET_WINDOWS_BOUND
RI_RET_RNIC_INVALID_HANDLE
RI_RET_MR_INVALID_HANDLE
RI_RET_INVALID_PARAMETER

If the call returns any other error, both “old” and “new” registrations are invalidated and any associated resources are freed.

If a remote agent is accessing a Memory Region while it is in the process of being reregistered, the behavior of a deregistration operation followed by a separate registration operation would be exhibited.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid region handle is returned in *outMem*.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to reregister the Memory Region.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The Memory Region handle passed is invalid, or belongs to a shared Memory Region.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_MEM_INVALID_ADDRESS	At least one of the physically addressed buffers is specified incorrectly.
RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.

RI_RET_MEM_INVALID_PBLE_LENGTH	Invalid PBL Length provided.
RI_RET_MEM_INVALID_OFFSET	The offset passed is invalid.
RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
RI_RET_WINDOWS_BOUND	Operation Denied; region still has bound Memory Window(s).
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
RI_RET_EXT_BMM_UNSUPPORTED	RNIC does not support memory extensions.
RI_RET_NO_PERMISSION	Caller does not have permission for physical registers.
RI_RET_FAIL	Unknown/fatal error. *ri_get_err_detail* can be used to get detailed error message.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

Where possible, resources are expected to be reused instead of deallocated and reallocated. Resources should not be available for other callers to claim between "deallocation" and "reallocation".

The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform physical registrations. This should not be the default.

SEE ALSO

ri_mr_dereg(), *ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_mr_rereg_phys*(), *ri_ltag_t*, *ri_mr_reg_phys_attr_t*

ri_mr_reg_shared()

NAME

ri_mr_reg_shared – register a Memory Region that shares the physical pages with an existing Memory Region

SYNOPSIS

```
ri_api_return_t ri_mr_reg_shared(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_mr_handle_t    in_mr,
    IN OUT  ri_mr_reg_attr_t *mr_reg_attr,
    OUT     ri_mr_handle_t   *out_mr,
    IN OUT  ri_ltag_t        *ltag,
    OUT     ri_rtag_t        *rtag
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC the region belongs to.

in_mr Handle to the Memory Region originally registered.

mr_reg_attr Attributes desired for the new Memory Region. Length is ignored on input, and matches the original region's length. RI_MEM_FLAGS_NONCOHERENT is ignored on input, and is only meaningful on output.

out_mr Handle to the cloned Memory Region.

ltag On output, *Ltag* for local access for the new region. See *ri_ltag_t*. On input, the "key" portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified. Otherwise, this argument is ignored on input.

rtag *rtag* assigned for remote access. This should be non-NULL and point to a valid value only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is requested by the caller in perms.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

Given an existing Memory Region, this API creates a new Memory Region associated with the same physical memory locations, with the intention that the new Memory Region share RNIC mapping resources to the extent possible.

The virtual address, Protection Domain, and Access Rights specified for the new Memory Region need not be the same as those of the existing Memory Region. The size of the new region is, by definition, the same as that of the original.

The caller supplies a requested virtual address to be associated with the first page in the new Memory Region, and the implementation returns the virtual address that is actually assigned. If the RI_MEM_FLAGS_SHARED_REG_IOVA flag is set in *mr_reg_attr* (in the *mem_attr* field), then the virtual address supplied is an I/O virtual address (as in physical memory registration). When interpreted as an IOVA, the virtual address (in *mr_reg_attr->va*) actually assigned may differ from that specified on input for InfiniBand adapters. The virtual address field is updated to reflect the actual value used in these cases.

If the RI_MEM_FLAGS_SHARED_REG_IOVA flag is not set, the virtual address should be interpreted relative to the appropriate address space (as in virtual memory registration). The RI_MEM_FLAGS_USER_VA indicates whether the address space to use is a user process address space or the kernel address space.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid region handle is returned in *outMem*.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to create a shared Memory Region.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MR_INVALID_HANDLE	The region handle passed (*mr*) is invalid.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_PD_INVALID	Invalid Protection Domain.
RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
RI_RET_FAIL	Unknown/fatal error. *ri_get_err_detail* can be used to get detailed error message.

CONSUMER USAGE

ri_mr_reg_shared() can be used to register the same memory with different access permissions and/or for a different Protection Domain without requiring redundant mapping resources.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mr_dereg(), *ri_mr_reg*(), *ri_mr_reg_shared*(), *ri_mr_rereg*(), *ri_mr_rereg_phys*(), *ri_ltag_t*, *ri_mr_reg_attr_t*

ri_mw_alloc()

NAME

ri_mw_alloc – allocate a Memory Window on the specified RNIC

SYNOPSIS

```
ri_api_return_t ri_mw_alloc(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_pd_handle_t    pd,
    IN      ri_mw_type_t      mw_type,
    OUT     ri_mw_handle_t   *mw,
    OUT     ri_rtag_t        *rtag,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic	Handle to the RNIC on which to allocate a window.
pd	Handle to Protection Domain the window is to be associated with.
mw_type	Type of the window – Type 1 (wide) or Type 2 (narrow).
mw	Handle to window returned by the call.
rtag	*Rtag* returned for remote access of the window.
vp_data	Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.
os_data	Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.
err_data	Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API allocates a Memory Window which is associated with the specified protection domain. It is not inherently associated with any Memory Region at the time of allocation. The *rtag* returned is not yet in a valid state and must be bound using *ri_qp_postsq*().

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid window handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.

RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
RI_RET_PD_INVALID	Invalid Protection Domain.
RI_RET_WINDOW_TYPE_UNSUPPORTED	The RNIC does not support the requested window type.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mw_dealloc(), *ri_mw_query*(), *ri_qp_postsq*(), *ri_ltag_t*

ri_mw_query()

NAME

ri_mw_query – return the attributes associated with the specified Memory Window

SYNOPSIS

```
ri_api_return_t ri_mw_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mw_handle_t    mw,
    OUT  ri_mw_type_t     *mw_type,
    OUT  ri_stag_state_t  *stag_state,
    OUT  ri_pd_handle_t   *pd,
    OUT  ri_mem_attr_t    *attr,
    OUT  ri_rtag_t        *rtag,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic	Handle to the RNIC the window belongs to.
mw	Handle to the window.
mw_type	Type 1 (wide) or Type 2 (narrow) window.
stag_state	Stag state – valid/invalid.
pd	Handle to Protection Domain the Memory Window is associated with.
attr	Attributes of the Memory Window, including permissions and ZBVA. Note that a Memory Window has no permissions before it is bound to a Memory Region.
rtag	*Rtag* for remote access of the window.
err_data	Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

This API returns the window type, state, Protection Domain, permissions, and *Rtag* of the Memory Window specified. The caller may supply NULL pointers for any data that is not desired.

Note: The window handle is passed in, not the current *Rtag*. This is contrary to what might be expected from the RDMAC verbs, but is required to also support InfiniBand verbs.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. *Rtag* and *pd* returned are valid.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MW_INVALID_HANDLE	The Memory Window handle passed is invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mw_alloc(), *ri_mw_dealloc*(), *ri_ltag_t*, *ri_stag_state_t*

ri_mw_dealloc()

NAME

ri_mw_dealloc – deallocate the specified Memory Window

SYNOPSIS

```
ri_api_return_t ri_mw_dealloc(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mw_handle_t    mw,
    OUT  ri_err_data_t     *err_data
);
```

DESCRIPTION

rnic Handle of the RNIC the window belongs to.

mw Handle to the window to be deallocated.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

Any remote operation requests that are in process and actively using a Memory Window while and after it is deallocated will fail with RI_COMPLT_STATUS_LOCAL_PROTECTION_ERROR. Any API calls involving window handles but for *ri_mw_alloc*() will fail with RI_RET_MW_INVALID_HANDLE if a deallocated window handle is specified.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The specified Memory Window is successfully deallocated.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_MW_INVALID_HANDLE	The window handle passed is invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mw_alloc()

ri_mem_pin(), ri_mem_unpin()

NAME

ri_mem_pin – pin memory through the RNIC services

ri_mem_unpin – unpin memory through the RNIC services

SYNOPSIS

```
ri_api_return_t ri_mem_pin (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

ri_api_return_t ri_mem_unpin (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle of RNIC, returned from *ri_rnic_get_hdl*().

vaddr Virtual address.

len Length of the virtual address range in bytes.

os_data Opaque data owned by OS modules. In general, IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged via “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

These functions allow the Consumer to pin and unpin memory using RNIC services. The range of memory is specified in terms of a virtual address and length in bytes using *vaddr* and *len.* The address range of an *ri_mem_unpin*() call must be the same as the *ri_mem_pin*() call originally used to pin the memory. Multiple *ri_mem_pin*() calls can be made on the same or overlapping memory ranges with all pin reference counting issues properly handled. Some implementations may also understand when a memory range need not be pinned and automatically optimize for this case, so that a Consumer does not have to distinguish between different pinning requirements of various memory resource pools.

If *os_data* is being used to manage the state of the virtual memory range *ri_mem_pin*(), then the same *os_data* value must be passed into *ri_mem_unpin*(). Further, it is required to use the same *os_data* for subsequent *ri_mem_map*() and *ri_mem_unmap*() calls on this memory range. In

some OSs, it is required to call this entry point instead of calling the device-independent memory pinning service. (Mostly because these OSs need to know which device will "own" this memory during subsequent operations.) For these OSs, the VP implementation of *ri_mem_pin*() (and *ri_mem_unpin*()) has to modify the *os_data* to include the proper device context information. The exact method of doing this is OS-dependent (since it is related to the format of *os_data* for that OS).

RNICPI supports two different styles of pinning: in one, Consumers pin memory using an RNIC-independent service; in the other, pinning requires device-dependent context and uses the RNIC facility provided by *ri_mem_pin*() and *ri_mem_unpin*().

These functions apply only to privileged callers. If an attempt is made by a non-privileged caller to invoke these functions, RI_RET_NO_PERMISSION is returned.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.
RI_RET_NO_PERMISSION	Caller is not privileged.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_INVALID_PARAMETER	One of the parameters other than the RNIC handle is invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

Two different styles of pinning are supported in RNICPI. One style uses a model independent of the RNIC. An OS using this first style will not have to call *ri_mem_pin*() and *ri_mem_unpin*(). The other style of pinning needs device-dependent context to efficiently pin. The calls *ri_mem_pin*() and *ri_mem_unpin*() aim at supporting this latter model. As a matter of compatibility, these entry points can be implemented for the first style as:

```
ri_api_return_t ri_mem_pin (
    ri_rnic_handle_t  rnic,
    ri_vaddr_t        vaddr,
    ri_length_t       len,
    ri_os_data_t      os_data,
    ri_err_data_t    *err_data)
{
    return ri_svc_mem_pin(vaddr, len, os_data, err_data);
}

ri_api_return_t ri_mem_unpin (
    ri_rnic_handle_t  rnic,
    ri_vaddr_t        vaddr,
    ri_length_t       len,
    ri_os_data_t      os_data,
```

```
        ri_err_data_t    *err_data)
{
        return ri_svc_mem_unpin(vaddr, len, os_data, err_data);
}
```

SEE ALSO

ri_mem_map(), *ri_mem_unmap*(), *ri_svc_mem_pin*(), *ri_svc_mem_unpin*()

ri_mem_map(), ri_mem_unmap()

NAME

ri_mem_map – map a virtual memory range to a physical memory range directly accessible by an interface adapter

ri_mem_unmap – remove a mapping of a virtual memory range to a physical memory range directly accessible by an interface adapter

SYNOPSIS

```
ri_api_return_t ri_mem_map (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    OUT     ri_phys_buf_t    *io,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

ri_api_return_t ri_mem_unmap (
    IN      ri_rnic_handle_t  rnic,
    IN      ri_phys_buf_t    *io,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC on which to map the memory range.

vaddr Virtual address.

len Length of the virtual address range in bytes.

io Address and length of the physical memory range.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to *ri_syscall*. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_mem_map() attempts to map a virtual memory range defined by *vaddr* and *len* to a physical memory range *io* addressable by the specified interface adapter. The size of the memory successfully mapped is returned in the *phys_len* field of *io* and it is safe to assume that the entire address range mapped is contiguous for the interface adapter. A Consumer is not guaranteed to get the entire address range mapped by the implementation. Before calling *ri_mem_map*(), a Consumer must pin the virtual address range either by using *ri_mem_pin*() or some other mechanism outside the scope of this specification.

ri_mem_unmap() unmaps the mapping of a virtual memory range to a physical memory range *io* addressable by the specified interface adapter. *os_data* is an opaque parameter used exclusively by the OSV code to manage the state of the mapping represented by *io*. The opaque *os_data* used to unmap *io* with *ri_mem_unmap*() must be the same opaque used when *io* was previously mapped with *ri_mem_map*().

ri_mem_map() and *ri_mem_unmap*() can be invoked from an interrupt context.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.
RI_RET_NO_PERMISSION	Caller is not privileged.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_INVALID_PARAMETER	One of the parameters other than the RNIC handle is invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

If the call does not entirely map the virtual address range, the Consumer can call again to obtain a mapping of the remaining part. In this case, the caller should offset *vaddr* by the amount of bytes that was successfully mapped in the last call to *ri_mem_map*() and subtract *len* by the same amount.

IMPLEMENTATION NOTES

kAL and uAL can use *os_data* to pass state information that characterizes a particular virtual memory range being mapped. Examples of state information are the process context or the device.

SEE ALSO

ri_mem_map(), *ri_mem_unmap*(), *ri_svc_mem_pin*(), *ri_svc_mem_unpin*()

ri_mem_sync_rread()

NAME

ri_mem_sync_rread – make memory changes visible to an incoming RDMA read operation

SYNOPSIS

```
ri_api_return_t ri_mem_sync_rread(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_data_seg_t    *local_seg_list,
    IN   uint_t            num_seg,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC where the memory segments are registered.

local_seg_list
Pointer to array of memory segments.

num_seg Number of segments in the array.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_mem_sync_rread*() function is used with non-coherent Memory Regions to make memory updates visible to subsequent incoming RDMA read operations. The *ri_sync_rread* function is needed if and only if a Memory Region was created in non-coherent mode using the RI_MEM_FLAGS_NONCOHERENT flag. After modifications are made to a non-coherent Memory Region, *ri_mem_sync_rread*() must be called to ensure the updates will be visible before incoming RDMA read operations on that memory start. It is permissible to batch synchronizations for multiple subsequent RDMA reads in a single operation by passing a *local_seg_list* containing all the modified memory ranges.

If the *ri_mem_sync_rread*() function is not called as described above, the results of an RDMA read operation on a non-coherent Memory Region are undefined.

The *local_seg_list* argument contains a list of memory segments of length *num_seg*. Only segments with *ltag*s for non-coherent regions will be synchronized. Segments of coherent Memory Regions (i.e., created without the RI_MEM_FLAGS_NONCOHERENT flag) in the list are unaffected.

This call is a no-op and always returns successfully if the implementation does not support non-coherent mode or if none of the memory segments reference non-coherent Memory Regions.

The kernel version of this function is suitable for interrupt usage.

The definitions for *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	RNIC handle passed is invalid.

RI_RET_INVALID_PARAMETER	*local_seg_list* is NULL, when *num_seg* is non-zero.
RI_RET_MEM_INVALID_LTAG	Invalid LTAG in memory segment list.
RI_RET_MEM_INVALID_ADDRESS	Invalid address in memory segment list.
RI_RET_MEM_INVALID_LENGTH	Invalid length in memory segment list.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

This function is described in terms of RDMA usage for non-coherent memory. While this is the common usage, it is possible to use it to support other operations using non-coherent memory.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mem_sync_rwrite()

ri_mem_sync_rwrite()

NAME

ri_mem_sync_rwrite – make results of incoming RDMA write operation visible to Consumers

SYNOPSIS

```
ri_api_return_t ri_mem_sync_rwrite(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_data_seg_t    *local_seg_list,
    IN   uint_t            num_seg,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic Handle to the RNIC where the memory segments are registered.

local_seg_list
Pointer to array of memory segments.

num_seg Number of segments in the array.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_mem_sync_rwrite*() function is used with non-coherent Memory Regions to make the results of an incoming RDMA write operation visible to Consumers. The *ri_mem_sync_rwrite*() function is needed if and only if a Memory Region was created in non-coherent mode using the RI_MEM_FLAGS_NONCOHERENT flag. After an RDMA write operation has completed on a non-coherent Memory Region, *ri_mem_sync_rwrite*() must be called to ensure the updates will be visible in memory before Consumers attempt to read the affected memory. It is permissible to batch synchronizations for multiple RDMA writes in a single operation by passing a *local_seg_list* containing all the modified memory ranges.

If the *ri_mem_sync_rwrite*() function is not called as described above, the results of an RDMA write operation on a non-coherent Memory Region are undefined.

The *local_seg_list* argument contains a list of memory segments of length *num_seg*. Only segments with *ltag*s for non-coherent regions will be synchronized. Segments of coherent Memory Regions (i.e., created without the RI_MEM_FLAGS_NONCOHERENT flag) in the list are unaffected.

This call is a no-op and always returns successfully if the implementation does not support non-coherent mode or if none of the memory segments reference non-coherent Memory Regions.

The kernel version of this function is suitable for interrupt usage.

The definitions for *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RNIC_INVALID_HANDLE	RNIC handle passed is invalid.

RI_RET_INVALID_PARAMETER	*local_seg_list* is NULL, when *num_seg* is non-zero.
RI_RET_MEM_INVALID_LTAG	Invalid LTAG in memory segment list.
RI_RET_MEM_INVALID_ADDRESS	Invalid address in memory segment list.
RI_RET_MEM_INVALID_LENGTH	Invalid length in memory segment list.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

This function is described in terms of RDMA usage for non-coherent memory. While this is the common usage, it is possible to use it to support other operations using non-coherent memory.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mem_sync_rread()

ri_ah_create()

NAME

ri_ah_create – create a handle for the specified address vector to be used in all IB Unreliable Datagram (UD) communication

SYNOPSIS

```
ri_api_return_t ri_ah_create(
    IN      ri_rnic_handle_t         rnic,
    IN      ri_pd_handle_t           pd,
    IN      ri_ud_address_vector_t *av,
    OUT     ri_address_handle_t    *ah,
    IN OUT  ri_vp_data_t             vp_data,
    IN OUT  ri_os_data_t             os_data,
    OUT     ri_err_data_t          *err_data
);

typedef struct ri_ud_address_vector ri_ud_address_vector_t;
struct ri_ud_address_vector {
    ri_gen_address_vector_t  av_common;
    uint8_t                  physical_port;
    /* number of the local physical port */
};
```

DESCRIPTION

rnic Handle to RNIC on which to create address handle.

pd Handle to Protection Domain the address handle is to be associated with.

av Address vector describing the remote node address for UD sends.

ah Address handle returned.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the “syscall” version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_ah_create() creates an address handle corresponding to the address vector passed. The address handle returned can be used by the caller to reference local and global destinations in all UD QP Post Sends.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. A valid address handle is returned.
RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_PD_INVALID	The Protection Domain passed is invalid.
RI_RET_INVALID_PARAMETER	*av* or *ah* passed is NULL or *av* points to an invalid address vector.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_ah_destroy(), *ri_ah_modify*(), *ri_ah_query*()

ri_ah_query()

NAME

ri_ah_query – return the address vector and Protection Domain associated with the specified address handle

SYNOPSIS

```
ri_api_return_t ri_ah_query(
    IN   ri_rnic_handle_t         rnic,
    IN   ri_address_handle_t      ah,
    OUT  ri_ud_address_vector_t *av,
    OUT  ri_pd_handle_t          *pd,
    OUT  ri_err_data_t           *err_data
);
```

DESCRIPTION

rnic	Handle to RNIC on which to create address handle.
ah	Address handle.
av	Address vector describing the remote node address.
pd	Handle to Protection Domain the address handle is associated with.
err_data	Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_ah_query() queries the specified address handle and returns the address vector and Protection Domain it was created with through the *ri_ah_create*() call.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. Valid address vector and Protection Domain are returned.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_ADDRESS_HANDLE_INVALID	The address handle passed is invalid.
RI_RET_INVALID_PARAMETER	*av* or *pd* passed is NULL.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_ah_create(), ri_ah_destroy(), *ri_ah_modify*()

ri_ah_modify()

NAME

ri_ah_modify – associate a new address vector with the specified address handle

SYNOPSIS

```
ri_api_return_t ri_ah_modify(
    IN   ri_rnic_handle_t        rnic,
    IN   ri_address_handle_t     ah,
    IN   ri_ud_address_vector_t *av,
    OUT  ri_err_data_t          *err_data
);
```

DESCRIPTION

rnic	Handle to RNIC on which to create address handle.
ah	Address handle.
av	Address vector describing the remote node address for UD sends.
err_data	Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_ah_modify() changes the address vector associated with the address handle passed in by the caller.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The address vector associated with the specified address handle is successfully modified.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_ADDRESS_HANDLE_INVALID	The address handle passed is invalid.
RI_RET_INVALID_PARAMETER	*av* passed is NULL or *av* points to an invalid address vector.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_ah_create(), *ri_ah_destroy*(), *ri_ah_query*()

ri_ah_destroy()

NAME

ri_ah_destroy – destroy the specified address handle used for Unreliable Datagram (UD) communication

SYNOPSIS

```
ri_api_return_t ri_ah_destroy(
    IN   ri_rnic_handle_t     rnic,
    IN   ri_address_handle_t  ah,
    OUT  ri_err_data_t       *err_data
);
```

DESCRIPTION

rnic Handle to RNIC the address handle belongs to.

ah Address handle to be destroyed.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_ah_destroy() invalidates an address vector and its associated address handle. After the address handle is invalidated, it can no longer be used to reference any destination.

RETURN VALUE

RI_RET_SUCCESS	Successful completion. The address handle is successfully destroyed.
RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
RI_RET_ADDRESS_HANDLE_INVALID	The address handle passed is invalid.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_ah_create(), *ri_ah_modify*(), *ri_ah_query*()

ri_mcast_attach()

NAME

ri_mcast_attach – attach a QP to a multicast group

SYNOPSIS

```
ri_api_return_t ri_mcast_attach(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_lid_t          mlid,
    IN      ri_gid_t         *mgid,
    IN      ri_qp_handle_t    qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

mlid Multicast group MLID.

mgid Multicast group MGID.

qp QP handle of the QP to be attached to the specified multicast group.

vp_data Opaque data owned by IHV modules; OS modules should not try to interpret it. This information gets communicated from uVP to the "syscall" version of this function which in turn gets passed unchanged by kAL to kVP while processing this call. This must be zero when passed by uAL to uVP and uVPs must ignore this parameter.

os_data Opaque data owned by OS modules; IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to the "syscall" version of this function. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

Attaches the QP to the specified multicast group. The only function of this Verb is to assign the Receive Work Queue of this QP to the specified multicast group; after the attachment completes, this QP will be provided with a copy of every multicast message addressed to the group specified by the MGID and received on the RNIC port with which the QP is associated. Creation of the multicast group, and reconfiguration of the fabric such that packets addressed to that group are routed to a local RNIC port, is described in Section 7.10, *IBA and Raw Packet Multicast* of the InfiniBand Architecture Release 1.2, Volume 1 [IB-R1.2]. The Service Type of the specified QP must be Unreliable Datagram. It is an error to specify a QP with any other Service Type. One or more QPs are allowed to be attached to a multicast group on the RNIC. If the maximum number of multicast group attachments has already been reached for the RNIC when a QP attempts to attach to the multicast group, an error is returned. The input modifier which determines the multicast group to attach to are the MGID and MLID. Both input

modifiers must be supplied. The IB unreliable multicast feature is optional. This Verb is required only if IB unreliable multicast is supported by the RNIC.

RETURN VALUE

RI_RET_SUCCESS	Operation completed successfully.
RI_RET_RESOURCE_SHORTAGE	Insufficient resources to complete request.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_MGID_INVALID	Invalid multicast MLID.
RI_RET_MLID_INVALID	Invalid Multicast group MGID.
RI_RET_QP_INVALID_HANDLE	Invalid QP handle.
RI_RET_QP_INVALID_SERVICE_TYPE	Invalid Service Type for this QP.
RI_RET_RNIC_QPS_PER_MCAST_PORT_EXCEEDED	Number of QPs attached to multicast groups exceeded.
RI_RET_OP_UNSUPPORTED	Multicast not supported by RNIC.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mcast_detach()

ri_mcast_detach()

NAME

ri_mcast_detach – detach a QP from the multicast group

SYNOPSIS

```
ri_api_return_t ri_mcast_detach(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_lid_t          mlid,
    IN   ri_gid_t         *mgid,
    IN   ri_qp_handle_t    qp,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

mlid Multicast group MLID.

mgid Multicast group MGID.

qp QP handle.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

Detaches the specified QP from a multicast group. The only function of this Verb is to detach the Receive Work Queue of this QP from the specified multicast group. All of the input modifiers must be correct for the QP to be detached. If the QP is attached to a different multicast group or port, an error will be returned. This Verb is required only if IB unreliable multicast is supported by the RNIC. The IB unreliable multicast feature is optional.

RETURN VALUE

RI_RET_SUCCESS	Operation completed successfully.
RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
RI_RET_MLID_INVALID	Invalid multicast MLID.
RI_RET_MGID_INVALID	Invalid Multicast group MGID.
RI_RET_QP_INVALID_HANDLE	Invalid QP handle.
RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
RI_RET_OP_UNSUPPORTED	Multicast not supported by RNIC.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mcast_attach()

ri_process_sma_mad()

NAME

ri_process_sma_mad – process the InfiniBand SMA MAD request

SYNOPSIS

```
ri_api_return_t ri_process_sma_mad(
    IN   ri_rnic_handle_t  rnic,
    IN   uint8_t           port_num,
    IN   char             *in_mad,
    OUT  char             *out_mad,
    OUT  ri_err_data_t    *err_data
);
```

DESCRIPTION

rnic RNIC handle.

port_num Port number on which the MAD is received.

in_mad Request MAD.

out_mad Response MAD.

err_data Data passed in all calls. How the recepient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

ri_process_mad can be used to build IHV-independent InfiniBand Subnet Management Agent. The SMA can pass the MAD message received on its UD QPs and get the response MAD built by the IHV code. The response MAD can then be sent back using the hardware-independent *ri_qp_postsq*() or its equivalent call. If *in_mad* is a *SubnGet*/*SubnSet* request, MAD and *out_mad* would be the corresponding response *SubnGetResp* MAD. If a *SubnTrap* message is originated in the CI without the knowledge of the Consumer, the message will be made available on the Receive Queue of QP0. Upon receiving the message, the Consumer (SMA) could forward the message to the SM and get back a *SubnTrapRepress* MAD. The SMA later passes on *SubnTrapRepress* MAD through *ri_process_sma_mad*() just to notify the HCA of SM acknowledgement. *out_mad* is undefined if *in_mad* contains *SubnTrapRepress*. If the trap is initiated by the Consumer, *SubnTrapRepress* MAD shall not be passed to *ri_process_sma_mad*().

This call can be invoked from an interrupt context.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_XPORT_UNSUPPORTED	Attempt to do the operation on a non-InfiniBand device.
RI_RET_FAIL	The request MAD could not be processed.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_svc_attach()

NAME

ri_svc_attach – attach an RNIC to the RDMA access layer

SYNOPSIS: KERNEL

```
ri_api_return_t ri_svc_attach(
    IN   ri_vp_handle_t     vp_hdl,
    IN   ri_dev_info_t      dev_info,
    IN   ri_svcs_dev_arg_t *svcs,
    IN   ri_ops_t          *al_locking_ops,
    IN   ri_ops_t          *vp_locking_ops,
    OUT  ri_err_data_t     *err_data
);

typedef struct ri_svcs_dev_arg ri_svcs_dev_arg_t;
struct ri_svcs_dev_arg {
    /* Major version of RNICPI implemented */
    uint32_t rnicpi_major_version;

    /* Minor version of RNICPI implemented */
    uint32_t rnicpi_minor_version;

    /* Major version of RNIC firmware */
    uint32_t fw_major_version;

    /* Minor version of RNIC firmware */
    uint32_t fw_minor_version;

    /* Sub minor version of RNIC firmware */
    uint32_t fw_sub_minor_version;

    /* size of buffer to allocate for query_rnic in bytes */
    uint_t query_size;

    /*
     * NULL terminated string pathname of the uVP shared
     * library. This will be used later to load the
     * appropriate uVP library.
     */
    char lib_path[RI_MAX_LIB_PATH_LEN];

