

1 *Consortium Specification*

2 **RNIC Programming Interface (RNICPI)**
3 **Version 1.0**

4 **The Interconnect Software Consortium**

5 in association with

6  THE *Open* GROUP
Making standards work™

7 Copyright © 2005, The Open Group

8 All rights reserved.

9 The copyright owner hereby grants permission for all or part of this publication to be reproduced, stored in a
10 retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying,
11 recording, or otherwise, provided that it remains unchanged and that this copyright statement is included in
12 all copies or substantial portions of the publication.

13 For any software code contained within this specification, permission is hereby granted, free-of-charge, to
14 any person obtaining a copy of this specification (the “Software”), to deal in the Software without restriction,
15 including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
16 copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the
17 above copyright notice and this permission notice being included in all copies or substantial portions of the
18 Software.

19 THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
20 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
21 FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. IN NO EVENT SHALL THE
22 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER
23 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
24 OUT OF, OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
25 THE SOFTWARE.

26 Permission is granted for implementers to use the names, labels, etc. contained within the specification. The
27 intent of publication of the specification is to encourage implementations of the specification.

28 This specification has not been verified for avoidance of possible third-party proprietary rights. In
29 implementing this specification, usual procedures to ensure the respect of possible third-party intellectual
30 property rights should be followed.

31

32 Consortium Specification

33 **RNIC Programming Interface (RNICPI) Version 1.0**

34 ISBN: 1-931624-58-5

35 Document Number: C054

36

37 Published by The Open Group, September 2005.

38

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

40 nic-pi-comments@opengroup.org

41

Contents

42

1	Introduction.....	1
1.1	Goals.....	1
1.2	RNICPI Architecture Model.....	2
1.2.1	RNICPI Interfaces Specified in this Document.....	3
1.2.2	Relation to the RDMAC Verbs Document.....	7
1.2.3	MPA Support.....	7
1.2.4	IETF/RDMAC Interoperability	8
1.3	InfiniBand Support	9
1.4	Referenced Documents.....	9

51

2	Definitions.....	11
---	------------------	----

52

3	Assumed RDMA Host Stack Implementation	22
3.1	Host Stack Components and Interfaces	22
3.2	Responsibilities of a Kernel Access Layer	23
3.2.1	Proxy Service	24
3.2.2	Registry Service	25
3.2.3	Interface Mapping Service	25
3.2.4	Event Handling Service.....	25
3.2.5	Pinning Service.....	25
3.2.6	Cleanup Service.....	25
3.3	Responsibilities of a User-Level Access Layer	26

62

4	Global Behavior of the RNICPI.....	27
4.1	Synchronous versus Asynchronous Operations.....	27
4.2	Interrupt Safety	27
4.3	Thread Safety.....	27
4.3.1	Supported Thread Safety Models	28
4.4	Error Handling.....	28
4.5	Object Handle Management	29
4.5.1	Scope of Object Handles	29
4.5.2	Validation of User-Level Object Handles	29
4.6	Opaque RNICPI Call Arguments	29
4.6.1	Operating System Private Data	30
4.6.2	Verbs Provider Private Data.....	30
4.6.3	OS/IHV Private Error Information.....	31
4.7	Event Handling	31
4.7.1	Event Handler Registration	31
4.7.2	Event Notification	32

78

5	Memory Management.....	33
---	------------------------	----

79	5.1	Overview	33
80	5.2	Extensions to the RDMAC Verbs.....	34
81	5.3	Memory Registration Procedure.....	34
82	5.3.1	Usage of Opaque Call Arguments.....	34
83	5.3.2	Registering Virtual Memory.....	35
84	5.3.3	Registering Physical Memory	35
85	5.3.4	Fast-Register Memory.....	35
86	5.3.5	Usage of Special STag	35
87	6	Connection Management	36
88	6.1	iWARP Connection Management	36
89	6.1.1	Network Addresses and RNIC Selection	36
90	6.1.2	Packet Filtering.....	36
91	6.1.3	LLP Connection Management before RDMA Mode Transition.....	37
92	6.1.4	LLP Connection Conversion	37
93	6.1.5	LLP Connection Management after RDMA Mode Transition.....	37
94	6.1.6	RDMA Association Termination	38
95	6.2	Non-TCP Connection Management.....	38
96	7	API Reference Pages.....	39
97		ri_nic_open()	40
98		ri_nic_close().....	42
99		ri_async_set_event_handler()	44
100		ri_nic_get_hdl()	48
101		ri_nic_query()	50
102		ri_nic_modify().....	52
103		ri_nic_free_hdl().....	55
104		ri_nic_diag()	57
105		ri_pd_alloc(), ri_pd_dealloc()	60
106		ri_cq_create()	62
107		ri_cq_query().....	64
108		ri_cq_modify()	66
109		ri_cq_destroy().....	68
110		ri_cq_set_event_handler().....	69
111		ri_cq_req_notification()	71
112		ri_cq_poll()	73
113		ri_qp_create().....	79
114		ri_qp_query()	82
115		ri_qp_modify().....	83
116		ri_qp_destroy().....	88
117		ri_special_qp_get()	90
118		ri_qp_postsq(), ri_qp_post_send(), ri_qp_post_rdma_read(),	
119		ri_qp_post_rdma_write(), ri_qp_post_bind(), ri_qp_post_fmr(),	
120		ri_qp_post_local_invalidate(), ri_qp_post_atomic(), ri_qp_post_datagram()	93
121		ri_qp_postrq(), ri_qp_post_recv(), ri_srq_post(), ri_srq_post_recv().....	100
122		ri_srq_create().....	104
123		ri_srq_query()	106
124		ri_srq_modify().....	107

125		ri_srq_destroy()	109
126		ri_ltag_alloc()	111
127		ri_mr_reg()	113
128		ri_mr_query()	115
129		ri_mr_rereg()	117
130		ri_mr_dereg()	120
131		ri_mr_reg_phys()	122
132		ri_mr_rereg_phys()	124
133		ri_mr_reg_shared()	127
134		ri_mw_alloc()	129
135		ri_mw_query()	131
136		ri_mw_dealloc()	133
137		ri_mem_pin(), ri_mem_unpin()	134
138		ri_mem_map(), ri_mem_unmap()	137
139		ri_mem_sync_rread()	139
140		ri_mem_sync_rwrite()	141
141		ri_ah_create()	143
142		ri_ah_query()	145
143		ri_ah_modify()	147
144		ri_ah_destroy()	148
145		ri_mcast_attach()	149
146		ri_mcast_detach()	151
147		ri_process_sma_mad()	153
148		ri_svc_attach()	155
149		ri_svc_detach()	158
150		ri_svc_lib_reg()	159
151		ri_svc_mem_pin(), ri_svc_mem_unpin()	161
152	8	Data Type Reference Pages	163
153		ri_api_return_t	164
154		ri_ops_t	167
155		ri_rnic_attr_t	169
156		ri_iwarp_rnic_attr_t	174
157		ri_ib_rnic_attr_t	176
158		ri_ltag_t, ri_rtag_t	179
159		ri_stag_state_t	180
160		ri_phys_buf_t	181
161		ri_mem_attr_t	182
162		ri_mr_reg_attr_t, ri_mr_rereg_t	184
163		ri_mr_reg_phys_attr_t	186
164		ri_mw_type_t	187
165		ri_qp_attr_t	188
166		ri_srq_attr_t	197
167		ri_wr_t	198
168	9	Implementers Guide	205
169		9.1 RNIC Management	205
170		9.1.1 Registration of an RNIC with the Operating System	205

171	9.1.2	Opening an RNIC	205
172	9.1.3	Registering a Consumer with the P-RNICPI.....	205
173	9.1.4	Selection and Registration of an uVP with the uAL	207
174	10	RNICPI Header Files	208
175	10.1	<rnicipi.h>	208
176	10.2	<rnicipi_osd.h>.....	245
177	10.3	<rnicipi_osdfunc.h>.....	247
178	List of Figures		
179	Figure 1: RNICPI Interfaces and Related Host Components		2
180	Figure 2: Typical RNICPI Host Environment.....		23
181	Figure 3: RNIC Management and Consumer Registration		206
182			
183	List of Tables		
184	Table 1: Available RNICPI Function Calls.....		5
185			

186

Preface

187

The Interconnect Software Consortium

188
189
190

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.

191

The Open Group

192
193
194
195
196
197
198
199
200

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.

201

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

202
203
204

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.

205

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

206
207
208
209

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.

210
211

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

212

This Document

213
214
215

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.

216
217
218

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.

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

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

231 **Typographical Conventions**

232 The following typographical conventions are used throughout this document:

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

238 **Reference Page Conventions**

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

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

250

Trademarks

251

252

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.

253

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

254

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

255

256

257

258

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.

259

Acknowledgements

260
261

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

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

262

262 1 Introduction

263 1.1 Goals

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

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

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

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

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

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

- 286 • An introduction (this section)
- 287 • A glossary (see Section 2)
- 288 • A section on global behavior (see Section 4)
- 289 • A section on the assumed RDMA host stack implementation (see Section 3)
- 290 • Sections on memory management and connection management (see Section 5 and 6)
- 291 • Reference pages for RNICPI calls and their supporting data type definitions (see Section 7
292 and 8)
- 293 • 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.

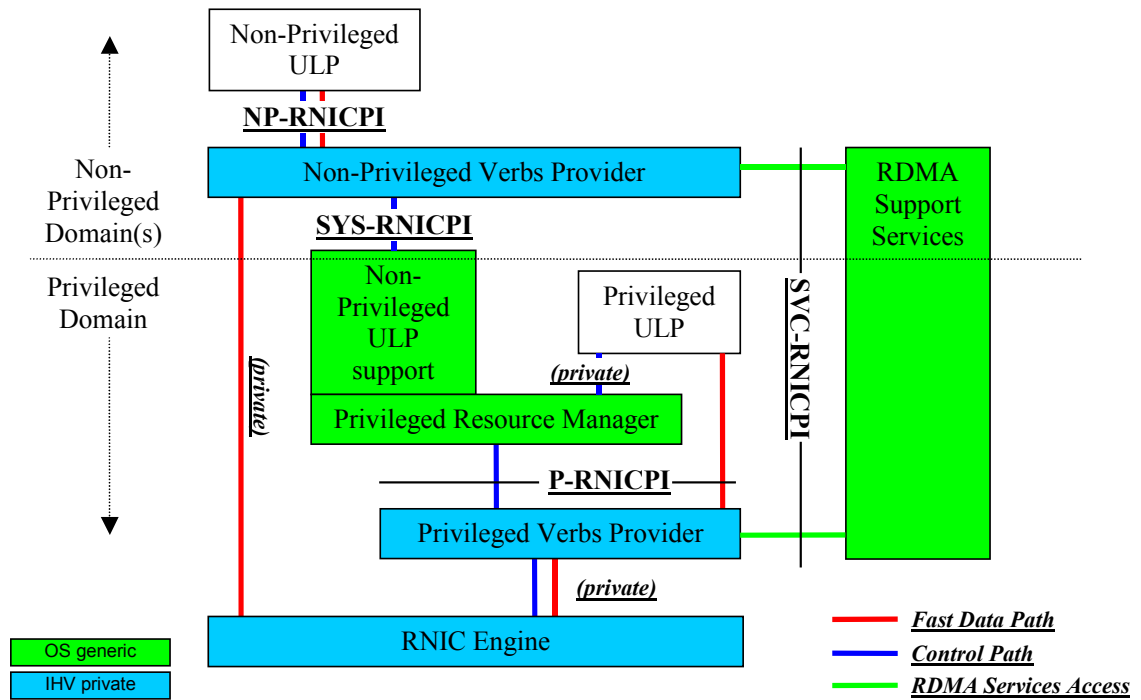


Figure 1: RNICPI Interfaces and Related Host Components

312 The components shown in Figure 1 are:

- 313 • RDMA Network Interface Controller (RNIC) Engine: The component that provides the
314 RDMA Services by implementing the iWARP protocol suite (RDMAP/DDP and
315 MPA/TCP and/or SCTP) and/or the HCA functionality of the InfiniBand stack or any
316 other future RDMA Transport.
- 317 • Privileged Verbs Provider: The component implementing the privileged standard interface
318 for accessing RDMA Services provided by the RNIC Engine.
- 319 • Privileged Resource Manager: The component responsible for managing resources
320 associated with RDMA services. While not sending or receiving data, the component is
321 policing all local resource reservations from both privileged and non-privileged ULPs.
322 This component is further checking and enforcing the credentials of non-privileged ULPs
323 to use only a subset of RDMA services. The granularity at which credentials can be
324 granted to ULPs is dependent on the operational environment and outside the scope of the
325 RNICPI.
- 326 • Privileged Upper Layer Protocol (ULP): The component representing the application or
327 middleware using RDMA services directly via the Privileged Verbs Provider. A typical
328 middleware would be the implementation of an RDMA-capable API for privileged
329 Consumers such as kDAPL [kDAPL]. A Privileged ULP is trusted by the local system to
330 not maliciously attack the operating environment, but is not trusted to deal fairly with
331 allocation and usage of RDMA resources [RDMASEC].
- 332 • Non-Privileged ULP Support: The component implementing proxy services to provide the
333 Non-Privileged ULP controlled access to privileged control operations of the RNICPI.
- 334 • Non-Privileged Verbs Provider: The component implementing the non-privileged
335 standard interface for accessing RDMA services provided by the RNIC Engine.
- 336 • Non-Privileged Upper Layer Protocol (ULP): The component residing in the non-
337 privileged domain that represents the application or middleware using RDMA services via
338 the Non-Privileged Verbs Provider. A typical middleware would be the implementation of
339 an RDMA-capable API for non-privileged Consumers such as uDAPL [uDAPL] or IT-
340 API [IT-APIv2]. A Non-Privileged ULP is assumed to be untrusted in the respect of fair
341 resource handling. Dependent on the credentials given to the ULP, several levels of trust
342 with respect to the operational environment are possible. The Privileged Resource
343 Manager may restrict the Non-Privileged ULP's credentials to use only a subset of the
344 RDMA services.
- 345 • RDMA Support Services: The component implementing a set of utility functions provided
346 for both the Privileged and Non-Privileged Verbs Provider. This specification assumes the
347 provision of services including memory pinning and RNIC device management.

348 1.2.1 RNICPI Interfaces Specified in this Document

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

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

- 356 1. Privileged RNICPI (P-RNICPI): This interface implements the full set of operations as
357 specified in the RDMA Verbs document [VERBS-RDMAC] and the InfiniBand
358 Specifications [IB-R1.1] and [IB-R1.2]. It further extends this set of functions as discussed
359 in Section 1.2.1.3. The P-RNICPI is exported to the Resource Manager (control type
360 operations) and to the Privileged ULP (fast path data type operations).
- 361 2. Non-Privileged RNICPI (NP-RNICPI): This interface implements most of the operations as
362 specified in the RDMA Verbs document [VERBS-RDMAC] and the InfiniBand
363 Specifications [IB-R1.1] and [IB-R1.2], including those specifically targeting user-level
364 usage. It further extends this set of functions as discussed in Section 1.2.1.3. The NP-
365 RNICPI is exported to the ULP by the Non-Privileged Verbs Provider. It is not part of the
366 NP-RNICPI to check the credentials of its Consumer (NP-ULP) to be sufficient to request a
367 certain RDMA service.
- 368 3. SYS-RNICPI: This interface implements the support for privileged RNICPI operations
369 (control type operations) for the Non-Privileged Verbs Provider. While the ULP accesses all
370 RDMA services via the NP-RNICPI, some operations (such as operations which will result
371 in the manipulation of host or RNIC resources; e.g., QP creation or communication buffer
372 memory registration) need the acknowledgement of the Resource Manager for further
373 system support. The SYS-RNICPI is exported to the Non-Privileged Verbs Provider by the
374 Non-Privileged ULP Support component.
- 375 4. RDMA Support Service Interface (SVC-RNICPI): This interface exports a set of utility
376 functions to the Privileged and Non-Privileged Verbs Providers. While not part of the
377 RDMA Verbs specification, the intention of the definition of this interface is to ease the
378 development of Verbs Provider components for potentially different (OS-specific) operating
379 environments.

380 1.2.1.1 Naming Conventions

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

- 382 1. Calls available at the P-RNICPI are named *ri_fn_name()*.
- 383 2. Calls available at the NP-RNICPI are named *ri_fn_name()*.
- 384 3. Calls available at the SYS-RNICPI are named *ri_sys_fn_name()*.
- 385 4. Calls available at the SVC-RNICPI are named *ri_svc_fn_name()*.

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

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

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

396 1.2.1.2 *Provided Calls per Interface*

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

398 **Table 1: Available RNICPI Function Calls**

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

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

Function Call	P-RNICPI	NP-RNICPI	SYS-RNICPI	SVC-RNICPI
<i>ri_mcast_attach()</i>	<i>x</i>	<i>x</i>	<i>x</i>	
<i>ri_mcast_detach()</i>	<i>x</i>	<i>x</i>	<i>x</i>	
<i>ri_process_sma_mad()</i>	<i>x</i>	<i>x</i>	<i>x</i>	
<i>ri_svc_attach()</i>				<i>x</i>
<i>ri_svc_detach()</i>				<i>x</i>
<i>ri_svc_lib_reg()</i>				<i>x</i>
<i>ri_svc_mem_pin()</i>				<i>x</i>
<i>ri_svc_mem_unpin()</i>				<i>x</i>

399 **1.2.1.3** *Experimental Extensions to this Specification*

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

404 **1.2.2** **Relation to the RDMAC Verbs Document**

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

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

419 **1.2.3** **MPA Support**

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

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

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

427 **1.2.4 IETF/RDMAC Interoperability**

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

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

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

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

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

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

463 **1.3 InfiniBand Support**

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

467 **1.4 Referenced Documents**

468 The following documents are referenced in this document:

- 469 [DDP-IETF] H. Shah et al, Direct Data Placement over Reliable Transports, IETF Internet Draft,
470 July 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-ddp-05.txt).
- 471 [DDP-RDMAC] H. Shah et al, Direct Data Placement over Reliable Transports (Version 1.0),
472 RDMA Consortium, October 2002 ([www.rdmaconsortium.org/home/draft-shah-
473 iwarp-ddp-v1.0.pdf](http://www.rdmaconsortium.org/home/draft-shah-iwarp-ddp-v1.0.pdf)).
- 474 [IB-R1.1] InfiniBand Architecture Specification Volume 1, Release 1.1, InfiniBand Trade
475 Association, November 2002.
- 476 [IB-R1.2] InfiniBand Architecture Specification Volume 1, Release 1.2, InfiniBand Trade
477 Association, October 2004.
- 478 [IT-API-V1.0] Interconnect Transport API (IT-API) Issue 1.0, The Interconnect Software
479 Consortium, February 2004.
- 480 [IT-API-V2.0] Interconnect Transport API (IT-API) Issue 2.0, The Interconnect Software
481 Consortium, May 2005.
- 482 [kDAPL] DAT Collaborative, kDAPL: Kernel Direct Access Programming Library, Version
483 1.2. September 2004 (www.datcollaborative.org/kDAPL12_091504.zip).
- 484 [MPA-IETF] P. Culley et al, Marker PDU Aligned Framing for TCP Specification, IETF Internet
485 Draft, February 2005 (www.ietf.org/internet-drafts/draft-ietf-rddp-mpa-02.pdf).
- 486 [MPA-RDMAC] P. Culley et al, Marker PDU Aligned Framing for TCP Specification (Version 1.0),
487 RDMA Consortium, October 2002 ([www.rdmaconsortium.org/home/draft-culley-
488 iwarp-mpa-v1.0.pdf](http://www.rdmaconsortium.org/home/draft-culley-iwarp-mpa-v1.0.pdf)).
- 489 [OpenRDMA] Work-in-progress (www.openrdma.org).
- 490 [RDMA-SEC] J. Pinkerton et al, DDP/RDMAP Security, April 2005 ([www.ietf.org/internet-
491 drafts/draft-ietf-rddp-security-07.txt](http://www.ietf.org/internet-drafts/draft-ietf-rddp-security-07.txt)).
- 492 [RDMAP-IETF] R. Recio et al, An RDMA Protocol Specification, IETF Internet Draft, July 2005
493 (www.ietf.org/internet-drafts/draft-ietf-rddp-rdmap-05.txt).
- 494 [RDMAP-RDMAC] R. Recio et al, An RDMA Protocol Specification (Version 1.0), RDMA
495 Consortium, October 2002 ([www.rdmaconsortium.org/home/draft-recio-iwarp-
496 rdmap-v1.0.pdf](http://www.rdmaconsortium.org/home/draft-recio-iwarp-rdmap-v1.0.pdf)).
- 497 [RFC 2119] S. Bradner, Keywords for use in RFCs to Indicate Requirement Levels, RFC 2119,
498 Internet Engineering Task Force, March 1997.

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

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

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

506 [uDAPL] DAT Collaborative. uDAPL: User Direct Access Programming Library, Version
507 1.2, September 2004 (www.datcollaborative.org/udapl12_091504.zip).