    /*
     * NULL terminated string of hardware path of the device
     * on the OSV environment. This could be used for
     * diagnostics or for persistent identification.
     */
    char hw_path[RI_MAX_HW_PATH_LEN];
};
```

DESCRIPTION

vp_hdl Verb provider handle to the RNIC. This handle must be unique for each RNIC.

dev_info OS-specific device identifier.

svcs High-level information about the device, its interfaces, and its compatible software.

al_locking_ops
Set of pointers to functions implementing the RNICPI operations using access layer enforced serialization.

vp_locking_ops
Set of pointers to functions implementing the RNICPI operations using verb provider layer enforced serialization. This vector is optional and, if not supported, this argument will be NULL.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_svc_attach*() function is an upcall from the RNIC driver to the OS RDMA access layer. The function "attaches" or registers the device with the RDMA access layer. Without calling this function, the OS may be unaware of a device's RDMA ability. The *vp_hdl* is an opaque handle used to uniquely identify the RNIC device. It is required that this handle be unique for each RNIC. The *vp_hdl* is also used in subsequent calls such as *ri_rnic_open*() and *ri_svc_detach*(). The *dev_info* argument is used by the OS to identify the device or network interface. Necessarily, it is an OS-dependent type. The *svcs* argument describes the version of the device and its driver interfaces. The RDMA access layer may use this information passed for diagnostic purposes. The values passed for the *rnicpi_major_version* and *rnicpi_minor_version* fields of *svcs* correspond to version of the RNICPI specification that the VP implements. The device driver passes the set of operation functions (serialized through the AL) in the *al_locking_ops* argument. Another vector of operations using VP enforced serialization is optionally provided in the *vp_locking_ops* argument (NULL if not provided). The *query_size* field of *svcs* is the size of a single buffer capable of containing all the information returned from *ri_rnic_query*(), including the main *ri_rnic_attr_t* struct, the transport-specific struct, and any related port information structs.

The RDMA infrastructure may start invoking the operation entry points even from within an invocation of *ri_svc_attach*(). In particular, *ri_svc_attach*() should be prepared for the possibility that *ri_rnic_open*() will be called from within the invocation of this function. The RI_RET_RNIC_INUSE error is returned if an attempt it made to attach the same RNIC (as identified through the *vp_rnic_handle)* more than once without first detaching it through an *ri_svc_detach*() call.

This function only exists in the kernel. There is no user-level version of this function. User-level initialization uses the *ri_svc_lib_reg*() function.

Definitions for *ri_vp_handle_t*, *ri_dev_info_t*, *ri_err_data_t,* RI_MAX_HW_PATH_LEN, and RI_MAX_LIB_PATH_LEN are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Device attached to RDMA access layer successfully.
RI_RET_RESOURCE_SHORTAGE	No resources available to attach the device.

RI_RET_VP_INVALID_HANDLE	Verb provider handle passed is invalid.
RI_RET_INVALID_DEV_INFO	*dev_info* parameter is invalid.
RI_RET_INVALID_PARAMETER	*svcs,* and/or *al_locking_ops* parameter is NULL.
RI_RET_INVALID_HW_PATH	*hw_path* in *svcs* is invalid.
RI_RET_INVALID_LIB_PATH	*lib_path* in *svcs* is invalid.
RI_RET_INVALID_VP_VERSION	Verb provider version is incompatible with that of the OS infrastructure.
RI_RET_RNIC_INUSE	RNIC is already attached.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

As stated above, it is required that *ri_svc_attach*() can uniquely distinguish each RNIC through the VP handle. This requirement means there is an agreement between AL (and/or OS) and the IHVs about the verb provider (VP) handle implementation, so that each IHV can supply a VP handle independently that will not clash with other IHV VP handles. Typically, this is accomplished by the VP returning either an identifier previously given to it by the AL and/or OS (e.g., OS-specific device identifier) or by using a pointer to memory that has been privately allocated for each VP. Because there is an agreement about the definition of the VP RNIC handle, it may be possible that an incorrect one could be recognized, which will cause the RI_RET_VP_INVALID_HANDLE error.

Another area requiring an agreement with the OS is the *dev_info* argument which is meant to be an OS-specific identifier/handle for the device or network interface. This allows the OS to properly coordinate the usage of the RNIC as a device and network interface as well.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_close*(), *ri_rnic_free_hdl*(), *ri_rnic_get_hdl*(), *ri_rnic_open*(), *ri_svc_detach*(), *ri_svc_lib_reg*(), *ri_ops_t*

ri_svc_detach()

NAME

ri_svc_detach – detach an RNIC from the RDMA access layer

SYNOPSIS: KERNEL

```
ri_api_return_t ri_svc_detach(
    IN   ri_vp_handle_t  vp_hdl,
    OUT  ri_err_data_t  *err_data
);
```

DESCRIPTION

vp_hdl Verb provider handle to the RNIC. This is the same handle passed in the *ri_svc_attach*() function.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_svc_detach*() function is an upcall from the RNIC driver to the RDMA access layer. This function "detaches" or deregisters the RNIC from the RDMA access layer. The *vp_hdl* argument uniquely specifies the RNIC device being detached from the RDMA access layer.

Upon successful return of *ri_svc_detach*(), the caller should not attempt to use the *vp_hdl* for any other function call. This function should only be called after a successful invocation of *ri_rnic_close*(). Otherwise, the RI_RET_RNIC_INUSE error is returned. The RDMA access layer shall be able to handle this call anytime after *ri_rnic_close*() returns.

This function only exists in the kernel. Similar functionality at user level uses other mechanisms.

The definitions for *ri_vp_handle_t* and *ri_err_data_t* are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_VP_INVALID_HANDLE	Verb provider handle to RNIC is invalid.
RI_RET_RNIC_INUSE	RNIC is still open.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_async_set_event_handler(), *ri_rnic_close*(), *ri_rnic_free_hdl*(), *ri_rnic_get_hdl*(), *ri_rnic_open*(), *ri_svc_attach*()

ri_svc_lib_reg()

NAME

ri_svc_lib_reg – register a user verb provider library with the user access layer

SYNOPSIS: USER

```
ri_api_return_t ri_svc_lib_reg(
    IN      char          *lib_path,
    IN      ri_ops_t      *al_locking_ops,
    IN      ri_ops_t      *vp_locking_ops,
    IN OUT  ri_os_data_t   os_data,
    OUT     ri_err_data_t *err_data
);
```

DESCRIPTION

lib_path Library pathname used by the uAL to identify the devices of this type. The pathname pointed to by this argument should be no longer than RI_MAX_LIB_PATH chars (including terminating NULL).

al_locking_ops
Set of pointers to functions implementing the RNICPI operations using access layer enforced serialization.

vp_locking_ops
Set of pointers to functions implementing the RNICPI operations using verb provider layer enforced serialization. This vector is optional and if not supported, this argument will be NULL.

os_data Opaque data owned by OS modules. IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged via "syscall". Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed on is also not defined. Some interested OSVs and IHVs may use it for private communication.

The *ri_svc_lib_reg*() function is an upcall used to register the user-level verb provider library with the user access layer. Once this function has been called, the uAL may call *ri_rnic_get_hdl*() on available RNICs that use this library. The *lib_path* argument identifies the library being loaded to the uAL. The *al_locking_ops* parameter is the operations vector exported by the library that implements AL enforced locking. An optional second VP enforced locking operations vector may be present in *vp_locking_ops*.

This function is called once for each library loaded. If there are several RNIC instances that use the same library, this function is still only called once for that library.

The *lib_path* should match a library name provided to the kAL by at least one currently open RNIC (see *ri_rnic_open*()). Library names are passed to the kAL through the *lib_path* field in the *svcs* argument of *ri_svc_attach*(). If the library path does not match a known library name, the RI_RET_INVALID_LIB_PATH error is returned.

This function is only available at user level. The kernel uses a different initialization mechanism (see *ri_svc_attach*()).

Definitions for *ri_os_data_t*, *ri_err_data_t,* and RI_MAX_LIB_PATH_LEN are OS-dependent.

RETURN VALUE

RI_RET_SUCCESS	Library registered with RDMA access layer successfully.
RI_RET_RESOURCE_SHORTAGE	No resources available to complete the operation.
RI_RET_INVALID_LIB_PATH	Library pathname does not match known library.
RI_RET_INVALID_PARAMETER	*al_locking_ops* parameter is NULL.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

Typically, the behavior of being called once per library is accomplished by calling this function from the "initializer" entry point of the library being dynamically loaded.

IMPLEMENTATION NOTES

How the uAL is informed of available RNICs which have successfully called *ri_svc_attach*() and have been opened (via *ri_rnic_open*()) in the kernel is outside the scope of RNICPI. Typically, there will be private syscall communication between uAL and kAL to facilitate this.

SEE ALSO

ri_rnic_get_hdl(), *ri_rnic_open*(), *ri_svc_attach*()

ri_svc_mem_pin(), ri_svc_mem_unpin()

NAME

ri_svc_mem_pin – pin a range of virtual memory

ri_svc_mem_unpin – unpin a range of virtual memory

SYNOPSIS: KERNEL

```
ri_api_return_t ri_svc_mem_pin (
    IN      ri_vaddr_t     vaddr,
    IN      ri_length_t    len,
    IN OUT  ri_os_data_t   os_data,
    OUT     ri_err_data_t *err_data
);

ri_api_return_t ri_svc_mem_unpin (
    IN      ri_vaddr_t     vaddr,
    IN      ri_length_t    len,
    IN OUT  ri_os_data_t   os_data,
    OUT     ri_err_data_t *err_data
);
```

DESCRIPTION

vaddr Virtual address.

len Length of the virtual address range in bytes.

os_data Opaque data owned by OS modules, IHV modules should not try to interpret it. This information gets communicated from uAL to uVP which in turn gets passed unchanged to *ri_syscall*. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to appropriate calls back into the OS.

err_data Data passed in all calls. How the recipient of the information uses it is outside the scope of RNICPI. The relation between the value received and later passed back is also not defined. Some interested OSVs and IHVs may use it for private communication.

These functions allow the verbs provider to pin and unpin a virtual memory range *vaddr* with length *len*.

Multiple *ri_svc_mem_pin*() calls can be made on the same or overlapping memory ranges with all pin reference counting issues properly handled.

The address range of an *ri_svc_mem_unpin*() call must be the same as the *ri_svc_mem_pin*() call originally used to pin the memory.

os_data is an opaque parameter used exclusively by the OSV code to manage the state of the virtual memory range *vaddr*. The opaque *os_data* used to unpin *vaddr* with *ri_svc_mem_unpin*() must be the same *os_data* opaque used when *vaddr* was previously pinned with *ri_svc_mem_pin*(). An OSV may also require that the *os_data* field used in these functions be the same as the *os_data* opaque passed by the RNICPI downcall that uses this service (e.g., the *os_data* passed by *ri_mr_reg*()).

These functions apply only to privileged callers and should only be invoked by the kVP.

RETURN VALUE

RI_RET_SUCCESS	Successful completion.
RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.
RI_RET_NO_PERMISSION	Caller is not privileged.
RI_RET_INVALID_PARAMETER	One of the parameters is invalid.
RI_RET_FAIL	Unknown/fatal error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mem_map(), *ri_mem_pin*(), *ri_mem_unmap*(), *ri_mem_unpin*()

8 Data Type Reference Pages

ri_api_return_t

NAME

ri_api_return_t – function call status return codes

SYNOPSIS

```
typedef enum ri_api_return ri_api_return_t
enum ri_api_return {
    RI_RET_SUCCESS = 0,

    /* generic errors */
    RI_RET_RESOURCE_SHORTAGE,
    RI_RET_INVALID_PARAMETER,
    RI_RET_NO_PERMISSION,
    RI_RET_XPORT_UNSUPPORTED,
    RI_RET_OP_UNSUPPORTED,
    RI_RET_BUFFER_TOO_SMALL,
    RI_RET_FAIL,

    /* options */
    RI_RET_QP_RESIZE_UNSUPPORTED,
    RI_RET_SRQ_UNSUPPORTED,
    RI_RET_SRQ_RESIZE_UNSUPPORTED,
    RI_RET_EXT_BMM_UNSUPPORTED,
    RI_RET_EXT_ZBVA_UNSUPPORTED,
    RI_RET_EXT_LIF_UNSUPPORTED,
    RI_RET_RNIC_ATOMICS_UNSUPPORTED,

    /* VP */
    RI_RET_VP_INVALID_HANDLE,
    RI_RET_INVALID_DEV_INFO,
    RI_RET_INVALID_HW_PATH,
    RI_RET_INVALID_LIB_PATH,
    RI_RET_INVALID_PBL_MODE,
    RI_RET_INVALID_VP_VERSION,

    /* RNIC */
    RI_RET_RNIC_INVALID_HANDLE,
    RI_RET_RNIC_INUSE,
    RI_RET_RNIC_NOT_OPEN,

    /* PD */
    RI_RET_PD_INVALID_HANDLE,
    RI_RET_PD_INUSE,

    /* CQ */
    RI_RET_CQ_INVALID_HANDLE,
    RI_RET_CQ_CAPACITY_EXCEEDED,
    RI_RET_CQ_EMPTY,
    RI_RET_CQ_ERROR,
    RI_RET_CQ_INUSE,
    RI_RET_CQ_INVALID_EVENT_HANDLER,
    RI_RET_CQ_NO_EVENT_HANDLER,
    RI_RET_CQ_WOULD_OVERFLOW,
```

```
5118                RI_RET_INVALID_HANDLER_ID,
5119
5120                /* QP */
5121                RI_RET_QP_INVALID_HANDLE,
5122                RI_RET_MIGRATION_STATE_INVALID,
5123                RI_RET_QP_ATTRIBUTE_CANT_CHANGE,
5124                RI_RET_QP_CAPACITY_EXCEEDED,
5125                RI_RET_QP_INVALID_SERVICE_TYPE,
5126                RI_RET_QP_INVALID_SPECIAL_TYPE,
5127                RI_RET_QP_INVALID_STATE,
5128                RI_RET_QP_CANNOT_SHRINK,
5129                RI_RET_QP_INUSE,
5130                RI_RET_RNIC_INVALID_MTU,
5131                RI_RET_RNIC_INVALID_PKEY_ENTRY,
5132                RI_RET_RNIC_INVALID_PKEY_INDEX,
5133                RI_RET_RNIC_INVALID_PORT,
5134                RI_RET_RNIC_INVALID_RNR_VALUE,
5135                RI_RET_RNIC_IRD_ORD_EXCEEDED,
5136                RI_RET_RNIC_NOT_IN_BLOCK_MODE,
5137                RI_RET_RNIC_NOT_IN_PAGE_MODE,
5138                RI_RET_RNIC_SG_LIMIT_EXCEEDED,
5139                RI_RET_RNIC_WR_LIMIT_EXCEEDED,
5140                RI_RET_STILL_FLUSHING_WQES,
5141                RI_RET_WR_INVALID_FLAGS,
5142                RI_RET_WR_INVALID_TYPE,
5143                RI_RET_WR_SG_INVALID_FORMAT,
5144
5145                /* SRQ */
5146                RI_RET_SRQ_INVALID_HANDLE,
5147                RI_RET_QP_TIED_TO_SRQ,
5148                RI_RET_SRQ_CANNOT_SHRINK,
5149                RI_RET_SRQ_IN_ERROR,
5150                RI_RET_SRQ_INUSE,
5151                RI_RET_SRQ_LIMIT_OUT_OF_RANGE,
5152
5153                /* MR */
5154                RI_RET_MR_INVALID_HANDLE,
5155                RI_RET_MEM_INVALID_ACCESS,
5156                RI_RET_MEM_INVALID_ADDRESS,
5157                RI_RET_MEM_INVALID_LENGTH,
5158                RI_RET_MEM_INVALID_OFFSET,
5159                RI_RET_MEM_INVALID_LTAG,
5160                RI_RET_MEM_INVALID_PBLE,
5161                RI_RET_MEM_INVALID_PBLE_LENGTH,
5162                RI_RET_MEM_INVALID_RTAG,
5163
5164                /* MW */
5165                RI_RET_MW_INVALID_HANDLE,
5166                RI_RET_INVALID_WINDOW_TYPE,
5167                RI_RET_WINDOWS_BOUND,
5168                RI_RET_WINDOW_TYPE_UNSUPPORTED,
5169
5170                /* AH */
5171                RI_RET_ADDRESS_HANDLE_INVALID,
```

```
        RI_RET_MGID_INVALID,
        RI_RET_MLID_INVALID,
        RI_RET_MULTICAST_ATTACHED,
        RI_RET_RNIC_QPS_PER_MCAST_PORT_EXCEEDED,

        /* LLP */
        RI_RET_INVALID_LLP_HANDLE,
        RI_RET_INVALID_DDP_RDMAP_VERSION,
        RI_RET_LLP_BAD_STATE,
        RI_RET_MPA_ATTR_UNSUPPORTED
    };
```

DESCRIPTION

The *ri_api_return_t* definition includes all function return values used in the RNICPI specification.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_ops_t

NAME

ri_ops_t – RNICPI operations vector

SYNOPSIS

```
/* Definition of the vector of function pointers. */
typedef struct ri_ops ri_ops_t;
struct ri_ops {
    ri_op_rnic_open_t                ri_op_rnic_open;
    ri_op_rnic_close_t               ri_op_rnic_close;
    ri_op_async_set_event_handler_t  ri_op_async_set_event_handler;
    ri_op_rnic_get_hdl_t             ri_op_rnic_get_hdl;
    ri_op_rnic_query_t               ri_op_rnic_query;
    ri_op_rnic_modify_t              ri_op_rnic_modify;
    ri_op_rnic_free_hdl_t            ri_op_rnic_free_hdl;
    ri_op_rnic_diag_t                ri_op_rnic_diag;
    ri_op_pd_alloc_t                 ri_op_pd_alloc;
    ri_op_pd_dealloc_t               ri_op_pd_dealloc;
    ri_op_cq_create_t                ri_op_cq_create;
    ri_op_cq_query_t                 ri_op_cq_query;
    ri_op_cq_modify_t                ri_op_cq_modify;
    ri_op_cq_destroy_t               ri_op_cq_destroy;
    ri_op_special_qp_get_t           ri_op_special_qp_get;
    ri_op_cq_set_event_handler_t     ri_op_cq_set_event_handler;
    ri_op_cq_req_notification_t      ri_op_cq_req_notification;
    ri_op_cq_poll_t                  ri_op_cq_poll;
    ri_op_qp_create_t                ri_op_qp_create;
    ri_op_qp_query_t                 ri_op_qp_query;
    ri_op_qp_modify_t                ri_op_qp_modify;
    ri_op_qp_destroy_t               ri_op_qp_destroy;
    ri_op_qp_postsq_t                ri_op_qp_postsq;
    ri_op_qp_post_send_t             ri_op_qp_post_send;
    ri_op_qp_post_rdma_read_t        ri_op_qp_post_rdma_read;
    ri_op_qp_post_rdma_write_t       ri_op_qp_post_rdma_write;
    ri_op_qp_post_bind_t             ri_op_qp_post_bind;
    ri_op_qp_post_fmr_t              ri_op_qp_post_fmr;
    ri_op_qp_post_local_invalidate_t ri_op_qp_post_local_invalidate;
    ri_op_qp_post_atomic_t           ri_op_qp_post_atomic;
    ri_op_qp_post_datagram_t         ri_op_qp_post_datagram;
    ri_op_qp_postrq_t                ri_op_qp_postrq;
    ri_op_qp_post_recv_t             ri_op_qp_post_recv;
    ri_op_srq_create_t               ri_op_srq_create;
    ri_op_srq_query_t                ri_op_srq_query;
    ri_op_srq_modify_t               ri_op_srq_modify;
    ri_op_srq_destroy_t              ri_op_srq_destroy;
    ri_op_srq_post_t                 ri_op_srq_post;
    ri_op_srq_post_recv_t            ri_op_srq_post_recv;
    ri_op_ltag_alloc_t               ri_op_ltag_alloc;
    ri_op_mr_reg_t                   ri_op_mr_reg;
    ri_op_mr_query_t                 ri_op_mr_query;
    ri_op_mr_rereg_t                 ri_op_mr_rereg;
    ri_op_mr_dereg_t                 ri_op_mr_dereg;
```

```
        ri_op_mr_reg_phys_t             ri_op_mr_reg_phys;
        ri_op_mr_rereg_phys_t           ri_op_mr_rereg_phys;
        ri_op_mr_reg_shared_t           ri_op_mr_reg_shared;
        ri_op_mw_alloc_t                ri_op_mw_alloc;
        ri_op_mw_dealloc_t              ri_op_mw_dealloc;
        ri_op_mw_query_t                ri_op_mw_query;
        ri_op_mem_pin_t                 ri_op_mem_pin;
        ri_op_mem_unpin_t               ri_op_mem_unpin;
        ri_op_mem_map_t                 ri_op_mem_map;
        ri_op_mem_unmap_t               ri_op_mem_unmap;
        ri_op_mem_sync_rread_t          ri_op_mem_sync_rread;
        ri_op_mem_sync_rwrite_t         ri_op_mem_sync_rwrite;
        ri_op_ah_create_t               ri_op_ah_create;
        ri_op_ah_query_t                ri_op_ah_query;
        ri_op_ah_modify_t               ri_op_ah_modify;
        ri_op_ah_destroy_t              ri_op_ah_destroy;
        ri_op_mcast_attach_t            ri_op_mcast_attach;
        ri_op_mcast_detach_t            ri_op_mcast_detach;
        ri_op_process_sma_mad_t         ri_op_process_sma_mad;
};
```

DESCRIPTION

The *ri_ops_t* structure defines the operations that a verb provider implementation supplies. For each "downcall" from AL to VP, there is a corresponding function pointer. This function call vector definition is used for both uVP and kVP. There are some functions which are not intended to be used at both kernel and user level. In these cases, a stub function is provided for the inappropriate context that always returns an error.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_svc_attach(), *ri_svc_lib_reg*()

ri_rnic_attr_t

NAME

ri_rnic_attr_t – RNIC attribute data structures

SYNOPSIS

```
typedef enum ri_rdma_transport_type ri_rdma_transport_type_t;
enum ri_rdma_transport_type {
    RI_RDMA_TRANSPORT_TYPE_IB,
    RI_RDMA_TRANSPORT_TYPE_IWARP
};

typedef union ri_transport_attr_ptr ri_transport_attr_ptr_t;
union ri_transport_attr_ptr {
    ri_ib_rnic_attr_t *ib;
    ri_iwarp_rnic_attr_t *iwarp;
};

typedef enum ri_rnic_flags ri_rnic_flags_t;
enum ri_rnic_flags {
    /* iWARP will always have BMM, ZBVA & LIF set */
    RI_RNIC_EXT_SRQ = 1 << 0,  /* IB 1.1: not set */
    RI_RNIC_EXT_BMM = 1 << 1,  /* IB 1.1: not set */

    /* note that ZBVA, LIF & BLOCK_MODE require BMM */
    RI_RNIC_EXT_ZBVA = 1 << 2,  /* IB 1.1: not set */
    RI_RNIC_EXT_LIF = 1 << 3,  /* IB 1.1: not set */

    /* if BLOCK_MODE not set, RNIC is in page mode */
    RI_RNIC_EXT_BLOCK_MODE = 1 << 4,  /* IB 1.1: not set */

    RI_RNIC_RQ_OVERFLOW_CHECKING = 1 << 5,  /* IB: set */
    RI_RNIC_CQ_OVERFLOW_CHECKING = 1 << 6,  /* IB: set */

    RI_RNIC_CAN_RESIZE_QP = 1 << 7,

    /* if RI_RNIC_EXT_SRQ not set, then SRQ stuff not defined */
    RI_RNIC_CAN_RESIZE_SRQ = 1 << 8,
    RI_RNIC_SRQ_SEQ_DEQUEUE = 1 << 9,  /* IB SRQ: not set */
    RI_RNIC_SRQ_LAST_WQE_ASYNC = 1 << 10,  /* IB SRQ: set */

    /*
     * safe to post on passive side after successful modify
     * to RTS
     */
    RI_RNIC_SAFE_POST_ON_RTS = 1 << 11,  /* IB: set */

    /* narrow windows support is part of RI_RNIC_EXT_BMM */
    RI_RNIC_WIDE_WINDOWS_SUPPORTED = 1 << 12,  /* IB: set */
    /*
     * extended key splitting
     * wide windows and virtual MRs also use index/key splitting
     */
```

```
5326                RI_RNIC_EXT_INDEX_KEY_SPLIT = 1 << 13,  /* IB 1.1: not set */
5327
5328                /* supported WR/WC opaque bits */
5329                RI_RNIC_CONSUMER0_OPAQUE = 1 << 14,
5330                RI_RNIC_CONSUMER1_OPAQUE = 1 << 15,
5331                RI_RNIC_CONSUMER2_OPAQUE = 1 << 16,
5332
5333                /*
5334                 * if local solicited is pass thru opaque, then
5335                 * LOCAL_SOLICITED_OPAQUE is defined,
5336                 * which is the same as CONSUMER2_OPAQUE.
5337                 * if local solicited is a real event, then
5338                 * LOCAL_SOLICITED_EVENT is defined.
5339                 * if NO local solicited support, then neither
5340                 * is defined.
5341                 */
5342                RI_RNIC_LOCAL_SOLICITED_OPAQUE = 1 << 16,
5343                RI_RNIC_LOCAL_SOLICITED_EVENT = 1 << 17,
5344
5345                /* non-coherent memory support */
5346                RI_RNIC_NON_COHERENT_MEMORY = 1 << 18
5347            };
5348
5349            typedef struct ri_rnic_attr ri_rnic_attr_t;
5350            struct ri_rnic_attr {
5351
5352                /* 64-bit and pointer types here to minimize padding */
5353
5354                /*
5355                 * supported memory page sizes bitmask for page mode,
5356                 * bit position i means page size 2^i size is supported,
5357                 * undefined if using block mode
5358                 */
5359                uint64_t  mem_page_sizes_bit_mask;
5360
5361                /* block mode limits, undefined if using page mode */
5362                uint64_t  block_size_low;  /* IB 1.1: undefined */
5363                uint64_t  block_size_hi;  /* IB 1.1: undefined */
5364
5365                /* transport-specific items */
5366                ri_transport_attr_ptr_t  trans_attr;
5367
5368                /*
5369                 * RNIC identification info
5370                 */
5371                uint32_t  vendor_id;  /* Vendor ID (IB: 24 bits)*/
5372                uint32_t  vendor_part_id;  /* Part ID (IB: 16 bits)*/
5373                uint32_t  hw_version;  /* Hardware Version */
5374
5375                /* reserved L_Key or stag0 (which is zero, of course) */
5376                uint32_t  res_ltag_value;  /* IB 1.1: undefined */
5377            #define stag0_value res_lkey_value
5378
```

```
5379                /*
5380                 * resource limits
5381                 */
5382
5383                uint_t  max_pds;  /* max num of Protection Domains */
5384
5385                uint32_t  max_qps;  /* max num of QPs (IB: 24 bits) */
5386                uint_t    max_wrs_qp;  /* max num of WRs on a WQ */
5387
5388                /* max number of SRQs, if SRQs unsupported: zero */
5389                uint_t  max_srq;  /* IB 1.1: zero */
5390                /*
5391                 * max num of outstanding WRs on a SRQ,
5392                 * if SRQs, unsupported: zero
5393                 */
5394                uint_t  max_wrs_srq;  /* IB 1.1: zero */
5395
5396                /*
5397                 * max number of scatter/gather entries per WR,
5398                 * for IB: send_recvqp = rdma_write = rdma_read,
5399                 * for iWARP: send_recvqp = srq and >= 4
5400                 */
5401                uint_t  max_sgl_send_recvqp;  /* send/recv on QP */
5402                uint_t  max_sgl_srq;  /* for SRQ */
5403                uint_t  max_sgl_rdma_write;  /* RDMA-W */
5404                uint_t  max_sgl_rdma_read;  /* RDMA-R */
5405
5406                /* largest inline size supported, zero if unsupported */
5407                uint32_t  max_unregistered_inline;
5408
5409                uint_t  max_cqs;  /* max number of CQs */
5410                uint_t  max_cqes_per_cq;  /* max number of CQEs per CQ */
5411
5412                /* max number of CQ event handlers */
5413                uint_t  max_cq_event_handlers;  /* IB 1.1: one */
5414
5415                uint_t  max_mrs;  /* max number of MRs */
5416                uint_t  max_mws;  /* max number of MWs */
5417
5418                /* max number of physical buffer entries per PBL */
5419                uint_t  max_pbl_size;  /* IB 1.1: undefined */
5420
5421                /* max number of RDMA-Reads & atomic ops */
5422                uint_t  max_ird;  /* incoming for whole RNIC */
5423                uint_t  max_ird_per_qp;  /* incoming per QP */
5424                uint_t  max_ord;  /* outgoing for whole RNIC */
5425                uint_t  max_ord_per_qp;  /* outgoing per QP */
5426
5427                /* IB or IWARP */
5428                ri_rdma_transport_type_t  transport_type;
5429
5430                ri_rnic_flags_t  rnic_flags;  /* various boolean flags */
5431
```

```
        /* Additional Vendor Info (NULL terminated) */
        char addl_vendor_info[RI_MAX_VENDOR_STRING_LEN];
};
```

DESCRIPTION

The *ri_rnic_attr_t* struct defines the transport-independent attributes of an RNIC. This structure is returned in *ri_rnic_query*(). Transport-specific attributes are in substructures accessed through the *trans_attr* pointer field. The type of the transport-specific struct will be *ri_ib_rnic_attr_t* if the value of *transport_type* is RI_RDMA_TRANSPORT_TYPE_IB. For RI_RDMA_TRANSPORT_TYPE_IWARP, the type of the transport-specific structure is *ri_iwarp_rnic_attr_t*.

The maximum values defined in this section are guaranteed not-to-exceed values. It is possible for an implementation to allocate various RNIC resources from a shared pool. In that case, all maximum values cannot be achieved simultaneously.

A number of optional features are flagged in *rnic_flags*. Several of the flags are named with the RI_RNIC_EXT_ prefix indicating that they are regarded as "verb extensions" in InfiniBand Version 1.2. RI_RNIC_EXT_SRQ indicates support for Shared Receive Queues. RI_RNIC_EXT_BMM indicates support for the "Memory Management" extensions including the use of *ri_ltag_alloc*(), fast register WRs, local invalidation WRs, remote invalidation WRs, *reserved_lkey*/Stag0, Narrow Memory Windows, and the explicit designation of whether a Memory Region is shared or not. RI_RNIC_EXT_ZBVA indicates support for "Zero Based" virtual addresses. RI_RNIC_EXT_LIF indicates support for Local Invalidate Fencing. RI_RNIC_BLOCK_MODE indicates support for "Block mode" physical buffer lists. The following RI_RNIC_EXT_ items are dependent on RI_RNIC_EXT_BMM being supported: ZBVA, LIF, and BLOCK_MODE. A strictly 1.1 implementation of InfiniBand will not have support for any of the RI_RNIC_EXT_ items. An iWARP implementation will always indicate support for the following RI_RNIC_EXT_ items: BMM, ZBVA, and LIF.

Note that there is no flag for the InfiniBand 1.2 "Queue Management" extension. If the underlying RNIC is not capable of posting more than one WR at a time, then the posting functions are required to iterate over the list of WRs posting one at a time. If the RNIC does not support multiple completion event handlers, then the number of supported completion handlers is one.

If the RI_RNIC_SAFE_POST_ON_RTS flag is set, then it is safe for the passive side to do an SQ post immediately after transitioning the QP to RTS. If not set, the Consumer must refrain from posting to the SQ until after a receive operation completes on the QP.

"Narrow windows" are RDMAC or IB type2 style windows which are associated with a single connection when bound. "Wide windows" are IB type1 windows which are access checked against a PD.

If the RI_RNIC_EXT_INDEX_KEY_SPLIT flag is set, then *LTags* and *RTags* can always use the index/key split scheme even when not otherwise required for virtually registered Memory Regions (through the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag) and "wide" Memory Windows. Even when not set, *Ltags* associated with fast memory registers and *RTags* associated with Narrow Memory Windows use the index/key split scheme.

If the RI_RNIC_CONSUMER*x*_OPAQUE flags are set, the corresponding pass-through flags from the WR are returned in the WC. Note that RI_RNIC_CONSUMER2_OPAQUE is used to

support local solicited behavior and has the same definition as RI_RNIC_LOCAL_SOLICITED_OPAQUE.

If the RI_RNIC_LOCAL_SOLICITED_OPAQUE flag is set, then a pass-through flag from the WR is returned in the WC. This pass-through flag is to be used by the AL to emulate local solicited behavior (i.e., generate an event). If the RI_RNIC_LOCAL_SOLICTED_EVENT flag is set, then the RNIC actually generates a completion event when completions with the local solicited flag are generated. If neither flag is set, then local solicited behavior is not supported.

InfiniBand does not allow for out-of-order receives for the Reliable Connected (RC) transport service type, so strictly speaking SRQ sequential *versus* arrival dequeuing order does not directly apply to RC SRQs. An out-of-order receive for an RC SRQ does not dequeue WQEs for either the arriving message (unlike iWARP) or for messages in the sequence gap. InfiniBand also allows UD SRQs, which don't have any ordering guarantees due to the nature of UD. So an out-of-order receive in this case acts more like arrival ordering, though there is no concept of a sequence gap for UD. Because of these two behaviors, InfiniBand RNICs supporting SRQ always clear the RI_RNIC_SEQ_DEQUEUE flag to indicate that out-of-order receives do not cause the dequeue of additional WQEs in the sequence gap.

An iWARP implementation that does not go beyond the RDMAC verbs specification will have a *max_sgl_rdma_read* value of one.

Inline data support is indicated by a non-zero *max_unregistered_inline* value. Data may be inlined if the WR has the inline flag on (it’s only a hint) and the message is less than or equal to *max_unregistered_inline* bytes in length.

The definition of RI_MAX_VENDOR_STRING_LEN is OS-dependent.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_query(), *ri_ib_rnic_attr_t*, *ri_iwarp_rnic_attr_t*

ri_iwarp_rnic_attr_t

NAME

ri_iwarp_rnic_attr_t – iWARP-specific RNIC attributes data structures

SYNOPSIS

```
typedef enum ri_iwarp_rnic_flags ri_iwarp_rnic_flags_t;
enum ri_iwarp_rnic_flags {
    RI_IWARP_CAN_MODIFY_IRD = 1 << 0,
    RI_IWARP_CAN_INCREASE_ORD = 1 << 1,

    /* supported IP transports */
    RI_IWARP_TCP_SUPPORTED = 1 << 2,
    RI_IWARP_SCTP_SUPPORTED = 1 << 3,

    /* if IETF and !PERMISSIVE, RNIC is non-permissive */
    RI_IWARP_IETF_PERMISSIVE = 1 << 4,

    /* marker & CRC preference for MPA negotiation on TCP */
    RI_IWARP_PREFER_MARKERS = 1 << 5,
    RI_IWARP_PREFER_CRC = 1 << 6,

    /* modify/query QP has KEEPALIVE option support */
    RI_IWARP_KEEPALIVE_SUPPORTED = 1 << 7
};

typedef struct ri_iwarp_port_attr ri_iwarp_port_attr_t;
struct ri_iwarp_port_attr {
    struct in6_addr *ipv6_addrs;
    struct in_addr *ipv4_addrs;
    uint_t port_num;
    uint_t num_ipv4_addrs;
    uint_t num_ipv6_addrs;
};

typedef struct ri_iwarp_rnic_attr ri_iwarp_rnic_attr_t;
struct ri_iwarp_rnic_attr {
    ri_iwarp_port_attr_t  *port_attrs;
    uint_t                 protocol_version; /* RDMAC = 0, IETF = 1 */
    uint_t                 num_ports;
    ri_iwarp_rnic_flags_t  iwarp_rnic_flags;
};
```

DESCRIPTION

The *ri_iwarp_rnic_attr_t* structure defines the iWARP transport-specific RNIC attributes based on the RDMAC 1.0 verb specification [VERBS-RDMAC] with small additions made to account for IETF differences and RNICPI additions.

The *protocol_version* field is zero for an RDMAC RNIC and one for an RNIC implementing the current IETF specifications. If the RNIC is IETF-compliant, then the RI_IWARP_IETF_PERMISSIVE flag distinguishes between a permissive and non-permissive implementation. The IP transports that an RNIC supports are signaled in the RI_IWARP_TCP_SUPPORTED and RI_IWARP_SCTP_SUPPORTED flags. For TCP, the

MPA marker and CRC preferences are signaled in the RI_IWARP_PREFER_MARKERS and RI_IWARP_PREFER_CRC; if a given flag is not set, then that item (markers or CRC) is not preferred.

If port information has not been requested in *ri_rnic_query*(), the *port_attrs* field will be set by the VP to NULL. However, the *num_ports* field is still filled in. If port information has been requested, then the *port_attrs* field points to an array of *ri_iwarp_port_attr_t* structs. That struct in turn describes the port number as well as having lists of the IPv4 and IPv6 addresses assigned to that port. If no addresses of a particular type are assigned to the port, then the pointer to the list of addresses will be NULL and the count of those addresses will be zero.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_query(), *ri_rnic_attr_t*

ri_ib_rnic_attr_t

NAME

ri_ib_rnic_attr_t – InfiniBand-specific RNIC attributes data structures

SYNOPSIS

```
typedef uint8_t ri_sl_t;  /* Service Level (4 bits) */
typedef uint8_t ri_lmc_t;  /* LID Mask Control (7 bits) */
typedef uint16_t ri_lid_t;  /* LID address */
typedef uint16_t ri_pkey_t;  /* P_Key */

typedef struct gid ri_gid_t;
struct ri_gid {  /* GID address */
    uint64_t  gid_prefix;
    uint64_t  gid_guid;
};

typedef enum ri_ib_port_state ri_ib_port_state_t;
enum ri_ib_port_state {  /* link/port states */
    RI_IB_LINK_DOWN = 1 << 1,
    RI_IB_LINK_INITIALIZE = 1 << 2,
    RI_IB_LINK_ARM = 1 << 3,
    RI_IB_LINK_ACT_DEFER = 1 << 4,  /* transient */
    RI_IB_LINK_ACTIVE = 1 << 5  /* "up" state */
};
/* how IB verbs define port up and down */
#define RI_IB_PORT_DOWN \
(RI_IB_LINK_DOWN|RI_IB_LINK_INITIALIZE|RI_IB_LINK_ARM)
#define RI_IB_PORT_UP \
(RI_IB_LINK_ACT_DEFER|RI_IB_LINK_ACTIVE)

typedef enum ri_ib_mtu ri_ib_mtu_t;
enum ri_ib_mtu {  /* MTU sizes */
    RI_IB_MTU_256 = 1,
    RI_IB_MTU_512,
    RI_IB_MTU_1K,
    RI_IB_MTU_2K,
    RI_IB_MTU_4K
};

/*
 * port capability flags
 * bit positions match portinfo mask
 */
typedef enum ri_ib_port_flags ri_ib_port_flags_t;
enum ri_ib_port_flags {
    RI_IB_PORT_SM = 1 << 0,
    RI_IB_PORT_SM_DISABLED = 1 << 10,
    RI_IB_PORT_SNMP_TUNNEL = 1 << 17,
    RI_IB_PORT_DEV_MGT = 1 << 19,
    RI_IB_PORT_VENDOR_CLASS = 1 << 20,
    RI_IB_PORT_CLIENT_REREG = 1 << 25
};
```

```
5619
5620        /* port attributes */
5621        typedef struct ri_ib_port_attr ri_ib_port_attr_t;
5622        struct ri_ib_port_attr {
5623            ri_gid_t          *src_gid_table;  /* valid in armed/active */
5624            ri_pkey_t         *pkey_table;  /* valid in armed/active */
5625            uint32_t           max_msg_size;  /* in bytes */
5626            ri_ib_mtu_t        max_mtu;  /* max MTU & UD msg size */
5627            ri_ib_port_state_t port_state;
5628            ri_ib_port_flags_t ib_port_flags;
5629            ri_lid_t           base_lid;  /* valid in armed/active */
5630            ri_lid_t           sm_lid;  /* valid in armed/active or 0xFFFF */
5631            uint16_t           bad_pkey_count;  /* zero if unsupported */
5632            uint16_t           bad_qkey_count;  /* zero if unsupported */
5633            uint8_t            src_gid_table_size;
5634            ri_lmc_t           lmc;  /* valid in armed/active */
5635            uint8_t            max_vls;  /* max VLs (4 bits) */
5636            ri_sl_t            sm_sl;  /* SL to reach SM addr */
5637            uint8_t            subnet_timeout;
5638            uint8_t            init_type_reply;  /* zero if unsupported */
5639        };
5640
5641        /* IB HCA flags */
5642        typedef enum ri_ib_rnic_flags ri_ib_rnic_flags_t;
5643        enum ri_ib_rnic_flags {
5644
5645            RI_IB_APM = 1 << 0,  /* auto path migrate */
5646            RI_IB_RNR_NAK = 1 << 1,  /* RC RNR-NAK */
5647            RI_IB_AH_PORT_CHECK = 1 << 2,  /* addr hdl port chk */
5648            RI_IB_CAN_CHANGE_PRIMARY_PORT = 1 << 3,  /* in SQD */
5649            RI_IB_CAN_CURRENT_QP_STATE = 1 << 4,  /* in ModifyQP */
5650
5651            RI_IB_ATOMICS = 1 << 5,  /* atomic ops */
5652            /* if atomics & !ALL_COMPONENTS, then atomic w/in HCA */
5653            RI_IB_ATOMICS_ALL_COMPONENTS = 1 << 6,
5654
5655            /* if BMM supported & !TYPE_2B, then type2A windows */
5656            RI_IB_TYPE_2B_WINDOWS = 1 << 7,
5657            RI_IB_MULTIPLE_BLOCK_SIZES_PER_REGION = 1 << 8,
5658
5659            RI_IB_UD_MULTICAST = 1 << 9,
5660
5661            RI_IB_PORT_ACTIVE_EVENT = 1 << 10,
5662            RI_IB_SHUTDOWN_PORT = 1 << 11,  /* in Modify RNIC */
5663
5664            RI_IB_PKEY_CNTR = 1 << 12,  /* P_Key Violated cnt */
5665            RI_IB_QKEY_CNTR = 1 << 13,  /* Q_Key Violated cnt */
5666            RI_IB_SI_GUID = 1 << 14,  /* Sys Image GUID */
5667            RI_IB_INIT_TYPE = 1 << 15  /* InitType support */
5668        };
5669
5670        typedef struct ri_ib_rnic_attr ri_ib_rnic_attr_t;
5671        struct ri_ib_rnic_attr {
5672            uint64_t           max_mr_size;  /* in bytes */
```

```
    uint64_t           node_guid;  /* Node GUID */
    uint64_t           system_image_guid;  /* zero if not supported */
    ri_ib_port_attr_t *port_attrs;
    ri_ib_rnic_flags_t ib_rnic_flags;
    uint_t             max_addr_hdls;  /* max UD Addr Handles */

    /* multicast fields are zero, if multicast unsupported */
    uint_t   max_mcgs;  /* max num multicast groups */
    uint32_t max_multicast_qps;  /* max MCG attachable QPs */
    uint32_t max_qps_per_mcg;  /* max num of QPs per MCG */

    uint16_t partition_table_size; /* size must be >= 1 */
    uint8_t  num_ports;
    uint8_t  local_ack_delay;  /* 5 bits */
};
```

DESCRIPTION

The *ri_iwarp_rnic_attr_t* structure defines the InfiniBand transport-specific RNIC attributes based on the 1.1 and 1.2 IBTA verb specifications. No attempt is made to distinguish earlier versions of the IBTA specifications. Further, no information is returned about the Reliable Datagram, Unreliable Connected, or Raw transport service types, which are all outside the scope of RNICPI.

The *port_attrs* field is a pointer to an array (*num_ports* long) of *ri_ib_port_attr_t* structs. An *ri_rnic_query*() invocation which does not retrieve port information will return *port_attrs* set to NULL by the VP. However, *num_ports* will still be valid to indicate how many ports are on the IB RNIC. The *ri_ib_port_attr_t* structure contains pointers to the source GID table (*source_gid_table* field) and the *P_Key* table (*partition_table* field). Note that the *P_Key* table for each port is of identical size which is given in the *partition_table_size* field of the *ri_iwarp_rnic_attr_t* structure.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_rnic_query(), *ri_rnic_attr_t*

ri_ltag_t, ri_rtag_t

NAME

ri_ltag_t, ri_rtag_t – *Ltag*/*Rtag* type definitons

SYNOPSIS

```
typedef uint32_t ri_ltag_t;
typedef uint32_t ri_rtag_t;
```

DESCRIPTION

These types describe the keys issued by the IHV to enable local (*LTag*) and remote (*RTag*) memory access. These map to InfiniBand *L_Keys* and *R_Keys*, and to iWARP STags.

Selection of the values for an *LTag* is frequently divided between the Consumer and the RNIC. The RNIC controlled portion is referred to as the "index", and the Consumer controlled portion as the "key". This is always the case for iWARP providers, and for all InfiniBand Memory Regions that can be fast registered. It may also be true for other InfiniBand Memory Regions and Memory Windows.

iWARP IHVs will use the same underlying type and value for the "*LTag*" and "*RTag*" of a Memory Region, and even potentially for the Memory Region/Window handle. These are defined as distinct types to support mapping to IB. Additionally, even for iWARP, Memory Region STags and Memory Window STags are clearly distinct types, even though they are both referred to as STags. An *LTag* maps to a Memory Region STag. An *RTag* maps either to a Memory Window STag or to the STag of a Memory Region that has been enabled for remote access. With iWARP the usage of an *LTag* and *RTag* may be different, but they are numbered as though they were a uniform resource. The index portion identifies the Memory Region or Memory Window for either. This is not necessarily true for IB.

Consumer editing of the key portion should be made to the *LTag* for Memory Regions, and to the *RTag* for Memory Windows.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_stag_state_t

NAME

ri_stag_state_t – state of an *Ltag* or *Rtag*

SYNOPSIS

```
enum ri_stag_state {
    RI_STAG_STATE_INVALID,
    RI_STAG_STATE_VALID
};
typedef enum ri_stag_state ri_stag_state_t;
```

DESCRIPTION

Both *Ltags* and *Rtags* can be in invalid or valid states.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_phys_buf_t

NAME

ri_phys_buf_t – descriptor of a physical memory block

SYNOPSIS

```
struct ri_phys_buf {
    ri_phys_addr_t     phys_address;
    ri_phys_buf_len_t  phys_len;
};
typedef struct ri_phys_buf ri_phys_buf_t;
```

DESCRIPTION

An *ri_phys_buf_t* defines a single entry in a physical buffer list when the length is supplied on a per-entry basis. Typically, this is when the associated parameter/field for the *pbl* entry length is zero. When there is a non-zero *pble_len* the *ri_phys_buf_t* pointer is actually an *ri_phys_addr_t* pointer.

Both *ri_phys_addr_t* and *ri_phys_buf_len_t* are OS-dependent types that may vary in size.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_mem_attr_t

NAME

ri_mem_attr_t – boolean attributes for Memory Regions/Windows

SYNOPSIS

```
enum ri_mem_attr {
    RI_MEM_FLAGS_LOCAL_READ        = 1 << 0,
    RI_MEM_FLAGS_LOCAL_WRITE       = 1 << 1,
    RI_MEM_FLAGS_LOCAL             = (RI_MEM_FLAGS_LOCAL_READ|
                                      RI_MEM_FLAGS_LOCAL_WRITE),

    RI_MEM_FLAGS_REMOTE_WRITE      = 1 << 2,
    RI_MEM_FLAGS_REMOTE_READ       = 1 << 3,
    RI_MEM_FLAGS_REMOTE            = (RI_MEM_FLAGS_REMOTE_WRITE|
                                      RI_MEM_FLAGS_REMOTE_READ),
    RI_MEM_FLAGS_ATOMICS           = 1 << 4,
    RI_MEM_FLAGS_BINDING           = 1 << 5,
    RI_MEM_FLAGS_CONSUMER_OWNS_KEY = 1 << 6,
    RI_MEM_FLAGS_ZBVA              = 1 << 7,
    RI_MEM_FLAGS_NONCOHERENT       = 1 << 8,
    RI_MEM_FLAGS_USER_VA           = 1 << 9,
    RI_MEM_FLAGS_SHARED_REG_IOVA   = 1 << 10
};
typedef ri_mem_attr ri_mem_attr_t;
```

DESCRIPTION

This type provides a set of flags that specify boolean characteristics of Memory Regions and/or Memory Windows.

Most of the flags specify the access permissions for the Memory Region/Window:

RI_MEM_FLAGS_LOCAL_READ	Permission flag that represents local read access. Local Read Access is actually assumed; this symbol is created for convenience. The Consumer can IOR this flag to represent Local Read Access without actually changing the flags.
RI_MEM_FLAGS_LOCAL_WRITE	Permission flag that enables local write access.
RI_MEM_FLAGS_REMOTE_WRITE	Permission flag that enables remote write access.
RI_MEM_FLAGS_REMOTE_READ	Permission flag that enables remote read access.
RI_MEM_FLAGS_ATOMIC	Permission flag that enables remote atomic access. This flag is irrelevant if the transport does not support atomic operations.
RI_MEM_FLAGS_BINDING	Permission flag that enables binding Memory Windows using this Memory Region. This flag is irrelevant for Memory Windows. The permission may also be granted automatically under certain transports.

RI_MEM_FLAGS_CONSUMER_OWNS_KEY	The *LTag*/*RTags* for this Region/Window are split between Index and a Key portions. The key portion is the least significant byte, and is specified by the Consumer.
RI_MEM_FLAGS_ZBVA	Flag that indicates that the Memory Region (or Window) uses zero-based addressing.
RI_MEM_FLAGS_NONCOHERENT	Specifies that the Memory Region does not require normal memory coherency rules. If non-coherent memory is enabled, the application should follow system or vendor-specific methods to guarantee data access is made to up-to-date buffers. Unless this flag is set, the Memory Region must adhere fully to all RDMA ordering rules. When non-coherent has been set for a Memory Region, it automatically applies to any redefinitions of the same memory.
RI_MEM_FLAGS_USER_VA	Indicates that the virtual address specified for virtual memory registration is to be interpreted in the context of the user process virtual address space associated with the RNIC handle used in the registration call. If this flag is not set, the virtual address should be considered a kernel virtual address.
RI_MEM_FLAGS_SHARED_REG_IOVA	Indicates that the virtual address specified for a shared memory registration is to be treated as an I/O virtual address (as in physical memory registration). If this flag is not set, the virtual address should be considered as a virtual memory registration (corresponding to some process/kernel address space). This flag is ignored for calls other than *ri_mr_reg_shared*().

CONSUMER USAGE

The Consumer creates the settings desired for a Memory Region/Window by bitwise OR'ing the entire set of desired permissions and attributes.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_mr_reg_attr_t, ri_mr_rereg_t

NAME

ri_mr_reg_attr_t – Memory Region creation attributes

ri_mr_rereg_t – Reregistration request types

SYNOPSIS

```
typedef struct ri_mr_reg_attr ri_mr_reg_attr_t;
struct ri_mr_reg_attr {
    ri_addr_t       va;
    ri_length_t     length;
    ri_mem_attr_t   mem_attr;
    ri_pd_handle_t  pd;
};
typedef enum ri_mr_rereg ri_mr_rereg_t;
enum ri_mr_rereg {
    RI_MR_REREG_TRANSLATE = 1<<0,
    RI_MR_REREG_PD        = 1<<1,
    RI_MR_REREG_ACCESS    = 1<<2
};
```

DESCRIPTION

The attributes required when creating a Memory Region.

va Virtual Address of the region and/or the address of the memory to be registered in the Consumer's current virtual memory map.

length Length in bytes of the Memory Region.

attr Boolean attributes for the Memory Region. See *ri_mem_attr_t*.

pd Handle of the Protection Domain.

The *ri_mr_rereg* attributes are used on reregister calls to indicate which Memory Region attributes are being changed:

RI_MR_REREG_TRANSLATE The address translation for the Memory Region may be changed by this reregister call.

RI_MR_REREG_PD The Protection Domain for the Memory Region may be changed by this reregister call.

RI_MR_REREG_ACCESS The access rights for the Memory Region may be changed by this reregister call.

CONSUMER USAGE

InfiniBand 1.1 adapters may enable a larger region than requested. Consumers should use *ri_mr_query*() to determine the actual region enabled if adjacent memory structures must be protected from RNIC access.

IMPLEMENTATION NOTES

None.

5892 **SEE ALSO**
5893 *ri_mr_query*(), *ri_mem_attr_t*

ri_mr_reg_phys_attr_t

NAME

ri_mr_reg_phys_attr_t – Memory Region physical creation attributes

SYNOPSIS

```
typedef struct ri_mr_reg_phys_attr ri_mr_reg_phys_attr_t;

struct ri_mr_reg_phys_attr {
    ri_mr_reg_attr_t  common;
    void             *phys_buf_list;
    uint_t            nbufs;
    uint_t            pble_size;
    uint_t            address_offset;
};
```

DESCRIPTION

The attributes required when creating a Memory Region using physical buffer lists.

common Attributes that apply to creating any Memory Region.

phys_buf_list
If *pble_size* is non-zero, this is an *ri_phys_addr_t* pointer to an array of physical addresses for blocks/pages that are all *pble_size* bytes long. If *pble_size* is zero, then this is a *ri_phys_buf_t* pointer to an array of block specifications that each contain the physical address and block size. This field is not returned on a query.

nbufs Number of buffers in *phys_buf_list*.

pble_size Size of each buffer in *phys_buf_list*, if non-zero. If zero, indicates that the lengths are in the *phys_buf_list*.

offset Offset within the first physical buffer corresponding to the starting virtual address.

CONSUMER USAGE

Consumers are advised to use *ri_mr_query*() to determine the actual boundaries of the Memory Region created.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_mr_reg_attr_t

ri_mw_type_t

NAME

ri_mw_type_t – type of Memory Window: narrow/wide

SYNOPSIS

```
enum ri_mw_type {
    RI_MW_TYPE_1, RI_MW_TYPE_WIDE=RI_MW_TYPE_1,
    RI_MW_TYPE_2, RI_MW_TYPE_NARROW=RI_MW_TYPE_2
};
typedef ri_mw_type ri_mw_type_t;
```

DESCRIPTION

Memory Windows must be created to support either type 1 (wide) or type 2 (narrow) binding.

Once bound, a wide-bind window is valid on all QPs with the matching protection domain. Narrow-bind windows are valid only on the QP where the bind operation was performed. The RNIC attributes indicate which types are supported. Generally, wide-binding MUST be supported on InfiiniBand and MAY be supported on iWARP. Narrow binding MAY be supported on InfiniBand and MUST be supported on iWARP.

Making a window narrow-bound typically simplifies internal processing for the RNIC, particularly when dealing with invalidations. Therefore, when both types are supported the Consumer is advised to use narrow-bind windows unless there is a specific application need for wide-bind windows.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

ri_qp_attr_t

NAME

ri_qp_attr_t – QP attributes

SYNOPSIS

```
enum ri_svc_type
{
    RI_SERVICE_RELIABLE_CONNECTION,
    RI_SERVICE_UNRELIABLE_DATAGRAM
};
typedef enum ri_svc_type ri_svc_type_t;

/* Some states are not valid for RDMAC/IB NICs. Attempts to modify the
state attribute will fail with RI_RET_INVALID_STATE if not appropriate
for the technology */

enum ri_qp_state
{
    RI_QP_STATE_IDLE,
    RI_QP_STATE_READY_TO_SEND,
    RI_QP_STATE_ERROR,

    /* The following states are valid only for iWARP */
    RI_QP_STATE_TERMINATE,
    RI_QP_STATE_CLOSING,

    /* The following states are valid only for IB */
    RI_QP_STATE_SEND_QUEUE_ERROR,
    RI_QP_STATE_INITIALIZED,
    RI_QP_STATE_READY_TO_RECEIVE,
    RI_QP_STATE_SEND_QUEUE_DRAINING,
    RI_QP_STATE_SEND_QUEUE_DRAINED,
    RI_QP_STATE_TIMEWAIT
};
typedef enum ri_qp_state ri_qp_state_t;

struct ri_qp_attr
{
    uint32_t qp_id;
    union {
        ri_cq_handle_t  scq_handle;
        ri_os_data_t    scq_os_data;
    } scq;
    union {
        ri_cq_handle_t  rcq_handle;
        ri_os_data_t    rcq_os_data;
    } rcq;
    union {
        ri_srq_handle_t  srq_handle;
        ri_os_data_t     srq_os_data;
    } srq;
    ri_pd_handle_t pd;
```

```
    uint32_t       sq_size;
    uint32_t       rq_size;
    ri_boolean_t   srq_valid;
    uint32_t       sq_max_sges;
    uint32_t       sq_max_sges_rdmaw;
    uint32_t       rq_max_sges;
    ri_qp_state_t  state;
    ri_boolean_t   enable_rdma_read;
    ri_boolean_t   enable_rdma_write;
    uint32_t       max_rdma_read_outgoing;
    uint32_t       max_rdma_read_incoming;
    ri_boolean_t   enable_stag0_fastreg;
    union {
        ri_iwarp_qp_attr_t  qp_iwarp;
        ri_ib_qp_attr_t     qp_ib;
    } qp_xport;
};
typedef struct ri_qp_attr ri_qp_attr_t;

struct ri_mpa_attr
{
    ri_boolean_t  recv_marker_enabled;
    ri_boolean_t  xmit_marker_enabled;
    ri_boolean_t  crc_enabled;
    uint32_t      ddp_rdmap_version;  /* 0 or 1 */
};
typedef struct ri_mpa_attr ri_mpa_attr_t;

struct ri_iwarp_qp_attr
{
    ri_boolean_t            enable_bind;
    char                    terminate_buffer[52];
    uint32_t                terminate_msg_length;
    ri_boolean_t            is_terminate_local;
    ri_llp_stream_handle_t  llp_stream_handle;
    ri_mpa_attr_t           mpa_attributes;
    char                   *stream_msg_buf;
    uint32_t                stream_msg_buf_length;
    uint32_t                rq_hiwat;
    ri_boolean_t            rq_armed;
};
typedef struct ri_iwarp_qp_attr ri_iwarp_qp_attr_t;

struct ri_gen_address_vector
{
    uint8_t   sl;
    uint8_t   send_global_routing_hdr_flg;
    uint8_t   hop_limit;
    uint8_t   source_gid_index;
    uint32_t  flow_label;
    ri_gid_t  dgid;
    ri_lid_t  dlid;
    uint8_t   traffic_class;
    uint8_t   max_static_rate;
```