508 [VERBS-RDMAC] J. Hilland et al, RDMA Protocol Verbs Specification (Version 1.0), RDMA
509 Consortium, April 2003 (www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf).
510

511 **2 Definitions**

512 **Access Rights**

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

515 **Affiliated Asynchronous Event**

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

518 **Affiliated Error**

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

521 **ARP**

522 Address Resolution Protocol. A [network layer protocol](#) used to convert an [IP address](#) into a
523 physical address, such as a MAC address.

524 **Asynchronous Error**

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

531 **Base Tagged Offset (Base TO)**

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

533 **Bind, Binding, Bound**

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

536	Block List
537	A list of physical addresses describing a set of memory blocks, which specifies the block size,
538	list of physical addresses, and offset to the start of the Memory Region of the first block. Each
539	block has the same length and that length can be any value in the range supported by the RNIC.
540	Each block may start at a byte granularity address. The starting address for the entire list may be
541	an offset into the first block and the entire list may have any length.
542	Complete (Completed, Completion, Completes)
543	When the Consumer can determine that a particular RDMA Operation has performed all
544	functions specified for the RDMA Operation, including Placement and Delivery. This can be
545	determined through a Work Completion for Signaled Work Requests. For Unsignaled Work
546	Requests, this means that the Completion Rules have been met.
547	Completion Error
548	A Processing Error reported through the Completion Queue.
549	Completion Queue (CQ)
550	A sharable queue containing one or more entries that can contain Completion Queue Entries. A
551	CQ is used to create a single point of completion notification for multiple Work Queues. The
552	Work Queues associated with a Completion Queue may be from different QPs and of differing
553	queue types (SQs or RQs).
554	Completion Queue Entry (CQE)
555	The RNIC Interface internal representation of a Work Completion.
556	Completion Status
557	The resultant status of a Work Request returned as part of a Work Completion.
558	Consumer, Verbs Consumer
559	A software process that communicates using RNICPI. The Consumer typically consists of an
560	application program, or an operating system adaptation layer such as the kAL, which provides
561	some API.
562	Device Driver Interface (DDI)
563	OS-specific standardized kernel interfaces used for writing device drivers. Its definition includes
564	the calling and return syntax of the kernel functions that are used for communication between
565	drivers and the devices they control, the calling and return syntax of the entry point routines and
566	kernel utility functions that are used for communication between drivers and the kernel, and the
567	contents of the data structures that are used to exchange information between drivers and the
568	kernel.

- 569 **Direct Data Placement Protocol (DDP)**
- 570 A wire protocol that supports Direct Data Placement by associating explicit memory buffer
571 placement information with the LLP payload units.
- 572 **Data Delivery (Delivery, Delivered, Delivers)**
- 573 Delivery is defined as the process of informing the ULP or Consumer that a particular Message
574 is available for use. This is specifically different from Data Placement, which may generally
575 occur in any order, while the order of Data Delivery is strictly defined.
- 576 **Data Placement (Placement, Placed, Places)**
- 577 A mechanism whereby ULP data contained within RDMA Transport Segments may be put
578 directly into its final destination in memory without processing by the ULP. This may occur
579 even when the Segments arrive out of order. Note that this differs from Data Delivery (see Data
580 Delivery). From the RNICPI viewpoint, Data Placement is only confirmed upon Completion.
- 581 **Event**
- 582 An indication provided by the RNICPI to the ULP to indicate a Completion or other condition
583 requiring immediate attention.
- 584 **First Byte Offset (FBO)**
- 585 The offset into the first Physical Buffer of a Memory Region. The value of the FBO cannot
586 exceed the size of the Physical Buffer Entry Size associated with the Memory Region.
- 587 **Handle**
- 588 An opaque identifier used to reference an RNIC or an RNIC Resource. Whether this is an index,
589 object, or some other construct is outside the scope of this specification.
- 590 **ICMP**
- 591 Internet Control Message Protocol. An extension to the [Internet Protocol \(IP\)](#). ICMP is defined
592 by [RFC 792].
- 593 **Immediate Error**
- 594 An error discovered by the RNICPI provider during invocation and reported back without
595 affecting the RNIC.
- 596 **Invalidate STag (Invalidate, Invalidated, etc.)**
- 597 A mechanism used to prevent the Remote Peer from re-using an Advertised STag, until the
598 Local Peer transitions the STag to the Valid state.

599 **Invalidate Local STag**

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

602 **iWARP**

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

606 **Kernel Access Layer (kAL)**

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

610 **Kernel Verbs Provider (kVP)**

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

614 **LTag**

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

620 **Local Access**

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

624 **Lower Layer Protocol (LLP)**

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

627 **LLP Closed (LLP Close)**

628 When the LLP Stream can no longer be used for data transmission. If there is a single LLP
629 Stream on an LLP Connection, it may also mean that the LLP Connection has been torn down.
630 For example, for TCP this could include the states TIME_WAIT, CLOSING, LAST-ACK, and
631 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.

662	ND
663 664	Neighbor Discovery. Protocol to discover relationships between neighboring nodes including discovery of physical network (MAC) addresses. ND is defined by [RFC 2461].
665	Non-Privileged Mode
666 667 668	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.
669	Non-Shared Memory Region
670 671 672	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.
673	Outstanding
674 675	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.
676	Page List
677 678 679 680 681	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.
682	Physical Address
683 684 685	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).
686	Physical Buffer
687 688 689	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.
690	Physical Buffer List (PBL)
691	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.

723	Remote Access
724	The Access Rights used to verify the RNIC's ability to access the Data Sink for incoming DDP
725	Tagged Messages and the Data Source for RDMA Read Request Messages.
726	Remote Direct Memory Access (RDMA)
727	A method of accessing memory on a remote system in which the local system specifies the
728	remote location of the data to be transferred. Employing an RNIC in the remote system allows
729	the access to take place without interrupting the processing of the CPU(s) on the system.
730	RDMA Network Interface Controller (RNIC)
731	A network I/O adapter or embedded controller with iWARP and/or InfiniBand functionality.
732	RDMA Transport
733	A protocol suite implementing end-to-end RDMA Operations, such as InfiniBand or iWARP.
734	RDMA Operation
735	A sequence of RDMAP or IB Messages, including control Messages, to transfer data from a
736	Data Source to a Data Sink.
737	RDMA Protocol (RDMAP)
738	A wire protocol that supports RDMA Operations to transfer ULP data between a Local Peer and
739	the Remote Peer. See [RDMAP-IETF].
740	RTag
741	An opaque reference to a Memory Region or Memory Window that can be advertised to the
742	remote peer to enable remote memory access. Additionally, it may be used to specify the local
743	Data Sink for an RDMA Read Work Request.
744	Scatter/Gather Element (SGE)
745	An individual entry in a Scatter/Gather List. Each SGE consists of STag, Tagged Offset, and
746	Length.
747	Scatter/Gather List (SGL)
748	A List of Scatter/Gather Elements. The list describes one or more ULP Buffers that will have
749	their data gathered on transmission or scattered upon reception.

- 750 **Send Queue (SQ)**
- 751 One of the two Work Queues associated with a Queue Pair. The Send Queue contains PostSQ
752 Work Queue Elements that have specific operation types, such as Send Type, RDMA Write, or
753 RDMA Read Type Operations, as well as STag operations such as Bind and Invalidate.
- 754 **Shared Memory Region**
- 755 An MR that currently shares, or at one time shared, the Physical Buffer List associated with the
756 Memory Region. Specifically, the PBL is currently shared or was previously shared with another
757 Memory Region.
- 758 **Shared Receive Queue (SRQ)**
- 759 An optional mechanism which allows the Receive Queues from multiple QPs to retrieve Receive
760 Queue Work Queue Elements from the same shared queue as needed.
- 761 **Signaled**
- 762 A WR that requires that the RNIC generate a Work Completion.
- 763 **Solicited Event (SE)**
- 764 A facility by which an RDMA Operation sender may cause an Event to be generated at the
765 recipient, if the recipient is configured to generate such an Event, when a Send with Solicited
766 Event or Send with Solicited Event and Invalidate Message is received.
- 767 **Steering Tag (STag)**
- 768 An iWARP-specific identifier of a Memory Window or Memory Region. STags are composed
769 of two components: STag Index and STag Key. The Consumer forms the STag by combining the
770 STag Index with the STag Key. See LTag and RTag for a more transport independent
771 representation of an MW or MR identifier.
- 772 **STag Key**
- 773 The least significant 8-bit portion of an STag. This field of an STag can be set to any value by
774 the Consumer when performing a Memory Registration operation, such as Bind Memory
775 Window, Fast-Register Memory Region, and Register Memory Region.
- 776 **STag Index**
- 777 The most significant 24 bits of an STag. This field of the STag is managed by the Verbs
778 Provider and is treated as an opaque object by the Consumer.

779	Tagged Buffer
780	A buffer that can be Advertised to a Remote Peer through exchange of an STag, Tagged Offset,
781	and length.
782	Tagged Offset (TO)
783	The offset within a Tagged Buffer.
784	Terminate
785	An RDMA Message used by a Node to pass an error indication to the Remote Peer on an RDMA
786	Stream.
787	Unaffiliated Asynchronous Event
788	This is an indication from the Verb layer to the Consumer that an event has occurred unrelated to
789	any single identifiable RNIC Resource.
790	Unaffiliated Error
791	An error that cannot be directly related back to a specific RNIC Resource, such as a QP, SRQ, or
792	CQ.
793	Unsignaled
794	A Work Request which only generates a Work Completion if it encounters an error during
795	processing.
796	Upper Layer Protocol (ULP)
797	The component representing the application or middleware using RDMA services directly via
798	the Privileged Verbs Provider.
799	User Access Layer (uAL)
800	Software module, which in a typical RDMA host stack implementation represents all OS generic
801	RDMA support in the non-privileged domain. The uAL exports the NP-RNICPI, the SVC-
802	RNICPI, and the SYS-RNICPI.
803	User Verbs Provider (uVP)
804	RNIC vendor private software module. It implements in a typical RDMA host stack
805	implementation vendor private code to access the RNIC hardware from the non-privileged
806	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.

831 **3 Assumed RDMA Host Stack Implementation**

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

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

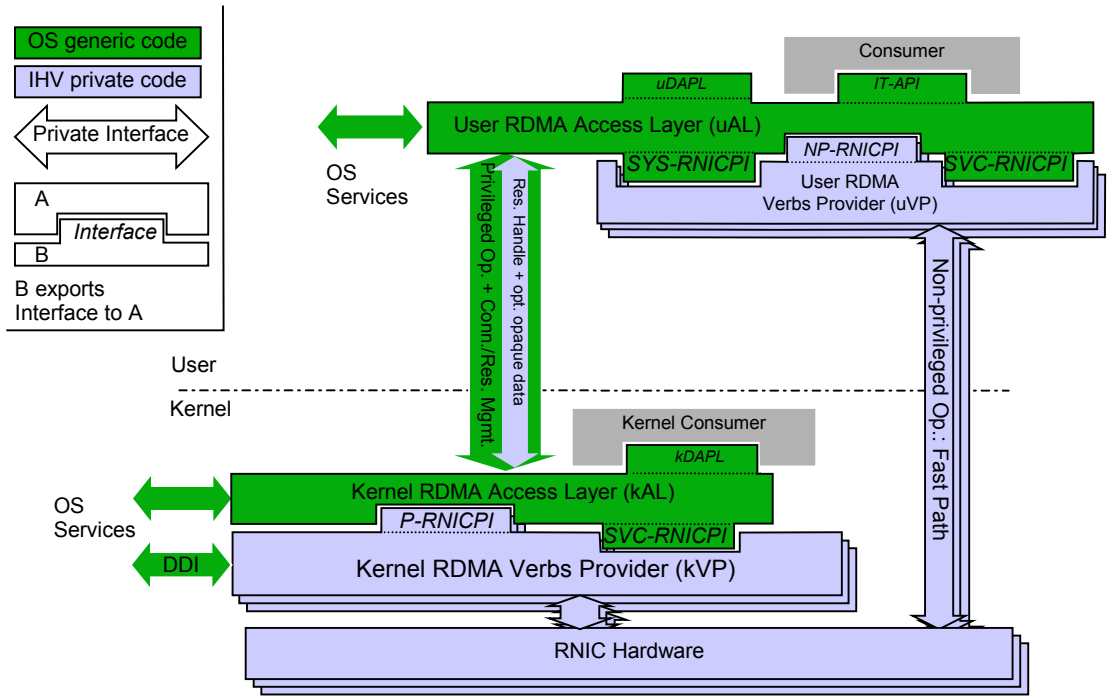
840 **3.1 Host Stack Components and Interfaces**

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

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

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

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



859
860

Figure 2: Typical RNICPI Host Environment

861 **3.2 Responsibilities of a Kernel Access Layer**

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

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

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

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

881 **3.2.1 Proxy Service**

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

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

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

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

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

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

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

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

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

918 **3.2.2 Registry Service**

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

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

923 **3.2.3 Interface Mapping Service**

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

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

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

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

933 **3.2.4 Event Handling Service**

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

941 **3.2.5 Pinning Service**

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

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

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

947 **3.2.6 Cleanup Service**

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

951 The Cleanup Service MUST be prompt enough to guarantee that no OS managed resource will
952 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.

958 **4 Global Behavior of the RNICPI**

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

960 **4.1 Synchronous versus Asynchronous Operations**

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

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

972 **4.2 Interrupt Safety**

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

980 **4.3 Thread Safety**

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

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

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

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

996 **4.3.1 Supported Thread Safety Models**

997 Ensuring mutual exclusion may affect the performance and so is only desirable when needed.
998 Furthermore, different implementation strategies (such as thread-affinity to single objects),
999 different locking methods (semaphores, spinlocks, blocking waits), and execution environments
1000 (single processor, SMP) may prefer to serialize access to critical objects at different points
1001 within the code path, mainly before or after crossing the RNICPI. The RNICPI thread safety
1002 model tries to take into account this potential heterogeneity by providing two locking strategies:

- 1003 • Verbs Consumer controlled serialization of critical object access: The verbs provider will
1004 not implement any thread safety mechanisms. The Access Layer and/or the Consumer are
1005 responsible for serializing those calls accessing critical objects. If the Access Layer is
1006 responsible for locking, then it may hold a spinlock across non-allocating RNICPI calls
1007 (i.e., all fast path operations and many modify operations). If invoked while spinlocked,
1008 the IHV routine **MUST NOT** do anything that would cause a context switch or block
1009 indefinitely.
- 1010 • Verbs provider controlled serialization of critical object access: Only the verbs provider
1011 will implement thread safety mechanisms. The Access Layer and Consumer are expected
1012 not to apply any locking mechanisms which might interfere with verbs provider locking.
1013 In particular the Access Layer **MUST NOT** be holding a spinlock when it invokes any
1014 RNICPI routine.

1015 Verbs Consumer controlled serialization is defined as the default behavior. Verbs provider
1016 controlled serialization is defined as optionally available. The verbs Consumer may choose from
1017 available behaviors during registering with the RNIC. See the description of the *ri_svc_attach()*
1018 and *ri_svc_reg_lib* calls for more details on how to choose the desired behavior.

1019 **4.4 Error Handling**

1020 The RNICPI defines three different mechanisms to provide to Consumers' Error Notification: as
1021 immediate error return values to interface calls, as completion status returned for an earlier
1022 request, and as asynchronous Events that are either associated with a particular object (affiliated
1023 errors) or not (non-affiliated errors). Immediate Error Notification is available for all calls of the
1024 RNICPI. Completion errors are polled to retrieve the results of previously posted Work Requests
1025 on the fast data path of the P-RNICPI and NP-RNICPI. Asynchronous errors are delivered by
1026 upcall to privileged Consumers in kernel mode. How to relay this information to user space
1027 Consumers is outside the scope of RNICPI.

1028 Immediate Error Notification generally happens when a call argument is invalid or incompatible
1029 with a condition relevant to the request. However, some errors of this type may be determined by
1030 the RNIC Engine below the RNICPI, and in this case the Consumer is notified of call parameter-
1031 related errors through an Event.

1032 Where there is a choice between immediate or completion error types to signal an error
1033 condition for a posting function, the kVP implementation chooses which single error type is used
1034 to report the error condition. The uVP for the same RNIC may make a different choice than the
1035 kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as
1036 noted in Section 4.5.2.

1037 Since the RNICPI is providing access to multiple RDMA Transports, some error values are
1038 defined only in the context of a specific RDMA Transport. See the reference pages for defined
1039 return values.

1040 **4.5 Object Handle Management**

1041 An Object Handle is an abstract representation of an RNICPI resource. It gets provided by the
1042 creator of this resource, typically an IHV component, during the resource creation call. The
1043 Consumer of the resource uses this handle to reference the resource in subsequent calls, but is
1044 typically not interpreting its value.

1045 **4.5.1 Scope of Object Handles**

1046 The scope of an RNIC handle is bound to the address domain where it was created. All resource
1047 handles created under an RNIC handle are only valid in the same domain and its lifetime is
1048 limited to the lifetime of the corresponding RNIC handle.

1049 It further gets defined that a child process is not authorized to use handles granted to the parent.
1050 Any unauthorized attempt MUST NOT interfere with the authorized holder's activities. The
1051 implementation should treat such an unauthorized activity like any other use of an invalid handle
1052 by an unauthorized process.

1053 **4.5.2 Validation of User-Level Object Handles**

1054 Resource handles are not expected to be strongly validated by the Verbs Provider code.
1055 Therefore, it is expected that the OS infrastructure (uAL/kAL) will ensure handle validity before
1056 passing any user-provided handle to the kVP. However, handles passed to the uVP only (as in
1057 post operations) may not be validated by the OS infrastructure. In this case, the uVP is allowed
1058 to terminate the application or incur an address fault if invalid handles are passed to it.

1059 **4.6 Opaque RNICPI Call Arguments**

1060 The RNICPI introduces the concept of Opaque Call Arguments. Present on some interface calls,
1061 an Opaque Call Argument passes so-called Opaque Data (OD). The RNICPI does not make any
1062 assumptions on the format of the Opaque Call Argument and on the format and content of the
1063 referenced OD. OD is introduced to help in the OSV and IHV-specific synchronization of

1064 RNICPI components and in the synchronization of RNICPI components with the OS-specific
1065 DDI.

1066 Typically, the caller will pass a pointer to a data structure, which contains some context
1067 information not defined in the RNICPI. From the perspective of the recipient of the OD, each
1068 OD argument can be further qualified by the definition of the following three orthogonal
1069 characteristics:

- 1070 1. Persistence: The RNICPI always specifies if the recipient must keep the handle referencing
1071 OD beyond the context of the actual call. Persistence is typically defined if the handle must
1072 be kept for reference in successive calls on the RNICPI or the DDI. If persistence is not
1073 explicitly defined by the RNICPI and the recipient must keep the OD information for local
1074 reference, it may have to copy the content.
- 1075 2. Interpretation: The RNICPI always specifies if the recipient may interpret the received OD.
- 1076 3. Passing: The RNICPI always specifies if the recipient must pass the OD to other RNICPI
1077 calls and lists those calls.

1078 Regarding the caller of a function passing OD, the RNICPI only specifies if it is the source of
1079 the data or is just passing along the OD received before by another RNICPI interface call.

1080 The RNICPI defines the following three different types of OD: Operating System Private Data,
1081 Verbs Provider Private Data, and Opaque Error Information. The following subsections define
1082 each of them in more detail.

1083 **4.6.1 Operating System Private Data**

1084 Operating System Private Data is always created in the OS generic part of the RDMA stack
1085 (uAL or kAL in Figure 2). Their typical usage is to synchronize OS components of the RDMA
1086 stack implementation across IHV private components. Specific to the call, this information may
1087 be transient to an intermediate IHV component (such as the uVP, which simply passes down OS
1088 Private Data received at the NP-RNICPI to the kAL via the SYS-RNICPI). The caller may have
1089 to keep this information for reference in further calls (such as the kVP which must present the
1090 data during a P-RNICPI event notification upcall), and the data may have to be interpreted by
1091 the recipient (such as the kAL, when the communication is meant to be uAL to kAL).

1092 In the reference pages, OS private OD is represented by the *os_data* argument.

1093 **4.6.2 Verbs Provider Private Data**

1094 Verbs Provider Private Data is always created in the IHV private part of the RDMA stack (uVP
1095 or kVP in Figure 2). Their typical usage is to synchronize IHV components of the RDMA stack
1096 (uVP and kVP) implementation across OS generic components. Verbs Provider Private Data is
1097 always handled transient in the OS components and is never interpreted during passing.

1098 A typical example for the usage of Verbs Provider Private Data is the creation of RDMA
1099 endpoint resources for a non-privileged ULP (e.g., creation of a QP): After requesting resource
1100 creation at the NP-RNICPI via *ri_qp_create()*, the uVP may establish some initial QP data
1101 structures. For completing the creation sequence, it will call for privileged support at the
1102 SYS-RNICPI. During this *ri_qp_create()* call, the uVP may want to pass down to the kVP some

1103 context information regarding the local QP structure (to enable RNIC Engines access to Work
1104 Request structures, etc.). Carried in the Verbs Provider Private Data, the kAL will hand over this
1105 information to the kVP in the following *ri_qp_create()* P-RNICPI call without any
1106 interpretation.

1107 In the reference pages, IHV private OD is represented by the *vp_data* argument.

1108 **4.6.3 OS/IHV Private Error Information**

1109 The RNICPI defines an additional OUT parameter to each interface call. This argument
1110 optionally transfers back context-specific error information in case of call failure. This extends
1111 the standard synchronous failure reporting mechanism (via the calls return value) by an
1112 OSV/IHV private method. This additional error reporting mechanism allows for the thread-safe
1113 handback of error information, which cannot always be achieved with an environment variable
1114 (such as *errno*, which in the current version of this specification never gets set or interpreted by
1115 the RNICPI). Furthermore, it supports the association of error information with the call instance
1116 rather than the object instance (e.g., multiple errors occurred on a single object or multiple
1117 objects are erroneous). The definition of valid error information and its interpretation is
1118 completely outside the scope of RNICPI.

1119 In the reference pages, OSV/IHV Private Error Information is represented by the *err_data*
1120 argument.

1121 **4.7 Event Handling**

1122 The IB and RDMA verbs specification define two types of events: one type is related to
1123 previously issued Work Requests which could complete in success or error (Work Completion
1124 Events), the other type is not related to a certain Work Request, but to a QP (affiliated error) or
1125 to another RNIC resource (non-affiliated error).

1126 **4.7.1 Event Handler Registration**

1127 In a typical implementation, the kAL is responsible for all Event Handler registration with the
1128 RNIC(s) available in the system. The kAL registers the single Asynchronous Event Handler
1129 (AEH) supported per RNIC as well as one or more Completion Event Handlers (CEHs). The
1130 maximum number of CEHs supported by the RNIC can be determined through *ri_rnic_query()*.

1131 The kAL registers its Event Handlers using the *ri_cq_set_event_handler()* resp.
1132 *ri_async_set_event_handler()* call. A successful call to *ri_cq_set_event_handler()* will return a
1133 Completion Event Handler ID. Since only one Asynchronous Event Handler can exist, each call
1134 to *ri_async_set_event_handler()* will replace the current handler.

1135 User-level event notification mechanisms are outside the scope of this specification. If user-level
1136 Consumer event mechanisms are used, it is envisioned that these mechanisms would have the
1137 uAL/kAL act as a proxy for the RNIC. The kAL/uAL will keep state to establish the appropriate
1138 mapping between the user registered mechanism and kVP registered Event Handler and related
1139 resource (CQ and/or RNIC Handle).

1140 A privileged Consumer could register Event Handler directly with the kVP. Since the number of
1141 supported Event Handlers is typically a scarce RNIC resource, it is expected that an Event
1142 Handler mapping would also apply.

1143 **4.7.2 Event Notification**

1144 Event processing typically needs the support of a privileged intermediate responsible for
1145 scheduling of the Consumer for event notification. The Event Notification path will naturally
1146 start with the event information transfer between RNIC and the kVP. The kVP will typically
1147 install an ISR which gets called after the RNIC placed event records within the event queue
1148 located in the context of the kVP. The ISR is responsible for transferring control and information
1149 to its Consumer, which in this model is always the kAL. Thus, the Event Handlers registered
1150 with the RNIC are always representing callbacks from the kVP into the kAL.

1151 If the kAL gets control during event passing (callback), it is able to de-multiplex events to
1152 individual Consumers based on the information passed with the call:

- 1153 • Asynchronous Events are described by an Event Record which contains the type of the
1154 corresponding resource (QP, CQ, RNIC, SRQ), a resource identifier (e.g., QP ID), and the
1155 Event Identifier which indicates the reason for the event. This information is sufficient to
1156 retrieve the Consumer or group of Consumers.
- 1157 • A CEH is called with the RNIC Handle and the CQ Handle as arguments. The Consumer
1158 can be directly derived from this information.

1159 For kernel-level Consumers, the kAL will trigger the execution of the Consumer's registered
1160 Event Handler directly. User-level Consumers get control via the kAL/uAL Interface.

1161 The Consumer can request completion notification by invoking *ri_cq_req_notification()* on a
1162 CQ that is enabled to receive notifications. A user-level Consumer may wait for notifications in
1163 an implementation-dependent manner (an uAL thread sleeping on a kAL device *read()* call, a
1164 *select()* on a file descriptor, etc.). When a completion event is generated by the RNIC, the kAL's
1165 CEH is invoked, upon which the kAL wakes up the waiting thread/updates the registered file
1166 descriptor. Alternatively, the Consumer can poll for completions using *ri_cq_poll()*.

1167 **5 Memory Management**

1168 **5.1 Overview**

1169 To enable read and write access to Consumer's communication buffers, the Consumer must
1170 register the corresponding memory with the RNIC Engine via the RNICPI. The result of any
1171 memory registration is the provision of a set of handles by the Verbs Provider:

- 1172 • A Memory Region Handle to be used in subsequent RNICPI calls
- 1173 • A Local Tag (LTag) that can be used along with a Tagged Offset (TO) and a length to
1174 reference portion of the Memory Region; LTags are used to specify the local source
1175 and/or destination in Work Requests
- 1176 • An optional Remote Tag (RTag) that can be used to enable remote access to the Memory
1177 Region; an RTag may also be used on some RNICs to specify the local Data Sink for an
1178 RDMA Read Work Request

1179 All memory registration is further qualified by the requested setting of remote and or local
1180 memory access rights. For iWARP, the LTag and RTag will always be the same value, and the
1181 non-key portion may be the Memory Region handle. For IB, these will be distinct values.

1182 A Memory Window (MW) can be bound to an MR to allow for fine grained access to a portion
1183 (window) of a Memory Region. Memory Windows are suitable for refining access permissions
1184 on a per-operation basis.

1185 During registration of an MR, a virtually contiguous block of Consumer memory (Virtual
1186 Memory, VM) gets pinned in physical memory, translated into a Physical Buffer List (PBL), and
1187 I/O-mapped for access by the RNIC Engine. Binding an MW to an already existent MR does not
1188 involve memory pinning or translation.

1189 Memory registration can be requested with a synchronously completing interface call (Register
1190 Memory; e.g., via *ri_mr_reg()*) or asynchronously via posting a Work Request onto the Send
1191 Queue (Fast Register). In case of Fast Register, a Steering Tag must be pre-registered with the
1192 RNIC. Fast Registration is available for privileged Consumers only.

1193 The usage of a special Steering Tag (STag with value 0 for iWARP) represents the single
1194 exception to the otherwise mandatory explicit memory registration. Using the special STag, the
1195 Consumer is able to reference an already pinned and translated physical buffer in a Work
1196 Request without its previous registration. The usage of this mechanism is restricted to
1197 referencing memory for local access and it is only available to privileged (kernel) Consumers.

1198 5.2 Extensions to the RDMAC Verbs

1199 The [VERBS-RDMAC] document describes one generic way to register memory with the
1200 RNIC. Independent of the registration method (Register Memory or Fast Register), the interface
1201 call or Work Request always presents already pinned and translated memory as a PBL. Pinning
1202 and translation of the memory is outside the scope of [VERBS-RDMAC].

1203 Due to the diversity of target operating systems, the RNICPI deviates from this simplistic
1204 approach and provides a set of additional interface calls to aid in the development of a
1205 standardized memory management. Main OS-specific differences for the management of
1206 communication buffers include:

- 1207 • Translation between kernel physical memory representation and RNIC drivers memory
1208 addressing mode (bus or I/O addresses). On some systems, a kernel created PBL could not
1209 always be synchronously mapped into addresses reachable for the RNIC Engine. For
1210 those systems, the PBL must be created under control of the driver.
- 1211 • Pinning and translation are not necessarily combined together within a single OS service.
- 1212 • System support for pinning of non-privileged Consumer memory may be available only as
1213 a kernel function or only outside the kernel (e.g., via a system call).
- 1214 • Privileged Consumer memory may always be pinned or may have to be explicitly pinned
1215 by the kAL.

1216 Therefore, besides providing RDMAC semantics (*ri_mr_reg_phys()*, *ri_mr_rereg_phys()*),
1217 RNICPI is extending the memory management by the following functionalities:

- 1218 • Support for the registration of virtual memory (VM) (*ri_mr_reg()*, *ri_mr_reg_shared()*,
1219 *ri_mr_rereg()*, *ri_qp_postsq()*).
- 1220 • Support for the mapping of memory for RNIC access. The *ri_mem_map()* function maps
1221 VM into an RNIC accessible PBL.
- 1222 • Memory pinning support from the SVC-RNICPI. The *ri_svc_mem_pin()* routine provided
1223 by the OS is pinning virtual memory ranges provided by the Verbs Provider.
- 1224 • Memory pinning downcall *ri_mem_pin()* to allow for RNIC driver controlled pinning of
1225 Consumer memory.

1226 5.3 Memory Registration Procedure

1227 5.3.1 Usage of Opaque Call Arguments

1228 Independent of the actual type of memory registration, all memory registration operations are
1229 accompanied by related calls provided by the P-RNICPI, SVC-RNICPI, and DDI. The opaque
1230 argument concept introduced in Section 4.6 will be used to keep and transfer Access Layer and
1231 Verbs Provider-specific context information during the resulting call sequences. See the
1232 appropriate reference pages for further details.

1233 **5.3.2 Registering Virtual Memory**

1234 During virtual memory registration, the Consumer calls the appropriate registration routine
1235 (*ri_mr_reg()*, *ri_mr_reg_shared()*, *ri_mr_rereg()*), and the Verbs Provider controls mapping and
1236 pinning of the memory. While the mapping is done under local responsibility of the Verbs
1237 Provider with typically some interaction via the DDI interface, the pinning is done by calling
1238 back the *ri_svc_mem_pin()* routine provided by the Access Layer. It is possible that the Verbs
1239 Provider calls the pinning service multiple times to allow for efficient mapping. This call
1240 completes synchronously and provides a handle back to the registered MR. The PBL type
1241 representation of the registered memory does not have to be provided to the Consumer.

1242 **5.3.3 Registering Physical Memory**

1243 Dependent on the OS architecture, either the Consumer pins the memory on its own
1244 responsibility (outside the RNICPI) or is using the *ri_mem_pin()* P-RNICPI downcall to the
1245 Verbs Provider. In a second step, the Consumer establishes the mapping of the pinned memory
1246 via *ri_mem_map()*. This call provides a PBL which now can be presented to the Verbs Provider
1247 via *ri_mr_reg_phys()* or *ri_mr_rereg_phys()*.

1248 **5.3.4 Fast-Register Memory**

1249 Fast Registration of memory is a Post operation on the Send Queue (*ri_qp_postsq()*). It expects a
1250 PBL as input which gets created by the very same procedure as described in Section 5.3.3.

1251 **5.3.5 Usage of Special STag**

1252 With this mechanism, a privileged Consumer may reference already pinned and translated
1253 memory in a Work Request without any previous registration with the RNICPI. The memory
1254 must be pinned and mapped using the same procedure as described in Section 5.3.3.

1255 **6 Connection Management**

1256 **6.1 iWARP Connection Management**

1257 An RDMA Stream is always bound to an established bi-directional LLP transport-level
1258 connection (TCP plus MPA, SCTP, or another Transport). Before conversion into RDMA Mode,
1259 an established LLP connection is represented by an operating system-specific LLP Handle which
1260 typically gets created by the operating system. After converting the LLP connection into RDMA
1261 mode, the LLP connection is owned by the RNICPI. The LLP handle may change during RDMA
1262 mode conversion.

1263 **6.1.1 Network Addresses and RNIC Selection**

1264 RNICPI assumes that an IP address associated with the RNIC device is consistent with the IP
1265 address administration of the underlying Ethernet device and host routing tables.

1266 How an RNICPI Consumer selects a given RNIC for communication is outside the scope of
1267 RNICPI. It is however expected that the Consumer select a given RNIC by consulting host
1268 routing tables. Once a given RNIC has been selected and an RDMA endpoint established, the
1269 route associated with the underlying LLP can dynamically change at any point in time (e.g.,
1270 ICMP PMTU update, ICMP destination unreachable, etc.), possibly resulting in an LLP
1271 connection loss (which in turn will generate an affiliated asynchronous LLP connection lost
1272 error on the QP). Similarly, the MAC address associated with a given source or next hop IP
1273 address may change. It is even possible for the IP addresses associated with an LLP connection
1274 to change for a certain class of LLPs (e.g., SCTP). In all cases, it is expected that the RNIC
1275 remain consistent with the host routing tables, ARP, and ND mechanisms.

1276 The list of IP addresses associated with the IP interface configured on the underlying Ethernet
1277 device is expected to be returned through *ri_rnic_query()*. The information provided includes
1278 both the list of all physical ports associated with the RNIC and the list of IP addresses associated
1279 with each of the ports. Note that a subsequent query MAY return a different list of IP addresses,
1280 to reflect dynamic IP address additions, deletions, and updates.

1281 **6.1.2 Packet Filtering**

1282 Any packet filtering rules that are associated with the IP address/LLP of a given RDMA
1283 endpoint MAY be applied only during the setup of the RDMA endpoint and will typically not
1284 remain effective if the endpoint entered RDMA mode. The RNICPI Consumer MUST NOT
1285 assume that all sophisticated filtering rules found in existing tools – such as *ipfilter*, *iptables*, etc.
1286 – can be applied on RDMA endpoints. Further, how the packet filtering rules are associated with
1287 a given RDMA endpoint is outside the scope of RNICPI.

1288 **6.1.3** **LLP Connection Management before RDMA Mode Transition**

1289 LLP Stream Management before RDMA Mode Transition is completely outside the scope of
1290 RNICPI. The RNICPI does not interfere with the end systems establishment of the LLP stream.
1291 With the one exception of optional MPA signaling as last streaming mode message (see
1292 [RDMAC-VERBS]) during RDMA mode conversion, it does not support any data exchange
1293 over the established LLP Connection. Explicitly, the RNICPI does not support access to any
1294 TOE mode services, which might be available on an already offloaded LLP connection.

1295 **6.1.4** **LLP Connection Conversion**

1296 The only reference of RNICPI to the LLP is the LLP handle provided by the ULP during RDMA
1297 mode conversion of the connection via the *ri_qp_modify()* interface. When provided by the ULP,
1298 it is assumed that it references an established transport connection in streaming mode. On many
1299 systems, this handle will equal with a socket file descriptor, referencing a fully established bi-
1300 directional transport connection. Both a non-privileged and a privileged ULP may provide an
1301 LLP handle to the RNICPI.

1302 As already introduced in Section 1.2.3 and 1.2.4, the RNICPI supports both RDMAC-style
1303 (negotiation of RDMA parameters at ULP level before conversion) and IETF-style RDMA mode
1304 initialization (negotiation of RDMA parameters during initial MPA request/reply exchange
1305 sequence). In case of the IETF style, the ULP may send the MPA request/reply header in
1306 streaming mode before requesting the conversion or as an argument to the *ri_qp_modify()* call
1307 itself. Typically, the MPA request frame will get issued in streaming mode and the reply frame
1308 as an argument of *ri_qp_modify()*.

1309 Since the RNICPI treats an LLP connection as completely opaque, it may also support the
1310 RDMA mode conversion of an already offloaded LLP connection. The only requirement is that
1311 this connection is completely referenced by the LLP handle and the handle is valid in the context
1312 of the kVP.

1313 **6.1.5** **LLP Connection Management after RDMA Mode Transition**

1314 After RDMA mode conversion, the LLP handle may change. This reflects the situation that the
1315 ownership of the LLP got transferred from the ULP/socket layer to the RDMA service provider.
1316 In fact, the DDP/RDMAP protocols became the new ULP of the LLP. The Consumer may query
1317 the new value of the handle by issuing an *ri_qp_query()*.

1318 It is understood that some ULPs may want to control the behavior of the LLP connection even
1319 after transition into RDMA mode. One example is the control of a transport protocols keepalive
1320 feature by the ULP. While this option is directly available at the LLP API if still in streaming
1321 mode (e.g., setting the SO_KEEPALIVE option on a TCP socket), after RDMA mode transition
1322 the result of direct operations on the potentially changed LLP handler is undefined. The RNICPI
1323 defines optional support to control the keepalive state of the LLP via *ri_qp_query()* and
1324 *ri_qp_modify()*. No other direct LLP control gets defined in the RNICPI. The Consumer may
1325 query keepalive support of the RNIC via the *ri_rnic_query()* call.

1326 **6.1.6 RDMA Association Termination**

1327 The RNICPI supports all five connection termination modes as defined in [VERBS-RDMAC]
1328 (normal close, ULP initiated termination, ULP initiated abortive teardown, remote termination,
1329 local termination, and local or remote abortive teardown).

1330 **6.2 Non-TCP Connection Management**

1331 RNICPI Connection Management interfaces were designed to transition an MPA/TCP
1332 connection into RDMA mode. However, the definition of the LLP Handle passed in is flexible
1333 enough to accommodate other Connection Management requirements.

1334 For SCTP, the LLP Handle would not reference a TCP connection, but rather an established
1335 SCTP association and the Stream ID to be used for this QP.

1336 For IB, the concept of LLP Handle does not apply.

1337 What all transports share is that the LLP Handle supplies all data required to initiate an RDMA
1338 layer connection with a remote QP. For MPA/TCP, that is typically a socket handle. RNICPI
1339 does not define this type to be a socket handle, however, because that would interfere with using
1340 the same API for SCTP, IB, and other RDMA transports.

1341

ri_rnic_open()

1343

1344 **NAME**

1345 `ri_rnic_open` – open an RNIC

1346 **SYNOPSIS: KERNEL**

```
1347 ri_api_return_t ri_rnic_open(  
1348     IN      ri_vp_handle_t  vp_hdl,  
1349     IN      ri_pbl_mode_t   pbl_mode,  
1350     IN OUT  ri_os_data_t    os_data,  
1351     OUT     ri_err_data_t   *err_data  
1352 );  
1353  
1354 typedef enum ri_pbl_mode ri_pbl_mode_t;  
1355 enum ri_pbl_mode {  
1356     RI_PHYS_PAGE_LIST_MODE,  
1357     RI_PHYS_BLOCK_LIST_MODE  
1358 };  
1359
```

1360 **DESCRIPTION**

1361 `vp_hdl` Verb provider handle to the RNIC as passed by the verb provider in the `ri_svc_attach()` call.

1362 `pbl_mode` Desired physical buffer list mode.

1363 `os_data` Opaque data owned by OS modules. IHV modules should not try to interpret it. This
1364 information gets communicated from uAL to uVP which in turn gets passed unchanged via
1365 “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
1366 appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

1367 `err_data` Data passed in all calls. How the recipient of the information uses it is outside the scope of
1368 RNICPI. The relation between the value received and later passed on is also not defined.
1369 Some interested OSVs and IHVs may use it for private communication.

1370 The `ri_rnic_open()` function allows the caller to open a given RNIC in page or block mode. The
1371 `ri_rnic_open()` operation is expected to prepare the RNIC for regular use by applications. If the
1372 RNIC does not support the desired physical buffer list mode, an error is returned. Note that at
1373 least “page” mode must always be supported by the hardware.

1374 This call can be invoked only in privileged mode in the kernel. While technically there is a user-
1375 level entry point and a “syscall” version for this call, they are never expected to be called. The
1376 user-level version of this call is simply a stub which returns the `RI_RET_NO_PERMISSION`
1377 error.

1378 Before the RNIC can be opened, verb providers are expected to call `ri_svc_attach()` to attach the
1379 RNIC to the RDMA access layer. Once the RNIC is successfully opened using `ri_rnic_open()`, it
1380 cannot be opened again until it is closed using `ri_rnic_close()`. Kernel verb providers shall be
1381 able to handle this call anytime after `ri_svc_attach()` is called including from within
1382 `ri_svc_attach()`.

1383 The definitions for `ri_vp_handle_t`, `ri_os_data_t`, and `ri_err_data_t` are OS-dependent.

1384 **RETURN VALUE**

1385	RI_RET_SUCCESS	Successful completion.
1386	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
1387	RI_RET_VP_INVALID_HANDLE	Verbs provider handle passed is invalid.
1388	RI_RET_INVALID_PARAMETER	<i>pbl_mode</i> parameter contains an undefined value.
1389	RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
1390	RI_RET_INVALID_PBL_MODE	Desired physical buffer list mode is not supported.
1391	RI_RET_RNIC_INUSE	<i>ri_rnic_open()</i> has already been called for this RNIC.
1392	RI_RET_FAIL	Unknown/fatal error.

1393 **CONSUMER USAGE**

1394 None.

1395 **IMPLEMENTATION NOTES**

1396 None.

1397 **SEE ALSO**

1398 *ri_async_set_event_handler()*, *ri_rnic_close()*, *ri_rnic_free_hdl()*, *ri_rnic_get_hdl()*,
1399 *ri_svc_attach()*, *ri_svc_detach()*

ri_rnic_close()

1400

1401 **NAME**

1402 `ri_rnic_close` – close an RNIC

1403 **SYNOPSIS: KERNEL**

```
1404     ri_api_return_t ri_rnic_close(  
1405         IN    ri_vp_handle_t  vp_hdl,  
1406         OUT   ri_err_data_t   *err_data  
1407     );
```

1408 **DESCRIPTION**

1409 `vp_hdl` Verb provider handle of the RNIC to close. This handle is the same one passed in the
1410 `ri_rnic_open()` function.

1411 `err_data` Data passed in all calls. How the recipient of the information uses it is outside the scope of
1412 RNICPI. The relation between the value received and later passed on is also not defined.
1413 Some interested OSVs and IHVs may use it for private communication.