```
    uint8_t   source_path_bits;
};
typedef struct ri_gen_address_vector ri_gen_address_vector_t;

struct ri_path_address_info
{
/* For UD QPs, pkey_index and physical_port are alone valid */
    uint16_t                pkey_index;
    uint8_t                 physical_port;
    uint16_t                path_mtu;
    uint8_t                 local_ack_timeout;
    uint8_t                 retry_count;
    uint8_t                 rnr_retry_count;
    ri_gen_address_vector_t  av;
};
typedef struct ri_path_address_info ri_path_address_info_t;

enum ri_path_migration_state
{
    RI_MIG_STATE_MIGRATED,
    RI_MIG_STATE_REARM,
    RI_MIG_STATE_ARMED
};
typedef enum ri_path_migration_state ri_path_migration_state_t;

struct ri_ib_qp_attr
{
    /* Common to RC and UD */
    uint32_t                  send_psn;
    ri_boolean_t              signal_completions;
    ri_svc_type_t             trans_type;
    uint32_t                  enable_async_notify_for_sqd;
    ri_path_address_info_t    primary_av;
    /* RC-specific attributes */
    uint32_t                  recv_psn;
    uint32_t                  dest_qpn;
    uint32_t                  min_rnrnak_timer;
    ri_path_address_info_t    alternate_av;
    ri_path_migration_state_t mig_state;
    ri_boolean_t              enable_atomic;
    /* UD-specific attributes */
    ri_qp_state_t             current_state;
    uint32_t                  recv_qkey;
};
typedef struct ri_ib_qp_attr ri_ib_qp_attr_t;
```

DESCRIPTION

The following table outlines the validity of each attribute in *ri_qp_create*() and *ri_qp_modify*(). All attributes are valid in *ri_qp_query*(). If an attribute is not defined as valid in *ri_qp_create*(), it must not be passed in the call and will be ignored by the implementation. If an attribute is defined as modifiable, it may be accepted by the implementation depending on the state of the QP.

QP Attributes	Description	Valid in *ri_qp_create*()	Modifiable?
Common Transport-Neutral QP Attributes			
qp_id	Local QP ID or number.	No	No
scq	Handle to the CQ associated with the Send Queue when passed in *ri_qp_create*(). Contains *os_data* of the CQ upon return from *ri_qp_query*().	Yes	No
rcq	Handle to the CQ associated with the Receive Queue when passed in *ri_qp_create*(). Contains *os_data* of the CQ upon return from *ri_qp_query*().	Yes	No
sq_size	Number of outstanding requests supported on the Send Queue.	Yes	Yes
rq_size	On input the minimum number of outstanding receive Work Requests required for this QP. If successful, this field on output will have the minimum number of receive Work Requests that will be supported for this QP. If no SRQ is associated with this QP, this is the actual number that will be supported. If an SRQ is associated the RNIC MAY allow more receive Work Requests to be associated with the QP, up to the capacity of the SRQ. A receive Work Request from an SRQ is considered to be "for" a QP when it is allocated to the QP. The sum of this field across all QPs using the same SRQ MUST NOT be pre-validated. The Consumer is expected to provision adequate buffers to the SRQ, but the quality of that estimation is inherently application-specific.	Yes	Yes
srq	Handle to the SRQ associated with this QP when passed in *ri_qp_create*(). Contains *os_data* of the SRQ upon return from *ri_qp_query*(). This field is valid only if *srq_valid* is TRUE.	Yes	No
srq_valid	TRUE if QP is associated with an SRQ. This must be set in *ri_qp_create*() if the QP needs to be associated with an SRQ.	Yes	No
sq_max_sges	Number of scatter/gather entries supported on Work Requests submitted to the Send Queue.	Yes	No

QP Attributes	Description	Valid in *ri_qp_create*()	Modifiable?
sq_max_sges_rdmaw	Number of scatter/gather entries supported on RDMA-write Work Requests submitted to the Send Queue. IB implementations must return the same value as *sq_max_sges* in *ri_qp_query*() and may need to allocate more resources to make this equal to *sq_max_sges* upon create and modify.	Yes	No
rq_max_sges	Number of scatter/gather entries supported on Work Requests submitted to the Receive Queue.	Yes	No
state	Current QP state (see *ri_qp_state_t*). If passed to *ri_qp_modify*(), this is the next desired state for the QP.	No	Yes
enable_rdma_read	Enable RDMA read operations on the QP.	Yes (iWARP) No (IB)	No (iWARP) Yes (IB)
enable_rdma_write	Enable RDMA write operations on the QP. iWarp implementations must enable responses to inbound RDMA read operations as well.	Yes (iWARP) No (IB)	No (iWARP) Yes (IB)
max_rdma_read_outgoing	Number of outgoing RDMA reads (and atomic operations for IB implementations) outstanding on the QP.	Yes (iWARP) No (IB)	Yes
max_rdma_read_incoming	Number of incoming RDMA reads (and atomic operations for IB implementations) outstanding on the QP.	Yes (iWARP) No (IB)	Yes
pd	Handle to the PD associated with the QP.	Yes	No
enable_stag0_fastreg	Enable Stag0 and Fast-register operations on the QP. Attempt to enable this will fail on IB 1.1 implementations.	Yes	No
iWarp-Specific QP Attributes			
enable_bind	Enable Memory Window bind operations.	Yes	No
terminate_buffer	If the QP is in Terminate or Error states, this is the buffer containing the Terminate Message that was received or sent (if possible).	No	No
terminate_msg_length	Number of bytes in *terminate_buffer* that contains a valid terminate message.	No	No
is_terminate_local	If the QP is in Terminate or Error states, this indicates if the Terminate message was generated locally or by the associated QP.	No	No

QP Attributes	Description	Valid in *ri_qp_create*()	Modifiable?
llp_stream_handle	Opaque handle of the LLP associated with the QP. This could either be a pointer to an underlying LLP object or a pointer to TCP connection information required to create a new LLP connection on the RNIC. The current value of this attribute is returned in *ri_qp_query*() and it may be different from what was passed by the Consumer in *ri_qp_modify*(). If this attribute has never been modified, *ri_qp_query*() must return NULL for this attribute. After passing this to a successful *ri_qp_modify*() operation, the Consumer must not use it to perform send/receive operations directly on the TCP connection, instead *ri_qp_postsq*(), *ri_qp_postrq*(), and their equivalent calls must be used to do send/receive on the QP. The set of control operations that are possible on the LLP may be substantially limited after this *ri_qp_modify*() and are implementation-dependent.	No	Yes
recv_marker_enable	Enable/disable accepting MPA frames with markers on the connection associated with the QP. *ri_qp_query*() reports the preference (on/off). The Consumer may attempt to override the preference through *ri_qp_modify*() which may fail. Permissive IETF RNICs will not fail such attempts. RDMAC-compliant RNICs shall fail an attempt to set this attribute to off.	No	Yes
xmit_marker_enable	Enable/disable send MPA frames with markers on the connection associated with the QP. *ri_qp_query*() reports the preference (on/off). The Consumer may attempt to override the preference through *ri_qp_modify*() which may fail. IETF RNICs shall not fail such attempts. RDMAC-compliant RNICs shall fail an attempt to set this attribute to off.	No	Yes
crc_enable	Enable/disable transmitting CRCs in outgoing MPA frames and checking CRCs in incoming MPA frames.	Yes	Yes

QP Attributes	Description	Valid in *ri_qp_create*()	Modifiable?
stream_msg_buf	Pointer to the last streaming mode message on the connection. Valid only for *ri_qp_modify*() if moving from Idle to RTS. A value of NULL is returned for this attribute in *ri_qp_query*(). When the Consumer is modifying this attribute by calling into kVP, this must be a valid kernel pointer, and when the Consumer is calling into uVP, this must be a valid user-space pointer.	No	Yes
stream_msg_buf_length	Number of bytes in *stream_msg_buf* containing the last streaming mode message.	No	Yes
ddp_rdmap_version	Version number to be used by the RNIC in DDP and RDMAP headers. This is to allow interoperability between RDMAC and IETF. See the MPA Interoperability draft.	Yes	Yes
rq_hiwat	A high water mark of number of receive WQEs allocated for the QP from the associated SRQ and are outstanding, when hit, will trigger an asynchronous event to the Consumer. The asynchronous event must not otherwise disrupt the QP operation. A zero *rq_hiwat* in *ri_qp_modify*() means that the asynchronous event is triggered when the QP consumes its first buffer from SRQ since last arm. If zero *rq_hiwat* is passed in *ri_qp_create*(), RQ limit arming must be disabled.	Yes	Yes
rq_armed	RQ limit armed indicator.	No	Yes
IB-Specific QP Attributes			
current_state	Current state of the QP. Valid only if *state* is RI_QP_STATE_READY_ TO_SEND.	No	Yes
recv_qkey	*Q_Key* for the UD Receive Queue.	No	Yes
*dest_qp*n	Destination QP number for RC and UC QPs.	No	Yes
min_rnrnak_timer	Minimum RNR NAK Timer Field Value. When a message arrives which is targeted at a local Receive Queue, and that Receive Queue has no receive Work Request outstanding, the RI may respond to the initiator with an RNR NAK packet. This modifier is the minimum value which shall be sent in the Timer Field of such an RNR NAK packet; it does not affect RNR NAKs sent for other reasons. Applicable only to RC QPs.	No	Yes

QP Attributes	Description	Valid in *ri_qp_create*()	Modifiable?
signal_completions	If set, all Work Requests submitted to the Send Queue always generate a completion entry. If not set, the Consumer must specify on each Work Request submitted to the Send Queue whether to generate a completion entry for successful completions.	Yes	No
trans_type	The transport service type requested for this QP. See *ri_svc_type_t*.	Yes	No
enable_async_notify_ for_sqd	Enable or disable Send Queue Drained Asynchronous Affiliated Event Notification. This modifier is only applicable when the QP state chosen is SQD.	No	Yes
mig_state	Path migration state. See *ri_path_migration_state_t*.	No	Yes
send_psn	Packet Sequence Number on the Send Queue.	No	Yes
recv_psn	Packet Sequence Number on the Receive Queue.	No	Yes
enable_atomic	Enable atomic operations on the QP.	No	Yes
pkey_index	Primary Key Index.	No	Yes
physical_port	Number of the local port to put the QP on.	No	Yes
The following attributes are valid only for RC.			
sl	Service Level.	No	Yes
send_global_routing_ hdr_flg	Send Global Routing Header Flag.	No	Yes
hop_limit	Hop Limit.	No	Yes
source_gid_index	Source GID Index.	No	Yes
flow_label	Flow Label.	No	Yes
dgid	Destination GID.	No	Yes
Dlid	Destination LID.	No	Yes
traffic_class	Traffic Class.	No	Yes
max_static_rate	Maximum Static Rate.	No	Yes
path_mtu	Path MTU.	No	Yes
local_ack_timeout	Local Ack Timeout.	No	Yes
retry_count	Retry Count.	No	Yes
rnr_retry_count	RNR Retry Count.	No	Yes
source_path_bits	Source Path Bits.	No	Yes

6108

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

rq_size can be used to guide allocation of per-QP resources to track receive Work Requests allocated to the QP. For example, an RNIC might allocate an RQ even when an SRQ is assigned, and copy the receive Work Request (or a pointer to it) when it is allocated to the QP.

The RNIC MAY simply ignore this field when an SRQ is specified.

SEE ALSO

ri_qp_create(), *ri_qp_modify*()

ri_srq_attr_t

NAME

ri_srq_attr_t – SRQ attributes

SYNOPSIS

```
struct ri_srq_attr {
    uint32_t        max_wrs;
    uint32_t        max_sgl;
    uint32_t        lowat;
    ri_pd_handle_t  pd;
    ri_boolean_t    srq_armed;
};
typedef struct ri_srq_attr ri_srq_attr_t;
```

DESCRIPTION

max_wrs On input, maximum number of Work Requests the Consumer expects to submit. On output, maximum number the Consumer is allowed to submit.

max_sgl On input, maximum number of scatter/gather elements the Consumer expects to submit On output, maximum number the Consumer is allowed to submit. This will be greater than or equal to the number requested for a successful creation. This field is not valid for *ri_srq_modify*().

lowat A low water mark of available number of posted WQEs. When the number available is lower than or equal to this, it will trigger an asynchronous event to the Consumer. The Consumer will have to re-arm explicitly for subsequent notifications. At most one event is generated per arming. The SRQ is automatically armed if *lowat* is non-zero in *ri_srq_create*().

pd Handle of the PD this SRQ is to be associated with. This is not valid for *ri_srq_modify*().

srq_armed TRUE when SRQ Limit is armed. This is an output only field that is ignored on input.

Attributes of a Shared Receive Queue that can be used as an IN OUT or OUT parameter.

It is used as an IN OUT parameter when created or modifying an SRQ. The desired attributes are passed in. On successful completion, the actual results are passed out.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

ri_srq_create(), *ri_srq_modify*()

ri_wr_t

NAME

ri_wr_t – Work Request structures for the Send Queue

SYNOPSIS

```
enum ri_wr_type{

    RI_WR_TYPE_SEND,

    RI_WR_TYPE_SEND_INVALIDATE,
    RI_WR_TYPE_RDMA_WRITE,
    RI_WR_TYPE_RDMA_READ,
    RI_WR_TYPE_BIND,
    RI_WR_TYPE_FAST_REGISTER,
    RI_WR_TYPE_LOCAL_INVALIDATE_MR,
    RI_WR_TYPE_LOCAL_INVALIDATE_MW,

    /* iWarp-specific opcodes */
    RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE,

    /* IB-specific opcodes */
    RI_WR_TYPE_COMPARE_AND_SWAP,
    RI_WR_TYPE_FETCH_AND_ADD,
    RI_WR_TYPE_SEND_IMMEDIATE,
    RI_WR_TYPE_RDMA_WRITE_IMMEDIATE

};
typedef enum ri_wr_type ri_wr_type_t;

enum ri_wr_flags{
    /* Generate completion event */
    RI_WR_SIGNALED_FLAG = 0x01,

    /* Generate notification for completion */
    RI_WR_LOCAL_SOLICITED_FLAG = 0x02,

    /* Notify the remote on completion */
    RI_WR_SOLICITED_EVENT_FLAG = 0x04,

    /* Block until all previous RDMA reads are completed */
    RI_WR_READ_FENCE_FLAG = 0x08,

    /* Block until all previous Work Requests are completed */
    RI_WR_LOCAL_FENCE_FLAG = 0x10,
    /* Coalesce Work Requests to the Send Queue */
    RI_WR_COALESCE_FLAG = 0x20,

    /* Hint to allow inlining of data */
    RI_WR_INLINE_OK_FLAG = 0x40,

    /* Pass-through in the absence of notification */
    RI_WR_CONSUMER_FLAG2 = RI_WR_LOCAL_SOLICITED_FLAG,
```

```
6204                RI_WR_CONSUMER_FLAG1 = 1 << 29,
6205                RI_WR_CONSUMER_FLAG0 = 1 << 30
6206
6207            };
6208            typedef enum ri_wr_flags ri_wr_flags_t;
6209
6210            typedef uint64_t ri_addr_t;
6211
6212            struct ri_data_seg {
6213                union {
6214                    ri_addr_t       vaddr;
6215                    ri_phys_addr_t  paddr;
6216                    ri_length_t     offset;
6217                } addr;
6218                union {
6219                    ri_ltag_t ltag;
6220                    ri_rtag_t rtag;
6221                } tag;
6222                uint32_t len;
6223            };
6224            typedef struct ri_data_seg ri_data_seg_t;
6225
6226            struct ri_wr_common {
6227                ri_wr_type_t   type;
6228                ri_wr_flags_t  flags;
6229                ri_wr_id_t     id;
6230            };
6231            typedef struct ri_wr_common ri_wr_common_t;
6232
6233            struct ri_wr_send {
6234                ri_wr_type_t    type;
6235                ri_wr_flags_t   flags;
6236                ri_wr_id_t      id;
6237                ri_data_seg_t  *sg_list;
6238                ri_rtag_t       rtag;
6239                uint_t          num_sgle;
6240                uint32_t        imm_data;
6241            };
6242            typedef struct ri_wr_send ri_wr_send_t;
6243
6244            struct ri_wr_rdma_write {
6245                ri_wr_type_t    type;
6246                ri_wr_flags_t   flags;
6247                ri_wr_id_t      id;
6248                ri_data_seg_t  *sg_list;
6249                uint64_t        rdma_addr;
6250                ri_rtag_t       rtag;
6251                uint_t          num_sgle;
6252                uint32_t        imm_data;
6253            };
6254            typedef struct ri_wr_rdma_write ri_wr_rdma_write_t;
6255
6256            struct ri_wr_rdma_read {
6257                ri_wr_type_t    type;
```

```
6258            ri_wr_flags_t   flags;
6259            ri_wr_id_t      id;
6260            ri_data_seg_t  *sg_list;
6261            uint64_t        rdma_addr;
6262            uint_t          num_sgle;
6263            ri_rtag_t       rtag;
6264        };
6265        typedef struct ri_wr_rdma_read ri_wr_rdma_read_t;
6266
6267        struct ri_wr_bind {
6268            ri_wr_type_t    type;
6269            ri_wr_flags_t   flags;
6270            ri_wr_id_t      id;
6271            ri_mw_handle_t  mw_handle;
6272            ri_rtag_t       rtag;
6273            ri_mr_handle_t  mr_handle;
6274            ri_ltag_t       ltag;
6275            ri_addr_t       vaddr;
6276            ri_length_t     length;
6277            ri_mem_attr_t   mem_attr;
6278        };
6279        typedef struct ri_wr_bind ri_wr_bind_t;
6280
6281        struct ri_wr_fmr{
6282            ri_wr_type_t           type;
6283            ri_wr_flags_t          flags;
6284            ri_wr_id_t             id;
6285            ri_mr_reg_phys_attr_t  attr;
6286            ri_ltag_t              ltag;
6287            ri_rtag_t              rtag;
6288            ri_mr_handle_t         mr_handle;
6289        };
6290        typedef struct ri_wr_fmr ri_wr_fmr_t;
6291
6292        struct ri_wr_local_invalidate{
6293            ri_wr_type_t     type;
6294            ri_wr_flags_t    flags;
6295            ri_wr_id_t       id;
6296            ri_ltag_t        ltag;
6297            ri_rtag_t        rtag;
6298            ri_mrw_handle_t  mrw_handle;
6299        };
6300        typedef struct ri_wr_local_invalidate ri_wr_local_invalidate_t;
6301
6302        union ri_mrw_handle{
6303            ri_mr_handle_t  mr_handle;
6304            ri_mw_handle_t  mw_handle;
6305        };
6306        typedef union ri_mrw_handle ri_mrw_handle_t;
6307
6308        struct ri_wr_atomic {
6309            ri_wr_type_t   type;
6310            ri_wr_flags_t  flags;
6311            ri_wr_id_t     id;
```

```
        ri_rtag_t      rtag;
        uint64_t       atomic_operand1;
        uint64_t       atomic_operand2;
        ri_data_seg_t  atomic_result;
    };
    typedef struct ri_wr_atomic ri_wr_atomic_t;

    struct ri_wr_datagram {
        ri_wr_type_t         type;
        ri_wr_flags_t        flags;
        ri_wr_id_t           id;
        ri_data_seg_t       *sg_list;
        uint_t               num_sgle;
        uint32_t             imm_data;
        ri_address_handle_t  remote_node_address;
        uint32_t             dest_qp;
        uint32_t             dest_qkey;
        uint8_t              max_static_rate;
    };
    typedef struct ri_wr_datagram ri_wr_datagram_t;

    union ri_wr{
        ri_wr_common_t            wr_common;
        ri_wr_send_t              wr_send;
        ri_wr_rdma_read_t         wr_rdma_read;
        ri_wr_rdma_write_t        wr_rdma_write;
        ri_wr_bind_t              wr_bind;
        ri_wr_fmr_t               wr_fmr;
        ri_wr_local_invalidate_t  wr_li;
        ri_wr_atomic_t            wr_atomic;
        ri_wr_datagram_t          wr_datagram;
    };
    typedef union ri_wr ri_wr_t;
```

DESCRIPTION

Work Request Field	Description
type	Type of Work Request. See *ri_wr_type_t*. Some flavors of Send, RDMA write, RDMA read, Memory Window Bind, Fast Register, Invalidate, and Atomic are supported. RI_WR_TYPE_BIND is specified for both Type1 (wide) and Type2 (narrow) windows.

Work Request Field	Description
flags	Modifier for a Work Request. See *ri_wr_flags_*t. RI_WR_SIGNALED_FLAG and RI_WR_LOCAL_SOLICITED_FLAG are valid for all send Work Request types. RI_WR_SOLICITED_EVENT_FLAG is valid only for Send Work Request types and RDMA write with immediate data. RI_WR_READ_FENCE_FLAG is valid only for Send and RDMA-writes on iWarp transport. It is extended to RDMA read, atomic, and Bind on IB transport. RI_WR_LOCAL_FENCE_FLAG is valid only for RI_WR_TYPE_LOCAL_INVALIDATE Work Request type. RI_WR_COALESCE_FLAG has the following characteristics: • Consumers typically set RI_WR_COALESCE_FLAG for post calls and eventually make a post call with RI_WR_COALESCE_FLAG cleared. If RI_WR_COALESCE_FLAG is set, the implementation could delay alerting the RNIC about the Work Request until a later post call. • Implementation MUST post to RNIC pending Work Requests including the current one if RI_WR_COALESCE_FLAG is cleared. • Implementation could post pending Work Requests to RNIC at any time even if RI_WR_COALESCE_FLAG is not cleared. This is what an implementation that does not do "true" WR coalescing is expected to do. • QP state changes may force implementation to implicitly post pending Work Requests to RNIC. • All immediate checks must be done at the time of post call regardless of the setting of RI_WR_COALESCE_FLAG. • Consumers MUST clear the flag before causing the Send Queue to be full. • RI_WR_CONSUMER_FLAG0, RI_WR_CONSUMER_FLAG1, and RI_WR_CONSUMER_FLAG2 are Consumer-defined tags to the Work Request. The flags are meant to be passed through by the implementation and must be preserved in the completion of the Work Request in addition to the Work Request ID. Whether or not these Consumer-defined flags are actually supported by the provider is indicated through *ri_rnic_query*(). Note that RI_WR_CONSUMER_FLAG2 is meant to be used by the Consumer when RNIC does not support generation of notification upon completion of a specific Work Request. • If RI_WR_INLINE_OK_FLAG is set, then the Verbs Provider MAY optimize implementation of the post by assuming that the Target Offset is usable as a local pointer (void *) and that there is no need to interpret it in the context of the *LTag* provided. If RNIC supports a non-zero *max_unregistered_inline* attribute, RNIC will not access the first *max_unregistered_inline* bytes in the Work Request through *Ltag*; instead this may allow the implementation to simply copy a payload to the binary form of the Work Request rather than copying the SGL itself. If the message size is greater than *max_unregistered_inline,* the Consumer must supply valid *Ltags* and RNIC may still inline some data if RI_WR_INLINE_OK_FLAG is specified. The Consumer is responsible for only selecting this option when using the addresses in the SGL as pointers will produce exactly the same result as if the *Ltags* had been fully interpreted. Failure to do so may result in undiagnosed errors (the wrong data being sent) or memory protection faults (if the address is invalid as a pointer).

Work Request Field	Description
sg_list	A triplet containing address, length, and STag for local use. *ri_data_seg_t* describes a contiguous data buffer that is accessible by the RNIC with the help of the STag for local use. It is part of Scatter/Gather lists posted to send or Receive Queue. *addr* can be a virtual address for virtually addressed regions or windows. For STag0 operations, *tag* must be set to *stag0_value* reported by *ri_rnic_query*(). If *tag* refers to ZBVA region or window, *addr.offset* should be set to the desired offset from which to perform the data transfer. If a Memory Window needs to be used as a data sink in RDMA-read type Work Request (iWarp only), then *tag.rtag* must be set since Memory Window does not have an *Ltag*.
id	An ID that identifies a Work Request. This is set by the Consumer at the time of posting a Work Request to send or Receive Queue and is returned by the implementation upon completion of the Work Request so that the Consumer can recognize which Work Request completed.
num_sgle	Number of valid scatter/gather elements in *sg_list.*
imm_data	4-byte Immediate data that can accompany either Send or RDMA-write Work Request types over IB transport. Not valid for iWarp transport. The message that contains immediate data always consumes a receive Work Request on the remote Receive Queue, but the 4-byte data is filled into the Work Completion.
rtag	RTag of the remote buffer for RDMA write/read and atomic operations. For bind Work Request type, this represents the current RTag of the window for remote use if the Consumer does not own a key portion of STag and new RTag of the window for remote use if the Consumer owns the key portion. (The Consumer always owns the key portion for Narrow Memory Windows. For Wide Memory Windows, the Consumer owns the key portion if the RNIC indicates “extended” index/key split support through the RI_RNIC_EXT_INDEX_KEY_SPLIT flag.) In any case, it contains the new RTag upon successful completion of the call. For fast register Work Request type, this represents the *rtag* of the region to be re-registered or unmapped RTag obtained through *ri_ltag_alloc*(). For local invalidate Work Request type, this represents the *rtag* of the region to be invalidated. For RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE, this represents the *rtag* of the remote region that is targeted. For RI_WR_TYPE_SEND_INVALIDATE, this represents the *rtag* of the remote region that is to be invalidated after the data is delivered. The region of the matching receive buffer on the remote may be different from the region that is invalidated. *rtag* is undefined and ignored by the implementation for fast register and local invalidate Work Request types if the region or window identified by *mr_handle/mw_handle* does not have remote access enabled.
rdma_addr	Target virtual address of the remote buffer for RDMA write/read and atomic operations.
mw_handle	Handle to the Memory Window to do bind on. For local invalidate Work Request type, this represents the handle to the window to be invalidated.

Work Request Field	Description
mr_handle	Handle to the Memory Region to bind to. For fast register Work Request type, this represents the handle to the region to be re-registered or unmapped LTag obtained through *ri_ltag_alloc*(). For local invalidate Work Request type, this represents the handle to the region to be invalidated.
ltag	*Ltag* (STag/L-Key for local use) of the region to bind to. For fast register Work Request type, the "index" portion of this field must be the same as the *Ltag* of the region to be re-registered or the unmapped *Ltag* obtained through *ri_ltag_alloc*(). The "key" portion of this field contains the new value for the *Ltag* of the region being registered. For local invalidate Work Request types, this represents the *Ltag* of the region to be invalidated. It is undefined if a Memory Window is invalidated.
vaddr	Virtual address of the scatter/gather element in Send, RDMA, and atomic operations. For window bind Work Request types, this represents the starting address of the Memory Window.
length	Length of the scatter/gather element in Send, RDMA, and atomic operations. For window bind Work Request types, this represents the length of the Memory Window.
mem_attr	Attributes for a fast registered Memory Region or a bound Memory Window. See *ri_mem_attr_t*.
phys_buf_list, nbufs pble_size	See *ri_mr_reg_phys*() for description of these fields.
IB-Specific Parameters	
atomic_operand1	Quantity used to compare against (RI_WR_TYPE_COMPARE_ AND_SWAP) or add to (RI_WR_TYPE_FETCH_AND_ADD).
atomic_operand2	Quantity that is swapped with the remote if compare operation succeeds. Ignored for RI_WR_TYPE_FETCH_AND_ADD.
atomic_result	Original value of the remote data prior to the atomic operation is deposited here.
remote_node_address	Valid only for UD. Represents the address handle.
dest_qp	Remote QP number/ID to be targeted.
dest_qkey	Destination QKey.
max_static_rate	Maximum static rate.

CONSUMER USAGE

None.

IMPLEMENTATION NOTES

None.

SEE ALSO

None.

9 Implementers Guide

9.1 RNIC Management

Figure 3 shows a possible scenario of complete RNICPI Consumer registrations in both privileged and non-privileged domains initiated by IT-API style interface creation calls. It assumes demand-loaded kVP and uVP components which are matching Consumer-required RNIC attributes. Using the example sequence chart of Figure 3, this section discusses the main activities during RNIC provider modules initialization and Consumer registration. Following the implementation model depicted in Figure 2, the RNIC-specific components of the end system's RDMA stack comprise uVP and kVP; OS generic modules are uAL and kAL.

9.1.1 Registration of an RNIC with the Operating System

In a typical OS environment, some initial mechanism outside the scope of RNICPI may load and/or initialize the "device driver" code module which implements the vendor-specific hardware control functions. In the described scenario it is assumed that module loading is triggered by a kernel Consumer's request to open an RDMA interface (*it_ia_create*). As a result of the module initialization it will have called the *ri_svc_attach*() interface to represent itself as the kernel RDMA verbs provider (kVP) to the kAL. This call provides the mandatory P-RNICPI function vector *ri_ops_t* implementing Consumer (e.g., kAL) controlled thread safety and may also provide the optional provider controlled thread safety interface (see Section 4.3). Furthermore, it provides to the kAL a Verb Provider handle to the RNIC (*vp_handle_t*) and the *ri_svcs_dev_arg_t* parameter which contains among others information about the supported RNICPI version and the load path of the user space library (uVP).

An RNIC is now unambiguously referenced by the combination of *vp_handle_t* and the current *ri_ops_t* interface function vector. Note that only the combination of both parameters can make the selection of an RNIC unambiguous, if potentially multiple kVPs and/or RNICs are available in the system. Point (1) in Figure 3 represents this status of the kAL.

9.1.2 Opening an RNIC

Before any RDMA service can be used, the RNIC device must be configured regarding its physical buffer list mode: the *ri_rnic_open*() call requests either Block List Mode or Page List Mode as RNIC global behavior (2). This mode remains selected for all offered RDMA services until the RNIC gets closed.

9.1.3 Registering a Consumer with the P-RNICPI

Before using RDMA services, each RNICPI Consumer must first obtain its private RNIC handle. The Consumer private RNIC handle is obtained via the *ri_rnic_get_hdl*() interface.

It is defined that only at this point the Consumer may choose the thread safety model of the VP: if the optional verbs provider controlled resource access serialization is available (i.e., the *vp_locking_ops* OUT parameter of the *ri_svc_attach*() call was non-zero and not equal to *al_locking_ops*), an *ri_rnic_get_hdl*() call using this method of the *vp_locking_ops* will set this behavior. When using the method from the default *al_locking_ops* vector, default behavior will be in place (3). After this implicit selection of the thread safety model, any RNICPI call using the same Consumer's RNIC handle but a method representing a different locking model will lead to undefined behavior of the RDMA services for this Consumer.

After a successful call to *ri_rnic_get_hdl*(), the privileged Consumer may access RDMA services using its private *rnic* handle (4).

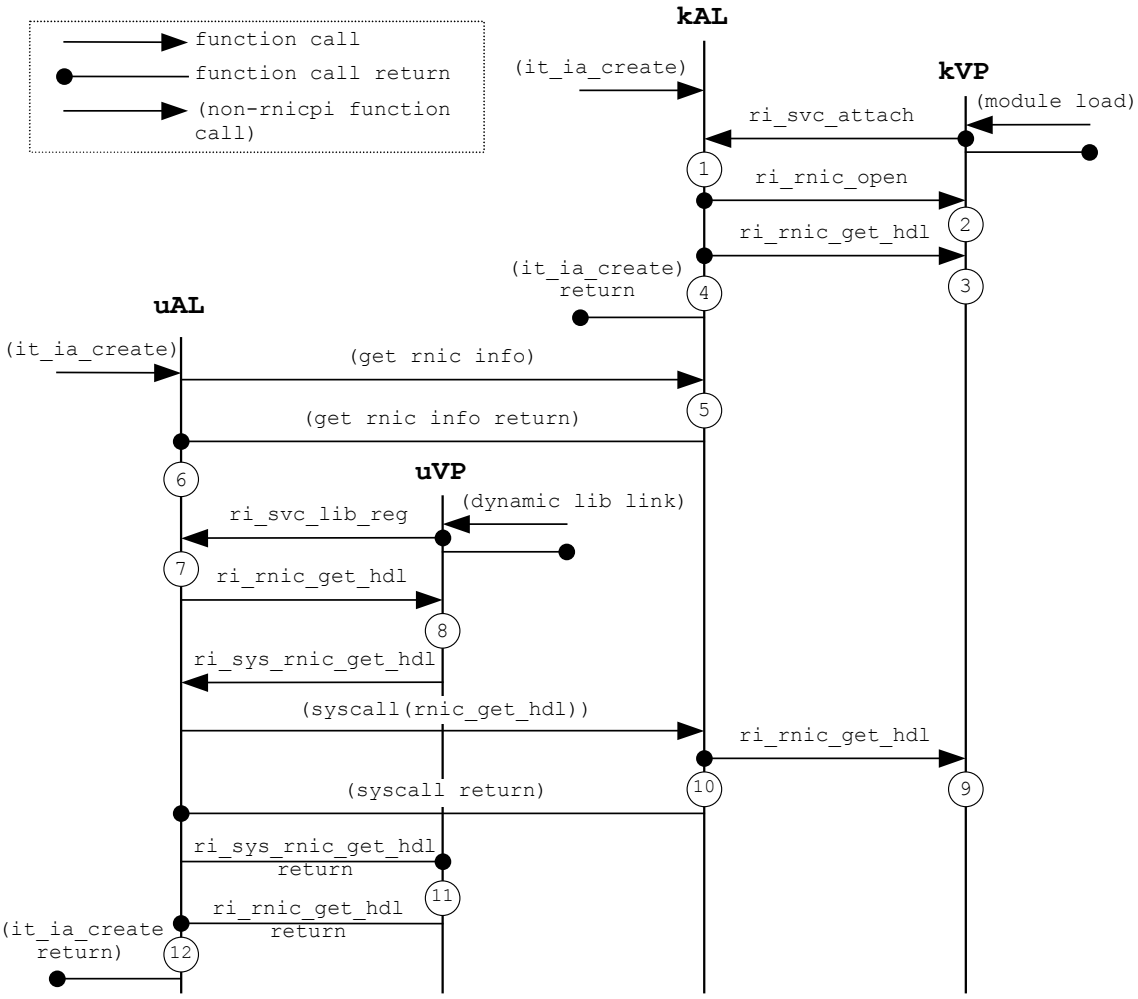

Figure 3: RNIC Management and Consumer Registration

9.1.4 Selection and Registration of an uVP with the uAL

The RNICPI assumes some private communication between uAL and kAL to allow the uAL selecting the correct uVP library. In a typical scenario, selection and loading of the uVP may be triggered by a Consumer's RDMA service request qualified by a remote IP address and a requested RNICPI interface version. By private uAL/kAL communication and relying on the kAL interface mapping service (see Section 3.2.3), the uAL will receive the library loading path for the uVP. This may imply that the kAL must load the corresponding kVP before providing this information back to the uAL (the library load path gets provided by the *ri_svc_attach*() routine; see above). It is expected that the kAL will also establish a unique mapping of the kVPs *vp_handle_t* to the application-level Consumer (5) and provide the mapped *vp_handle_t* together with the uVP library load path back to the uAL (6).

The uAL now can dynamically link the uVP code. Similar to the *ri_svc_attach*() operation described for the privileged domain, during linking and initializing the user-level module will call the *ri_svc_lib_reg*() interface to register itself with the uAL as uVP (7).

In a next step, the Consumer will request its RNIC handle via *ri_rnic_get_hdl*() and thereby implicitly select the desired level of thread safety provision (8). The *ri_rnic_get_handle* call will result in the uVP's call to the SYS-RNICPI (*ri_sys_rnic_get_hdl*) which is routed to the kAL and then mapped into a P-RNICPI call to the appropriate kVP. After the successful P-RNICPI call, the kVP will have established context information related to this new Consumer (9) and the kAL will have created the necessary state information to police and forward all further slow path activities of the new user space RNICPI Consumer (10). The complete call stack of the *ri_sys_rnic_get_hdl*, uAL/kAL communication, and *ri_rnic_get_hdl*() at the P-RNICPI transports opaque *vp_data* to synchronize uVP and kVP during resource setup. After returning from the SYS-RNICPI call, the uVP will have initialized its *rnic* handle-related structures (11) and the *rnic* handle is valid to be used for requesting further RDMA services (12).

Note that the RNICPI does not allow the setting of the PBL mode from the NP-RNICPI. The *ri_rnic_open*() call is available only at the P-RNICPI.

10 RNICPI Header Files

10.1 <rnicpi.h>

```
/*
 * rnicpi.h
 *
 * Copyright © 2005 The Open Group
 * All rights reserved.
 * The copyright owner hereby grants permission for all or part of this
 * publication to be reproduced, stored in a retrieval system, or
 * transmitted, in any form or by any means, electronic, mechanical,
 * photocopying, recording, or otherwise, provided that it remains
 * unchanged and that this copyright statement is included in all
 * copies or substantial portions of the publication.
 *
 * For any software code contained within this specification,
 * permission is hereby granted, free-of-charge, to any person
 * obtaining a copy of this specification (the "Software"), to deal in
 * the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom
 * the Software is furnished to do so, subject to the above copyright
 * notice and this permission notice being included in all copies or
 * substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
 * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Permission is granted for implementers to use the names, labels,
 * etc. contained within the specification. The intent of publication
 * of the specification is to encourage implementations of the
 * specification. This specification has not been verified for
 * avoidance of possible third-party proprietary rights. In
 * implementing this specification, usual procedures to ensure the
 * respect of possible third-party intellectual property rights should
 * be followed.
 */

#ifndef _RNICPI_H_
#define _RNICPI_H_
#include <rnicpi_osd.h>
```

```
6472
6473            typedef enum ri_api_return ri_api_return_t;
6474            enum ri_api_return {
6475                RI_RET_SUCCESS=0,
6476
6477                /* generic errors */
6478                RI_RET_RESOURCE_SHORTAGE,
6479                RI_RET_INVALID_PARAMETER,
6480                RI_RET_NO_PERMISSION,
6481                RI_RET_XPORT_UNSUPPORTED,
6482                RI_RET_OP_UNSUPPORTED,
6483                RI_RET_BUFFER_TOO_SMALL,
6484                RI_RET_FAIL,
6485
6486                /* options */
6487                RI_RET_QP_RESIZE_UNSUPPORTED,
6488                RI_RET_SRQ_UNSUPPORTED,
6489                RI_RET_SRQ_RESIZE_UNSUPPORTED,
6490                RI_RET_EXT_BMM_UNSUPPORTED,
6491                RI_RET_EXT_ZBVA_UNSUPPORTED,
6492                RI_RET_EXT_LIF_UNSUPPORTED,
6493                RI_RET_RNIC_ATOMICS_UNSUPPORTED,
6494
6495                /* VP */
6496                RI_RET_VP_INVALID_HANDLE,
6497                RI_RET_INVALID_DEV_INFO,
6498                RI_RET_INVALID_HW_PATH,
6499                RI_RET_INVALID_LIB_PATH,
6500                RI_RET_INVALID_PBL_MODE,
6501                RI_RET_INVALID_VP_VERSION,
6502
6503                /* RNIC */
6504                RI_RET_RNIC_INVALID_HANDLE,
6505                RI_RET_RNIC_INUSE,
6506                RI_RET_RNIC_NOT_OPEN,
6507
6508                /* PD */
6509                RI_RET_PD_INVALID_HANDLE,
6510                RI_RET_PD_INUSE,
6511
6512                /* CQ */
6513                RI_RET_CQ_INVALID_HANDLE,
6514                RI_RET_CQ_CAPACITY_EXCEEDED,
6515                RI_RET_CQ_EMPTY,
6516                RI_RET_CQ_ERROR,
6517                RI_RET_CQ_INUSE,
6518                RI_RET_CQ_INVALID_EVENT_HANDLER,
6519                RI_RET_CQ_NO_EVENT_HANDLER,
6520                RI_RET_CQ_WOULD_OVERFLOW,
6521                RI_RET_INVALID_HANDLER_ID,
6522
6523                /* QP */
6524                RI_RET_QP_INVALID_HANDLE,
6525                RI_RET_MIGRATION_STATE_INVALID,
```

```
6526                RI_RET_QP_ATTRIBUTE_CANT_CHANGE,
6527                RI_RET_QP_CAPACITY_EXCEEDED,
6528                RI_RET_QP_INVALID_SERVICE_TYPE,
6529                RI_RET_QP_INVALID_SPECIAL_TYPE,
6530                RI_RET_QP_INVALID_STATE,
6531                RI_RET_QP_CANNOT_SHRINK,
6532                RI_RET_QP_INUSE,
6533                RI_RET_RNIC_INVALID_MTU,
6534                RI_RET_RNIC_INVALID_PKEY_ENTRY,
6535                RI_RET_RNIC_INVALID_PKEY_INDEX,
6536                RI_RET_RNIC_INVALID_PORT,
6537                RI_RET_RNIC_INVALID_RNR_VALUE,
6538                RI_RET_RNIC_IRD_ORD_EXCEEDED,
6539                RI_RET_RNIC_NOT_IN_BLOCK_MODE,
6540                RI_RET_RNIC_NOT_IN_PAGE_MODE,
6541                RI_RET_RNIC_SG_LIMIT_EXCEEDED,
6542                RI_RET_RNIC_WR_LIMIT_EXCEEDED,
6543                RI_RET_STILL_FLUSHING_WQES,
6544                RI_RET_WR_INVALID_FLAGS,
6545                RI_RET_WR_INVALID_TYPE,
6546                RI_RET_WR_SG_INVALID_FORMAT,
6547
6548                /* SRQ */
6549                RI_RET_SRQ_INVALID_HANDLE,
6550                RI_RET_QP_TIED_TO_SRQ,
6551                RI_RET_SRQ_CANNOT_SHRINK,
6552                RI_RET_SRQ_IN_ERROR,
6553                RI_RET_SRQ_INUSE,
6554                RI_RET_SRQ_LIMIT_OUT_OF_RANGE,
6555
6556                /* MR */
6557                RI_RET_MR_INVALID_HANDLE,
6558                RI_RET_MEM_INVALID_ACCESS,
6559                RI_RET_MEM_INVALID_ADDRESS,
6560                RI_RET_MEM_INVALID_LENGTH,
6561                RI_RET_MEM_INVALID_OFFSET,
6562                RI_RET_MEM_INVALID_LTAG,
6563                RI_RET_MEM_INVALID_PBLE,
6564                RI_RET_MEM_INVALID_PBLE_LENGTH,
6565                RI_RET_MEM_INVALID_RTAG,
6566
6567                /* MW */
6568                RI_RET_MW_INVALID_HANDLE,
6569                RI_RET_INVALID_WINDOW_TYPE,
6570                RI_RET_WINDOWS_BOUND,
6571                RI_RET_WINDOW_TYPE_UNSUPPORTED,
6572
6573                /* AH */
6574                RI_RET_ADDRESS_HANDLE_INVALID,
6575                RI_RET_MGID_INVALID,
6576                RI_RET_MLID_INVALID,
6577                RI_RET_MULTICAST_ATTACHED,
6578                RI_RET_RNIC_QPS_PER_MCAST_PORT_EXCEEDED,
6579
```

```
6580            /* LLP */
6581            RI_RET_INVALID_LLP_HANDLE,
6582            RI_RET_INVALID_DDP_RDMAP_VERSION,
6583            RI_RET_LLP_BAD_STATE,
6584            RI_RET_MPA_ATTR_UNSUPPORTED
6585        };
6586
6587        typedef uint8_t ri_sl_t;  /* Service Level (4 bits) */
6588        typedef uint8_t ri_lmc_t;  /* LID Mask Control (7 bits) */
6589        typedef uint16_t ri_lid_t;  /* LID address */
6590        typedef uint16_t ri_pkey_t;  /* P_Key */
6591
6592        typedef struct ri_gid ri_gid_t;
6593        struct ri_gid {  /* GID address */
6594            uint64_t  gid_prefix;
6595            uint64_t  gid_guid;
6596        };
6597
6598        typedef enum ri_ib_port_state ri_ib_port_state_t;
6599        enum ri_ib_port_state {  /* link/port states */
6600            RI_IB_LINK_DOWN = 1 << 1,
6601            RI_IB_LINK_INITIALIZE = 1 << 2,
6602            RI_IB_LINK_ARM = 1 << 3,
6603            RI_IB_LINK_ACT_DEFER = 1 << 4,  /* transient */
6604            RI_IB_LINK_ACTIVE = 1 << 5  /* "up" state */
6605        };
6606        /* how IB verbs define port up and down */
6607        #define RI_IB_PORT_DOWN \
6608        (RI_IB_LINK_DOWN|RI_IB_LINK_INITIALIZE|RI_IB_LINK_ARM)
6609        #define RI_IB_PORT_UP \
6610        (RI_IB_LINK_ACT_DEFER|RI_IB_LINK_ACTIVE)
6611
6612        typedef enum ri_ib_mtu ri_ib_mtu_t;
6613        enum ri_ib_mtu {  /* MTU sizes */
6614            RI_IB_MTU_256 = 1,
6615            RI_IB_MTU_512,
6616            RI_IB_MTU_1K,
6617            RI_IB_MTU_2K,
6618            RI_IB_MTU_4K
6619        };
6620
6621        /* port capability flags */
6622        typedef enum ri_ib_port_flags ri_ib_port_flags_t;
6623        enum ri_ib_port_flags {  /* bit positions match portinfo mask */
6624            RI_IB_PORT_SM = 1 << 0,
6625            RI_IB_PORT_SM_DISABLED = 1 << 10,
6626            RI_IB_PORT_SNMP_TUNNEL = 1 << 17,
6627            RI_IB_PORT_DEV_MGT = 1 << 19,
6628            RI_IB_PORT_VENDOR_CLASS = 1 << 20,
6629            RI_IB_PORT_CLIENT_REREG = 1 << 25
6630        };
6631
6632        /* port attributes */
6633        typedef struct ri_ib_port_attr ri_ib_port_attr_t;
```

```
6634        struct ri_ib_port_attr {
6635            ri_gid_t           *src_gid_table;  /* valid in armed/active */
6636            ri_pkey_t          *pkey_table;  /* valid in armed/active */
6637            uint32_t            max_msg_size;  /* in bytes */
6638            ri_ib_mtu_t         max_mtu;  /* max MTU & UD msg size */
6639            ri_ib_port_state_t  ib_port_state;
6640            ri_ib_port_flags_t  ib_port_flags;
6641            ri_lid_t            base_lid;  /* valid in armed/active */
6642            ri_lid_t            sm_lid;  /* valid in armed/active or 0xFFFF */
6643            uint16_t            bad_pkey_count;  /* zero if unsupported */
6644            uint16_t            bad_qkey_count;  /* zero if unsupported */
6645            uint8_t             src_gid_table_size;
6646            ri_lmc_t            lmc;  /* valid in armed/active */
6647            uint8_t             max_vls;  /* max VLs (4 bits) */
6648            ri_sl_t             sm_sl;  /* SL to reach SM address */
6649            uint8_t             subnet_timeout;
6650            uint8_t             init_type_reply;  /* zero if unsupported */
6651        };
6652
6653        /* IB HCA flags */
6654        typedef enum ri_ib_rnic_flags ri_ib_rnic_flags_t;
6655        enum ri_ib_rnic_flags {
6656            RI_IB_APM = 1 << 0,  /* auto path migration */
6657            RI_IB_RNR_NAK = 1 << 1,  /* RC RNR-NAK support */
6658            RI_IB_AH_PORT_CHECK = 1 << 2,  /* addr hdl port check */
6659            RI_IB_CAN_CHANGE_PRIMARY_PORT = 1 << 3,  /* in SQD state */
6660            RI_IB_CAN_CURRENT_QP_STATE = 1 << 4,  /* in Modify QP */
6661
6662            RI_IB_ATOMICS = 1 << 5,  /* atomic operations */
6663            /* if atomics & !ALL_COMPONENTS, then atomic only in HCA */
6664            RI_IB_ATOMICS_ALL_COMPONENTS = 1 << 6,
6665
6666            /* if BMM supported & !TYPE_2B, then type2A windows */
6667            RI_IB_TYPE_2B_WINDOWS = 1 << 7,
6668            RI_IB_MULTIPLE_BLOCK_SIZES_PER_REGION = 1 << 8,
6669
6670            RI_IB_UD_MULTICAST = 1 << 9,
6671
6672            RI_IB_PORT_ACTIVE_EVENT = 1 << 10,
6673            RI_IB_SHUTDOWN_PORT = 1 << 11,  /* in Modify RNIC */
6674
6675            RI_IB_PKEY_CNTR = 1 << 12,  /* P_Key Violation cnt */
6676            RI_IB_QKEY_CNTR = 1 << 13,  /* Q_Key Violation cnt */
6677            RI_IB_SI_GUID = 1 << 14,  /* System Image GUID */
6678            RI_IB_INIT_TYPE = 1 << 15  /* InitType support */
6679        };
6680
6681        typedef struct ri_ib_rnic_attr ri_ib_rnic_attr_t;
6682        struct ri_ib_rnic_attr {
6683            uint64_t            max_mr_size;  /* in bytes */
6684            uint64_t            node_guid;  /* Node GUID */
6685            uint64_t            system_image_guid;  /* zero if not supported */
6686            ri_ib_port_attr_t  *port_attrs;
6687            ri_ib_rnic_flags_t  ib_rnic_flags;
```

```
6688            uint_t              max_addr_hdls;  /* max UD Address Handles */
6689
6690            /* multicast fields are zero, if multicast unsupported */
6691            uint_t    max_mcgs;  /* max num multicast groups */
6692            uint32_t  max_multicast_qps;  /* max MCG attachable QPs */
6693            uint32_t  max_qps_per_mcg;  /* max num of QPs per MCG */
6694
6695            uint16_t  partition_table_size;  /* size must be >= 1 */
6696            uint8_t   num_ports;
6697            uint8_t   local_ack_delay;  /* 5 bits */
6698        };
6699
6700        typedef enum ri_rdma_transport_type ri_rdma_transport_type_t;
6701        enum ri_rdma_transport_type {
6702            RI_RDMA_TRANSPORT_TYPE_IB,
6703            RI_RDMA_TRANSPORT_TYPE_IWARP
6704        };
6705
6706        typedef enum ri_iwarp_rnic_flags ri_iwarp_rnic_flags_t;
6707        enum ri_iwarp_rnic_flags {
6708            RI_IWARP_CAN_MODIFY_IRD = 1 << 0,
6709            RI_IWARP_CAN_INCREASE_ORD = 1 << 1,
6710
6711            /* supported IP transports */
6712            RI_IWARP_TCP_SUPPORTED = 1 << 2,
6713            RI_IWARP_SCTP_SUPPORTED = 1 << 3,
6714
6715            /* if IETF and !PERMISSIVE, RNIC is non-permissive */
6716            RI_IWARP_IETF_PERMISSIVE = 1 << 4,
6717
6718            /* marker & CRC preference for MPA negotiation (over TCP) */
6719            RI_IWARP_PREFER_MARKERS = 1 << 5,
6720            RI_IWARP_PREFER_CRC = 1 << 6,
6721
6722            /* modify/query QP has KEEPALIVE option support */
6723            RI_IWARP_KEEPALIVE_SUPPORTED = 1 << 7
6724        };
6725
6726        typedef struct ri_iwarp_port_attr ri_iwarp_port_attr_t;
6727        struct ri_iwarp_port_attr {
6728            struct in6_addr *ipv6_addrs;
6729            struct in_addr  *ipv4_addrs;
6730            uint_t           port_num;
6731            uint_t           num_ipv4_addrs;
6732            uint_t           num_ipv6_addrs;
6733        };
6734
6735        typedef struct ri_iwarp_rnic_attr ri_iwarp_rnic_attr_t;
6736        struct ri_iwarp_rnic_attr {
6737            ri_iwarp_port_attr_t  *port_attrs;
6738            uint_t                 protocol_version;  /* RDMAC = 0, IETF = 1 */
6739            uint_t                 num_ports;
6740            ri_iwarp_rnic_flags_t  iwarp_rnic_flags;
6741        };
```

```
6742
6743        typedef union ri_transport_attr_ptr ri_transport_attr_ptr_t;
6744        union ri_transport_attr_ptr {
6745            ri_ib_rnic_attr_t    *ib;
6746            ri_iwarp_rnic_attr_t *iwarp;
6747        };
6748
6749        typedef enum ri_rnic_flags ri_rnic_flags_t;
6750        enum ri_rnic_flags {
6751            /* iWARP will always have BMM, ZBVA, & LIF set */
6752            RI_RNIC_EXT_SRQ = 1 << 0,  /* IB 1.1: not set */
6753            RI_RNIC_EXT_BMM = 1 << 1,  /* IB 1.1: not set */
6754
6755            /* note that ZBVA, LIF, & BLOCK_MODE require BMM */
6756            RI_RNIC_EXT_ZBVA = 1 << 2,  /* IB 1.1: not set */
6757            RI_RNIC_EXT_LIF = 1 << 3,  /* IB 1.1: not set */
6758
6759            /* if BLOCK_MODE not set, RNIC is in page mode */
6760            RI_RNIC_EXT_BLOCK_MODE = 1 << 4,  /* IB 1.1: not set */
6761
6762            RI_RNIC_RQ_OVERFLOW_CHECKING = 1 << 5,  /* IB: set */
6763            RI_RNIC_CQ_OVERFLOW_CHECKING = 1 << 6,  /* IB: set */
6764
6765            RI_RNIC_CAN_RESIZE_QP = 1 << 7,
6766
6767            /* if RI_RNIC_EXT_SRQ not set, then SRQ stuff not defined */
6768            RI_RNIC_CAN_RESIZE_SRQ = 1 << 8,
6769            RI_RNIC_SRQ_SEQ_DEQUEUE = 1 << 9,  /* IB SRQ: not set */
6770
6771            RI_RNIC_SRQ_LAST_WQE_ASYNC = 1 << 10,  /* IB SRQ: set */
6772
6773            /* safe to post after successful modify to RTS */
6774            RI_RNIC_SAFE_POST_ON_RTS = 1 << 11,  /* IB: set */
6775
6776            /* narrow windows support is part of RI_RNIC_EXT_BMM */
6777            RI_RNIC_WIDE_WINDOWS_SUPPORTED = 1 << 12,  /* IB: set */
6778            /*
6779            * extended key splitting
6780            * wide windows and virtual MRS, also use index/key splitting
6781            */
6782            RI_RNIC_KEY_INDEX_KEY_SPLIT = 1 << 13, /* IB 1.1: 5320 : not set */
6783
6784            /* supported WR/WC opaque bits */
6785            RI_RNIC_CONSUMER0_OPAQUE = 1 << 14,
6786            RI_RNIC_CONSUMER1_OPAQUE = 1 << 15,
6787            RI_RNIC_CONSUMER2_OPAQUE = 1 << 16,
6788
6789            /*
6790             * if local solicited is pass thru opaque, then
6791             * LOCAL_SOLICITED_OPAQUE is defined,
6792             * which is the same as CONSUMER2_OPAQUE.
6793             * if local solicited is a real event, then
6794             * LOCAL_SOLICITED_EVENT is defined.
6795             * if NO local solicited support, then neither
```

```
6796                 * is defined.
6797                 */
6798                RI_RNIC_LOCAL_SOLICITED_OPAQUE = 1 << 16,
6799                RI_RNIC_LOCAL_SOLICITED_EVENT = 1 << 17,
6800
6801                /* non-coherent memory support */
6802                RI_RNIC_NON_COHERENT_MEMORY = 1 << 18
6803            };
6804
6805            typedef struct ri_rnic_attr ri_rnic_attr_t;
6806            struct ri_rnic_attr {
6807
6808                /* 64-bit and pointer types here to minimize padding */
6809
6810                /*
6811                 * supported memory page sizes bitmask for page mode,
6812                 * bit position i means page size 2^i size is supported,
6813                 * undefined if using block mode
6814                 */
6815                uint64_t  mem_page_sizes_bit_mask;
6816
6817                /* block mode limits, undefined if using page mode */
6818                uint64_t  block_size_low;  /* IB 1.1: undefined */
6819                uint64_t  block_size_hi;  /* IB 1.1: undefined */
6820
6821                /* transport-specific items */
6822                ri_transport_attr_ptr_t  trans_attr;
6823
6824                /*
6825                 * RNIC identification info
6826                 */
6827
6828                uint32_t  vendor_id;  /* Vendor ID (IB: 24 bits)*/
6829                uint32_t  vendor_part_id;  /* Vendor Part ID (IB: 16 bits)*/
6830                uint32_t  hw_version;  /* Hardware Version */
6831
6832                /* reserved L_Key or stag0 (which is zero, of course) */
6833                uint32_t  res_ltag_value;  /* IB 1.1: undefined */
6834            #define stag0_value res_ltag_value
6835
6836                /*
6837                 * resource limits
6838                 */
6839
6840                uint_t  max_pds;  /* max num of Protection Domains */
6841
6842                uint32_t  max_qps;  /* max num of QPs (IB: 24 bits) */
6843                uint_t    max_wrs_qp;  /* max num of outstanding WRs on a WQ */
6844
6845                /* max number of SRQs, if SRQs unsupported: zero */
6846                uint_t  max_srq;  /* IB 1.1: zero */
6847                /*
6848                 * max num of outstanding WRs on a SRQ,
6849                 * if SRQs, unsupported: zero
```

```
6850                 */
6851                uint_t  max_wrs_srq;  /* IB 1.1: zero */
6852
6853                /*
6854                 * max number of scatter/gather entries per WR,
6855                 * note IB reliable datagram has a separate limit,
6856                 * for IB: send_recvqp = rdma_write = rdma_read,
6857                 * for iWARP: send_recvqp = srq and >= 4
6858                 */
6859                uint_t  max_sgl_send_recvqp;  /* send/recv on QP, not RD */
6860                uint_t  max_sgl_srq;  /* for SRQ */
6861                uint_t  max_sgl_rdma_write;  /* RDMA-W */
6862                uint_t  max_sgl_rdma_read;  /* RDMA-R */
6863
6864                /* largest inline size supported, zero if unsupported */
6865                uint32_t  max_unregistered_inline;
6866
6867                uint_t  max_cqs;  /* max number of CQs */
6868                uint_t  max_cqes_per_cq;  /* max # entries per CQ */
6869
6870                /* max number of CQ event handlers */
6871                uint_t  max_cq_event_handlers;  /* IB 1.1: one */
6872
6873                uint_t  max_mrs;  /* max # of Memory Regions */
6874                uint_t  max_mws;  /* max number of MWs */
6875
6876                /* max number of physical buffer entries per PBL */
6877                uint_t  max_pbl_size;  /* IB 1.1: undefined */
6878
6879                /* max number of RDMA-Reads & atomic (if supported) ops */
6880                uint_t  max_ird;  /* incoming for whole RNIC */
6881                uint_t  max_ird_per_qp;  /* incoming per QP */
6882                uint_t  max_ord;  /* outgoing for whole RNIC */
6883                uint_t  max_ord_per_qp;  /* outgoing per QP */
6884
6885                ri_rdma_transport_type_t  transport_type;  /* IB or IWARP */
6886
6887                ri_rnic_flags_t  rnic_flags;  /* various boolean flags */
6888
6889                /* Additional Vendor Info (NULL terminated) */
6890                char  addl_vendor_info[RI_MAX_VENDOR_STRING_LEN];
6891            };
6892
6893            typedef enum ri_pbl_mode ri_pbl_mode_t;
6894            enum ri_pbl_mode {
6895                RI_PHYS_PAGE_LIST_MODE,
6896                RI_PHYS_BLOCK_LIST_MODE
6897            };
6898
6899            typedef ri_api_return_t (*ri_op_rnic_open_t)(
6900                IN      ri_vp_handle_t  vp_hdl,
6901                IN      ri_pbl_mode_t   pbl_mode,
6902                IN OUT  ri_os_data_t    os_data,
6903                OUT     ri_err_data_t  *err_data
```

```
);

typedef ri_api_return_t (*ri_op_rnic_close_t)(
    IN   ri_vp_handle_t  vp_hdl,
    OUT  ri_err_data_t  *err_data
);

typedef enum ri_consumer_domain ri_consumer_domain_t;
enum ri_consumer_domain {
    RI_CONSUMER_LOCATION_KERNEL,
    RI_CONSUMER_LOCATION_USER
};

typedef uint32_t ri_ltag_t;
typedef uint32_t ri_rtag_t;

typedef enum ri_stag_state ri_stag_state_t;
enum ri_stag_state {
    RI_STAG_STATE_INVALID,
    RI_STAG_STATE_VALID
};

typedef struct ri_phys_buf ri_phys_buf_t;
struct ri_phys_buf {
    ri_phys_addr_t     phys_address;
    ri_phys_buf_len_t  phys_len;
};

typedef enum ri_mr_rereg ri_mr_rereg_t;
enum ri_mr_rereg {
    RI_MR_REREG_TRANSLATE  = 1 << 0,
    RI_MR_REREG_PD         = 1 << 1,
    RI_MR_REREG_ACCESS     = 1 << 2
};

typedef enum ri_mem_attr ri_mem_attr_t;
enum ri_mem_attr {
    RI_MEM_FLAGS_LOCAL_READ   = 1 << 0,
    RI_MEM_FLAGS_LOCAL_WRITE  = 1 << 1,
    RI_MEM_FLAGS_LOCAL        = (RI_MEM_FLAGS_LOCAL_READ|
                                 RI_MEM_FLAGS_LOCAL_WRITE),