1414 This API allows the caller to close a given RNIC. This call is responsible for deallocating all
1415 resources that were allocated by `ri_rnic_open()`. The verb provider handle specified is the same
1416 one that was used as an input parameter for the `ri_rnic_open()` call. After this call completes,
1417 there will be no upcalls (i.e., asyncs) in progress for this RNIC. Note that an RNIC must be
1418 closed before an `ri_svc_detach()` call can occur.

1419 This call can be invoked only in privileged mode in the kernel. While technically there is a user-
1420 level entry point and a “syscall” version for this call, they are never expected to be called. The
1421 user-level version of this call is simply a stub which returns the `RI_RET_NO_PERMISSION`
1422 error.

1423 If the verb provider handle has already been closed (via `ri_rnic_close()`) or if the specified verb
1424 provider handle does not match one that was passed to the `ri_rnic_open()` call, this call will fail
1425 with a return value of `RI_RET_VP_INVALID_HANDLE`. Consumers are expected to have
1426 freed all RNIC handles via `ri_rnic_free_hdl()` calls before invoking this entry point. Otherwise,
1427 `RI_RET_RNIC_INUSE` is returned.

1428 The definitions for `ri_vp_handle_t` and `ri_err_data_t` are OS-dependent.

1429 **RETURN VALUE**

1430 `RI_RET_SUCCESS` Successful completion. The specified RNIC is
1431 successfully closed.

1432 `RI_RET_VP_INVALID_HANDLE` The verb provider handle passed is invalid.

1433 `RI_RET_RNIC_INUSE` RNIC handles are still associated with the RNIC.

1434 `RI_RET_NO_PERMISSION` Caller is not privileged to perform this operation.

1435 `RI_RET_FAIL` Unknown/fatal error.

1436 **CONSUMER USAGE**

1437 None.

1438 **IMPLEMENTATION NOTES**

1439 None.

1440 **SEE ALSO**

1441 *ri_async_set_event_handler()*, *ri_rnic_free_hdl()*, *ri_rnic_get_hdl()*, *ri_rnic_open()*,
1442 *ri_svc_attach()*, *ri_svc_detach()*

1443

ri_async_set_event_handler()

1444 **NAME**

1445 ri_async_set_event_handler – register an event handler for asynchronous events from the
1446 specified RNIC

1447 **SYNOPSIS: KERNEL**

```
1448 ri_api_return_t ri_async_set_event_handler (  
1449     IN    ri_vp_handle_t      vp_hdl,  
1450     IN    ri_async_event_callback_t  async_event_callback,  
1451     OUT   ri_err_data_t      *err_data  
1452 );  
1453  
1454 typedef struct ri_ib_port_event ri_ib_port_event_t;  
1455 struct ri_ib_port_event {  
1456     /* IB verbs do not specify that port number has to be returned in a  
1457     port event. Port number 0 indicates that there is no knowledge of which  
1458     port this event is associated with. */  
1459     uint8_t      port_number;  
1460     ri_ib_port_state_t  port_state;  
1461 };  
1462  
1463 typedef union ri_event_data ri_event_data_t;  
1464 union ri_event_data {  
1465     ri_ib_port_event_t  port_event;  
1466     ri_os_data_t      resource_os_data;  
1467 };  
1468  
1469 typedef struct ri_event_record ri_event_record_t;  
1470 struct ri_event_record {  
1471     ri_async_event_type_t  event_type;  
1472     ri_event_data_t      event_data;  
1473     ri_err_data_t      err_data;  
1474 };  
1475  
1476 typedef void (*ri_async_event_callback_t) (  
1477     IN OUT ri_os_data_t      vp_hdl_os_data,  
1478     IN     ri_event_record_t  *event  
1479 );  
1480  
1481 typedef enum ri_async_event_type ri_async_event_type_t;  
1482 enum ri_async_event_type {  
1483  
1484     /* iWARP-only events */  
1485     RI_EVENT_QP_LLQ_CLOSE_COMPLETE,  
1486     RI_EVENT_QP_TERMINATE_MESSAGE_RECEIVED,  
1487     RI_EVENT_QP_LLQ_CONNECTION_RESET,  
1488     RI_EVENT_QP_LLQ_CONNECTION_LOST,  
1489     RI_EVENT_QP_LLQ_INTEGRITY_ERROR_SIZE,  
1490     RI_EVENT_QP_LLQ_INTEGRITY_ERROR_CRC,  
1491     RI_EVENT_QP_LLQ_INTEGRITY_ERROR_BAD_FPDU,  
1492     RI_EVENT_QP_REMOTE_OP_ERROR_DDP_VERSION,  
1493     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_LLQ_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 };

```

1545 **DESCRIPTION**

1546 *vp_hdl* Handle to the RNIC to get events from. This must be same as the handle used in
1547 *ri_rnic_open()*.

1548 *async_event_callback*
1549 A handler that is called when there is an asynchronous event from the RNIC. See
1550 *ri_async_event_callback_t* for arguments passed back in the callback.

1551 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
1552 RNICPI. The relation between the value received and later passed on is also not defined.
1553 Some interested OSVs and IHVs may use it for private communication.

1554 *ri_async_set_event_handler()* allows privileged Consumers to register an asynchronous event
1555 handler for the specified RNIC. Only one asynchronous event handler can be registered per
1556 RNIC. Subsequent calls to *ri_async_set_event_handler()* with a given RNIC handle will
1557 overwrite the asynchronous event handler routine to be called. If *async_event_callback* is
1558 NULL, no callback function is invoked upon any asynchronous events or errors. Any
1559 asynchronous event that occurs prior to the registration of the event handler will be lost.

1560 The *os_data* passed by the Consumer at the time of *ri_rnic_open()* is provided as the
1561 *vp_hdl_os_data* parameter to the callback function. The event record *event* contains the
1562 appropriate event type and additional information on the resource affiliated to the event.
1563 *resource_os_data* is set to the *os_data* of the resource set by the Consumer at the time of
1564 resource creation. Any fatal error detected in the RNIC not specifically related to a QP, CQ,
1565 SRQ, or port results in an RI_EVENT_RNIC_LOCAL_CATASTROPHIC_ERROR event and
1566 *resource_os_data* is set to the *os_data* passed by the Consumer at the time of *ri_rnic_open()*. If
1567 the *event_type* field in the event record indicates that it is a CQ-related event (i.e., events with
1568 RI_EVENT_CQ prefix), *resource_os_data* is the *os_data* of the CQ set by the Consumer during
1569 *ri_cq_create()*. If the event type indicates it is an SRQ-related event (i.e., events with the
1570 RI_EVENT_SRQ prefix), *resource_os_data* is the *os_data* of the SRQ set by the Consumer
1571 during *ri_srq_create()*. For events related to QP (i.e., events with RI_EVENT_QP prefix),
1572 *resource_os_data* is the *os_data* set by the Consumer at the time of *ri_qp_create()*.

1573 IB devices generate an unaffiliated RI_EVENT_PORT_EVENT event if the link state of a port
1574 changes. *port_state* indicates if the link is active or has become unavailable. A *port_number* of 0
1575 indicates that no additional port-specific information is provided in the event.

1576 Most of the errors are transport-specific. See the related verb specifications for more details on
1577 the cause of the events or errors.

1578 *ri_async_set_event_handler()* can be invoked only in the kernel. The user mode entry point of
1579 this call is simply a stub that returns RI_RET_NO_PERMISSION.

1580 **RETURN VALUE**

1581 RI_RET_SUCCESS The specified asynchronous RNIC event handler has been
1582 successfully registered.

1583 RI_RET_VP_INVALID_HANDLE The verb provider handle passed is invalid.

1584 RI_RET_NO_PERMISSION Caller is not privileged to perform this operation.

1585 RI_RET_FAIL Unknown/fatal error.

1586 **CONSUMER USAGE**

1587 None.

1588 **IMPLEMENTATION NOTES**

1589 None.

1590 **SEE ALSO**

1591 *ri_cq_create()*, *ri_cq_set_event_handler()*, *ri_qp_create()* *ri_rnic_open()*, *ri_srq_create()*

ri_rnic_get_hdl()

1592

1593 **NAME**

1594 `ri_rnic_get_hdl` – obtain an RNIC handle usable by applications

1595 **SYNOPSIS**

```
1596 ri_api_return_t ri_rnic_get_hdl(  
1597     IN      ri_vp_handle_t      vp_hdl,  
1598     IN      ri_consumer_domain_t location,  
1599     OUT     ri_rnic_handle_t     *rnic,  
1600     IN OUT  ri_vp_data_t         vp_data,  
1601     IN OUT  ri_os_data_t         os_data,  
1602     OUT     ri_err_data_t        *err_data  
1603 );  
1604  
1605 typedef struct ri_consumer_domain ri_consumer_domain_t;  
1606 struct ri_consumer_domain {  
1607     RI_CONSUMER_LOCATION_KERNEL,  
1608     RI_CONSUMER_LOCATION_USER  
1609 };
```

1610 **DESCRIPTION**

1611 *vp_hdl* Verb provider handle to the RNIC originally passed by the verb provider in *ri_svc_attach()*.

1612 *location* Consumer’s domain (user/kernel). This argument is ignored in the user-level version of this
1613 call (the location is inherently user-level).

1614 *rnic* Returned handle to the RNIC for regular operations.

1615 *vp_data* Opaque data owned by IHV modules, OS modules should not try to interpret it. This
1616 information gets communicated from uVP through a “syscall” which in turn gets passed
1617 unchanged by kAL to kVP while processing this call. This must be zero when passed by
1618 uAL to uVP and uVPs must ignore this parameter.

1619 *os_data* Opaque data owned by OS modules, IHV modules should not try to interpret it. This
1620 information gets communicated from uAL to uVP which in turn gets passed unchanged via
1621 “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
1622 appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

1623 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
1624 RNICPI. The relation between the value received and later passed on is also not defined.
1625 Some interested OSVs and IHVs may use it for private communication.

1626 This function allows the caller to create a handle to the RNIC indicated by the verb provider
1627 handle. The RNIC handle returned is specified as an input parameter to most subsequent API
1628 calls (there are a few exceptions such as *ri_rnic_close()*). Further, the returned RNIC handle is
1629 used to set the context of these subsequent operations. This call may be made more than once to
1630 get different handles for the same RNIC.

1631 The verb provider handle specified as the input parameter to this call must be opened with
1632 *ri_rnic_open()* prior to invoking *ri_rnic_get_hdl()*. If it is called on an RNIC that is not currently
1633 open, `RI_RET_RNIC_NOT_OPEN` is returned.

1634 The definitions for *ri_rnic_handle_t*, *ri_vp_handle_t*, *ri_vp_data_t*, *ri_os_data_t*, and
1635 *ri_err_data_t* are OS-dependent.

1636 RETURN VALUE

1637	RI_RET_SUCCESS	Successful completion. A valid RNIC handle is returned.
1638	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
1639	RI_RET_VP_INVALID_HANDLE	The verb provider handle passed is invalid.
1640	RI_RET_INVALID_PARAMETER	<i>location</i> has an undefined value or <i>rnic</i> is NULL.
1641	RI_RET_RNIC_NOT_OPEN	RNIC is not open.
1642	RI_RET_FAIL	Unknown/fatal error.

1643 CONSUMER USAGE

1644 None.

1645 IMPLEMENTATION NOTES

1646 How the uAL finds out about available RNICs to call *ri_rnic_get_hdl()* is outside of the scope of
1647 RNICPI. However, typically it is done through private communication (e.g., syscalls) between
1648 uAL and kAL.

1649 The *location* parameter specified will be used by the implementation to make choices about type
1650 of memory copy, kind of protection, need to do memory pinning, etc. One common usage of this
1651 function is to create a new RNIC handle per user process (i.e., different address space). Some
1652 implementations may be able to share resources between objects created through the same RNIC
1653 handle.

1654 The number of handles that can be supported by this function is implementation-dependent. On
1655 the other hand, if an implementation does not really support the concept of different RNIC
1656 handles, it might simply return the same handle to all.

1657 SEE ALSO

1658 *ri_async_set_event_handler()*, *ri_rnic_close()*, *ri_rnic_free_hdl()*, *ri_rnic_open()*,
1659 *ri_svc_attach()*, *ri_svc_detach()*

ri_rnic_query()

1660

1661 **NAME**

1662 `ri_rnic_query` – return the attributes of an RNIC

1663 **SYNOPSIS**

```
1664 ri_api_return_t ri_rnic_query(  
1665     IN     ri_rnic_handle_t    rnic,  
1666     IN     ri_rnic_query_flags_t flags,  
1667     OUT    ri_rnic_attr_t      *attrs,  
1668     IN OUT uint_t              *attr_size,  
1669     OUT    ri_err_data_t       *err_data  
1670 );  
1671  
1672 typedef enum ri_rnic_query_flags ri_rnic_query_flags_t;  
1673 enum ri_rnic_query_flags {  
1674     RI_QUERY_RNIC_PORT_INFO = 1 << 0  
1675 };
```

1676 **DESCRIPTION**

1677 *rnic* Handle to the RNIC to query obtained from `ri_rnic_get_hdl()`.

1678 *flags* Flag to indicate whether port information is also desired.

1679 *attrs* Consumer allocated buffer for attributes of the RNIC.

1680 *attr_size* The input value is the size of the buffer. The output value is the actual size of the requested
1681 attributes.

1682 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
1683 RNICPI. The relation between the value received and later passed on is also not defined.
1684 Some interested OSVs and IHVs may use it for private communication.

1685 This function returns the attributes of the specified RNIC in *attrs*. The format of the attribute
1686 information is described in the `ri_rnic_attr_t` reference page and the related reference pages for
1687 each transport-specific attribute. The *flags* argument specifies whether the information queried
1688 will also include information about the ports of the RNIC. Undefined flags cause the error
1689 `RI_RET_INVALID_PARAMETER` to be returned. The size of the *attrs* buffer is specified in
1690 *attr_size*. The *attr_size* parameter should be large enough to contain all requested information,
1691 including the main `ri_rnic_attr_t` struct, any transport-specific struct, and – if port information
1692 has been requested – the associated port-related structs (which may include lists of address
1693 information). When the function returns, *attr_size* contains the size of the queried data.

1694 If *attr_size* is not large enough to fit the requested attributes, then the error
1695 `RI_RET_BUFFER_TOO_SMALL` is returned and no attributes are returned. Note that *attr_size*
1696 is still updated in that case to provide the correct size. An initial size for attributes including port
1697 data is returned in the *query_size* field of `ri_svc_attach()`. Note that some transports allow the
1698 size of the attributes to vary dynamically, so *attr_size* may only indicate the size of the attributes
1699 at the moment that `ri_svc_attach()` was called.

1700 The definitions of `ri_rnic_handle_t` and `ri_err_data_t` are OS-dependent.

1701	RETURN VALUE	
1702	RI_RET_SUCCESS	Successful completion. <i>attrs</i> contains the RNIC attributes.
1703		
1704	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
1705	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
1706	RI_RET_INVALID_PARAMETER	Undefined <i>flags</i> parameter or <i>attrs</i> and/or <i>attr_size</i> parameter is NULL.
1707		
1708	RI_RET_BUFFER_TOO_SMALL	Input <i>attr_size</i> was not large enough to contain the requested attributes.
1709		
1710	RI_RET_FAIL	Unknown/fatal error.
1711	CONSUMER USAGE	
1712	None.	
1713	IMPLEMENTATION NOTES	
1714	None.	
1715	SEE ALSO	
1716	<i>ri_rnic_modify()</i> , <i>ri_rnic_attr_t</i>	

ri_rnic_modify()

1717

1718 **NAME**

1719 `ri_rnic_modify` – modify the attributes of an RNIC

1720 **SYNOPSIS: KERNEL**

```
1721 ri_api_return_t ri_rnic_modify(  
1722     IN   ri_rnic_handle_t    rnic,  
1723     IN   ri_rnic_attr_mask_t attr_mask,  
1724     IN   ri_rnic_modify_req_t *attrs,  
1725     OUT  ri_err_data_t      *err_data  
1726 );  
1727  
1728 typedef enum ri_rnic_attr_mask ri_rnic_attr_mask_t;  
1729 enum ri_rnic_attr_mask {  
1730     RI_IB_RNIC_ATTR_SI_GUID = 1 << 0  
1731 };  
1732  
1733 typedef enum ri_port_attr_mask ri_port_attr_mask_t;  
1734 enum ri_port_attr_mask {  
1735     RI_IB_PORT_ATTR_SHUTDOWN = 1 << 0,  
1736     RI_IB_PORT_ATTR_INIT_TYPE = 1 << 1,  
1737     RI_IB_PORT_ATTR_QKEY_RESET = 1 << 2,  
1738     RI_IB_PORT_ATTR_CAP_SET = 1 << 3, /* set FLAGS below */  
1739     RI_IB_PORT_ATTR_CAP_FLAG_SM = 1 << 4,  
1740     RI_IB_PORT_ATTR_CAP_FLAG_SNMP = 1 << 5,  
1741     RI_IB_PORT_ATTR_CAP_FLAG_DEVMGT = 1 << 6,  
1742     RI_IB_PORT_ATTR_CAP_FLAG_VENDOR = 1 << 7  
1743 };  
1744  
1745 #define RI_IB_ALL_PORTS (0)  
1746  
1747 typedef struct ri_port_modify_req ri_port_modify_req_t;  
1748 struct ri_port_modify_req {  
1749     ri_port_attr_mask_t port_attr_mask;  
1750     uint8_t              port_number;  
1751     uint8_t              port_init_type; /* 4 bits */  
1752 };  
1753  
1754 typedef struct ri_rnic_modify_req ri_rnic_modify_req_t;  
1755 struct ri_rnic_modify_req {  
1756     uint64_t             si_guid;  
1757     ri_port_modify_req_t *port_attrs;  
1758     uint8_t              port_list_size;  
1759 };
```

1760 **DESCRIPTION**

1761 *rnic* RNIC handle of the RNIC to modify.

1762 *attr_mask* Mask to select attributes to be modified.

1763 *attrs* Attributes structure which contains the requested values for the attributes corresponding to
1764 the mask bits chosen.

1765 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
 1766 RNICPI. The relation between the value received and later passed on is also not defined.
 1767 Some interested OSVs and IHVs may use it for private communication.

1768 This function allows the caller to modify the attributes of the RNIC associated with the specified
 1769 handle. The attributes to be modified are selected using *attr_mask* which the caller can construct
 1770 by inclusive OR-ing the required values of *ri_rnic_attr_mask_t*. Any bits outside the definition
 1771 of *ri_rnic_attr_mask_t* will cause the error `RI_RET_INVALID_PARAMETER` to be returned.
 1772 Upon successful completion of the call, a subsequent successful *ri_rnic_query()* call will see the
 1773 new values reflected in the attributes returned. Because relatively few attributes of an RNIC can
 1774 be modified, this function uses a different version of the attribute structure which is not shared
 1775 by *ri_rnic_query()*.

1776 This call applies only to privileged callers. Callers without sufficient privilege will return the
 1777 `RI_RET_NO_PERMISSION` error.

1778 Currently, only the InfiniBand transport supports modifying RNIC attributes. Calling this
 1779 function on RNICs using other transports will return `RI_RET_XPORT_UNSUPPORTED`. The
 1780 *attrs* argument contains the requested changes to be made to the RNIC. If the
 1781 `RI_IB_RNIC_ATTR_SI_GUID` flag is set, the new system image GUID is specified in the
 1782 *si_guid* field of the *attrs* argument. The *port_attrs* field is a pointer to an array of port
 1783 modification requests. The length of the array is specified in the *port_list_size* field. If
 1784 *port_list_size* is zero, no port attributes will be changed and the *port_attrs* field is ignored.

1785 Each *ri_port_modify_req_t* structure contains a *port_attr_mask* field which specifies the port
 1786 attributes to be changed. The mask can be constructed by inclusive OR-ing the required values
 1787 of *ri_port_attr_mask_t*. Any bits outside the definition of *ri_port_attr_mask_t* will cause the
 1788 error `RI_RET_INVALID_PARAMETER` to be returned. The *port_number* field specifies the
 1789 port to be modified. If `RI_IB_ALL_PORTS` is specified, then the change is made to all ports on
 1790 the RNIC. If a non-existent port is specified, then `RI_RET_INVALID_PARAMETER` is
 1791 returned.

1792 If `RI_IB_PORT_ATTR_CAP_SET` is specified, then the `RI_IB_PORT_ATTR_CAP_FLAG_`
 1793 `fields specified are set and those not specified are cleared. If RI_IB_PORT_ATTR_CAP_SET is
 1794 not specified, then the RI_IB_PORT_ATTR_CAP_FLAG_ items will trigger the
 1795 RI_RET_INVALID_PARAMETER error. If RI_IB_PORT_ATTR_INIT_TYPE is set, then the
 1796 port_init_type field specified the new InitType value.`

1797 The system image GUID, port shutdown capability, *InitType*, and *Q_Key* violation counter are
 1798 all optional features. If an attempt is made to use an unsupported optional item,
 1799 `RI_RET_OP_UNSUPPORTED` is returned.

1800 The definitions of *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

1801 RETURN VALUE

1802	<code>RI_RET_SUCCESS</code>	Successful completion.
1803	<code>RI_RET_RNIC_INVALID_HANDLE</code>	The RNIC handle passed is invalid.
1804	<code>RI_RET_INVALID_PARAMETER</code>	<i>attrs</i> passed is NULL or unknown flag in <i>attr_mask</i> 1805 specified.
1806	<code>RI_RET_NO_PERMISSION</code>	Caller is not privileged to perform this operation.

1807	RI_RET_OP_UNSUPPORTED	Specified operation not supported (e.g., setting an unsupported optional item).
1808		
1809	RI_RET_XPORT_UNSUPPORTED	The operation was called on a non-InfiniBand device.
1810	RI_RET_FAIL	Unknown/fatal error.
1811	CONSUMER USAGE	
1812	None.	
1813	IMPLEMENTATION NOTES	
1814	None.	
1815	SEE ALSO	
1816	<i>ri_rnic_query()</i>	

ri_rnic_free_hdl()

1817

1818 NAME

1819 ri_rnic_free_hdl – invalidate an application RNIC handle

1820 SYNOPSIS

```
1821 ri_api_return_t ri_rnic_free_hdl(  
1822     IN   vp_handle_t      vp_hdl,  
1823     IN   ri_rnic_handle_t rnic,  
1824     OUT  ri_err_data_t    *err_data  
1825 );
```

1826 DESCRIPTION

1827 *vp_hdl* Verb provider handle to the RNIC, same as the one passed in *ri_rnic_get_hdl()*.

1828 *rnic* RNIC handle to free, originally returned through *ri_rnic_get_hdl()*.

1829 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
1830 RNICPI. The relation between the value received and later passed on is also not defined.
1831 Some interested OSVs and IHVs may use it for private communication.

1832 This function allows the caller to destroy association of the specified handle (from
1833 *ri_rnic_get_hdl()*) with its RNIC. Further calls using the specified RNIC handle may fail with a
1834 return value of RI_RET_RNIC_INVALID_HANDLE. Note that it is necessary to free all
1835 handles from *ri_rnic_get_hdl()* belonging to an RNIC, before *ri_rnic_close()* can be called on
1836 that RNIC.

1837 The Consumer is expected to free, deallocate, or deregister all resources obtained through this
1838 RNIC handle (for example, Protection Domains, Queue Pairs, Completion Queues, Memory
1839 Registrations, etc.) before invoking this call. As each object is released through the appropriate
1840 function call, no further upcalls (i.e., CQ completion events and asyncs) will be in progress that
1841 can reference the released object. So by the time *ri_rnic_free_hdl()* completes, there should be
1842 no upcalls in progress for any of the objects that were obtained through the released RNIC
1843 handle. If *ri_rnic_free_hdl()* is called and there are outstanding RNIC resources associated with
1844 the RNIC handle specified, RI_RET_RNIC_INUSE is returned.

1845 The definitions of *ri_vp_handle_t*, *ri_rnic_handle_t*, and *ri_err_data_t* are OS-dependent.

1846 RETURN VALUE

1847 RI_RET_SUCCESS Successful completion. The specified RNIC handle is
1848 successfully destroyed.

1849 RI_RET_VP_INVALID_HANDLE Verb provider handle to the RNIC is invalid.

1850 RI_RET_RNIC_INVALID_HANDLE RNIC handle passed is invalid.

1851 RI_RET_RNIC_INUSE There are outstanding resources – like Protection
1852 Domains, Queue Pairs, Memory Registrations, etc. – still
1853 associated with the RNIC handle.

1854 RI_RET_FAIL Unknown/fatal error.

1855 **CONSUMER USAGE**

1856 None.

1857 **IMPLEMENTATION NOTES**

1858 None.

1859 **SEE ALSO**

1860 *ri_async_set_event_handler()*, *ri_rnic_close()*, *ri_rnic_get_hdl()*, *ri_rnic_open()*,

1861 *ri_svc_attach()*, *ri_svc_detach()*

ri_rnic_diag()

1862

1863 **NAME**

1864 `ri_rnic_diag` – perform privileged diagnostic operations on the RNIC

1865 **SYNOPSIS: KERNEL**

```

1866 ri_api_return_t ri_rnic_diag (
1867     IN      ri_rnic_handle_t  rnic,
1868     IN      ri_diag_op_type_t op,
1869     IN      void              *arg,
1870     IN OUT  unsigned char     *opval,
1871     IN OUT  ri_length_t       *oplen,
1872     OUT     ri_err_data_t     *err_data
1873 );
    
```

1874 **DESCRIPTION**

1875 *rnic* Handle to RNIC to perform diagnostic operation on.

1876 *op* Opcode of the diagnostic operation.

1877 *arg* Operation-dependent argument.

1878 *opval* Pointer to location to hold kVP-specific information.

1879 *oplen* Value result parameter indicating size of *opval*.

1880 *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.

<i>op</i>	<i>arg</i>	<i>opval</i>	<i>oplen</i>
RI_DIAG_RNIC_DETAILS	Ignored	Upon successful return, <i>opval</i> 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 <i>opval</i> is NULL or is not big enough to hold the entire information as reflected by <i>oplen</i> , an error is returned and the correct size is returned in <i>oplen</i> .	IN: size of <i>opval</i> OUT: expected/ actual size of information returned in <i>opval</i>

<i>op</i>	<i>arg</i>	<i>opval</i>	<i>oplen</i>
RI_DIAG_QP_DETAILS	QP handle	Upon successful return, <i>opval</i> 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 <i>opval</i> is NULL or is not big enough to hold the entire information as reflected by <i>oplen</i> , an error is returned and the correct size is returned in <i>oplen</i> .	IN: size of <i>opval</i> OUT: expected/actual size of information returned in <i>opval</i>
RI_DIAG_RNIC_DUMP	Ignored	Upon successful return, <i>opval</i> contains kVP-dependent dump of RNIC internal memory, contexts, traces, etc. This information may be logged by a diagnostic tool. If <i>opval</i> is NULL or is not big enough to hold the entire information as reflected by <i>oplen</i> , an error is returned and the correct size is returned in <i>oplen</i> .	IN: size of <i>opval</i> OUT: expected/actual size of information returned in <i>opval</i>
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

1883
1884
1885
1886

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.

1887
1888
1889

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.

1890
1891

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.

1892 RETURN VALUE

1893	RI_RET_SUCCESS	Successful completion.
1894	RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
1895	RI_RET_INVALID_PARAMETER	Invalid <i>arg</i> .

1896	RI_RET_BUFFER_TOO_SMALL	The size of <i>opval</i> as specified in <i>oplen</i> is insufficient to hold the kVP-dependent information, or <i>opval</i> is NULL.
1897		
1898		
1899	RI_RET_OP_UNSUPPORTED	The specified <i>opcode</i> is not supported by the kVP.
1900		
1901	RI_RET_NO_PERMISSION	The caller is not privileged.
1902	RI_RET_FAIL	Unknown/fatal error.

1903 CONSUMER USAGE

1904 Here is a sample usage:

```

1905     ri_length_t      oplen;
1906     unsigned char    *opval;
1907     ri_rnic_handle_t rnic;
1908     ri_api_return_t  ret;
1909
1910     /* Determine the amount of memory required for the operation */
1911     oplen = 0;
1912     if (ri_rnic_diag(rnic, RI_DIAG_RNIC_DETAILS, NULL, NULL, &oplen)
1913         == RI_RET_BUFFER_TOO_SMALL) {
1914
1915         /* Allocate memory for opval */
1916         ret = ri_rnic_diag(rnic, RI_DIAG_RNIC_DETAILS,
1917             NULL, opval, &oplen);
1918     }

```

1919 IMPLEMENTATION NOTES

1920 The kAL, or its equivalent, is responsible for ensuring that the caller has the required
1921 permissions to perform diagnostic operations on the RNIC.

1922 SEE ALSO

1923 [ri_rnic_open\(\)](#), [ri_rnic_get_hdl\(\)](#)

ri_pd_alloc(), ri_pd_dealloc()

1924

1925 NAME

1926 `ri_pd_alloc` – allocate a Protection Domain

1927 `ri_pd_dealloc` – deallocate a Protection Domain

1928 SYNOPSIS

```
1929 ri_api_return_t ri_pd_alloc (  
1930     IN      ri_rnic_handle_t  rnic,  
1931     OUT     ri_pd_handle_t    *pd,  
1932     IN OUT  ri_vp_data_t      vp_data,  
1933     IN OUT  ri_os_data_t      os_data,  
1934     OUT     ri_err_data_t     *err_data  
1935 );
```

1936

```
1937 ri_api_return_t ri_pd_dealloc (  
1938     IN  ri_rnic_handle_t  rnic,  
1939     IN  ri_pd_handle_t    pd,  
1940     OUT ri_err_data_t     *err_data  
1941 );
```

1942 DESCRIPTION

1943 *rnic* Handle to the RNIC on which to allocate/deallocate a Protection Domain.

1944 *pd* Handle to the Protection Domain.

1945 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
1946 information gets communicated from uVP through a “syscall” which in turn gets passed
1947 unchanged by kAL to kVP while processing this call. This must be zero when passed by
1948 uAL to uVP and uVPs must ignore this parameter.

1949 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
1950 information gets communicated from uAL to uVP which in turn gets passed unchanged via
1951 “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
1952 appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

1953 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
1954 RNICPI. The relation between the value received and later passed on is also not defined.
1955 Some interested OSVs and IHVs may use it for private communication.

1956 `ri_pd_alloc()` allocates a Protection Domain within the context of the RNIC specified by *rnic*.
1957 `ri_pd_dealloc()` deallocates a Protection Domain that was previously allocated by `ri_pd_alloc()`
1958 within the context of the RNIC specified by *rnic*.

1959 RETURN VALUE

1960 `RI_RET_SUCCESS` Successful completion. A valid handle to the allocated
1961 Protection Domain is returned.

1962 `RI_RET_RESOURCE_SHORTAGE` Not enough resources to complete the call.

1963	RI_RET_PD_INUSE	The Protection Domain indicated by <i>pd</i> is still being used
1964		by some other resource.
1965	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle indicated by <i>rnice</i> is invalid.
1966	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
1967	RI_RET_FAIL	Unknown/fatal error.
1968	CONSUMER USAGE	
1969	None.	
1970	IMPLEMENTATION NOTES	
1971	None.	
1972	SEE ALSO	
1973	<i>ri_rnic_get_hdl()</i> , <i>ri_rnic_free_hdl()</i>	

ri_cq_create()

1974

1975 **NAME**

1976 `ri_cq_create` – create a Completion Queue (CQ) on the specified RNIC

1977 **SYNOPSIS**

```
1978 ri_api_return_t ri_cq_create (  
1979     IN     ri_rnic_handle_t  rnic,  
1980     IN OUT ri_cq_attr_t     *cq_attr,  
1981     OUT    ri_cq_handle_t    *cq,  
1982     IN OUT ri_vp_data_t     vp_data,  
1983     IN OUT ri_os_data_t     os_data,  
1984     OUT    ri_err_data_t     *err_data  
1985 );  
1986  
1987 typedef uint_t ri_cq_event_handler_id_t;  
1988 typedef struct ri_cq_attr ri_cq_attr_t;  
1989 struct ri_cq_attr {  
1990     uint32_t          num_cqe;  
1991     ri_cq_event_handler_id_t handler_id;  
1992 };
```

1993 **DESCRIPTION**

1994 *rnic* Handle to the RNIC on which to create a CQ.

1995 *cq_attr* Pointer to CQ attribute structure.

1996 *cq* Handle to the newly created CQ.

1997 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
1998 information gets communicated from uVP to the “syscall” version of this function which in
1999 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
2000 when passed by uAL to uVP and uVPs must ignore this parameter.

2001 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
2002 information gets communicated from uAL to uVP which in turn gets passed unchanged to
2003 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
2004 get passed unchanged to appropriate calls back into the OS.

2005 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2006 RNICPI. The relation between the value received and later passed on is also not defined.
2007 Some interested OSVs and IHVs may use it for private communication.

2008 `ri_cq_create()` allows the caller to create a Completion Queue on the specified RNIC. The
2009 Consumer fills in the desired number of CQ entries and completion handler ID in *cq_attr*. Upon
2010 successful completion, the actual number of allocated CQ entries is returned in the *num_cqe*
2011 field of *cq_attr*, and a handle to the newly created CQ is returned in *cq*. The number of CQ
2012 entries returned can be greater than or equal to the one requested. If the number of CQ entries
2013 requested is greater than the maximum number of CQ entries supported by the RNIC,
2014 `RI_RET_CQ_CAPACITY_EXCEEDED` is returned and the CQ is not created.

2015 A privileged RNICPI Consumer such as kAL can specify the completion handler that is
2016 associated with the CQ in the *handler_id* field of *cq_attr*. Privileged Consumers can obtain the

2017 completion handler through the *ri_cq_set_event_handler()* call. If the *handler_id* specified in
 2018 *cq_attr* is NULL, no completion handler is associated with the given CQ and the Consumer
 2019 cannot enable notifications on the CQ. If an invalid *handler_id* is specified, *ri_cq_create()* fails
 2020 with an RI_RET_CQ_INVALID_EVENT_HANDLER error. It is not possible to change the
 2021 *handler_id* associated with the CQ once the CQ is created. However, privileged Consumers can
 2022 change the completion handler that is associated with a *handler_id* through the
 2023 *ri_cq_set_event_handler()* call.

2024 Non-privileged RNICPI Consumers such as uAL cannot specify completion handlers for CQ.
 2025 uAL must set *handler_id* in *cq_attr* to zero and uVP must ignore the *handler_id* field passed.

2026 If a completion handler is registered for a given CQ event handler ID and the CQ is enabled for
 2027 notification, the kVP shall call the completion handler with the *os_data* passed in during
 2028 *ri_cq_create()* as the CQ handle.

2029 **RETURN VALUE**

2030	RI_RET_SUCCESS	The call completed successfully and a valid CQ
2031		handle is returned.
2032	RI_RET_RESOURCE_SHORTAGE	Inadequate resources to complete the call.
2033	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
2034	RI_RET_INVALID_PARAMETER	<i>cq</i> and/or <i>cq_attr</i> is NULL.
2035	RI_RET_CQ_CAPACITY_EXCEEDED	Number of CQ entries requested exceeds RNIC
2036		capability.
2037	RI_RET_CQ_INVALID_EVENT_HANDLER	Invalid CQ Event handler.
2038	RI_RET_FAIL	Unknown/fatal error.

2039 **CONSUMER USAGE**

2040 None.

2041 **IMPLEMENTATION NOTES**

2042 None.

2043 **SEE ALSO**

2044 *ri_cq_destroy()*, *ri_cq_modify()*, *ri_cq_query()*, *ri_cq_set_event_handler()*

ri_cq_query()

2045

2046 NAME

2047

ri_cq_query – query the attributes associated with a given CQ

2048 SYNOPSIS

2049

2050

2051

2052

2053

2054

```
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  
);
```

2055 DESCRIPTION

2056

rnic Handle to the RNIC the CQ belongs to.

2057

cq Handle to CQ.

2058

cq_attr Pointer to CQ attribute information.

2059

2060

2061

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.

2062

2063

2064

2065

2066

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*.

2067

2068

2069

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()*.

2070 RETURN VALUE

2071

2072

RI_RET_SUCCESS

CQ attributes are returned in *cq_attr* upon successful completion.

2073

RI_RET_RNIC_INVALID_HANDLE

The RNIC handle passed is invalid.

2074

RI_RET_CQ_INVALID_HANDLE

The CQ handle passed is invalid.

2075

RI_RET_INVALID_PARAMETER

cq_attr is NULL.

2076

RI_RET_CQ_ERROR

CQ has an overrun or has become inaccessible.

2077

RI_RET_FAIL

Unknown/fatal error.

2078 CONSUMER USAGE

2079

None.

2080 **IMPLEMENTATION NOTES**

2081 None.

2082 **SEE ALSO**

2083 [*ri_cq_create\(\)*](#), [*ri_cq_destroy\(\)*](#), [*ri_cq_modify\(\)*](#), [*ri_cq_set_event_handler\(\)*](#)

2084

ri_cq_modify()

2085

2086 **NAME**

2087

ri_cq_modify – modify some attributes of a CQ

2088 **SYNOPSIS**

2089

2090

2091

2092

2093

2094

2095

2096

2097

2098

2099

2100

```
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  
};
```

2101 **DESCRIPTION**

2102

rnic Handle to the RNIC the CQ belongs to.

2103

cq Handle to CQ.

2104

cq_attr Pointer to CQ attribute information.

2105

mask Mask for the CQ modify operation.

2106

2107

2108

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.

2109

2110

2111

2112

2113

2114

2115

2116

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.

2117

2118

2119

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()*.

2120 **RETURN VALUE**

2121

RI_RET_SUCCESS Successful completion.

2122

RI_RET_RESOURCE_SHORTAGE Inadequate resources to complete the call.

2123

RI_RET_RNIC_INVALID_HANDLE The RNIC handle passed is invalid.

2124	RI_RET_CQ_INVALID_HANDLE	The CQ handle passed is invalid.
2125	RI_RET_INVALID_PARAMETER	<i>cq_attr</i> is NULL or <i>num_cqe</i> is 0 or invalid <i>mask</i> specified.
2126		
2127	RI_RET_CQ_CAPACITY_EXCEEDED	Number of CQ entries requested exceeds RNIC capability.
2128		
2129	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.
2130		
2131		
2132	RI_RET_CQ_ERROR	CQ has an overrun or has become inaccessible.
2133	RI_RET_FAIL	Unknown/fatal error.
2134	CONSUMER USAGE	
2135	None.	
2136	IMPLEMENTATION NOTES	
2137	None.	
2138	SEE ALSO	
2139	<i>ri_cq_create()</i> , <i>ri_cq_destroy()</i> , <i>ri_cq_modify()</i> , <i>ri_cq_query()</i>	
2140		

ri_cq_destroy()

2141

2142 NAME

2143 `ri_cq_destroy` – destroy the specified CQ

2144 SYNOPSIS

```
2145 ri_api_return_t ri_cq_destroy (  
2146     IN  ri_rnic_handle_t  rnic,  
2147     IN  ri_cq_handle_t    cq,  
2148     OUT ri_err_data_t     *err_data  
2149 );
```

2150 DESCRIPTION

2151 *rnic* Handle of the RNIC the CQ belongs to.

2152 *cq* Handle to the CQ to be destroyed.

2153 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2154 RNICPI. The relation between the value received and later passed on is also not defined.
2155 Some interested OSVs and IHVs may use it for private communication.

2156 `ri_cq_destroy()` destroys the specified CQ. Resources allocated by `ri_cq_create()` and
2157 subsequently by `ri_cq_modify()` will be deallocated by this call. If there is a QP still associated
2158 with the CQ, an `RI_RET_CQ_INUSE` error is returned. Upon successful return, the VP shall not
2159 invoke the CQ event handler or post any further CQ event to this CQ. Also, no completion event
2160 handling for this CQ will be in progress upon successful return. Any completion that has not
2161 been retrieved from the CQ is discarded. Upon successful return, no async error/event for this
2162 CQ will be in progress, and no further ones will be invoked for this CQ.

2163 RETURN VALUE

2164 `RI_RET_SUCCESS` The specified CQ has been destroyed successfully.

2165 `RI_RET_RNIC_INVALID_HANDLE` The RNIC handle passed is invalid.

2166 `RI_RET_CQ_INVALID_HANDLE` The CQ handle passed is invalid.

2167 `RI_RET_CQ_INUSE` One or more Work Queues are still associated with the
2168 CQ.

2169 `RI_RET_FAIL` Unknown/fatal error.

2170 CONSUMER USAGE

2171 None.

2172 IMPLEMENTATION NOTES

2173 None.

2174 SEE ALSO

2175 `ri_cq_create()`, `ri_cq_modify()`

2176

2177

ri_cq_set_event_handler()

2178 **NAME**

2179 ri_cq_set_event_handler – register a handler to be called when the next Work Completion
2180 becomes available on the specified RNIC

2181 **SYNOPSIS: KERNEL**

```

2182 ri_api_return_t ri_cq_set_event_handler (
2183     IN      ri_vp_handle_t      vp_hdl,
2184     IN OUT  ri_cq_event_handler_id_t *handler_id,
2185     IN      ri_cq_event_callback_t cq_callback,
2186     OUT     ri_err_data_t       *err_data
2187 );
2188
2189 /* Completion callback function pointer type */
2190 typedef void (*ri_cq_event_callback_t) (
2191     IN OUT  ri_os_data_t  vp_hdl_os_data,
2192     IN OUT  ri_os_data_t  cq_os_data
2193 );

```

2194 **DESCRIPTION**

2195 *vp_hdl* Handle to RNIC to register completion handler on. This must be the same as the handle used
2196 in *ri_rnic_open()*.

2197 *handler_id* Pointer to CQ event handler identifier.

2198 *cq_callback* A callback function to be called when the next Work Completion becomes available on any
2199 CQ that is associated with the specified/new *handler_id* and that has requested notification
2200 through *ri_cq_req_notification()*. See *ri_cq_event_callback_t* for a description of arguments
2201 passed to the callback function.

2202 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2203 RNICPI. The relation between the value received and later passed on is also not defined.
2204 Some interested OSVs and IHVs may use it for private communication.