    RI_MEM_FLAGS_REMOTE_WRITE       = 1 << 2,
    RI_MEM_FLAGS_REMOTE_READ        = 1 << 3,
    RI_MEM_FLAGS_REMOTE             = (RI_MEM_FLAGS_REMOTE_WRITE|
                                       RI_MEM_FLAGS_REMOTE_READ),
    RI_MEM_FLAGS_ATOMICS            = 1 << 4,
    RI_MEM_FLAGS_BINDING            = 1 << 5,
    RI_MEM_FLAGS_CONSUMER_OWNS_KEY  = 1 << 6,
    RI_MEM_FLAGS_ZBVA               = 1 << 7,
    RI_MEM_FLAGS_NONCOHERENT        = 1 << 8,
    RI_MEM_FLAGS_USER_VA            = 1 << 9,
    RI_MEM_FLAGS_SHARED_REG_IOVA    = 1 << 10
};
```

```
typedef uint64_t ri_addr_t;

typedef struct ri_mr_reg_attr ri_mr_reg_attr_t;
struct ri_mr_reg_attr {
    ri_addr_t       va;
    ri_length_t     length;
    ri_mem_attr_t   mem_attr;
    ri_pd_handle_t  pd;
};

typedef struct ri_mr_reg_phys_attr ri_mr_reg_phys_attr_t;
struct ri_mr_reg_phys_attr {
    ri_mr_reg_attr_t  common;
    void             *phys_buf_list;
    uint_t            nbufs;
    uint_t            pble_size;
    uint_t            address_offset;
};

/*
 * alternate: type of phys_buf_t could be union of
 * ri_phys_addr_t * and ri_phys_buf_t *
 */

typedef enum ri_mw_type ri_mw_type_t;
enum ri_mw_type {
    RI_MW_TYPE_1, RI_MW_TYPE_WIDE=RI_MW_TYPE_1,
    RI_MW_TYPE_2, RI_MW_TYPE_NARROW=RI_MW_TYPE_2
};

typedef struct ri_srq_attr ri_srq_attr_t;
struct ri_srq_attr {
    uint32_t        max_wrs;
    uint32_t        max_sgl;
    uint32_t        lowat;
    ri_pd_handle_t  pd;
    ri_boolean_t    srq_armed;
};

typedef enum ri_svc_type ri_svc_type_t;
enum ri_svc_type {
    RI_SERVICE_RELIABLE_CONNECTION,
    RI_SERVICE_UNRELIABLE_DATAGRAM
};

/* Some states are not valid for iWARP/IB NICs.
 * Attempts to modify the state attribute will fail
 * with RI_RET_INVALID_STATE if not appropriate
 * for the technology.
 */

typedef enum ri_qp_state ri_qp_state_t;
enum ri_qp_state {
```

```
        RI_QP_STATE_IDLE,
        RI_QP_STATE_READY_TO_SEND,
        RI_QP_STATE_ERROR,

        /* The following states are valid only for iWARP */
        RI_QP_STATE_TERMINATE,
        RI_QP_STATE_CLOSING,

        /* The following states are valid only for IB */
        RI_QP_STATE_SEND_QUEUE_ERROR,
        RI_QP_STATE_INITIALIZED,
        RI_QP_STATE_READY_TO_RECEIVE,
        RI_QP_STATE_SEND_QUEUE_DRAINING,
        RI_QP_STATE_SEND_QUEUE_DRAINED,
        RI_QP_STATE_TIMEWAIT
    };

    typedef struct ri_mpa_attr ri_mpa_attr_t;
    struct ri_mpa_attr {
        ri_boolean_t  recv_marker_enabled;
        ri_boolean_t  xmit_marker_enabled;
        ri_boolean_t  crc_enabled;
        uint32_t      ddp_rdmap_version;  /* 0 or 1 */
    };

    typedef struct ri_iwarp_qp_attr ri_iwarp_qp_attr_t;

    struct ri_iwarp_qp_attr {
        ri_boolean_t            enable_bind;
        char                    terminate_buffer[52];
        uint32_t                terminate_msg_length;
        ri_boolean_t            is_terminate_local;
        ri_llp_stream_handle_t  llp_stream_handle;
        ri_mpa_attr_t           mpa_attrs;
        char                    *stream_msg_buf;
        uint32_t                stream_msg_buf_length;
        uint32_t                rq_hiwat;
        ri_boolean_t            rq_armed;
    };

    typedef struct ri_gen_address_vector ri_gen_address_vector_t;
    struct ri_gen_address_vector {
        uint8_t   sl;
        uint8_t   send_global_routing_hdr_flg;
        uint8_t   hop_limit;
        uint8_t   source_gid_index;
        uint32_t  flow_label;
        ri_gid_t  dgid;
        ri_lid_t  dlid;
        uint8_t   traffic_class;
        uint8_t   max_static_rate;
        uint8_t   source_path_bits;
    };
```

```
typedef struct ri_path_address_info ri_path_address_info_t;
struct ri_path_address_info {
    uint16_t                pkey_index;
    uint8_t                 physical_port;
    uint16_t                path_mtu;
    uint8_t                 local_ack_timeout;
    uint8_t                 retry_count;
    uint8_t                 rnr_retry_count;
    ri_gen_address_vector_t  av;
};

typedef enum ri_path_migration_state  ri_path_migration_state_t;
enum ri_path_migration_state {
    RI_MIG_STATE_MIGRATED,
    RI_MIG_STATE_REARM,
    RI_MIG_STATE_ARMED
};

typedef struct ri_ib_qp_attr ri_ib_qp_attr_t;
struct ri_ib_qp_attr {
    /* common to RC and UD */
    uint32_t                  send_psn;
    ri_boolean_t              signal_completions;
    ri_svc_type_t             trans_type;
    uint32_t                  enable_async_notify_for_sqd;
    ri_path_address_info_t    primary_av;
    /* RC-specific attributes */
    uint32_t                  recv_psn;
    uint32_t                  dest_qpn;
    uint32_t                  min_rnrnak_timer;
    ri_path_address_info_t    alternate_av;
    ri_path_migration_state_t mig_state;
    ri_boolean_t              enable_atomic;
    /* UD-specific attributes */
    ri_qp_state_t             current_state;
    uint32_t                  recv_qkey;
};

typedef struct ri_qp_attr ri_qp_attr_t;
struct ri_qp_attr {
    uint32_t                    qp_id;
    union {
        ri_cq_handle_t  scq_handle;
        ri_os_data_t    scq_os_data;
    } scq;
    union {
        ri_cq_handle_t  rcq_handle;
        ri_os_data_t    rcq_os_data;
    } rcq;
    union {
        ri_srq_handle_t  srq_handle;
        ri_os_data_t     srq_os_data;
    } srq;
    ri_pd_handle_t              pd;
```

```
7120            uint32_t                    sq_size;
7121            uint32_t                    rq_size;
7122            ri_boolean_t                srq_valid;
7123            uint32_t                    sq_max_sges;
7124            uint32_t                    sq_max_sges_rdmaw;
7125            uint32_t                    rq_max_sges;
7126            ri_qp_state_t               state;
7127            ri_boolean_t                enable_rdma_read;
7128            ri_boolean_t                enable_rdma_write;
7129            uint32_t                    max_rdma_read_outgoing;
7130            uint32_t                    max_rdma_read_incoming;
7131            ri_boolean_t                enable_stag0_fastreg;
7132            union {
7133                ri_iwarp_qp_attr_t  qp_iwarp;
7134                ri_ib_qp_attr_t     qp_ib;
7135            } qp_xport;
7136        };
7137
7138        typedef enum ri_wr_type ri_wr_type_t;
7139        enum ri_wr_type {
7140            RI_WR_TYPE_SEND,
7141            RI_WR_TYPE_SEND_INVALIDATE,
7142            RI_WR_TYPE_RDMA_WRITE,
7143            RI_WR_TYPE_RDMA_READ,
7144            RI_WR_TYPE_BIND,
7145            RI_WR_TYPE_FAST_REGISTER,
7146            RI_WR_TYPE_LOCAL_INVALIDATE_MR,
7147            RI_WR_TYPE_LOCAL_INVALIDATE_MW,
7148
7149            /* iWarp-specific opcodes */
7150            RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE,
7151
7152            /* IB-specific opcodes */
7153            RI_WR_TYPE_COMPARE_AND_SWAP,
7154            RI_WR_TYPE_FETCH_AND_ADD,
7155            RI_WR_TYPE_SEND_IMMEDIATE,
7156            RI_WR_TYPE_RDMA_WRITE_IMMEDIATE
7157        };
7158
7159        typedef enum ri_wr_flags ri_wr_flags_t;
7160        enum ri_wr_flags {
7161            /* Generate completion event */
7162            RI_WR_SIGNALED_FLAG  = 0x01,
7163
7164            /* Generate notification for completion */
7165            RI_WR_LOCAL_SOLICITED_FLAG  = 0x02,
7166
7167            /* Notify the remote on completion */
7168            RI_WR_SOLICITED_EVENT_FLAG  = 0x04,
7169
7170            /* Block until all previous RDMA reads are completed */
7171            RI_WR_READ_FENCE_FLAG  = 0x08,
7172
7173            /* Block until all previous Work Requests are completed */
```

```
        RI_WR_LOCAL_FENCE_FLAG  = 0x10,

        /* Coalesce Work Requests to the Send queue */
        RI_WR_COALESCE_FLAG  = 0x20,

        /* Hint to allow inlining of data */
        RI_WR_INLINE_OK_FLAG = 0x40

        RI_WR_CONSUMER_FLAG2  = RI_WR_LOCAL_SOLICITED_FLAG,
        RI_WR_CONSUMER_FLAG1  = 1 << 29,
        RI_WR_CONSUMER_FLAG0  = 1 << 30
    };

    typedef struct ri_data_seg ri_data_seg_t;
    struct ri_data_seg {
        union {
            ri_addr_t       vaddr;
            ri_phys_addr_t  paddr;
            ri_length_t     offset;
        } addr;
        union {
            ri_ltag_t  ltag;
            ri_rtag_t  rtag;
        } tag;
        uint32_t len;
    };

    typedef uint64_t  ri_wr_id_t;

    typedef struct ri_wr_common ri_wr_common_t;
    struct ri_wr_common {
        ri_wr_type_t   type;
        ri_wr_flags_t  flags;
        ri_wr_id_t     id;
    };

    typedef struct ri_wr_send ri_wr_send_t;
    struct ri_wr_send {
        ri_wr_type_t   type;
        ri_wr_flags_t  flags;
        ri_wr_id_t     id;
        ri_data_seg_t  *sg_list;
        ri_rtag_t      rtag;
        uint_t         num_sgle;
        uint32_t       imm_data;
    };

    typedef struct ri_wr_rdma_write ri_wr_rdma_write_t;
    struct ri_wr_rdma_write {
        ri_wr_type_t   type;
        ri_wr_flags_t  flags;
        ri_wr_id_t     id;
        ri_data_seg_t  *sg_list;
        uint64_t       rdma_addr;
```

```
    ri_rtag_t      rtag;
    uint_t         num_sgle;
    uint32_t       imm_data;
};

typedef struct ri_wr_rdma_read ri_wr_rdma_read_t;
struct ri_wr_rdma_read {
    ri_wr_type_t   type;
    ri_wr_flags_t  flags;
    ri_wr_id_t     id;
    ri_data_seg_t  *sg_list;
    uint64_t       rdma_addr;
    uint_t         num_sgle;
    ri_rtag_t      rtag;
};

typedef struct ri_wr_bind ri_wr_bind_t;
struct ri_wr_bind {
    ri_wr_type_t    type;
    ri_wr_flags_t   flags;
    ri_wr_id_t      id;
    ri_mw_handle_t  mw_handle;
    ri_rtag_t       rtag;
    ri_mr_handle_t  mr_handle;
    ri_ltag_t       ltag;
    ri_addr_t       vaddr;
    ri_length_t     length;
    ri_mem_attr_t   mem_attr;
};

typedef struct ri_wr_fmr ri_wr_fmr_t;
struct ri_wr_fmr {
    ri_wr_type_t           type;
    ri_wr_flags_t          flags;
    ri_wr_id_t             id;
    ri_mr_reg_phys_attr_t  attr;
    ri_ltag_t              ltag;
    ri_rtag_t              rtag;
    ri_mr_handle_t         mr_handle;
};

typedef union ri_mrw_handle ri_mrw_handle_t;
union ri_mrw_handle {
    ri_mr_handle_t  mr_handle;
    ri_mw_handle_t  mw_handle;
};

typedef struct ri_wr_local_invalidate ri_wr_local_invalidate_t;
struct ri_wr_local_invalidate {
    ri_wr_type_t     type;
    ri_wr_flags_t    flags;
    ri_wr_id_t       id;
    ri_ltag_t        ltag;
    ri_rtag_t        rtag;
```

```
    ri_mrw_handle_t  mrw_handle;
};

typedef struct ri_wr_atomic ri_wr_atomic_t;
struct ri_wr_atomic {
    ri_wr_type_t   type;
    ri_wr_flags_t  flags;
    ri_wr_id_t     id;
    ri_rtag_t      rtag;
    uint64_t       atomic_operand1;
    uint64_t       atomic_operand2;
    ri_data_seg_t  atomic_result;
};

typedef struct ri_wr_datagram  ri_wr_datagram_t;
struct ri_wr_datagram {
    ri_wr_type_t         type;
    ri_wr_flags_t        flags;
    ri_wr_id_t           id;
    ri_data_seg_t        *sg_list;
    uint_t               num_sgle;
    uint32_t             imm_data;
    ri_address_handle_t  remote_node_address;
    uint32_t             dest_qp;
    uint32_t             dest_qkey;
    uint8_t              max_static_rate;
};

typedef union ri_wr ri_wr_t;
union ri_wr {
    ri_wr_common_t            wr_common;
    ri_wr_send_t              wr_send;
    ri_wr_rdma_read_t         wr_rdma_read;
    ri_wr_rdma_write_t        wr_rdma_write;
    ri_wr_bind_t              wr_bind;
    ri_wr_fmr_t               wr_fmr;
    ri_wr_local_invalidate_t  wr_li;
    ri_wr_atomic_t            wr_atomic;
    ri_wr_datagram_t          wr_datagram;
};

typedef ri_api_return_t (*ri_op_rnic_get_hdl_t)(
    IN      ri_vp_handle_t        vp_hdl,
    IN      ri_consumer_domain_t  location,
    OUT     ri_rnic_handle_t     *rnic,
    IN OUT  ri_vp_data_t          vp_data,
    IN OUT  ri_os_data_t          os_data,
    OUT     ri_err_data_t        *err_data
);

typedef ri_api_return_t (*ri_op_rnic_free_hdl_t)(
    IN   ri_vp_handle_t    vp_hdl,
    IN   ri_rnic_handle_t  rnic,
    OUT  ri_err_data_t    *err_data
```

```
);

typedef enum ri_rnic_query_flags ri_rnic_query_flags_t;
enum ri_rnic_query_flags {
    RI_QUERY_RNIC_PORT_INFO = 1 << 0
};

typedef ri_api_return_t (*ri_op_rnic_query_t)(
    IN      ri_rnic_handle_t       rnic,
    IN      ri_rnic_query_flags_t  flags,
    OUT     ri_rnic_attr_t        *attrs,
    IN OUT  uint_t                *attr_size,
    OUT     ri_err_data_t         *err_data
);

typedef enum  ri_rnic_attr_mask ri_rnic_attr_mask_t;
enum ri_rnic_attr_mask {
    RI_IB_RNIC_ATTR_SI_GUID = 1 << 0
};

typedef enum ri_port_attr_mask ri_port_attr_mask_t;
enum ri_port_attr_mask {
    RI_IB_PORT_ATTR_SHUTDOWN = 1 << 0,
    RI_IB_PORT_ATTR_INIT_TYPE = 1 << 1,
    RI_IB_PORT_ATTR_QKEY_RESET = 1 << 2,
    RI_IB_PORT_ATTR_CAP_SET = 1 << 3,
    RI_IB_PORT_ATTR_CAP_FLAG_SM = 1 << 4,
    RI_IB_PORT_ATTR_CAP_FLAG_SNMP = 1 << 5,
    RI_IB_PORT_ATTR_CAP_FLAG_DEVMGT = 1 << 6,
    RI_IB_PORT_ATTR_CAP_FLAG_VENDOR = 1 << 7
};

#define RI_IB_ALL_PORTS (0)

typedef struct ri_port_modify_req ri_port_modify_req_t;
struct ri_port_modify_req {
    ri_port_attr_mask_t  port_attr_mask;
    uint8_t              port_number;
    uint8_t              port_init_type; /* 4 bits */
};

typedef struct ri_rnic_modify_req ri_rnic_modify_req_t;
struct ri_rnic_modify_req {
    uint64_t               si_guid;
    ri_port_modify_req_t  *port_attrs;
    uint8_t                port_list_size;
};

typedef ri_api_return_t (*ri_op_rnic_modify_t)(
    IN   ri_rnic_handle_t       rnic,
    IN   ri_rnic_attr_mask_t    attr_mask,
    IN   ri_rnic_modify_req_t  *attrs,
    OUT  ri_err_data_t            *err_data
);
```

```
typedef ri_api_return_t (*ri_op_pd_alloc_t)(
    IN      ri_rnic_handle_t  rnic,
    OUT     ri_pd_handle_t   *pd,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_pd_dealloc_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_pd_handle_t    pd,
    OUT  ri_err_data_t    *err_data
);

typedef uint_t ri_cq_event_handler_id_t;

typedef struct ri_cq_attr ri_cq_attr_t;
struct ri_cq_attr {
    uint32_t                 num_cqe;
    ri_cq_event_handler_id_t  handler_id;
};

typedef ri_api_return_t (*ri_op_cq_create_t)(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  ri_cq_attr_t     *cq_attr,
    OUT     ri_cq_handle_t   *cq,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_cq_destroy_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_cq_handle_t    cq,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_cq_query_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_cq_handle_t    cq,
    OUT  ri_cq_attr_t     *cq_attr,
    OUT  ri_err_data_t    *err_data
);

typedef enum ri_cq_modify_mask ri_cq_modify_mask_t;
enum ri_cq_modify_mask {
    RI_CQ_MODIFY_MASK_NUM_CQE = 0x1
};

typedef ri_api_return_t (*ri_op_cq_modify_t)(
    IN      ri_rnic_handle_t     rnic,
    IN      ri_cq_handle_t       cq,
    IN      ri_cq_modify_mask_t  mask,
```

```
7444            IN OUT  ri_cq_attr_t        *cq_attr,
7445            OUT     ri_err_data_t       *err_data
7446        );
7447
7448        typedef enum ri_async_event_type ri_async_event_type_t;
7449        enum ri_async_event_type {
7450            /* iWarp-only events */
7451            RI_EVENT_QP_LLP_CLOSE_COMPLETE,
7452            RI_EVENT_QP_TERMINATE_MESSAGE_RECEIVED,
7453            RI_EVENT_QP_LLP_CONNECTION_RESET,
7454            RI_EVENT_QP_LLP_CONNECTION_LOST,
7455            RI_EVENT_QP_LLP_INTEGRITY_ERROR_SIZE,
7456            RI_EVENT_QP_LLP_INTEGRITY_ERROR_CRC,
7457            RI_EVENT_QP_LLP_INTEGRITY_ERROR_BAD_FPDU,
7458            RI_EVENT_QP_REMOTE_OP_ERROR_DDP_VERSION,
7459            RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_VERSION,
7460            RI_EVENT_QP_REMOTE_OP_ERROR_OPCODE,
7461            RI_EVENT_QP_REMOTE_OP_ERROR_DDP_QN,
7462            RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_RD_DISABLED,
7463            RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_WR_RD_RESP_DISABLED,
7464            RI_EVENT_QP_REMOTE_OP_ERROR_RDMA_RD_INVALID,
7465            RI_EVENT_QP_REMOTE_OP_ERROR_NO_L_BIT,
7466            RI_EVENT_QP_PROTECTION_ERROR_STAG_QP_MISMATCH,
7467            RI_EVENT_QP_PROTECTION_ERROR_BOUNDS_VIOLATION,
7468            RI_EVENT_QP_PROTECTION_ERROR_ACCESS_VIOLATION,
7469            RI_EVENT_QP_PROTECTION_ERROR_INVALID_PD,
7470            RI_EVENT_QP_PROTECTION_ERROR_WRAP_ERROR,
7471            RI_EVENT_TYPE_BAD_CLOSE,
7472            RI_EVENT_QP_BAD_LLP_CLOSE,
7473            RI_EVENT_QP_RQ_PROTECTION_ERROR_INVALID_MSN,
7474            RI_EVENT_QP_RQ_PROTECTION_ERROR_MSN_GAP,
7475            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_TOO_MANY_RDMA_READS,
7476            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_MSN_GAP,
7477            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_MSN,
7478            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_STAG,
7479            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_BOUNDS_VIOLATION,
7480            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_ACCESS_VIOLATION,
7481            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_INVALID_PD,
7482            RI_EVENT_QP_IRRQ_PROTECTION_ERROR_WRAP_ERROR,
7483            RI_EVENT_QP_CQ_SQ_ERROR,
7484            RI_EVENT_QP_CQ_RQ_ERROR,
7485            RI_EVENT_CQ_OVERFLOW_DETECTED,  /*optional for iWARP */
7486            RI_EVENT_CQ_OP_ERROR,
7487            RI_EVENT_QP_SRQ_ERROR,
7488            RI_EVENT_QP_LOCAL_CATASTROPHIC_ERROR,
7489            RI_EVENT_QP_RQ_LIMIT_REACHED,
7490
7491            /* common events */
7492            RI_EVENT_SRQ_LIMIT_REACHED,
7493            RI_EVENT_SRQ_CATASTROPHIC_ERROR,
7494            RI_EVENT_LOCAL_CATASTROPHIC_ERROR,
7495            RI_EVENT_RNIC_LOCAL_CATASTROPHIC_ERROR =
7496            RI_EVENT_LOCAL_CATASTROPHIC_ERROR,
7497            RI_EVENT_QP_LAST_WQE_REACHED,  /* optional for iWARP */
```

```
    /* IB-specific events */
    RI_EVENT_TYPE_QP_PATH_MIGRATED,
    RI_EVENT_TYPE_QP_COMMUNICATION_ESTABLISHED,
    RI_EVENT_TYPE_QP_SEND_QUEUE_DRAINED,
    RI_EVENT_CQ_ERROR,
    RI_EVENT_QP_LOCAL_WQ_CATASTROPHIC_ERROR,
    RI_EVENT_QP_INVALID_REQ_LOCAL_WQ_ERROR,
    RI_EVENT_QP_LOCAL_ACCESS_VIOLATION_WQ_ERROR,
    RI_EVENT_QP_PATH_MIG_REQ_ERROR,
    RI_EVENT_PORT_EVENT,
    RI_EVENT_CLIENT_REG_EVENT
};

typedef struct ri_ib_port_event ri_ib_port_event_t;
struct ri_ib_port_event {
    /* IB verbs do not specify that the port
     * number has to be returned. Port number 0
     * indicates that there is no knowledge of
     * which port this event is associated with.
     */
    uint8_t             port_number;
    ri_ib_port_state_t  port_state;
};

typedef union ri_event_data ri_event_data_t;
union ri_event_data {
    ri_ib_port_event_t  port_event;
    ri_os_data_t        resource_os_data;
};

typedef struct ri_event_record ri_event_record_t;
struct ri_event_record {
    ri_async_event_type_t  event_type;
    ri_event_data_t        event_data;
    ri_err_data_t          err_data;
};

typedef void (*ri_async_event_callback_t)(
    IN OUT  ri_os_data_t        vp_hdl_os_data,
    IN      ri_event_record_t  *event
);

typedef ri_api_return_t (*ri_op_async_set_event_handler_t)(
    IN   ri_vp_handle_t             vp_hdl,
    IN   ri_async_event_callback_t  async_event_callback,
    OUT  ri_err_data_t             *err_data
);

typedef void (*ri_cq_event_callback_t)(
    IN OUT  ri_os_data_t  vp_hdl_os_data,
    IN OUT  ri_os_data_t  cq_os_data
);
```

```
typedef ri_api_return_t (*ri_op_cq_set_event_handler_t)(
    IN      ri_vp_handle_t             vp_hdl,
    IN OUT  ri_cq_event_handler_id_t  *handler_id,
    IN      ri_cq_event_callback_t     cq_callback,
    OUT     ri_err_data_t             *err_data
);

typedef enum ri_cmplt_notify_type ri_cmplt_notify_type_t;
enum ri_cmplt_notify_type {
    RI_CMPLT_NOTIFY_SOLICITED,
    RI_CMPLT_NOTIFY_ANY
};

typedef ri_api_return_t (*ri_op_cq_req_notification_t)(
    IN   ri_rnic_handle_t        rnic,
    IN   ri_cq_handle_t          cq,
    IN   ri_cmplt_notify_type_t  type,
    OUT  ri_err_data_t          *err_data
);

/* Performance Note: An attempt to align the wc types with IT API's
 * event types has been done for the first five types. It is highly
 * desirable that the definitions do not change with future versions
 * of IT API.
 */
typedef enum ri_wc_type ri_wc_type_t;
enum ri_wc_type {
    RI_WC_TYPE_SEND = 0x1,
    RI_WC_TYPE_DATA_RECEIVED = 0x2,
    RI_WC_TYPE_RDMA_WRITE = 0x4,
    RI_WC_TYPE_RDMA_READ = 0x5,
    RI_WC_TYPE_BIND = 0x6,
    RI_WC_TYPE_FAST_REGISTER,
    RI_WC_TYPE_LOCAL_INVALIDATE,
    RI_WC_TYPE_RDMA_READ_LOCAL_INVALIDATE,
    RI_WC_TYPE_COMPARE_AND_SWAP,
    RI_WC_TYPE_FETCH_AND_ADD,
    RI_WC_TYPE_RDMA_IMMEDIATE_RECEIVED
};

typedef enum ri_wc_flags ri_wc_flags_t;
enum ri_wc_flags {
    RI_WC_FLAG_SOLICITED_EVENT = 1 << 0,
    RI_WC_FLAG_LOCAL_SOLICITED = 1 << 1,
    RI_WC_FLAG_RTAG_INVALIDATE = 1 << 2,
    RI_WC_FLAG_IMMEDIATE_DATA = 1 << 3,
    /* Pass through if there is no local solicited notification */
    RI_WC_FLAG_CONSUMER2 = RI_WC_FLAG_LOCAL_SOLICITED,
    RI_WC_FLAG_CONSUMER1 = 1 << 29,
    RI_WC_FLAG_CONSUMER0 = 1 << 30
};

/* Performance Note: An attempt to align the completion status with
 * IT API's dto_status_t has been done for the first five types.
```

```
 * It is highly desirable that the definitions do not change with
 * future versions of IT API.
 */
typedef enum ri_wc_status ri_wc_status_t;
enum ri_wc_status {
    RI_WC_STATUS_SUCCESS = 0,
    RI_WC_STATUS_WR_FLUSHED_ERROR = 1,
    RI_WC_STATUS_LOCAL_QP_CATASTROPHIC = 2,
    RI_WC_STATUS_LOCAL_QP_OP_ERROR = 2,
        /* same as LOCAL_QP_CATASTROPHIC */

    /* iWarp-specific status */
    RI_WC_STATUS_WQE_FORMAT_ERROR,
    RI_WC_STATUS_REMOTE_TERMINATION_ERROR,
    RI_WC_STATUS_INVALID_STAG_ERROR,
    RI_WC_STATUS_BASE_BOUNDS_VIOLATION_ERROR,
    RI_WC_STATUS_ACCESS_VIOLATION_ERROR,
    RI_WC_STATUS_INVALID_PD_ERROR,
    RI_WC_STATUS_ADDRESS_WRAP_ERROR,
    RI_WC_STATUS_STAG_INVALIDATE_ERROR,
    RI_WC_STATUS_ZERO_ORD_ERROR,
    RI_WC_STATUS_FMR_QP_NOT_PRIVILEGED_ERROR,
    RI_WC_STATUS_STAG_NOT_IN_INVALID_STATE_ERROR,
    RI_WC_STATUS_INVALID_FMR_PAGE_SIZE_ERROR,
    RI_WC_STATUS_INVALID_FMR_BLOCK_SIZE_ERROR,
    RI_WC_STATUS_INVALID_FMR_PBL_ENTRY_ERROR,
    RI_WC_STATUS_INVALID_FMR_FBO_ERROR,
    RI_WC_STATUS_INVALID_FMR_LENGTH_ERROR,
    RI_WC_STATUS_INVALID_FMR_PERMS_ERROR,
    RI_WC_STATUS_FMR_PBL_TOO_LONG_ERROR,
    RI_WC_STATUS_INVALID_FMR_VA_ERROR,
    RI_WC_STATUS_INVALID_BIND_MR_ERROR,
    RI_WC_STATUS_INVALID_BIND_MW_ERROR,
    RI_WC_STATUS_HUGE_PAYLOAD_ERROR,