2205 *ri_cq_set_event_handler()* registers a completion event handler. Only one completion event
2206 handler can be registered per CQ event handler ID on an RNIC. However, an RNIC may support
2207 more than one CQ event handler ID.

2208 If *handler_id* points to zero, a new CQ event handler ID is returned in *handler_id* and
2209 *cq_callback* is registered on the newly generated handler ID. Subsequent calls to
2210 *ri_cq_set_event_handler()* with a valid CQ event handler ID will overwrite the CQ event handler
2211 routine to be called. The CQ event handler ID is associated with a CQ at the time of
2212 *ri_cq_create()*. This call does not automatically request a notification on a completion event.
2213 The caller must call *ri_cq_req_notification()* in order to request notification. Specifying a NULL
2214 *cq_callback* for a given CQ event handler ID clears the completion handler associated with that
2215 handler ID. No callback function will be invoked upon completion on CQs associated with that
2216 handler ID even if *ri_cq_req_notification()* has been called on the CQ.

2217 *ri_cq_set_event_handler()* can be invoked only in the kernel. The user mode entry point of this
2218 call is simply a stub that returns RI_RET_NO_PERMISSION.

2219 The *vp_hdl_os_data* parameter that is passed to *cq_callback* is identical to *os_data* passed by the
2220 Consumer at the time of *ri_rnic_open()*. The *cq_os_data* parameter that is passed to *cq_callback*
2221 is identical to *os_data* passed by the Consumer at the time of *ri_cq_create()*.

2222 **RETURN VALUE**

2223	RI_RET_SUCCESS	The completion event handler is successfully registered on the specified RNIC.
2224		
2225	RI_RET_RESOURCE_SHORTAGE	Inadequate resources to complete the call. RNIC may have run out of handler IDs.
2226		
2227	RI_RET_VP_INVALID_HANDLE	The VP handle passed is invalid.
2228	RI_RET_INVALID_PARAMETER	<i>handler_id</i> is NULL.
2229	RI_RET_INVALID_HANDLER_ID	Invalid <i>handler_id</i> passed.
2230	RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
2231	RI_RET_FAIL	Unknown/fatal error.

2232 **CONSUMER USAGE**

2233 None.

2234 **IMPLEMENTATION NOTES**

2235 None.

2236 **SEE ALSO**

2237 *ri_cq_create()*, *ri_cq_req_notification()*, *ri_rnic_open()*

2238

ri_cq_req_notification()

2239 **NAME**

2240 *ri_cq_req_notification* – cause the completion handler to be called when the next Work
2241 Completion becomes available on the specified CQ

2242 **SYNOPSIS**

```
2243 ri_api_return_t ri_cq_req_notification (  
2244     IN    ri_rnic_handle_t      rnic,  
2245     IN    ri_cq_handle_t       cq,  
2246     IN    ri_cmplt_notify_type_t type,  
2247     OUT   ri_err_data_t        *err_data  
2248 );  
2249  
2250 typedef enum ri_cmplt_notify_type ri_cmplt_notify_type_t;  
2251 enum ri_cmplt_notify_type {  
2252     RI_CMPLT_NOTIFY_SOLICITED,  
2253     RI_CMPLT_NOTIFY_ANY  
2254 };
```

2255 **DESCRIPTION**

2256 *rnic* Handle to RNIC the specified CQ belongs to.

2257 *cq* Handle to CQ to enable notification on.

2258 *type* Completion notification type. See *ri_cmplt_notify_type_t*.

2259 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2260 RNICPI. The relation between the value received and later passed on is also not defined.
2261 Some interested OSVs and IHVs may use it for private communication.

2262 *ri_cq_req_notification()* requests the CQ event handler to be called when the next completion
2263 entry of the specified type is added to the specified CQ. The CQ event handler is registered using
2264 the *ri_cq_set_event_handler()* call. The handler is called at most once per
2265 *ri_cq_req_notification()* call for a particular CQ. Any CQ entries that existed before notification
2266 is enabled will not result in a call to the handler.

2267 Completion Events are one of two types: solicited or unsolicited. A solicited completion event
2268 occurs when an incoming RDMA Write with Immediate Data (valid only for IB) or Send
2269 message, with the solicited event header bit set causes a successful receive Work Completion to
2270 be added to a CQ, or when a Work Request that was posted with RI_WR_NOTIFY_FLAG
2271 enabled causes a successful completion to be added to a CQ, or when an unsuccessful Work
2272 Completion is added to a CQ. An unsolicited completion event occurs when any other successful
2273 receive Work Completion, or any successful send Work Completion, is added to a CQ.

2274 If an *ri_cq_req_notification()* is pending, subsequent calls to *ri_cq_req_notification()* for the
2275 same CQ prior to the completion event affect only when the notification occurs. A
2276 *ri_cq_req_notification()* for any completion event takes precedence over an
2277 *ri_cq_req_notification()* for a solicited event completion for the same CQ. If multiple calls to
2278 *ri_cq_req_notification()* have been made for the same CQ and at least one of the requests set the
2279 notification type to the next completion, the CQ event handler will be called when the next
2280 completion is added to that CQ. The CQ event handler will be called only once, even though

2281 multiple CQ notification requests were made prior to the completion event for the specified CQ.
2282 Once the CQ event handler is called, another completion event handler must be enabled before it
2283 gets called again. It is the responsibility of the caller to call *ri_cq_poll()* to retrieve a Work
2284 Completion.

2285 If *ri_cq_req_notification()* is invoked on a CQ that is not associated with an event handler (i.e.,
2286 the CQ was created with a zero *handler_id*), *RI_RET_CQ_NO_EVENT_HANDLER* is
2287 returned. If there is no completion callback function registered with the handler ID that is
2288 associated with the CQ, no notification will be generated, even if *ri_cq_req_notification()* is
2289 invoked.

2290 This call can be invoked from an interrupt context.

2291 RETURN VALUE

2292 *RI_RET_SUCCESS* The completion event handler will be called on the next
2293 completion event of the specified type.

2294 *RI_RET_RNIC_INVALID_HANDLE* The RNIC handle passed is invalid.

2295 *RI_RET_CQ_INVALID_HANDLE* The CQ handle passed is invalid.

2296 *RI_RET_INVALID_PARAMETER* The completion notification type passed is invalid.

2297 *RI_RET_CQ_NO_EVENT_HANDLER* The CQ was not associated with any event handler ID
2298 during CQ creation and hence cannot be enabled for
2299 notifications.

2300 *RI_RET_FAIL* Unknown/fatal error.

2301 CONSUMER USAGE

2302 None.

2303 IMPLEMENTATION NOTES

2304 None.

2305 SEE ALSO

2306 *ri_cq_create()*, *ri_cq_poll()*, *ri_cq_set_event_handler()*

2307

ri_cq_poll()

2308 **NAME**

2309 ri_cq_poll – poll the specified CQ for one or more new Work Completions

2310 **SYNOPSIS**

```
2311 ri_api_return_t ri_cq_poll (
2312     IN      ri_rnic_handle_t  rnic,
2313     IN      ri_cq_handle_t    cq,
2314     OUT     ri_wc_t           *wcs,
2315     IN OUT  uint_t            *num_wcs,
2316     OUT     ri_err_data_t     *err_data
2317 );
2318
2319 /* Performance Note: An attempt to align the WC types with IT API's
2320 event types has been done for the first five types. It is highly
2321 desirable that the definitions do not change with future versions of IT
2322 API */
2323
2324 typedef enum ri_wc_type ri_wc_type_t;
2325 enum ri_wc_type {
2326     RI_WC_TYPE_SEND = 0x1,
2327     RI_WC_TYPE_DATA_RECEIVED = 0x2,
2328     RI_WC_TYPE_RDMA_WRITE = 0x4,
2329     RI_WC_TYPE_RDMA_READ = 0x5,
2330     RI_WC_TYPE_BIND = 0x6,
2331     RI_WC_TYPE_FAST_REGISTER,
2332     RI_WC_TYPE_LOCAL_INVALIDATE,
2333     RI_WC_TYPE_RDMA_READ_LOCAL_INVALIDATE,
2334     RI_WC_TYPE_COMPARE_AND_SWAP,
2335     RI_WC_TYPE_FETCH_AND_ADD,
2336     RI_WC_TYPE_RDMA_WRITE_IMMEDIATE_RECEIVED
2337 };
2338
2339 typedef enum ri_wc_flags ri_wc_flags_t;
2340 enum ri_wc_flags {
2341     RI_WC_FLAG_SOLICITED_EVENT = 1 << 0,
2342     RI_WC_FLAG_LOCAL_SOLICITED = 1 << 1,
2343     RI_WC_FLAG_RTAG_INVALIDATE = 1 << 2,
2344     RI_WC_FLAG_IMMEDIATE_DATA = 1 << 3,
2345     /* Pass through if there is no local solicited notification */
2346     RI_WC_FLAG_CONSUMER2 = RI_WC_FLAG_LOCAL_SOLICITED,
2347     RI_WC_FLAG_CONSUMER1 = 1 << 29,
2348     RI_WC_FLAG_CONSUMER0 = 1 << 30
2349 };
2350
2351 /* Performance Note: An attempt to align the completion status with IT
2352 API's dto_status_t has been done for the first five types. It is highly
2353 desirable that the definitions do not change with future versions of IT
2354 API */
2355
2356 typedef enum ri_wc_status ri_wc_status_t;
2357 enum ri_wc_status {
```

```

2358     RI_WC_STATUS_SUCCESS = 0,
2359     RI_WC_STATUS_WR_FLUSHED_ERROR = 1,
2360     RI_WC_STATUS_LOCAL_QP_CATASTROPHIC = 2,
2361     RI_WC_STATUS_LOCAL_QP_OP_ERROR = 2,
2362     /* same as LOCAL_QP_CATASTROPHIC */
2363
2364     /* iWARP-specific status */
2365     RI_WC_STATUS_WQE_FORMAT_ERROR,
2366     RI_WC_STATUS_REMOTE_TERMINATION_ERROR,
2367     RI_WC_STATUS_INVALID_STAG_ERROR,
2368     RI_WC_STATUS_BASE_BOUNDS_VIOLATION_ERROR,
2369     RI_WC_STATUS_ACCESS_VIOLATION_ERROR,
2370     RI_WC_STATUS_INVALID_PD_ERROR,
2371     RI_WC_STATUS_ADDRESS_WRAP_ERROR,
2372     RI_WC_STATUS_STAG_INVALIDATE_ERROR,
2373     RI_WC_STATUS_ZERO_ORD_ERROR,
2374     RI_WC_STATUS_FMR_QP_NOT_PRIVILEGED_ERROR,
2375     RI_WC_STATUS_STAG_NOT_IN_INVALID_STATE_ERROR,
2376     RI_WC_STATUS_INVALID_FMR_PAGE_SIZE_ERROR,
2377     RI_WC_STATUS_INVALID_FMR_BLOCK_SIZE_ERROR,
2378     RI_WC_STATUS_INVALID_FMR_PBL_ENTRY_ERROR,
2379     RI_WC_STATUS_INVALID_FMR_FBO_ERROR,
2380     RI_WC_STATUS_INVALID_FMR_LENGTH_ERROR,
2381     RI_WC_STATUS_INVALID_FMR_PERMS_ERROR,
2382     RI_WC_STATUS_FMR_PBL_TOO_LONG_ERROR,
2383     RI_WC_STATUS_INVALID_FMR_VA_ERROR,
2384     RI_WC_STATUS_INVALID_BIND_MR_ERROR,
2385     RI_WC_STATUS_INVALID_BIND_MW_ERROR,
2386     RI_WC_STATUS_HUGE_PAYLOAD_ERROR,
2387
2388     /* IB-specific status */
2389     RI_WC_STATUS_LOCAL_LENGTH_ERROR,
2390     RI_WC_STATUS_LOCAL_PROTECTION_ERROR,
2391     RI_WC_STATUS_MEMORY_MGMT_ERROR,
2392     RI_WC_STATUS_BAD_RESPONSE_ERROR,
2393     RI_WC_STATUS_LOCAL_ACCESS_ERROR,
2394     RI_WC_STATUS_REMOTE_INVALID_REQ_ERROR,
2395     RI_WC_STATUS_REMOTE_ACCESS_ERROR,
2396     RI_WC_STATUS_REMOTE_OP_ERROR,
2397     RI_WC_STATUS_TRANSPORT_RETRY_EXCEEDED,
2398     RI_WC_STATUS_RNR_RETRY_EXCEEDED,
2399     RI_WC_STATUS_REMOTE_ABORTED_ERROR,
2400 };
2401
2402 /* Performance Note: An attempt to align RC (non-immediate data) Work
2403 Completion structure with IT API's dto_cmpl_event_t structure has been
2404 made. It is highly desirable that the definitions do not change with
2405 future versions of IT API */
2406
2407 typedef struct ri_rc_wc ri_rc_wc_t;
2408 struct ri_rc_wc {
2409     /* op_type should be the first field in the structure */
2410     ri_wc_type_t op_type;
2411

```

```

2412     /* Placeholder for it_evd_handle_t in IT API. OSVs may need to define
2413     flags and invalidated_rtag to take the size of it_evd_handle_t on their
2414     platforms. */
2415
2416         ri_os_data_t    cq_os_data;
2417         ri_wc_flags_t   flags;
2418         ri_rtag_t       invalidated_rtag;
2419
2420     /* Placeholder for it_ep_handle_t in IT API. OSVs may need to define
2421     ri_os_data_t identical to it_ep_handle_t on their platforms. */
2422
2423         ri_os_data_t    qp_os_data;
2424
2425         ri_wr_id_t      id;
2426         ri_wc_status_t  op_status;
2427         uint32_t        bytes_transferred; /* valid only for receive */
2428         uint32_t        immediate_data; /* valid only if immediate data is
2429                                         received along with incoming
2430                                         Send or RDMA Write (IB only) */
2431         ri_err_data_t   err_data;
2432     };
2433
2434     typedef struct ri_gen_wc ri_gen_wc_t;
2435     struct ri_gen_wc {
2436         /* op_type should be the first field in the structure */
2437         ri_wc_type_t    op_type;
2438
2439     /* Placeholder for it_evd_handle_t in IT API. OSVs may need to pad this
2440     appropriately to take the size of it_evd_handle_t on their platforms.*/
2441
2442         ri_os_data_t    cq_os_data;
2443         ri_wc_flags_t   flags;
2444         uint32_t        qpn;
2445
2446     /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
2447     ri_handle_t identical to it_ep_handle_t on their platforms. */
2448         ri_os_data_t    qp_os_data;
2449         ri_wr_id_t      id;
2450         ri_wc_status_t  op_status;
2451         uint32_t        bytes_transferred;
2452         uint32_t        immediate_data;
2453
2454         ri_lid_t        remote_lid;
2455         uint32_t        remote_qpn;
2456         uint8_t         remote_sl;
2457         uint8_t         local_lid_pathbits;
2458         uint16_t        ether_type;
2459         uint32_t        grh_present;
2460         uint32_t        pkey_index;
2461         uint32_t        freed_resource_count;
2462         ri_err_data_t   err_data;
2463     };
2464
2465     typedef union ri_wc ri_wc_t;

```

```

2466 union ri_wc {
2467     ri_wc_type_t  type_wc;
2468     ri_rc_wc_t    rc_wc;
2469     ri_gen_wc_t   ud_wc;
2470 };

```

2471 **DESCRIPTION**

2472 *rnrc* Handle to the RNIC the specified CQ belongs to.

2473 *cq* Handle to the CQ to poll for Work Completion.

2474 *wcs* Work Completion(s) returned containing information related to the completed Work
2475 Request from a QP the specified CQ is associated with.

2476 *num_wcs* Number of Work Completions desired by the Consumer on input. On output, *num_wcs*
2477 returns the actual number of Work Completions returned by the Verbs Provider.

2478 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2479 RNICPI. The relation between the value received and later passed on is also not defined.
2480 Some interested OSVs and IHVs may use it for private communication.

2481 *ri_cq_poll()* polls the specified CQ for one or more Work Completions. The Consumer specifies
2482 the number of Work Completions desired in *num_wcs* upon input. If a Work Request has
2483 completed, a corresponding Work Completion is returned in *wcs*. The actual number of Work
2484 Completions returned by the Verbs Provider will be less than or equal to the desired number of
2485 Work Completions and will be returned in *num_wcs*. If no Work Completions have been found,
2486 then the call returns RI_RET_CQ_EMPTY. The Consumer must not assume that all Work
2487 Completions have been reaped if the actual number of Work Completions returned is less than
2488 the desired number of Work Completions. A Verbs Provider may return any number of
2489 completions less than *num_wcs* even if there are more completions to be reaped. The Consumer
2490 is guaranteed that all Work Completions have been reaped only when the call returns
2491 RI_RET_CQ_EMPTY. The Consumer is advised to pass an array of *ri_wc_t* even if the
2492 Consumer knows that the CQ is only associated with RC QP; e.g., a Consumer of only iWARP
2493 devices should not pass an array of *ri_rc_wc_t* to *ri_cq_poll()*.

2494 This call can be invoked from an interrupt context.

Field Name	Description
<i>op_type</i>	Type of the Work Request that completed. Most types returned in completion have one-one correspondence with <i>ri_wr_t</i> . 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.
<i>flags</i>	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 <i>invalidated_rtag</i> field is valid suggesting the message received also invalidated an <i>Rtag</i> . For Sends or RDMA Write with immediate data (IB only), RI_WC_FLAG_IMMEDIATE_DATA indicates the presence of immediate data.

Field Name	Description
<i>invalidated_rtag</i>	The <i>Rtag</i> that was invalidated due to an incoming Send with Invalidate. This field is valid only if the <code>RI_WC_FLAG_RTAG_INVALIDATED</code> flag is set.
<i>qp_os_data</i>	The opaque <i>os_data</i> parameter passed by the Consumer at the time of <i>ri_qp_create()</i> of the QP. This Work Completion relates to one of the Work Requests posted on the QP or the SRQ associated with the QP.
<i>id</i>	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.
<i>op_status</i>	Status of the operation. This is always valid regardless of <i>op_type</i> .
<i>bytes_transferred</i>	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.
<i>immediate_data</i>	4-byte immediate data. This is valid only if <code>RI_WC_FLAG_IMMEDIATE_DATA</code> is enabled in flags.
<i>grh_present</i>	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.
<i>remote_lid</i> <i>remote_sl</i> <i>remote_qpn</i> <i>local_lid_pathbits</i> <i>ether_type</i>	Remote node address. Returned only for Datagram services.
<i>freed_resource_count</i>	This is always valid, regardless of the status of the operation.
<i>pkey_index</i>	P_Key index, for GSI only.
<i>qpn</i>	QP number of the local QP that caused the completion.
<i>err_data</i>	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.

2495 RETURN VALUE

2496 2497	<code>RI_RET_SUCCESS</code>	Successful completion. A new Work Completion is returned in <i>wc</i> .
2498	<code>RI_RET_RNIC_INVALID_HANDLE</code>	The RNIC handle passed is invalid.
2499	<code>RI_RET_CQ_INVALID_HANDLE</code>	The CQ handle passed is invalid.
2500	<code>RI_RET_CQ_EMPTY</code>	The CQ does not have any completions.
2501	<code>RI_RET_FAIL</code>	Unknown/fatal error.

2502 **CONSUMER USAGE**

2503 None.

2504 **IMPLEMENTATION NOTES**

2505 None.

2506 **SEE ALSO**

2507 *ri_cq_create()*, *ri_cq_req_notification()*, *ri_mr_reg()*, *ri_qp_create()*, *ri_qp_postrq()*,
2508 *ri_qp_postsq()*, *ri_rnic_open()*

2509

ri_qp_create()

2510

2511 **NAME**

2512 `ri_qp_create` – create a Queue Pair (QP) on the specified RNIC

2513 **SYNOPSIS**

```
2514 ri_api_return_t ri_qp_create(  
2515     IN     ri_rnic_handle_t  rnic,  
2516     IN OUT ri_qp_attr_t     *attrs,  
2517     OUT    ri_qp_handle_t    *qp,  
2518     IN OUT ri_vp_data_t     vp_data,  
2519     IN OUT ri_os_data_t     os_data,  
2520     OUT    ri_err_data_t     *err_data  
2521 );
```

2522 **DESCRIPTION**

2523 *rnica* Handle to RNIC.

2524 *attrs* Attributes of the QP. This contains the desired values when the caller makes this call. Upon
2525 successful completion of the call, this is updated with the number of Work Requests and
2526 scatter/gather elements indicating the actual values the RNIC supports. See *ri_qp_attr_t* for
2527 a description of QP attributes.

2528 *qp* Handle to the newly created QP.

2529 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
2530 information gets communicated from uVP to the “syscall” version of this function which in
2531 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
2532 when passed by uAL to uVP and uVPs must ignore this parameter.

2533 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
2534 information gets communicated from uAL to uVP which in turn gets passed unchanged to
2535 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
2536 get passed unchanged to appropriate calls back into the OS.

2537 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2538 RNICPI. The relation between the value received and later passed on is also not defined.
2539 Some interested OSVs and IHVs may use it for private communication.

2540 *ri_qp_create()* allows the caller to create a QP for the specified RNIC. A set of initial QP
2541 attributes must be specified by the caller. See *ri_qp_attr_t* for which attributes are required to be
2542 specified based on the transport. If any of the required initial attributes are illegal, an appropriate
2543 error is returned and a QP will not be created.

2544 Upon successful completion, *attrs* will indicate the actual values the RNIC assigned to the
2545 following fields of *ri_qp_attr_t*:

2546 *qp_id*
2547 *sq_size*
2548 *rq_size*
2549 *sq_max_sges*
2550 *sq_max_sges_rdmaw*
2551 *rq_max_sges*

2552 *max_rdma_read_outgoing*
 2553 *max_rdma_read_incoming*

2554 The *qp_id* field is set to the newly assigned QP ID or number. The values of the remaining fields
 2555 are guaranteed to be greater than or equal to the values requested. The QP is created in
 2556 RI_QP_STATE_IDLE state.

2557 **RETURN VALUE**

2558	RI_RET_SUCCESS	Successful completion. A valid QP handle is returned.
2559	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
2560	RI_RET_INVALID_PARAMETER	<i>attrs</i> passed is NULL.
2561	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
2562	RI_RET_CQ_INVALID_HANDLE	<i>scq</i> or <i>rcq</i> in <i>ri_qp_attr_t</i> is invalid.
2563	RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
2564		
2565	RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements requested exceeds RNIC capability.
2566		
2567	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
2568	RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
2569	RI_RET_QP_INVALID_SERVICE_TYPE	Invalid Transport Service Type for this QP.
2570	RI_RET_SRQ_LIMIT_OUT_OF_RANGE	RQ limit specified by <i>rq_hiwat</i> exceeds the maximum supported by the RNIC.
2571		
2572	RI_RET_SRQ_UNSUPPORTED	SRQs not supported in the RNIC.
2573	RI_RET_EXT_BMM_UNSUPPORTED	Base Memory management extensions not supported.
2574	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.
2575		
2576		
2577		
2578		
2579		
2580	RI_RET_INVALID_DDP_RDMAP_VERSION	Specified DDP and RDMAP version number not supported.
2581		
2582	RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
2583	RI_RET_FAIL	Unknown/fatal error.

2584 **CONSUMER USAGE**

2585 None.

2586 **IMPLEMENTATION NOTES**

2587 None.

2588 **SEE ALSO**

2589 [*ri_qp_query\(\)*](#), [*ri_qp_destroy\(\)*](#), [*ri_qp_modify\(\)*](#)

ri_qp_query()

2590

2591 **NAME**

2592 `ri_qp_query` – query the attributes of the specified QP

2593 **SYNOPSIS**

```
2594 ri_api_return_t ri_qp_query(  
2595     IN ri_rnic_handle_t rnic,  
2596     IN ri_qp_handle_t qp,  
2597     OUT ri_qp_attr_t *attrs,  
2598     OUT ri_err_data_t *err_data  
2599 );
```

2600 **DESCRIPTION**

2601 *rnic* Handle to an RNIC to which the QP belongs.

2602 *qp* Handle to the QP to query.

2603 *attrs* Attributes of the QP filled in as a result of the query operation.

2604 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2605 RNICPI. The relation between the value received and later passed on is also not defined.
2606 Some interested OSVs and IHVs may use it for private communication.

2607 `ri_qp_query()` returns the current values of the attributes of the specified QP. See [ri_qp_attr_t](#)
2608 for a description of the QP attributes. Either all the attributes are returned or none is returned in
2609 case of error.

2610 **RETURN VALUE**

2611 `RI_RET_SUCCESS` Successful completion. *attrs* contains the QP attributes.

2612 `RI_RET_INVALID_PARAMETER` *attrs* parameter passed is NULL.

2613 `RI_RET_RNIC_INVALID_HANDLE` The RNIC handle passed is invalid.

2614 `RI_RET_QP_INVALID_HANDLE` Invalid QP handle passed.

2615 `RI_RET_INVALID_PARAMETER` *attrs* is NULL or invalid.

2616 `RI_RET_FAIL` Unknown/fatal error.

2617 **CONSUMER USAGE**

2618 None.

2619 **IMPLEMENTATION NOTES**

2620 None.

2621 **SEE ALSO**

2622 [ri_qp_create\(\)](#), [ri_qp_destroy\(\)](#), [ri_qp_modify\(\)](#)

ri_qp_modify()

2623

2624 **NAME**

2625

ri_qp_modify – modify selected attributes of the specified QP

2626 **SYNOPSIS**

2627

2628

2629

2630

2631

2632

2633

2634

2635

2636

2637

2638

2639

2640

2641

2642

2643

2644

2645

2646

2647

2648

2649

2650

2651

2652

2653

2654

2655

2656

2657

2658

2659

2660

2661

2662

2663

2664

2665

2666

2667

2668

```
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_LLQ_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;
```

2669 **DESCRIPTION**

2670

rnic Handle to an RNIC to which the QP belongs.

2671 *qp* Handle to the QP whose attributes are to be modified.

2672 *qp_mask* Bitwise OR of *ri_qp_attr_mask_t* to select attributes to be modified.

2673 *attrs* Attributes structure which contains the requested values for the attributes corresponding to
2674 the mask bits chosen in *qp_mask* by the caller. Upon successful completion of the call, this
2675 contains the actual values of some of the attributes.

2676 *ri_qp_modify()* allows the caller to modify the attributes of the QP identified by the specified
2677 handle. The attributes to be modified are identified by *qp_mask*. This call will cause the QP to
2678 transition to the specified state if *RI_QP_ATTR_NEXT_STATE* is part of *qp_mask*. Only a
2679 subset of the QP attributes can be modified in each of the QP states. If any of the QP attributes to
2680 be modified are invalid for the current state of the QP or the requested state transition is illegal
2681 for the underlying transport, none of the QP attributes will be modified, the QP state will remain
2682 unchanged, and an appropriate error will be returned. Some QP attributes can be modified with
2683 outstanding Work Requests. Any outstanding Work Request on a Work Queue may not execute
2684 as expected if the QP modifiers are changed. For instance, if RDMA Reads, which were
2685 successfully posted, are outstanding when the QP is modified to no longer allow RDMA Reads,
2686 some outstanding in-flight RDMA Reads may complete while pending WRs may fail.¹

2687 Upon successful completion, *attrs* will indicate actual values the RNIC supports if any of the
2688 following fields of *ri_qp_attr_t* are being modified:

2689 *sq_size*
2690 *rq_size*
2691 *max_rdma_read_outgoing*
2692 *max_rdma_read_incoming*

2693 The values are guaranteed to be greater than or equal to the values requested. *sq_size* and *rq_size*
2694 can be modified only if the RNIC supports resizing of the Work Queues. If the QP is associated
2695 with an SRQ and the Consumer attempts to modify *rq_size* or if the QP is not associated with an
2696 SRQ and the Consumer attempts to modify either *rq_hiwat* or *rq_armed*, then
2697 *RI_RET_QP_ATTRIBUTE_CANT_CHANGE* is returned. The error is also returned when the
2698 RNIC does not support resizing of Work Queues and the Consumer attempts to modify either
2699 *sq_size* or *rq_size*.

2700 A value of zero for *rq_hiwat* in *ri_qp_modify()* means the Consumer wants to get an
2701 asynchronous notification for the first message since arming the Receive Queue of the QP for
2702 high water mark notification. If the QP's Receive Queue is not armed or has not been armed
2703 since last notification, the QP can, however, consume more than one receive Work Request from
2704 its SRQ. The zero value for *rq_hiwat* will not be interpreted similarly if this is specified in an
2705 *ri_qp_create()* call. It merely leaves the newly created QP disarmed for high water mark
2706 notification.

2707 If *RI_QP_ATTR_LLQ_STREAM_HANDLE* is set in *qp_mask*, the *llq_stream_handle* field of
2708 *attrs* indicates the LLP stream handle that is to be associated with the QP. This could point to an
2709 opaque data structure that is interpreted alike by the RNICPI Consumer and the RNIC driver.
2710 For example, it may point to an LLP context that is being handed off from the operating
2711 system's host stack to the RNIC, or it may be a handle to an already offloaded LLP connection
2712 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].

2713 Consumer and the driver are outside the scope of RNICPI. Keep-alive on the LLP could be
 2714 turned by passing vendor-dependent data in *llp_stream_handle* if RNIC supports it.

2715 If *RI_QP_ATTR_STREAM_MSG_BUFFER* is set in *qp_mask*, *stream_msg_buf* points to the
 2716 last streaming mode message that needs to be sent on the LLP before the QP goes into RDMA
 2717 mode and *stream_msg_buf_length* indicates the length of the message.

2718 The mask *RI_QP_ATTR_MPA_ATTR* can be used to modify the MPA-specific attributes to be
 2719 used for the IETF MPA startup framing protocol. *recv_marker_enabled* declares a receiver's
 2720 requirement for markers. Setting this field will cause the RNIC to look for and accept markers in
 2721 FPDUs received on the LLP connection. If the field is not set, the RNIC will assume markers
 2722 will not be present in the received FPDUs on the connection. Similarly, setting
 2723 *xmit_marker_enabled* declares that markers must be enabled in FPDUs that are transmitted by
 2724 the RNIC and not setting it will cause markers not to be inserted on the stream. *crc_enabled*
 2725 declares preferred CRC usage. Setting this field will cause the RNIC to generate and check
 2726 CRCs on the connection. Not setting the field would mean CRCs must not be checked and need
 2727 not be generated on the connection. Values of all the fields of *ri_mpa_attr_t* must be specified
 2728 by the Consumer if *RI_QP_ATTR_MPA_ATTR* is set in *qp_mask*. It is the responsibility of the
 2729 caller to modify the MPA attributes as desired before the QP goes into the RDMA mode. The
 2730 MPA attributes can be set either in a call prior to or in the same call as the one that modifies the
 2731 LLP stream handle.

2732 **RETURN VALUE**

2733	<i>RI_RET_SUCCESS</i>	Successful completion.
2734 2735	<i>RI_RET_RESOURCE_SHORTAGE</i>	Adequate resources not available to complete the call.
2736	<i>RI_RET_RNIC_INVALID_HANDLE</i>	The RNIC handle passed is invalid.
2737	<i>RI_RET_QP_INVALID_HANDLE</i>	Invalid QP handle passed.
2738 2739	<i>RI_RET_INVALID_PARAMETER</i>	The <i>attrs</i> parameter passed is NULL or <i>qp_mask</i> is invalid.
2740	<i>RI_RET_RNIC_INVALID_PKEY_INDEX</i>	PKey index is out of range.
2741 2742	<i>RI_RET_RNIC_INVALID_PKEY_ENTRY</i>	PKey index specifies invalid entry in PKey table.
2743	<i>RI_RET_QP_INVALID_STATE</i>	Invalid state transition.
2744	<i>RI_RET_RNIC_INVALID_MTU</i>	MTU of RNIC port exceeded.
2745	<i>RI_RET_RNIC_INVALID_PORT</i>	Invalid port specified.
2746	<i>RI_RET_MIGRATION_STATE_INVALID</i>	Invalid path migration state.
2747	<i>RI_RET_RNIC_INVALID_RNR_VALUE</i>	Invalid RNR NAK Timer Field Value.
2748	<i>RI_RET_QP_INVALID_SERVICE_TYPE</i>	Invalid Transport Service Type for this QP.
2749	<i>RI_RET_INVALID_LLP_HANDLE</i>	Invalid LLP Handle (iWarp only).

2750 2751	RI_RET_QP_ATTRIBUTE_CANT_CHANGE	Requested attribute modification is not valid for the state transition asked for.
2752	RI_RET_QP_RESIZE_UNSUPPORTED	RNIC does not support resizing QP.
2753 2754	RI_RET_RNIC_ATOMICS_NOT_SUPPORTED	Cannot enable incoming atomic operations as RNIC does not support atomic operations.
2755 2756	RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
2757 2758 2759	RI_RET_RNIC_IRD_ORD_EXCEEDED	<i>max_rdma_read_outgoing</i> or <i>max_rdma_read_outgoing</i> exceeds RNIC capability.
2760 2761	RI_RET_QP_CANNOT_SHRINK	More outstanding entries on WQ than size specified.
2762	RI_RET_WINDOWS_BOUND	Type 2A Memory Windows still bound to QP.
2763 2764	RI_RET_STILL_FLUSHING_WQES	QP is in error state and transition to Idle cannot happen as RI is busy flushing WQES.
2765 2766	RI_RET_SRQ_LIMIT_OUT_OF_RANGE	RQ limit specified by <i>rq_hiwat</i> exceeds the maximum supported by the RNIC.
2767 2768	RI_RET_NO_PERMISSION	Caller is not privileged to perform this operation.
2769 2770 2771 2772 2773 2774 2775	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.
2776 2777	RI_RET_INVALID_DDP_RDMAP_VERSION	Specified DDP and RDMAP version number not supported.
2778 2779 2780	RI_RET_LLP_BAD_STATE	Invalid state for modifying MPA attributes. LLP has already moved to RDMA mode and may be exchanging FPDUs.
2781	RI_RET_FAIL	Unknown/fatal error.

2782 **CONSUMER USAGE**

2783 iWARP Consumers that wish to move a QP to Error state are advised to follow it up with
2784 potentially repeated attempts to *ri_qp_modify()* to Idle if the error
2785 RI_RET_STILL_FLUSHING_WQES is returned. When the operation succeeds, they should
2786 reap all the expected completions for the QP. If the QP is associated with an SRQ, iWARP
2787 Consumers of RNICs that support generating an RI_EVENT_QP_LAST_WQE_REACHED
2788 asynchronous event may simply wait for this event in order to see *ri_qp_modify()* to Idle stop
2789 getting the RI_RET_STILL_FLUSHING_WQES error. If the QP is associated with an SRQ, IB

2790 Consumers should wait for an `RI_EVENT_QP_LAST_WQE_REACHED` asynchronous event.
2791 As a subsequent step, either IB or iWARP Consumers should perform *ri_poll* on the associated
2792 CQs until they return `RI_RET_CQ_EMPTY` or a full scan through the CQs is complete. At this
2793 point, Consumers may safely reuse the QP as though it is newly created or go ahead and call
2794 *ri_qp_destroy()* on the QP. Not taking the steps suggested here may lead to loss of or corruption
2795 of the completions.

2796 **IMPLEMENTATION NOTES**

2797 None.

2798 **SEE ALSO**

2799 *ri_qp_create()*, *ri_qp_destroy()*, *ri_qp_query()*

ri_qp_destroy()

2800

2801 NAME

2802 ri_qp_destroy – destroy the specified QP

2803 SYNOPSIS

```
2804 ri_api_return_t ri_qp_destroy(  
2805     IN  ri_rnic_handle_t  rnic,  
2806     IN  ri_qp_handle_t   qp,  
2807     OUT ri_err_data_t    *err_data  
2808 );
```

2809 DESCRIPTION

2810 *rnic* Handle to an RNIC to which the QP belongs.

2811 *qp* Handle to the QP that is to be destroyed.

2812 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2813 RNICPI. The relation between the value received and later passed on is also not defined.
2814 Some interested OSVs and IHVs may use it for private communication.

2815 This API allows the caller to destroy the specified QP and deallocate all the resources allocated
2816 to process Work Requests. A QP cannot be destroyed if it has a Type 2A Memory Window
2817 bound to it, in which case, this call will fail with RI_RET_WINDOWS_BOUND. If the QP is
2818 bound to only Type 2B Memory Windows, this call will not fail with this error. However, a QP
2819 instance is allowed to have Work Requests outstanding when an *ri_qp_destroy()* call is made.
2820 When a QP is destroyed, all outstanding Work Requests are implicitly cancelled. It is the
2821 responsibility of the caller to clean up resources associated with such Work Requests. No
2822 outstanding Work Requests will be completed after this call returns. Incoming operations
2823 destined for a QP that has been destroyed are discarded. Any completions sitting on the CQs
2824 associated with this QP may or may not be returned to the Consumer. The Consumer may also
2825 have the risk of overflowing the CQs if they have not successfully polled the CQs at least up to
2826 the number of expected completions. Upon successful return of this call, there will no async
2827 errors/events for this QP in progress and no further ones invoked.

2828 RETURN VALUE

2829	RI_RET_SUCCESS	Successful completion. The specified QP handle is
2830		successfully destroyed.
2831	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
2832	RI_RET_QP_INVALID_HANDLE	Invalid QP handle passed.
2833	RI_RET_WINDOWS_BOUND	Type 2A Memory Windows still bound to QP.
2834	RI_RET_MULTICAST_ATTACHED	The QP is still attached to one or more IB multicast
2835		groups.
2836	RI_RET_FAIL	Unknown/fatal error.

2837 **CONSUMER USAGE**

2838 None.

2839 **IMPLEMENTATION NOTES**

2840 None.

2841 **SEE ALSO**

2842 [*ri_qp_destroy\(\)*](#), [*ri_qp_modify\(\)*](#), [*ri_qp_query\(\)*](#)

ri_special_qp_get()

2843

2844 **NAME**

2845 `ri_special_qp_get` –return a handle to an IB special QP of specified type

2846 **SYNOPSIS**

```
2847 ri_api_return_t ri_special_qp_get(  
2848     IN     ri_rnic_handle_t  rnic,  
2849     IN     uint8_t          port,  
2850     IN     ri_sqp_type_t    qp_type,  
2851     IN OUT ri_qp_attr_t     *attrs,  
2852     OUT    ri_qp_handle_t   *qp,  
2853     IN OUT ri_vp_data_t    vp_data,  
2854     IN OUT ri_os_data_t    os_data,  
2855     OUT    ri_err_data_t   *err_data  
2856 );  
2857  
2858 typedef enum  
2859 {  
2860     RI_SQP_TYPE_SMI,  
2861     RI_SQP_TYPE_GSI,  
2862 } ri_sqp_type_t;
```

2863 **DESCRIPTION**

2864 *rnic* RNIC handle on which a special QP needs to be allocated.

2865 *port* RNIC port number.

2866 *qp_type* QP type requested. The allowed types are: SMI QP (QP0) and GSI QP (QP1).

2867 *attrs* Attributes of the QP filled in as a result of the query operation. Upon successful completion
2868 of the call, this becomes a copy of *attrs* with the number of Work Requests and
2869 scatter/gather elements indicating the actual values the RNIC is capable of.

2870 *qp* Handle to the newly created QP.

2871 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
2872 information gets communicated from uVP to the “syscall” version of this function which in
2873 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
2874 when passed by uAL to uVP and uVPs must ignore this parameter.

2875 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
2876 information gets communicated from uAL to uVP which in turn gets passed unchanged to
2877 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
2878 get passed unchanged to appropriate calls back into the OS.

2879 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
2880 RNICPI. The relation between the value received and later passed on is also not defined.
2881 Some interested OSVs and IHVs may use it for private communication.

2882 *ri_special_qp_get()* returns the handle for the specified QP type for the specified RNIC port. The
2883 special QP types include: SMI QP (QP0) and GSI QP (QP1). Handles associated with the SMI
2884 QP and the GSI QP will only be given out once for each QP per RNIC port. Subsequent

2885 invocations of this call, without an intervening *ri_qp_destroy()*, will return an error. SMI/GSI
 2886 QPs cannot share a completion queue with any non-SMI/GSI QP. An attempt to do so will result
 2887 in an RI_RET_CQ_HANDLE_INVALID error. This operation cannot be performed on a device
 2888 that does not support InfiniBand transport.

2889 A set of initial QP attributes must be specified by the caller. If any of the required initial
 2890 attributes are illegal, an appropriate error is returned and a QP will not be created. The following
 2891 fields of *ri_qp_attr_t* are required to be specified in *attrs* with this call:

- 2892 *scq*
- 2893 *rcq*
- 2894 *sq_size*
- 2895 *rq_size*
- 2896 *sq_max_sges*
- 2897 *rq_max_sges*
- 2898 *signal_completions*
- 2899 *pd*
- 2900 *trans_type*

2901 Upon successful completion, *attrs* will indicate the actual values the RNIC is capable of
 2902 supporting for the following fields of *ri_qp_attr_t*:

- 2903 *sq_size*
- 2904 *rq_size*
- 2905 *sq_max_sges*
- 2906 *rq_max_sges*

2907 These are guaranteed to be greater than or equal to the values requested.

2908 **RETURN VALUE**

2909	RI_RET_SUCCESS	Successful completion. A valid QP handle is returned.
2910		
2911	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
2912		
2913	RI_RET_INVALID_PARAMETER	The <i>attrs</i> or <i>qp</i> parameter passed is NULL.
2914	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
2915	RI_RET_QP_INVALID_SPECIAL_TYPE	Invalid Special QP type.
2916	RI_RET_QP_INUSE	QP already in use (applies only to SMI and GSI QPs).
2917		
2918	RI_RET_RNIC_INVALID_PORT	Invalid Port number specified.
2919	RI_RET_CQ_INVALID_HANDLE	<i>scq</i> or <i>rcq</i> in <i>ri_qp_attr_t</i> is invalid.
2920	RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
2921		
2922	RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements requested exceeds RNIC capability.
2923		

2924	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
2925	RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
2926	RI_RET_FAIL	Unknown/fatal error.
2927	CONSUMER USAGE	
2928	None.	
2929	IMPLEMENTATION NOTES	
2930	None.	
2931	SEE ALSO	
2932	None.	