    /* IB-specific status */
    RI_WC_STATUS_LOCAL_LENGTH_ERROR,
    RI_WC_STATUS_LOCAL_PROTECTION_ERROR,
    RI_WC_STATUS_MEMORY_MGMT_ERROR,
    RI_WC_STATUS_BAD_RESPONSE_ERROR,
    RI_WC_STATUS_LOCAL_ACCESS_ERROR,
    RI_WC_STATUS_REMOTE_INVALID_REQ_ERROR,
    RI_WC_STATUS_REMOTE_ACCESS_ERROR,
    RI_WC_STATUS_REMOTE_OP_ERROR,
    RI_WC_STATUS_TRANSPORT_RETRY_EXCEEDED,
    RI_WC_STATUS_RNR_RETRY_EXCEEDED,
    RI_WC_STATUS_REMOTE_ABORTED_ERROR
};

/* Performance Note: An attempt to align RC (non-immediate data) work
 * completion structure with IT API's dto_cmpl_event_t structure has
 * been made. It is highly desirable that the definitions do not change
 * with future versions of IT API.
 */
```

```
typedef struct ri_rc_wc ri_rc_wc_t;
struct ri_rc_wc {
    /* op_type should be the first field in the structure */
    ri_wc_type_t  op_type;

    /* Placeholder for it_evd_handle_t in ITAPI. OSVs may need to
     * define flags and invalidated_rtag to take the size of
     * it_evd_handle_t on their platforms.
     */
    ri_os_data_t   cq_os_data;
    ri_wc_flags_t  flags;
    ri_rtag_t      invalidated_rtag;

    /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
     * ri_os_data_t identical to it_ep_handle_t on their platforms
     */
    ri_os_data_t  qp_os_data;

    ri_wr_id_t id;
    ri_wc_status_t  op_status;
    uint32_t        bytes_transferred;  /* valid only for receive */
    uint32_t        immediate_data;  /* valid only if immediate data
                                      * is recevied along with incoming
                                      * Send or RDMA Write (IB only)
                                      */
    ri_err_data_t  err_data;
};

typedef struct ri_gen_wc ri_gen_wc_t;
struct ri_gen_wc {
    /* op_type should be the first field in the structure */
    ri_wc_type_t  op_type;

    /* Placeholder for it_evd_handle_t in ITAPI. OSVs may need to pad
     * this appropriately to take the size of it_evd_handle_t on their
     * platforms.
     */
    ri_os_data_t   cq_os_data;
    ri_wc_flags_t  flags;
    uint32_t       qpn;

    /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
     * ri_handle_t identical to it_ep_handle_t on their platforms.
     */
    ri_os_data_t    qp_os_data;
    ri_wr_id_t      id;
    ri_wc_status_t  op_status;
    uint32_t        bytes_transferred;
    uint32_t        immediate_data;
    ri_lid_t        remote_lid;
    uint32_t        remote_qpn;
    uint8_t         remote_sl;
    uint8_t         local_lid_pathbits;
```

```
    uint16_t        ether_type;
    uint32_t        grh_present;
    uint32_t        pkey_index;
    uint32_t        freed_resource_count;
    ri_err_data_t   err_data;
};

typedef union ri_wc ri_wc_t;
union ri_wc {
    ri_wc_type_t  type_wc;
    ri_rc_wc_t    rc_wc;
    ri_gen_wc_t   ud_wc;
};

typedef ri_api_return_t (*ri_op_cq_poll_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_cq_handle_t    cq,
    OUT     ri_wc_t          *wcs,
    IN OUT  uint_t           *num_wcs,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_qp_create_t)(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  ri_qp_attr_t     *attrs,
    OUT     ri_qp_handle_t   *qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_qp_query_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    OUT  ri_qp_attr_t     *attrs,
    OUT  ri_err_data_t    *err_data
);

typedef enum ri_qp_attr_mask ri_qp_attr_mask_t;
enum ri_qp_attr_mask {
    RI_QP_ATTR_NEXT_STATE = 1 << 0,
    RI_QP_ATTR_SQ_SIZE = 1 << 1,
    RI_QP_ATTR_RQ_SIZE = 1 << 2,
    RI_QP_ATTR_MAX_RDMA_READ_OUTGOING = 1 << 3,
    RI_QP_ATTR_MAX_RDMA_READ_INCOMING = 1 << 4,

    /* iWARP-specific masks */
    RI_QP_ATTR_RQ_HIWAT = 1 << 5,
    RI_QP_ATTR_RQ_ARMED = 1 << 6,
    RI_QP_ATTR_LLP_STREAM_HANDLE = 1 << 7,
    RI_QP_ATTR_STREAM_MSG_BUFFER = 1 << 8,
    RI_QP_ATTR_MPA_ATTR = 1 << 9,

    /* IB-specific masks */
```

```
7768            RI_QP_ATTR_ENABLE_RDMA_READ  = 1 << 5,
7769            RI_QP_ATTR_ENABLE_RDMA_WRITE = 1 << 6,
7770            RI_QP_ATTR_ENABLE_ATOMICS = 1 << 7,
7771            RI_QP_ATTR_DEST_QPID = 1 << 8,
7772            RI_QP_ATTR_ENABLE_SQD = 1 << 9,
7773            RI_QP_ATTR_QKEY = 1 << 10,
7774            RI_QP_ATTR_SEND_PSN = 1 << 11,
7775            RI_QP_ATTR_RECV_PSN = 1 << 12,
7776            RI_QP_ATTR_MIN_RNR_TIMER = 1 << 13,
7777            RI_QP_ATTR_PRIMARY_PATH = 1 << 14,
7778            RI_QP_ATTR_ALTERNATE_PATH = 1 << 15,
7779            RI_QP_ATTR_MIGSTATE = 1 << 16,
7780            RI_QP_ATTR_PRIMARY_PKEY = 1 << 17,
7781            RI_QP_ATTR_PORT_NUM = 1 << 18
7782
7783            /* cannot go beyond << 31 */
7784        };
7785
7786        typedef ri_api_return_t (*ri_op_qp_modify_t)(
7787            IN      ri_rnic_handle_t   rnic,
7788            IN      ri_qp_handle_t     qp,
7789            IN      ri_qp_attr_mask_t  qp_mask,
7790            IN OUT  ri_qp_attr_t      *attrs,
7791            OUT     ri_err_data_t     *err_data
7792        );
7793
7794        typedef ri_api_return_t (*ri_op_qp_destroy_t)(
7795            IN   ri_rnic_handle_t  rnic,
7796            IN   ri_qp_handle_t    qp,
7797            OUT  ri_err_data_t    *err_data
7798        );
7799
7800        typedef ri_api_return_t (*ri_op_srq_create_t)(
7801            IN      ri_rnic_handle_t  rnic,
7802            IN OUT  ri_srq_attr_t    *srq_attrs,
7803            OUT     ri_srq_handle_t  *srq,
7804            IN OUT  ri_vp_data_t      vp_data,
7805            IN OUT  ri_os_data_t      os_data,
7806            OUT     ri_err_data_t    *err_data
7807        );
7808
7809        typedef enum ri_srq_attr_mask ri_srq_attr_mask_t;
7810        enum ri_srq_attr_mask {
7811            RI_SRQ_ATTR_MAX_WRS = 1 << 0,
7812            /* max_sgl attribute is NOT modifiable */
7813            RI_SRQ_ATTR_LOWAT = 1 << 1
7814            /* pd is NOT modifiable *
7815             * modifying lowat implicitly arms the alarm.*/
7816        };
7817
7818        typedef ri_api_return_t (*ri_op_srq_query_t)(
7819            IN   ri_rnic_handle_t  rnic,
7820            IN   ri_srq_handle_t   srq,
7821            OUT  ri_srq_attr_t    *srq_attrs,
```

```
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_srq_modify_t)(
    IN   ri_rnic_handle_t    rnic,
    IN   ri_srq_handle_t     srq,
    IN   ri_srq_attr_mask_t  srqmask,
    OUT  ri_srq_attr_t      *attrs,
    OUT  ri_err_data_t      *err_data
);

typedef struct ri_wr_recv ri_wr_recv_t;
struct ri_wr_recv {
    ri_wr_flags_t   flags;
    ri_wr_id_t      id;
    ri_data_seg_t  *sg_list;
    uint_t          num_sgle;
};

typedef ri_api_return_t (*ri_op_srq_post_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_srq_handle_t   srq_handle,
    IN      ri_wr_recv_t     *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_srq_post_recv_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_srq_handle_t   srq_handle,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint_t            num_sgle,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_srq_destroy_t)(
   IN   ri_rnic_handle_t  rnic,
   IN   ri_srq_handle_t   srq,
   OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_qp_postsq_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_qp_handle_t    qp,
    IN      ri_wr_t          *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_qp_post_send_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
```

```
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_data_seg_t    *sg_list,
        IN   ri_rtag_t         rtag,
        IN   uint_t            num_sgle,
        IN   uint32_t          imm_data,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_rdma_write_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_data_seg_t    *sg_list,
        IN   uint64_t          rdma_addr,
        IN   ri_rtag_t         rtag,
        IN   uint_t            num_sgle,
        IN   uint32_t          imm_data,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_rdma_read_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_data_seg_t    *sg_list,
        IN   uint64_t          rdma_addr,
        IN   uint_t            num_sgle,
        IN   ri_rtag_t         rtag,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_bind_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_mw_handle_t    mw_handle,
        IN   ri_rtag_t         rtag,
        IN   ri_mr_handle_t    mr_handle,
        IN   ri_ltag_t         ltag,
        IN   ri_addr_t         vaddr,
        IN   ri_length_t       length,
        IN   ri_mem_attr_t     mem_attr,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_fmr_t)(
```

```
        IN   ri_rnic_handle_t        rnic,
        IN   ri_qp_handle_t          qp,
        IN   ri_wr_type_t            type,
        IN   ri_wr_flags_t           flags,
        IN   ri_wr_id_t              id,
        IN   ri_mr_reg_phys_attr_t  *attr,
        IN   ri_ltag_t               ltag,
        IN   ri_rtag_t               rtag,
        IN   ri_mr_handle_t          mr_handle,
        OUT  ri_err_data_t          *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_local_invalidate_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_rtag_t         rtag,
        IN   ri_ltag_t         ltag,
        IN   ri_mrw_handle_t   mrw_handle,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_atomic_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp,
        IN   ri_wr_type_t      type,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_rtag_t         rtag,
        IN   uint64_t          atomic_operand1,
        IN   uint64_t          atomic_operand2,
        IN   ri_data_seg_t    *sg_list,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_datagram_t)(
        IN   ri_rnic_handle_t     rnic,
        IN   ri_qp_handle_t       qp,
        IN   ri_wr_type_t         type,
        IN   ri_wr_flags_t        flags,
        IN   ri_wr_id_t           id,
        IN   ri_data_seg_t       *sg_list,
        IN   uint_t               num_sgle,
        IN   uint32_t             imm_data,
        IN   ri_address_handle_t  remote_node_address,
        IN   uint32_t             dest_qp,
        IN   uint32_t             dest_qkey,
        IN   uint8_t              max_static_rate,
        OUT  ri_err_data_t       *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_postrq_t)(
```

```
        IN      ri_rnic_handle_t  rnic,
        IN      ri_qp_handle_t    qp_handle,
        IN      ri_wr_recv_t     *wr,
        IN OUT  uint_t           *num_wrs,
        OUT     ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_qp_post_recv_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_qp_handle_t    qp_handle,
        IN   ri_wr_flags_t     flags,
        IN   ri_wr_id_t        id,
        IN   ri_data_seg_t    *sg_list,
        IN   uint_t            num_sgle,
        OUT  ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_ltag_alloc_t)(
        IN      ri_rnic_handle_t  rnic,
        IN OUT  uint_t           *address_list_length,
        IN      ri_pd_handle_t    pd,
        IN      ri_mem_attr_t     attr,
        OUT     ri_mr_handle_t   *mr,
        IN OUT  ri_ltag_t        *ltag,
        OUT     ri_rtag_t        *rtag,
        IN OUT  ri_vp_data_t      vp_data,
        IN OUT  ri_os_data_t      os_data,
        OUT     ri_err_data_t    *err_data
    );

    typedef ri_api_return_t (*ri_op_mr_reg_t)(
        IN        ri_rnic_handle_t   rnic,
        IN const  ri_mr_reg_attr_t  *mr_reg_attr,
        OUT       ri_mr_handle_t    *mr,
        IN OUT    ri_ltag_t         *ltag,
        OUT       ri_rtag_t         *rtag,
        IN OUT    ri_vp_data_t       vp_data,
        IN OUT    ri_os_data_t       os_data,
        OUT       ri_err_data_t     *err_data
    );

    typedef ri_api_return_t (*ri_op_mr_dereg_t)(
        IN   ri_rnic_handle_t  rnic,
        IN   ri_mr_handle_t    mr,
        OUT  ri_err_data_t    *err_data_t
    );

    typedef struct ri_mr_info ri_mr_info_t;
    struct ri_mr_info {
        ri_mr_reg_attr_t  attr;
        ri_ltag_t         ltag;
        ri_rtag_t         rtag;
        ri_boolean_t      is_shared_region;
        ri_addr_t         local_lb;
```

```
    ri_addr_t         local_ub;
    ri_addr_t         remote_lb;
    ri_addr_t         remote_ub;
    ri_stag_state_t   stag_state;
};

typedef ri_api_return_t (*ri_op_mr_query_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mr_handle_t    mr,
    OUT  ri_mr_info_t     *mr_info,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mr_rereg_t)(
    IN        ri_rnic_handle_t   rnic,
    IN        ri_mr_handle_t     in_mr,
    IN        ri_mr_rereg_t      change_mask,
    IN const  ri_mr_reg_attr_t  *mr_attr,
    OUT       ri_mr_handle_t    *out_mr,
    IN OUT    ri_ltag_t         *ltag,
    OUT       ri_rtag_t         *rtag,
    IN OUT    ri_vp_data_t       vp_data,
    IN OUT    ri_os_data_t       os_data,
    OUT       ri_err_data_t     *err_data
);

typedef ri_api_return_t (*ri_op_mr_reg_phys_t)(
    IN      ri_rnic_handle_t        rnic,
    IN OUT  ri_mr_reg_phys_attr_t  *mr_phys_attr,
    OUT     ri_mr_handle_t         *mr,
    IN OUT  ri_ltag_t              *ltag,
    OUT     ri_rtag_t              *rtag,
    IN OUT  ri_vp_data_t            vp_data,
    IN OUT  ri_os_data_t            os_data,
    OUT     ri_err_data_t          *err_data
);

typedef ri_api_return_t (*ri_op_mr_rereg_phys_t)(
    IN      ri_rnic_handle_t        rnic,
    IN      ri_mr_handle_t          in_mr,
    IN      ri_mr_rereg_t           change_mask,
    IN OUT  ri_mr_reg_phys_attr_t  *mr_phys_attr,
    OUT     ri_mr_handle_t         *out_mr,
    IN OUT  ri_ltag_t              *ltag,
    IN OUT  ri_rtag_t              *rtag,
    IN OUT  ri_vp_data_t            vp_data,
    IN OUT  ri_os_data_t            os_data,
    OUT     ri_err_data_t          *err_data
);

typedef ri_api_return_t (*ri_op_mr_reg_shared_t)(
    IN      ri_rnic_handle_t   rnic,
    IN      ri_mr_handle_t     in_mr,
    IN OUT  ri_mr_reg_attr_t  *mr_reg_attr,
```

```
8092            OUT     ri_mr_handle_t    *out_mr,
8093            IN OUT  ri_ltag_t         *ltag,
8094            OUT     ri_rtag_t         *rtag,
8095            IN OUT  ri_vp_data_t       vp_data,
8096            IN OUT  ri_os_data_t       os_data,
8097            OUT     ri_err_data_t     *err_data
8098        );
8099
8100        typedef ri_api_return_t (*ri_op_mw_alloc_t)(
8101            IN      ri_rnic_handle_t  rnic,
8102            IN      ri_pd_handle_t    pd,
8103            IN      ri_mw_type_t      mw_type,
8104            OUT     ri_mw_handle_t   *mw,
8105            OUT     ri_rtag_t        *rtag,
8106            IN OUT  ri_vp_data_t      vp_data,
8107            IN OUT  ri_os_data_t      os_data,
8108            OUT     ri_err_data_t    *err_data
8109        );
8110
8111        typedef ri_api_return_t (*ri_op_mw_query_t)(
8112            IN   ri_rnic_handle_t  rnic,
8113            IN   ri_mw_handle_t    mw,
8114            OUT  ri_mw_type_t     *mw_type,
8115            OUT  ri_stag_state_t  *stag_state,
8116            OUT  ri_pd_handle_t   *pd,
8117            OUT  ri_mem_attr_t    *attr,
8118            OUT  ri_rtag_t        *rtag,
8119            OUT  ri_err_data_t    *err_data
8120        );
8121
8122        typedef struct ri_ud_address_vector ri_ud_address_vector_t;
8123        struct ri_ud_address_vector {
8124            ri_gen_address_vector_t  av_common;
8125            uint8_t                  physical_port;
8126            /* number of the local physical port */
8127        };
8128
8129        typedef ri_api_return_t (*ri_op_ah_create_t)(
8130            IN      ri_rnic_handle_t         rnic,
8131            IN      ri_pd_handle_t           pd,
8132            IN      ri_ud_address_vector_t  *av,
8133            OUT     ri_address_handle_t     *ah,
8134            IN OUT  ri_vp_data_t             vp_data,
8135            IN OUT  ri_os_data_t             os_data,
8136            OUT     ri_err_data_t           *err_data
8137        );
8138
8139        typedef ri_api_return_t (*ri_op_ah_query_t)(
8140            IN   ri_rnic_handle_t         rnic,
8141            IN   ri_address_handle_t      ah,
8142            OUT  ri_ud_address_vector_t  *av,
8143            OUT  ri_pd_handle_t          *pd,
8144            OUT  ri_err_data_t           *err_data
8145        );
```

```
typedef ri_api_return_t (*ri_op_ah_modify_t)(
    IN   ri_rnic_handle_t         rnic,
    IN   ri_address_handle_t      ah,
    IN   ri_ud_address_vector_t  *av,
    OUT  ri_err_data_t           *err_data
);

typedef ri_api_return_t (*ri_op_ah_destroy_t)(
    IN   ri_rnic_handle_t     rnic,
    IN   ri_address_handle_t  ah,
    OUT  ri_err_data_t       *err_data
);

typedef enum ri_sqp_type ri_sqp_type_t;
enum ri_sqp_type {
    RI_SQP_TYPE_SMI,
    RI_SQP_TYPE_GSI
};

typedef ri_api_return_t (*ri_op_special_qp_get_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      uint8_t           port,
    IN      ri_sqp_type_t     qp_type,
    IN OUT  ri_qp_attr_t     *attrs,
    OUT     ri_qp_handle_t   *qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_process_sma_mad_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   uint8_t           port_num,
    IN   char             *in_mad,
    OUT  char             *out_mad,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mcast_attach_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_lid_t          mlid,
    IN      ri_gid_t         *mgid,
    IN      ri_qp_handle_t    qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mcast_detach_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_lid_t          mlid,
    IN   ri_gid_t         *mgid,
    IN   ri_qp_handle_t    qp,
```

```
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mw_dealloc_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mw_handle_t    mw,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_sync_rread_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_data_seg_t    *local_seg_list,
    IN   uint_t            num_seg,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_sync_rwrite_t)(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_data_seg_t    *local_seg_list,
    IN   uint_t            num_seg,
    OUT  ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_pin_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_unpin_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_map_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    OUT     ri_phys_buf_t    *io,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);

typedef ri_api_return_t (*ri_op_mem_unmap_t)(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_phys_buf_t    *io,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data
);
```

```
8254
8255            typedef enum ri_diag_op_type ri_diag_op_type_t;
8256            enum ri_diag_op_type {
8257                RI_DIAG_RNIC_DETAILS,
8258                RI_DIAG_QP_DETAILS,
8259                RI_DIAG_RNIC_DUMP,
8260                RI_DIAG_RDMA_RESET
8261            };
8262
8263            typedef ri_api_return_t (*ri_op_rnic_diag_t)(
8264                IN      ri_rnic_handle_t   rnic,
8265                IN      ri_diag_op_type_t  op,
8266                IN      void              *arg,
8267                IN OUT  unsigned char     *opval,
8268                IN OUT  ri_length_t       *oplen,
8269                OUT     ri_err_data_t     *err_data
8270            );
8271
8272            typedef struct ri_services_dev_arg ri_services_dev_arg_t;
8273            struct ri_services_dev_arg {
8274                /* Major version of RNICPI implemented */
8275                uint32_t rnicpi_major_version;
8276
8277                /* Minor version of RNICPI implemented */
8278                uint32_t rnicpi_minor_version;
8279
8280                /* Major version identifying the firmware on the RNIC */
8281                uint32_t fw_major_version;
8282
8283                /* Minor version identifying the firmware on the RNIC */
8284                uint32_t fw_minor_version;
8285
8286                /* Sub-minor version identifying the firmware on the RNIC */
8287                uint32_t fw_sub_minor_version;
8288
8289                /* size of buffer to allocate for query_rnic in bytes */
8290                uint_t query_size;
8291
8292                /* Character string indicating the pathname of the uVP
8293                 * shared library. This will be used later to load the
8294                 * appropriate uVP library.
8295                 */
8296                char lib_path[RI_MAX_LIB_PATH_LEN];
8297
8298                /* Character string that uniquely identifies the hardware
8299                 * path of the device on the OSV environment. This could
8300                 * be used for diagnostics or for persistent
8301                 * identification.
8302                 */
8303                char hw_path[RI_MAX_HW_PATH_LEN];
8304            };
8305
8306            /* Definitions of the function pointers. */
8307            typedef struct ri_ops ri_ops_t;
```

```
8308        struct ri_ops {
8309            ri_op_rnic_open_t                ri_op_rnic_open;
8310            ri_op_rnic_close_t               ri_op_rnic_close;
8311            ri_op_async_set_event_handler_t  ri_op_async_set_event_handler;
8312            ri_op_rnic_get_hdl_t             ri_op_rnic_get_hdl;
8313            ri_op_rnic_query_t               ri_op_rnic_query;
8314            ri_op_rnic_modify_t              ri_op_rnic_modify;
8315            ri_op_rnic_free_hdl_t            ri_op_rnic_free_hdl;
8316            ri_op_rnic_diag_t                ri_op_rnic_diag;
8317            ri_op_pd_alloc_t                 ri_op_pd_alloc;
8318            ri_op_pd_dealloc_t               ri_op_pd_dealloc;
8319            ri_op_cq_create_t                ri_op_cq_create;
8320            ri_op_cq_query_t                 ri_op_cq_query;
8321            ri_op_cq_modify_t                ri_op_cq_modify;
8322            ri_op_cq_destroy_t               ri_op_cq_destroy;
8323            ri_op_special_qp_get_t           ri_op_special_qp_get;
8324            ri_op_cq_set_event_handler_t     ri_op_cq_set_event_handler;
8325            ri_op_cq_req_notification_t      ri_op_cq_req_notification;
8326            ri_op_cq_poll_t                  ri_op_cq_poll;
8327            ri_op_qp_create_t                ri_op_qp_create;
8328            ri_op_qp_query_t                 ri_op_qp_query;
8329            ri_op_qp_modify_t                ri_op_qp_modify;
8330            ri_op_qp_destroy_t               ri_op_qp_destroy;
8331            ri_op_qp_postsq_t                ri_op_qp_postsq;
8332            ri_op_qp_post_send_t             ri_op_qp_post_send;
8333            ri_op_qp_post_rdma_read_t        ri_op_qp_post_rdma_read;
8334            ri_op_qp_post_rdma_write_t       ri_op_qp_post_rdma_write;
8335            ri_op_qp_post_bind_t             ri_op_qp_post_bind;
8336            ri_op_qp_post_fmr_t              ri_op_qp_post_fmr;
8337            ri_op_qp_post_local_invalidate_t ri_op_qp_post_local_invalidate;
8338            ri_op_qp_post_atomic_t           ri_op_qp_post_atomic;
8339            ri_op_qp_post_datagram_t         ri_op_qp_post_datagram;
8340            ri_op_qp_postrq_t                ri_op_qp_postrq;
8341            ri_op_qp_post_recv_t             ri_op_qp_post_recv;
8342            ri_op_srq_create_t               ri_op_srq_create;
8343            ri_op_srq_query_t                ri_op_srq_query;
8344            ri_op_srq_modify_t               ri_op_srq_modify;
8345            ri_op_srq_destroy_t              ri_op_srq_destroy;
8346            ri_op_srq_post_t                 ri_op_srq_post;
8347            ri_op_srq_post_recv_t            ri_op_srq_post_recv;
8348            ri_op_ltag_alloc_t               ri_op_ltag_alloc;
8349            ri_op_mr_reg_t                   ri_op_mr_reg;
8350            ri_op_mr_query_t                 ri_op_mr_query;
8351            ri_op_mr_rereg_t                 ri_op_mr_rereg;
8352            ri_op_mr_dereg_t                 ri_op_mr_dereg;
8353            ri_op_mr_reg_phys_t              ri_op_mr_reg_phys;
8354            ri_op_mr_rereg_phys_t            ri_op_mr_rereg_phys;
8355            ri_op_mr_reg_shared_t            ri_op_mr_reg_shared;
8356            ri_op_mw_alloc_t                 ri_op_mw_alloc;
8357            ri_op_mw_dealloc_t               ri_op_mw_dealloc;
8358            ri_op_mw_query_t                 ri_op_mw_query;
8359            ri_op_mem_pin_t                  ri_op_mem_pin;
8360            ri_op_mem_unpin_t                ri_op_mem_unpin;
8361            ri_op_mem_map_t                  ri_op_mem_map;
```

```
    ri_op_mem_unmap_t                ri_op_mem_unmap;
    ri_op_mem_sync_rread_t           ri_op_mem_sync_rread;
    ri_op_mem_sync_rwrite_t          ri_op_mem_sync_rwrite;
    ri_op_ah_create_t                ri_op_ah_create;
    ri_op_ah_query_t                 ri_op_ah_query;
    ri_op_ah_modify_t                ri_op_ah_modify;
    ri_op_ah_destroy_t               ri_op_ah_destroy;
    ri_op_mcast_attach_t             ri_op_mcast_attach;
    ri_op_mcast_detach_t             ri_op_mcast_detach;
    ri_op_process_sma_mad_t          ri_op_process_sma_mad;
};

/*
 * the actual svc routines
 */

ri_api_return_t ri_svc_attach(
    IN   ri_vp_handle_t          vp_hdl,
    IN   ri_dev_info_t           dev_info,
    IN   ri_services_dev_arg_t  *svcs,
    IN   ri_ops_t               *al_locking_ops,
    IN   ri_ops_t               *vp_locking_ops,
    OUT  ri_err_data_t          *err_data
);

ri_api_return_t ri_svc_detach(
    IN   ri_vp_handle_t  vp_hdl,
    OUT  ri_err_data_t  *err_data
);

ri_api_return_t ri_svc_lib_reg(
    IN      char           *lib_path,
    IN      ri_ops_t       *al_locking_ops,
    IN      ri_ops_t       *vp_locking_ops,
    IN OUT  ri_os_data_t    os_data,
    OUT     ri_err_data_t  *err_data
);

ri_api_return_t ri_svc_mem_pin(
    IN   ri_vaddr_t      vaddr,
    IN   ri_length_t     len,
    IN   ri_os_data_t    os_data,
    OUT  ri_err_data_t  *err_data
);

ri_api_return_t ri_svc_mem_unpin(
    IN   ri_vaddr_t      vaddr,
    IN   ri_length_t     len,
    IN   ri_os_data_t    os_data,
    OUT  ri_err_data_t  *err_data
);

#include "rnicpi_osdfunc.h"
#endif
```

10.2 <rnicpi_osd.h>

```
/*
 * rnicpi_osd.h -- prototype of OS-specific file to support rnicpi.h
 *
 * Copyright © 2005 The Open Group
 * All rights reserved.
 * The copyright owner hereby grants permission for all or part of this
 * publication to be reproduced, stored in a retrieval system, or
 * transmitted, in any form or by any means, electronic, mechanical,
 * photocopying, recording, or otherwise, provided that it remains
 * unchanged and that this copyright statement is included in all
 * copies or substantial portions of the publication.
 *
 * For any software code contained within this specification,
 * permission is hereby granted, free-of-charge, to any person
 * obtaining a copy of this specification (the "Software"), to deal in
 * the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom
 * the Software is furnished to do so, subject to the above copyright
 * notice and this permission notice being included in all copies or
 * substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
 * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Permission is granted for implementers to use the names, labels,
 * etc. contained within the specification. The intent of publication
 * of the specification is to encourage implementations of the
 * specification. This specification has not been verified for
 * avoidance of possible third-party proprietary rights. In
 * implementing this specification, usual procedures to ensure the
 * respect of possible third-party intellectual property rights should
 * be followed.
 */

#ifndef _RNICPI_OSD_H_
#define _RNICPI_OSD_H_

#define IN
#define OUT

typedef int             ri_llp_stream_handle_t;
typedef unsigned int    uint_t;
typedef unsigned short  ri_boolean_t;
typedef unsigned int    ri_length_t;
```

```
struct ri_handle_struct {
    const struct ri_ops  *ops;
    /* implementation and type-dependent fields follow
     * This format MUST be used for vp_handle and ri_rnic_handle_t.
     * It MAY be used for all other handles.
     *
     * Other implementations may vary, but must provide a mechanism to
     * translate from a vp_handle or rnic_handle to a struct ri_ops. *
     */
};

typedef void        *ri_handle_t;
typedef ri_handle_t  ri_vp_handle_t;
typedef ri_handle_t  ri_rnic_handle_t;
typedef ri_handle_t  ri_pd_handle_t;
typedef ri_handle_t  ri_qp_handle_t;
typedef ri_handle_t  ri_cq_handle_t;
typedef ri_handle_t  ri_mr_handle_t;
typedef ri_handle_t  ri_mw_handle_t;
typedef ri_handle_t  ri_srq_handle_t;
typedef ri_handle_t  ri_address_handle_t;
typedef void        *ri_vp_data_t;
typedef void        *ri_os_data_t;
typedef uint64_t     ri_err_data_t;

typedef char        *ri_dev_info_t;

typedef void        *ri_vaddr_t;

static __inline__ const struct ri_ops *ri_svc_ops(
    IN  void  *handle
)
{
    return ((struct ri_handle_struct *)handle)->ops;
}

static __inline__ void *ri_svc_handle_ptr(
    IN  ri_handle_t  handle
)
{
    return (void *)handle;
}

static __inline__ ri_handle_t ri_svc_make_handle(
    IN  struct  ri_handle_struct *p
)
{
    return (ri_handle_t)p;
}

typedef uint64_t  ri_phys_addr_t;
typedef uint64_t  ri_phys_buf_len_t;

#define RI_MAX_LIB_PATH_LEN 64
```

```
#define RI_MAX_HW_PATH_LEN 64
#define RI_MAX_VENDOR_STRING_LEN 64

#endif
```

10.3 <rnicpi_osdfunc.h>

```
/*
 * rnicpi_osdfunc.h - Platform-specific function declarations
 *
 * Copyright © 2005 The Open Group
 * All rights reserved.
 * The copyright owner hereby grants permission for all or part of this
 * publication to be reproduced, stored in a retrieval system, or
 * transmitted, in any form or by any means, electronic, mechanical,
 * photocopying, recording, or otherwise, provided that it remains
 * unchanged and that this copyright statement is included in all
 * copies or substantial portions of the publication.
 *
 * For any software code contained within this specification,
 * permission is hereby granted, free-of-charge, to any person
 * obtaining a copy of this specification (the "Software"), to deal in
 * the Software without restriction, including without limitation the
 * rights to use, copy, modify, merge, publish, distribute, sublicense,
 * and/or sell copies of the Software, and to permit persons to whom
 * the Software is furnished to do so, subject to the above copyright
 * notice and this permission notice being included in all copies or
 * substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
 * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Permission is granted for implementers to use the names, labels,
 * etc. contained within the specification. The intent of publication
 * of the specification is to encourage implementations of the
 * specification. This specification has not been verified for
 * avoidance of possible third-party proprietary rights. In
 * implementing this specification, usual procedures to ensure the
 * respect of possible third-party intellectual property rights should
 * be followed.
 */
/*
 * Inline re-direction functions. Each platform is encouraged
 * to provide these named symbols, using these inline definitions
 * or by other means. If the symbols are not provided, platform-
 * specific instructions on how to invoke the correct function
 * through the provideer's ri_ops_t should be provided.
```

```
 */

#ifndef _RNICPI_OSDFUNC_H_
#define _RNICPI_OSDFUNC_H_

static __inline__ ri_api_return_t ri_async_set_event_handler(
    IN   ri_vp_handle_t             vp_hdl,
    IN   ri_async_event_callback_t  async_event_callback,
    OUT  ri_err_data_t             *err_data
)
{
    return ri_svc_ops(vp_hdl)->ri_op_async_set_event_handler(vp_hdl,
        async_event_callback,err_data);
}

static __inline__ ri_api_return_t ri_rnic_open(
    IN      ri_vp_handle_t  vp_hdl,
    IN      ri_pbl_mode_t   pbl_mode,
    IN OUT  ri_os_data_t    os_data,
    OUT     ri_err_data_t  *err_data)
{
    return ri_svc_ops(vp_hdl)->ri_op_rnic_open(vp_hdl,pbl_mode,
        os_data,err_data);
}

static __inline__ ri_api_return_t ri_rnic_close(
    IN   ri_vp_handle_t  vp_hdl,
    OUT  ri_err_data_t  *err_data)
{
    return ri_svc_ops(vp_hdl)->ri_op_rnic_close(vp_hdl,err_data);
}

static __inline__ ri_api_return_t ri_rnic_free_hdl(
    IN   ri_vp_handle_t    vp_hdl,
    IN   ri_rnic_handle_t  rnic,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(vp_hdl)->ri_op_rnic_free_hdl(vp_hdl,
        rnic,err_data);
}

static __inline__ ri_api_return_t ri_rnic_query(
    IN      ri_rnic_handle_t       rnic,
    IN      ri_rnic_query_flags_t  flags,
    OUT     ri_rnic_attr_t        *attrs,
    IN OUT  uint_t                *attr_size,
    OUT     ri_err_data_t         *err_data)
{
    return ri_svc_ops(rnic)->ri_op_rnic_query(rnic,flags,
        attrs,attr_size,err_data);
}

static __inline__ ri_api_return_t ri_modify_rnic(
    IN   ri_rnic_handle_t           rnic,
```

```
8628            IN   ri_rnic_attr_mask_t        attr_mask,
8629            IN   ri_rnic_modify_req_t      *attrs,
8630            OUT  ri_err_data_t             *err_data)
8631        {
8632             return ri_svc_ops(rnic)->ri_op_rnic_modify(rnic,attr_mask,attrs,
8633                err_data);
8634        }
8635
8636        static __inline__ ri_api_return_t ri_pd_alloc(
8637            IN      ri_rnic_handle_t  rnic,
8638            OUT     ri_pd_handle_t   *pd,
8639            IN OUT  ri_vp_data_t     *vp_data,
8640            IN OUT  ri_os_data_t     *os_data,
8641            OUT     ri_err_data_t    *err_data)
8642        {
8643            return ri_svc_ops(rnic)->ri_op_pd_alloc(rnic,pd,vp_data,os_data,
8644                err_data);
8645        }
8646
8647        static __inline__ ri_api_return_t ri_pd_dealloc(
8648            IN   ri_rnic_handle_t  rnic,
8649            IN   ri_pd_handle_t    pd,
8650            OUT  ri_err_data_t    *err_data
8651        )
8652        {
8653            return ri_svc_ops(rnic)->ri_op_pd_dealloc(rnic,pd,err_data);
8654        }
8655
8656        static __inline__ ri_api_return_t ri_cq_create(
8657            IN      ri_rnic_handle_t  rnic,
8658            IN OUT  ri_cq_attr_t     *cq_attr,
8659            OUT     ri_cq_handle_t   *cq,
8660            IN OUT  ri_vp_data_t      vp_data,
8661            IN OUT  ri_os_data_t      os_data,
8662            OUT     ri_err_data_t    *err_data
8663        )
8664        {
8665            return ri_svc_ops(rnic)->ri_op_cq_create(rnic,cq_attr,cq,vp_data,
8666                os_data,err_data);
8667        }
8668
8669        static __inline__ ri_api_return_t ri_cq_poll(
8670            IN      ri_rnic_handle_t   rnic,
8671            IN      ri_cq_handle_t     cq,
8672            OUT     ri_wc_t           *wc,
8673            IN OUT  uint_t            *num_wcs,
8674            OUT     ri_err_data_t     *err_data)
8675        {
8676            return ri_svc_ops(rnic)->ri_op_cq_poll(rnic,cq,wc,
8677                num_wcs,err_data);
8678        }
8679
8680        static __inline__ ri_api_return_t ri_cq_destroy(
8681            IN   ri_rnic_handle_t  rnic,
```

```
8682            IN   ri_cq_handle_t    cq,
8683            OUT  ri_err_data_t    *err_data
8684        )
8685        {
8686            return ri_svc_ops(rnic)->ri_op_cq_destroy(rnic,cq,err_data);
8687        }
8688
8689        static __inline__ ri_api_return_t ri_cq_req_notification(
8690            IN   ri_rnic_handle_t        rnic,
8691            IN   ri_cq_handle_t          cq,
8692            IN   ri_cmplt_notify_type_t  type,
8693            OUT  ri_err_data_t          *err_data)
8694        {
8695            return ri_svc_ops(rnic)->ri_op_cq_req_notification(rnic,cq,type,
8696                err_data);
8697        }
8698
8699        static __inline__ ri_api_return_t ri_cq_query(
8700            IN   ri_rnic_handle_t  rnic,
8701            IN   ri_cq_handle_t    cq,
8702            OUT  ri_cq_attr_t     *cq_attr,
8703            OUT  ri_err_data_t    *err_data
8704        )
8705        {
8706            return ri_svc_ops(rnic)->ri_op_cq_query(rnic,cq,cq_attr,err_data);
8707        }
8708
8709        static __inline__ ri_api_return_t ri_cq_set_event_handler(
8710            IN      ri_vp_handle_t             vp_hdl,
8711            IN OUT  ri_cq_event_handler_id_t  *handler_id,
8712            IN      ri_cq_event_callback_t     cq_callback,
8713            OUT     ri_err_data_t             *err_data)
8714        {
8715            return ri_svc_ops(vp_hdl)->ri_op_cq_set_event_handler(vp_hdl,
8716                handler_id,cq_callback,err_data);
8717        }
8718
8719        static __inline__ ri_api_return_t ri_cq_modify(
8720            IN      ri_rnic_handle_t     rnic,
8721            IN      ri_cq_handle_t       cq,
8722            IN      ri_cq_modify_mask_t  mask,
8723            IN OUT  ri_cq_attr_t        *cq_attrs,
8724            OUT     ri_err_data_t       *err_data
8725        )
8726        {
8727            return ri_svc_ops(rnic)->ri_op_cq_modify(rnic,cq,mask,cq_attrs,
8728                err_data);
8729        }
8730
8731        static __inline__ ri_api_return_t ri_qp_create(
8732            IN      ri_rnic_handle_t  rnic,
8733            IN OUT  ri_qp_attr_t     *attrs,
8734            OUT     ri_qp_handle_t   *qp,
8735            IN OUT  ri_vp_data_t      vp_data,
```

```
8736                IN OUT  ri_os_data_t      os_data,
8737                OUT     ri_err_data_t    *err_data)
8738            {
8739                return ri_svc_ops(rnic)->ri_op_qp_create(rnic,attrs,qp,vp_data,
8740                    os_data,err_data);
8741            };
8742
8743            static __inline__ ri_api_return_t ri_qp_query(
8744                IN   ri_rnic_handle_t  rnic,
8745                IN   ri_qp_handle_t    qp,
8746                OUT  ri_qp_attr_t     *attrs,
8747                OUT  ri_err_data_t    *err_data)
8748            {
8749                return ri_svc_ops(rnic)->ri_op_qp_query(rnic,qp,attrs,err_data);
8750            }
8751
8752            static __inline__ ri_api_return_t ri_qp_modify(
8753                IN      ri_rnic_handle_t   rnic,
8754                IN      ri_qp_handle_t     qp,
8755                IN      ri_qp_attr_mask_t  qp_mask,
8756                IN OUT  ri_qp_attr_t      *attrs,
8757                OUT     ri_err_data_t*     err_data)
8758            {
8759                return ri_svc_ops(rnic)->ri_op_qp_modify(rnic,qp,qp_mask,attrs,
8760                    err_data);
8761            }
8762
8763            static __inline__ ri_api_return_t ri_qp_destroy(
8764                IN   ri_rnic_handle_t  rnic,
8765                IN   ri_qp_handle_t    qp,
8766                OUT  ri_err_data_t    *err_data)
8767            {
8768                return ri_svc_ops(rnic)->ri_op_qp_destroy(rnic,qp,err_data);
8769            }
8770
8771            static __inline__ ri_api_return_t ri_srq_create(
8772                IN      ri_rnic_handle_t  rnic,
8773                IN OUT  ri_srq_attr_t    *srq_attrs,
8774                OUT     ri_srq_handle_t  *srq,
8775                IN OUT  ri_vp_data_t      vp_data,
8776                IN OUT  ri_os_data_t      os_data,
8777                OUT     ri_err_data_t    *err_data)
8778            {
8779                return ri_svc_ops(rnic)->ri_op_srq_create(rnic,srq_attrs,
8780                    srq,vp_data,os_data,err_data);
8781            }
8782
8783            static __inline__ ri_api_return_t ri_srq_query(
8784                IN   ri_rnic_handle_t  rnic,
8785                IN   ri_srq_handle_t   srq,
8786                OUT  ri_srq_attr_t    *srq_attrs,
8787                OUT  ri_err_data_t*    err_data)
8788            {
8789                return ri_svc_ops(rnic)->ri_op_srq_query(rnic,srq,
```

```
        srq_attrs,err_data);
}

static __inline__ ri_api_return_t ri_srq_modify(
    IN   ri_rnic_handle_t    rnic,
    IN   ri_srq_handle_t     srq,
    IN   ri_srq_attr_mask_t  srqmask,
    OUT  ri_srq_attr_t      *attrs,
    OUT  ri_err_data_t      *err_data)
{
    return ri_svc_ops(rnic)->ri_op_srq_modify(rnic,srq,srqmask,attrs,
        err_data);
}

static __inline__ ri_api_return_t ri_srq_destroy(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_srq_handle_t   srq,
    OUT  ri_err_data_t*    err_data)
{
    return ri_svc_ops(rnic)->ri_op_srq_destroy(rnic,srq,err_data);
}

static __inline__ ri_api_return_t ri_qp_postsq(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_qp_handle_t    qp,
    IN      ri_wr_t          *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_qp_postsq(rnic,qp,wr,
        num_wrs,err_data);
}

static __inline__ ri_api_return_t ri_qp_post_send(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   ri_rtag_t         rtag,
    IN   uint_t            num_sgle,
    IN   uint32_t          imm_data,
    OUT  ri_err_data_t    *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_qp_post_send(rnic,qp,type,flags,id,
        sg_list,rtag,num_sgle,imm_data,err_data);
}

static __inline__ ri_api_return_t ri_qp_post_rdma_write(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
```

```
8844            IN   ri_wr_flags_t     flags,
8845            IN   ri_wr_id_t        id,
8846            IN   ri_data_seg_t    *sg_list,
8847            IN   uint64_t          rdma_addr,
8848            IN   ri_rtag_t         rtag,
8849            IN   uint_t            num_sgle,
8850            IN   uint32_t          imm_data,
8851            OUT  ri_err_data_t    *err_data
8852        )
8853        {
8854            return ri_svc_ops(rnic)->ri_op_qp_post_rdma_write(rnic,qp,type,
8855                flags,id,sg_list,rdma_addr,rtag,num_sgle,imm_data,err_data);
8856        }
8857
8858        static __inline__ ri_api_return_t ri_qp_post_rdma_read(
8859            IN   ri_rnic_handle_t  rnic,
8860            IN   ri_qp_handle_t    qp,
8861            IN   ri_wr_type_t      type,
8862            IN   ri_wr_flags_t     flags,
8863            IN   ri_wr_id_t        id,
8864            IN   ri_data_seg_t    *sg_list,
8865            IN   uint64_t          rdma_addr,
8866            IN   uint_t            num_sgle,
8867            IN   ri_rtag_t         rtag,
8868            OUT  ri_err_data_t    *err_data
8869        )
8870        {
8871            return ri_svc_ops(rnic)->ri_op_qp_post_rdma_read(rnic,qp,type,
8872                flags,id,sg_list,rdma_addr,num_sgle,rtag,err_data);
8873        }
8874
8875        static __inline__ ri_api_return_t ri_qp_post_bind(
8876            IN   ri_rnic_handle_t  rnic,
8877            IN   ri_qp_handle_t    qp,
8878            IN   ri_wr_type_t      type,
8879            IN   ri_wr_flags_t     flags,
8880            IN   ri_wr_id_t        id,
8881            IN   ri_mw_handle_t    mw_handle,
8882            IN   ri_rtag_t         rtag,
8883            IN   ri_mr_handle_t    mr_handle,
8884            IN   ri_ltag_t         ltag,
8885            IN   ri_addr_t         vaddr,
8886            IN   ri_length_t       length,
8887            IN   ri_mem_attr_t     mem_attr,
8888            OUT  ri_err_data_t    *err_data
8889        )
8890        {
8891            return ri_svc_ops(rnic)->ri_op_qp_post_bind(rnic,qp,type,flags,id,
8892                mw_handle,rtag,mr_handle,ltag,vaddr,length,mem_attr,err_data);
8893        }
8894
8895        static __inline__ ri_api_return_t ri_qp_post_fmr(
8896            IN   ri_rnic_handle_t        rnic,
8897            IN   ri_qp_handle_t          qp,
```

```
    IN   ri_wr_type_t            type,
    IN   ri_wr_flags_t           flags,
    IN   ri_wr_id_t              id,
    IN   ri_mr_reg_phys_attr_t  *attr,
    IN   ri_ltag_t               ltag,
    IN   ri_rtag_t               rtag,
    IN   ri_mr_handle_t          mr_handle,
    OUT  ri_err_data_t          *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_qp_post_fmr(rnic,qp,type,flags,
        id,attr,ltag,rtag,mr_handle,err_data);
}

static __inline__ ri_api_return_t ri_qp_post_local_invalidate(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_rtag_t         rtag,
    IN   ri_ltag_t         ltag,
    IN   ri_mrw_handle_t   mrw_handle,
    OUT  ri_err_data_t    *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_qp_post_local_invalidate(rnic,qp,
        type,flags,id,rtag,ltag,mw_handle,err_data);
}

static __inline__ ri_api_return_t ri_qp_post_atomic(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp,
    IN   ri_wr_type_t      type,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_rtag_t         rtag,
    IN   uint64_t          atomic_operand1,
    IN   uint64_t          atomic_operand2,
    IN   ri_data_seg_t    *atomic_result,
    OUT  ri_err_data_t    *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_qp_post_atomic(rnic,qp,type,flags,
        id,rtag,atomic_operand1,atomic_operand2,atomic_result,
        err_data);
}

static __inline__ ri_api_return_t ri_qp_postrq(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_qp_handle_t    qp_handle,
    IN      ri_wr_recv_t     *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data)
```

```
{
    return ri_svc_ops(rnic)->ri_op_qp_postrq(rnic,qp_handle,wr,
        num_wrs,err_data);
}

static __inline__ ri_api_return_t ri_qp_post_recv(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_qp_handle_t    qp_handle,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        id,
    IN   ri_data_seg_t    *sg_list,
    IN   uint_t            num_sgle,
    OUT  ri_err_data_t    *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_qp_post_recv(rnic,qp_handle,flags,
        id,sg_list,num_sgle,err_data);
}

static __inline__ ri_api_return_t ri_srq_post(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_srq_handle_t   srq_handle,
    IN      ri_wr_recv_t     *wr,
    IN OUT  uint_t           *num_wrs,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_srq_post(rnic,srq_handle,wr,num_wrs,
        err_data);
}

static __inline__ ri_api_return_t ri_srq_post_recv(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_handle_t       srq_handle,
    IN   ri_wr_flags_t     flags,
    IN   ri_wr_id_t        wrid,
    IN   ri_data_seg_t    *sg_list,
    IN   uint_t            num_sgle,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_srq_post_recv(rnic,srq_handle,
        flags,wrid,sg_list,num_sgle,err_data);
}

static __inline__ ri_api_return_t ri_ltag_alloc(
    IN      ri_rnic_handle_t  rnic,
    IN OUT  uint_t           *address_list_length,
    IN      ri_pd_handle_t    pd,
    IN      ri_mem_attr_t     attr,
    OUT     ri_mr_handle_t   *mr,
    IN OUT  ri_ltag_t        *ltag,
    IN OUT  ri_rtag_t        *rtag,
    IN      ri_vp_data_t      vp_data,
    IN      ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data)
```

```
{
    return ri_svc_ops(rnic)->ri_op_ltag_alloc(rnic,
        address_list_length,pd,attr,mr,ltag,rtag,vp_data,
        os_data,err_data);
}

static __inline__ ri_api_return_t  ri_mr_reg(
    IN        ri_rnic_handle_t   rnic,
    IN const  ri_mr_reg_attr_t  *mr_reg_attr,
    OUT       ri_mr_handle_t    *mem,
    OUT       ri_ltag_t         *ltag,
    OUT       ri_rtag_t         *rtag,
    IN        ri_vp_data_t       vp_data,
    IN        ri_os_data_t       os_data,
    OUT       ri_err_data_t     *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_reg(rnic,mr_reg_attr,mem,ltag,
        rtag,vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t  ri_mr_dereg(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mr_handle_t    mem,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_dereg(rnic,mem,err_data);
}

static __inline__ ri_api_return_t  ri_mr_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mr_handle_t    mr,
    OUT  ri_mr_info_t     *mr_info,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_query(rnic,mr,mr_info,
        err_data);
}

static __inline__ ri_api_return_t  ri_mr_rereg(
    IN        ri_rnic_handle_t   rnic,
    IN        ri_mr_handle_t     in_mr,
    IN        ri_mr_rereg_t      change_mask,
    IN const  ri_mr_reg_attr_t  *mem_attr,
    OUT       ri_mr_handle_t    *out_mr,
    IN OUT    ri_ltag_t         *ltag,
    OUT       ri_rtag_t         *rtag,
    IN        ri_vp_data_t       vp_data,
    IN        ri_os_data_t       os_data,
    OUT       ri_err_data_t     *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_rereg(rnic,in_mr,change_mask,
        mem_attr,out_mr,ltag,rtag,vp_data,os_data,err_data);
}
```

```
static __inline__ ri_api_return_t  ri_mr_reg_phys(
    IN      ri_rnic_handle_t        rnic,
    IN OUT  ri_mr_reg_phys_attr_t  *mr_phys_attr,
    OUT     ri_mr_handle_t         *mr,
    IN OUT  ri_ltag_t              *ltag,
    OUT     ri_rtag_t              *rtag,
    IN      ri_vp_data_t            vp_data,
    IN      ri_os_data_t            os_data,
    OUT     ri_err_data_t          *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_reg_phys(rnic,mr_phys_attr,mr,
        ltag,rtag,vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t ri_mr_rereg_phys(
    IN      ri_rnic_handle_t        rnic,
    IN      ri_mr_handle_t          in_mr,
    IN      ri_mr_rereg_t           change_mask,
    IN OUT  ri_mr_reg_phys_attr_t  *mr_phys_attr,
    OUT     ri_mr_handle_t         *out_mr,
    IN OUT  ri_ltag_t              *ltag,
    IN OUT  ri_rtag_t              *rtag,
    IN OUT  ri_vp_data_t            vp_data,
    IN OUT  ri_os_data_t            os_data,
    OUT     ri_err_data_t          *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_rereg_phys(rnic,in_mr,
        change_mask,mr_phys_attr,out_mr,ltag,rtag,vp_data,
        os_data,err_data);
}

static __inline__ ri_api_return_t  ri_mr_reg_shared(
    IN      ri_rnic_handle_t   rnic,
    IN      ri_mr_handle_t     in_mr,
    IN OUT  ri_mr_reg_attr_t  *mem_reg_attr,
    OUT     ri_mr_handle_t    *out_mr,
    IN OUT  ri_ltag_t         *ltag,
    OUT     ri_rtag_t         *rtag,
    IN      ri_vp_data_t       vp_data,
    IN      ri_os_data_t       os_data,
    OUT     ri_err_data_t     *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mr_reg_shared(rnic,in_mr,
        mem_reg_attr,out_mr,ltag,rtag,vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t  ri_mw_alloc(
    IN   i_rnic_handle_t  rnic,
    IN   i_pd_handle_t    pd,
    IN   i_mw_type_t      mw_type,
    OUT  i_mw_handle_t   *mw,
    OUT  i_rtag_t        *rtag,
    IN   i_vp_data_t      vp_data,
    IN   i_os_data_t      os_data,
```

```
    OUT  i_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mw_alloc(rnic,pd,mw_type,mw,rtag,
        vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t  ri_mw_query(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mw_handle_t    mw,
    OUT  ri_mw_type_t     *mw_type,
    OUT  ri_stag_state_t  *stag_state,
    OUT  ri_pd_handle_t   *pd,
    OUT  ri_mem_attr_t    *attr,
    OUT  ri_rtag_t        *rtag,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mw_query(rnic,mw,mw_type,
        stag_state,pd,attr,rtag,err_data);
}

static __inline__ ri_api_return_t  ri_mw_dealloc(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_mw_handle_t    mw,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mw_dealloc(rnic,mw,err_data);
}

static __inline__ ri_api_return_t ri_ah_create(
    IN      ri_rnic_handle_t         rnic,
    IN      ri_pd_handle_t           pd,
    IN      ri_ud_address_vector_t  *av,
    OUT     ri_address_handle_t     *ah,
    IN OUT  ri_vp_data_t             vp_data,
    IN OUT  ri_os_data_t             os_data,
    OUT     ri_err_data_t           *err_data)
{
    return ri_svc_ops(rnic)->ri_op_ah_create(rnic,pd,av,ah,vp_data,
        os_data,err_data);
}

static __inline__ ri_api_return_t ri_ah_query(
    IN   ri_rnic_handle_t         rnic,
    IN   ri_address_handle_t      ah,
    OUT  ri_ud_address_vector_t  *av,
    OUT  ri_pd_handle_t          *pd,
    OUT  ri_err_data_t           *err_data)
{
    return ri_svc_ops(rnic)->ri_op_ah_query(rnic,ah,av,pd,err_data);
}

static __inline__ ri_api_return_t ri_ah_modify(
    IN   ri_rnic_handle_t         rnic,
    IN   ri_address_handle_t      ah,
```

```
    IN   ri_ud_address_vector_t  *av,
    OUT  ri_err_data_t           *err_data)
{
    return ri_svc_ops(rnic)->ri_op_ah_modify(rnic,ah,av,err_data);
}

static __inline__ ri_api_return_t ri_ah_destroy(
    IN   ri_rnic_handle_t     rnic,
    IN   ri_address_handle_t  ah,
    OUT  ri_err_data_t       *err_data)
{
    return ri_svc_ops(rnic)->ri_op_ah_destroy(rnic,ah,err_data);
}

static __inline__ ri_api_return_t ri_special_qp_get(
    IN      ri_rnic_handle_t  rnic,
    IN      uint8_t           port,
    IN      ri_sqp_type_t     qp_type,
    IN OUT  ri_qp_attr_t     *attrs,
    OUT     ri_qp_handle_t   *qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_special_qp_get(rnic,port,qp_type,
        attrs,qp,vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t ri_process_sma_mad(
    IN   ri_rnic_handle_t   rnic,
    IN   uint8_t            port_num,
    IN   char              *in_mad,
    OUT  char              *out_mad,
    OUT  ri_err_data_t     *err_data)
{
    return ri_svc_ops(rnic)->ri_op_process_sma_mad(rnic,port_num,
        in_mad,out_mad,err_data);
}

static __inline__ ri_api_return_t ri_mcast_attach(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_lid_t          dlid,
    IN      ri_gid_t         *mgid,
    IN      ri_qp_handle_t    qp,
    IN OUT  ri_vp_data_t      vp_data,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mcast_attach(rnic,dlid,mgid,qp,
        vp_data,os_data,err_data);
}

static __inline__ ri_api_return_t ri_mcast_detach(
    IN   ri_rnic_handle_t  rnic,
```

```
    IN   uint32_t          dlid,
    IN   ri_gid_t         *gid,
    IN   ri_qp_handle_t    qp,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mcast_detach(rnic,dlid,gid,qp,
        err_data);
}

static __inline__ ri_api_return_t ri_mem_sync_rread(
    IN   ri_rnic_handle_t   rnic,
    IN   ri_data_seg_t     *local_seg_list,
    IN   uint_t             num_seg,
    OUT  ri_err_data_t     *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_mem_sync_rread(rnic,local_seg_list,
        num_seg,err_data);
}

static __inline__ ri_api_return_t ri_mem_sync_rwrite(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_data_seg_t    *local_seg_list,
    IN   uint_t            num_seg,
    OUT  ri_err_data_t    *err_data
)
{
    return ri_svc_ops(rnic)->ri_op_mem_sync_rwrite(rnic,local_seg_list,
        num_seg,err_data);
}

static __inline__ ri_api_return_t  ri_mem_pin(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_vaddr_t        vaddr,
    IN   ri_length_t       len,
    IN   ri_os_data_t      os_data,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mem_pin(rnic,vaddr,len,os_data,
        err_data);
}

static __inline__ ri_api_return_t  ri_mem_unpin(
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mem_unpin(rnic,vaddr,len,os_data,
        err_data);
}

static __inline__ ri_api_return_t ri_mem_map(
```

```
    IN      ri_rnic_handle_t  rnic,
    IN      ri_vaddr_t        vaddr,
    IN      ri_length_t       len,
    OUT     ri_phys_buf_t    *io,
    IN OUT  ri_os_data_t      os_data,
    OUT     ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mem_map(rnic,vaddr,len,io,os_data,
        err_data);
}

static __inline__ ri_api_return_t ri_mem_unmap(
    IN   ri_rnic_handle_t  rnic,
    IN   ri_phys_buf_t    *io,
    IN   ri_os_data_t      os_data,
    OUT  ri_err_data_t    *err_data)
{
    return ri_svc_ops(rnic)->ri_op_mem_unmap(rnic,io,os_data,
        err_data);
}

static __inline__ ri_api_return_t ri_op_rnic_diag(
    IN      ri_rnic_handle_t   rnic,
    IN      ri_diag_op_type_t  op,
    IN      void              *arg,
    IN OUT  unsigned char     *opval,
    IN OUT  ri_length_t       *oplen,
    OUT     ri_err_data_t     *err_data)
{
    return ri_svc_ops(rnic)->ri_op_rnic_diag(rnic,op,arg,opval,oplen,
        err_data);
}

/* get_hdl must be an actual function
 */

extern ri_api_return_t ri_rnic_get_hdl(
    IN   ri_vp_handle_t        vp_hdl,
    IN   ri_consumer_domain_t  location,
    OUT  ri_rnic_handle_t     *rnic,
    IN   ri_vp_data_t          vp_data,
    IN   ri_os_data_t          os_data,
    OUT  ri_err_data_t        *err_data);

#endif
```