2933 **ri_qp_postsq(), ri_qp_post_send(), ri_qp_post_rdma_read(),**
 2934 **ri_qp_post_rdma_write(), ri_qp_post_bind(), ri_qp_post_fmr(),**
 2935 **ri_qp_post_local_invalidate(), ri_qp_post_atomic(),**
 2936 **ri_qp_post_datagram()**

2937 **NAME**

2938 ri_qp_postsq – post operations on the Send Queue of the specified QP
 2939 ri_qp_post_send – post a Send operation on the Send Queue of the specified QP
 2940 ri_qp_post_rdma_read – post RDMA Read operation on the Send Queue of the specified QP
 2941 ri_qp_post_rdma_write – post RDMA Write operation on the Send Queue of the specified QP
 2942 ri_qp_post_bind – post Memory Window Bind operation on the Send Queue of the specified QP
 2943 ri_qp_post_fmr – post fast Memory (Region) registration operation on the Send Queue of the
 2944 specified QP
 2945 ri_qp_post_local_invalidate – post local (Memory Region) invalidation operation on the Send
 2946 Queue of the specified QP
 2947 ri_qp_post_atomic – post atomic operation on the Send Queue of the specified QP
 2948 ri_qp_post_datagram – post datagram operation on the Send Queue of the specified QP

2949 **SYNOPSIS**

```

2950 ri_api_return_t ri_qp_postsq (
2951     IN     ri_rnic_handle_t  rnic,
2952     IN     ri_qp_handle_t   qp,
2953     IN     ri_wr_t          *wr,
2954     IN OUT uint_t           *num_wrs,
2955     OUT    ri_err_data_t    *err_data
2956 );
2957
2958 /* See ri_wr_t for a description of the parameters in
2959 the following send methods */
2960 ri_api_return_t ri_qp_post_send(
2961     IN     ri_rnic_handle_t  rnic,
2962     IN     ri_qp_handle_t   qp,
2963     IN     ri_wr_type_t     type,
2964     IN     ri_wr_flags_t    flags,
2965     IN     ri_wr_id_t       id,
2966     IN     ri_data_seg_t    *sg_list,
2967     IN     ri_rtag_t        rtag,
2968     IN     uint_t           num_sgls,
2969     IN     uint32_t         imm_data,
2970     OUT    ri_err_data_t    *err_data
2971 );
2972
2973 ri_api_return_t ri_qp_post_rdma_read (
2974     IN     ri_rnic_handle_t  rnic,
2975     IN     ri_qp_handle_t   qp,
2976     IN     ri_wr_type_t     type,
  
```

```

2977         IN    ri_wr_flags_t    flags,
2978         IN    ri_wr_id_t      id,
2979         IN    ri_data_seg_t    *sg_list,
2980         IN    uint64_t         rdma_addr,
2981         IN    uint_t           num_sgls,
2982         IN    ri_rtag_t        rtag,
2983         OUT   ri_err_data_t    *err_data
2984     );
2985
2986     ri_api_return_t ri_qp_post_rdma_write(
2987         IN    ri_rnic_handle_t  rnic,
2988         IN    ri_qp_handle_t    qp,
2989         IN    ri_wr_type_t      type,
2990         IN    ri_wr_flags_t     flags,
2991         IN    ri_wr_id_t        id,
2992         IN    ri_data_seg_t     *sg_list,
2993         IN    uint64_t          rdma_addr,
2994         IN    ri_rtag_t         rtag,
2995         IN    uint_t            num_sgls,
2996         IN    uint32_t          imm_data,
2997         OUT   ri_err_data_t     *err_data
2998     );
2999
3000     ri_api_return_t ri_qp_post_bind (
3001         IN    ri_rnic_handle_t  rnic,
3002         IN    ri_qp_handle_t    qp,
3003         IN    ri_wr_type_t      type,
3004         IN    ri_wr_flags_t     flags,
3005         IN    ri_wr_id_t        id,
3006         IN    ri_mw_handle_t    mw_handle,
3007         IN    ri_rtag_t         rtag,
3008         IN    ri_mr_handle_t    mr_handle,
3009         IN    ri_ltag_t         ltag,
3010         IN    ri_addr_t         vaddr,
3011         IN    ri_length_t       length,
3012         IN    ri_mem_attr_t     mem_attr,
3013         OUT   ri_err_data_t     *err_data
3014     );
3015
3016     ri_api_return_t ri_qp_post_fmr (
3017         IN    ri_rnic_handle_t  rnic,
3018         IN    ri_qp_handle_t    qp,
3019         IN    ri_wr_type_t      type,
3020         IN    ri_wr_flags_t     flags,
3021         IN    ri_wr_id_t        id,
3022         IN    ri_mr_reg_phys_attr_t attr;
3023         IN    ri_ltag_t         ltag,
3024         IN    ri_rtag_t         rtag,
3025         IN    ri_mr_handle_t    mr_handle,
3026         OUT   ri_err_data_t     *err_data
3027     );
3028
3029     ri_api_return_t ri_qp_post_local_invalidate(
3030         IN    ri_rnic_handle_t  rnic,

```



```

3031         IN    ri_qp_handle_t    qp,
3032         IN    ri_wr_type_t      type,
3033         IN    ri_wr_flags_t     flags,
3034         IN    ri_wr_id_t        id,
3035         IN    ri_ltag_t         ltag,
3036         IN    ri_rtag_t         rtag,
3037         IN    ri_mrw_handle_t    mrw_handle,
3038         OUT   ri_err_data_t     *err_data
3039     );
3040
3041     ri_api_return_t ri_qp_post_atomic (
3042         IN    ri_rnic_handle_t    rnic,
3043         IN    ri_qp_handle_t      qp,
3044         IN    ri_wr_type_t        type,
3045         IN    ri_wr_flags_t       flags,
3046         IN    ri_wr_id_t          id,
3047         IN    ri_rtag_t           rtag,
3048         IN    uint64_t            atomic_operand1,
3049         IN    uint64_t            atomic_operand2,
3050         IN    ri_data_seg_t        atomic_result,
3051         OUT   ri_err_data_t       *err_data
3052     );
3053
3054     ri_api_return_t ri_qp_post_datagram (
3055         IN    ri_rnic_handle_t    rnic,
3056         IN    ri_qp_handle_t      qp,
3057         IN    ri_wr_type_t        type,
3058         IN    ri_wr_flags_t       flags,
3059         IN    ri_wr_id_t          id,
3060         IN    ri_data_seg_t        *sg_list;
3061         IN    uint_t              num_sgls;
3062         IN    uint32_t            imm_data;
3063         IN    ri_address_handle_t  remote_node_address,
3064         IN    uint32_t            dest_qp,
3065         IN    uint32_t            dest_qkey,
3066         IN    uint8_t             max_static_rate,
3067         OUT   ri_err_data_t       *err_data
3068     );

```

3069 DESCRIPTION

3070	<i>rnic</i>	Handle to the RNIC to send a message out on.
3071	<i>qp</i>	Handle to the <i>qp</i> on which to post the send.
3072	<i>wr</i>	Pointer to an array of Work Requests containing the information required to perform the send. The fields that must be specified in <i>ri_wr_t</i> are dependent on the operation type specified.
3073		
3074		
3075	<i>num_wrs</i>	Number of Work Requests submitted. Upon return, it will indicate the number of Work Requests processed.
3076		
3077	<i>err_data</i>	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.
3078		
3079		

3080 *ri_qp_postsq()* builds one or more WQEs from the information contained in the Work Requests
3081 and posts to the Send Queue of the specified QP. If the QP is enabled for selectable completion
3082 notification, the caller must specify whether a successful completion of this Work Request
3083 results in a completion entry on the CQ through RI_WR_SIGNED_FLAG in the *flags* field
3084 of *wr*. When the call returns, *wr* is in the scope of the caller and can no longer be accessed by the
3085 implementation. *wr* points to an array of *ri_wr_t* structures filled in by the Consumer. *wr* can
3086 contain a mixture of Work Request types. There is no difference, semantically, between posting
3087 the different Work Requests in the same or different *ri_qp_postsq()* calls.

3088 If the call returns successfully, all the Work Requests in *wr* will be posted to the Send Queue of
3089 the QP. If the call returns with an error, *num_wr* contains the number of Work Requests
3090 successfully posted to the Send Queue and the error corresponds to *num_wr*+1st Work Request in
3091 *wr*. Each successfully posted Work Request may generate a completion on the CQ associated
3092 with the Send Queue depending on *flags*. The Work Requests in *wr* are posted in the same order
3093 as they appear in the array. If RNIC supports enabling notification on a per-Work Request basis,
3094 setting RI_WR_LOCAL_SOLICITED_FLAG will cause the completion handler of the CQ to be
3095 invoked.

3096 If RI_WR_SIGNED_FLAG is not set, the implementation must ignore the setting of
3097 RI_WR_LOCAL_SOLICITED_FLAG and behave as though the flag is not set.

3098 In general, the Consumer must use the union structure *ri_wr_t*. Depending on the operation type,
3099 an appropriate sub-structure needs to be filled in. For example, the Consumer must not pass a
3100 pointer to an array of *ri_wr_send_t* typecast to a pointer to *ri_wr_t* unless *num_wr* is 1. See
3101 *ri_wr_t* for a description of the different types of Work Requests that can be posted using
3102 *ri_qp_postsq()*. Consumers should not assume all the contents of the Work Requests are
3103 validated until Work Completion is received for the Work Request on the associated CQ.

3104 If RI_WR_TYPE_FAST_REGISTER type is specified, the implementation may fail with
3105 RI_RET_NO_PERMISSION if called by a non-privileged Consumer.

3106 An attempt to post an RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE type Work
3107 Request will fail with RI_RET_WR_SG_INVALID_FORMAT if *num_sg* is greater than 1.

3108 More than one type of Work Request can be outstanding on the same QP at the same time. The
3109 ordering and fencing considerations for atomic operations are the same as for RDMA Read.

3110 The following allow Consumers to post one Work Request of a specific type at a time:

3111 *ri_qp_post_atomic*
3112 *ri_qp_post_bind*
3113 *ri_qp_post_datagram*
3114 *ri_qp_post_fmr*
3115 *ri_qp_post_local_invalidate*
3116 *ri_qp_post_rdma_read*
3117 *ri_qp_post_rdma_write*
3118 *ri_qp_post_send*

3119 All that is said in the context of *ri_qp_postsq()* is also valid for these calls unless it is about
3120 posting more than one Work Request in the same call.

3121 Where there is a choice between immediate or completion error types to signal an error
3122 condition for a posting function, the kVP implementation chooses which single error type is used
3123 to report the error condition. The uVP for the same RNIC may make a different choice than the

3124		kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as
3125		noted in Section 4.5.2.
3126		The calls described in this reference page can all be invoked from an interrupt context.
3127	RETURN VALUE	
3128	RI_RET_SUCCESS	Successful completion. The send is successfully
3129		posted on the specified QP.
3130	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
3131	RI_RET_QP_INVALID_HANDLE	The QP handle passed is invalid.
3132	RI_RET_QP_CAPACITY_EXCEEDED	Too many Work Requests posted.
3133	RI_RET_WR_INVALID_TYPE	Invalid operation type.
3134	RI_RET_QP_INVALID_STATE	Invalid QP state. This error may be returned on
3135		iWARP transport if the QP is not in RTS state
3136		and it cannot complete the Work Request with
3137		flushed status. On IB transport, this error is
3138		returned only when the QP is in the Idle, Init, or
3139		RTR states. It is not returned when the QP is in
3140		the Error state due to race conditions that could
3141		result in indeterminate behavior. Work Requests
3142		posted to the Send Queue while the QP is in the
3143		Error state, are completed with a flush error.
3144	RI_RET_WR_INVALID_FLAGS	Invalid flags specified.
3145	RI_RET_WR_SG_INVALID_FORMAT	Invalid Scatter/Gather list format. Note that for
3146		Raw IPv6 QPs, the first 40 bytes of the buffer(s)
3147		referred to by the Scatter/Gather list must
3148		contain the IPv6 header of the outgoing
3149		message.
3150	RI_ADDRESS_HANDLE_INVALID	Address handle specified in the Work Request is
3151		invalid.
3152	RI_RET_INVALID_WINDOW_TYPE	Consumer attempted to perform either: an
3153		Invalidate operation on a Type 1 Memory
3154		Window; or a zero-length Bind operation on a
3155		Type 2 Memory Window.
3156	RI_RET_MW_INVALID_HANDLE	The Memory Window handle passed is invalid.
3157	RI_RET_MR_INVALID_HANDLE	The MR handle passed is invalid.
3158	RI_MEM_INVALID_RTAG	RTag passed is invalid.
3159	RI_MEM_INVALID_LTAG	LTag passed is invalid.
3160	RI_RET_MEM_INVALID_ADDRESS	Invalid virtual address passed.
3161	RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.

3162	RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
3163	RI_RET_RNIC_SG_LIMIT_EXCEEDED	Invalid Scatter/Gather list length.
3164 3165	RI_RET_RNIC_ATOMICS_NOT_SUPPORTED	Cannot enable incoming atomic operations as RNIC does not support atomic operations.
3166 3167	RI_RET_BASE_MEM_EXTN_UNSUPPORTED	Base Memory management extensions not supported
3168 3169	RI_RET_EXT_LIF_UNSUPPORTED	RNIC does not support Local Invalidate Fencing.
3170 3171	RI_RET_NOT_IN_BLOCK_MODE	RNIC does not support Block Type Physical Buffers or was not opened in this mode.
3172	RI_RET_NOT_IN_PAGE_MODE	RNIC was not opened in Page mode.
3173 3174	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support Zero-Based Virtual Address (ZBVA).
3175 3176	RI_RET_NO_PERMISSION	Only privileged users are allowed to fast registrations.
3177	RI_RET_FAIL	Unknown/fatal error.

3178 **CONSUMER USAGE**

3179 None.

3180 **IMPLEMENTATION NOTES**

3181 As a VP must implement both the WR list model (*ri_qp_postsq()*) and the parameterized model
3182 (example: *ri_qp_post_send()*), the following is provided as a simple implementation aid.

3183 If the underlying implementation is the parameterized version, the Work Request list model can
3184 be implemented trivially as follows:

```

3185 ri_qp_postsq( ri_work_request_t *wr, ... ) {
3186
3187     For each work_request in wr
3188     /* Call the appropriate type-specific function passing the fields
3189        of wr as parameters. Clear RI_WR_COALESCE_FLAG for the last Work
3190        Request in the array. */
3191     ri_qp_post_xxxx();
3192 }

```

3193
3194 If the underlying implementation is a Work Request list, parameterized versions can be
3195 supported as follows:

```

3196 #define ASSIGN_WR_FIELD( wr, field_name ) \
3197     wr.field_name = field_name;
3198
3199 ri_qp_post_xxxx( rnic, qp, parameters, ... ) {
3200
3201     ri_work_request_xxxx_t wr;
3202     For each parameter
3203     ASSIGN_WR_FIELD(wr, parameter) = parameter;

```

```
3204         ri_qp_postsq(rnic, qp, &wr, 1, err_data );
3205     }
```

3206

3207

3208

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

3209 **SEE ALSO**

3210

3211

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*

3212 **ri_qp_postrq(), ri_qp_post_recv(), ri_srq_post(), ri_srq_post_recv()**

3213 **NAME**

3214 ri_qp_postrq – post list of receive Work Requests on the specified QP
3215 ri_qp_post_recv – post buffers for a receive operation on the specified QP
3216 ri_srq_post – post list of receive Work Requests on the specified SRQ
3217 ri_srq_post_recv – post buffers for a receive operation on the specified SRQ

3218 **SYNOPSIS**

```
3219 ri_api_return_t ri_qp_postrq (  
3220     IN     ri_rnic_handle_t  rnic,  
3221     IN     ri_qp_handle_t   qp_handle,  
3222     IN     ri_wr_recv_t     *wr,  
3223     IN OUT uint_t           *num_wrs,  
3224     OUT    ri_err_data_t    *err_data  
3225 );  
3226  
3227 ri_api_return_t ri_qp_post_recv (  
3228     IN     ri_rnic_handle_t  rnic,  
3229     IN     ri_qp_handle_t   qp_handle,  
3230     IN     ri_wr_flags_t    flags,  
3231     IN     ri_wr_id_t       id,  
3232     IN     ri_data_seg_t    *sg_list,  
3233     IN     uint_t           num_sgls,  
3234     OUT    ri_err_data_t    *err_data  
3235 );  
3236  
3237 ri_api_return_t ri_srq_post (  
3238     IN     ri_rnic_handle_t  rnic,  
3239     IN     ri_srq_handle_t  srq_handle,  
3240     IN     ri_wr_recv_t     *wr,  
3241     IN OUT uint_t           *num_wrs,  
3242     OUT    ri_err_data_t    *err_data  
3243 );  
3244  
3245 ri_api_return_t ri_srq_post_recv (  
3246     IN     ri_rnic_handle_t  rnic,  
3247     IN     ri_srq_handle_t  srq_handle,  
3248     IN     ri_wr_flags_t    flags,  
3249     IN     ri_wr_id_t       id,  
3250     IN     ri_data_seg_t    *sg_list,  
3251     IN     uint_t           num_sgls,  
3252     OUT    ri_err_data_t    *err_data  
3253 );  
3254  
3255 struct ri_wr_recv {  
3256     ri_wr_flags_t  flags;  
3257     ri_wr_id_t     id;  
3258     ri_data_seg_t *sg_list;  
3259     uint_t         num_sgls;  
3260 };
```

3261 typedef struct ri_wr_recv ri_wr_recv_t;

3262 **DESCRIPTION**

3263 *rnrc* Handle to the RNIC to receive data on.

3264 *qp_handle* Handle to the QP on which to post buffers for receive operation.

3265 *srq_handle* Handle to the SRQ on which to post buffers for receive operation.

3266 *wr* Pointer to an array of Work Requests that contain the information required to perform the
3267 buffer posting.

3268 *flags* Flags for the receive operation. See *ri_wr_flags_t*.

3269 *id* Work Request ID.

3270 *sg_list* See *ri_wr_t*.

3271 *num_sgls* Number of valid scatter/gather elements in *sg_list*.

3272 *num_wrs* Number of Work Requests submitted. Upon return, it will indicate the number of Work
3273 Requests processed.

3274 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3275 RNICPI. The relation between the value received and later passed on is also not defined.
3276 Some interested OSVs and IHVs may use it for private communication.

3277 *ri_qp_postrq()* builds one or more WQEs from the information contained in the Work Requests
3278 and posts to the Receive Queue of the specified QP. *ri_srq_post()* builds one or more WQEs
3279 from the information contained in the Work Requests and posts to the Receive Queue of the
3280 specified SRQ. When the call returns, the Work Request is in the scope of the caller and will no
3281 longer be accessed by the implementation. *wr* points to an array of *ri_wr_recv_t* structures filled
3282 in by the Consumer. Each Work Request can contain a gather list of receive buffers. Each
3283 incoming message on the Receive Queue of the QP satisfies only one of the Work Requests
3284 posted.

3285 If the call returns successfully, all the Work Requests in *wr* will be posted to the Receive Queue
3286 of the QP. If the call returns with error, *num_wr* contains the number of Work Requests
3287 successfully posted to the Receive Queue and the error corresponds to *num_wr*+1st Work
3288 Request in *wr*. Each successfully posted Work Request will generate a completion on the CQ
3289 associated with the Receive Queue. The Work Requests in *wr* are posted in the same order as
3290 they appear in the array. Flags other than `RI_WR_LOCAL_SOLICITED_FLAG` and
3291 `RI_WR_COALESCE_FLAG` are not valid in *ri_qp_postrq()/ri_srq_post()*. If RNIC supports
3292 enabling notification on a per-Work Request basis, setting
3293 `RI_WR_LOCAL_SOLICITED_FLAG` will cause the completion handler of the CQ to be
3294 invoked. Consumers should not assume all the contents of the Work Requests are validated until
3295 Work Completion is received for the Work Request on the associated CQ.

3296 *ri_qp_post_recv()* and *ri_srq_post_recv()* let Consumers post one Work Request at a time rather
3297 than a list of Work Requests as permitted by *ri_qp_postrq()* and *ri_srq_post()*. All that is said in
3298 the context of *ri_qp_postrq()/ri_srq_post()* is also valid for *ri_qp_post_recv()* and
3299 *ri_srq_post_recv()* unless it is about posting more than one Work Request in the same call.

3300 Note that for IB UD QPs, the first 40 bytes of the buffer(s) referred to by the Scatter/Gather list
3301 will contain the GRH of the incoming message. If no GRH is present, the contents of the first 40
3302 bytes of the buffer(s) will be undefined. The presence of the GRH will be indicated by a bit in
3303 the Work Completion.

3304 Where there is a choice between immediate or completion error types to signal an error
3305 condition for a posting function, the kVP implementation chooses which single error type is used
3306 to report the error condition. The uVP for the same RNIC may make a different choice than the
3307 kVP. Regardless of the error type chosen, each error condition MUST still be reported, except as
3308 noted in Section 4.5.2.

3309 The calls described in this reference page can all be invoked from an interrupt context

3310 RETURN VALUE

3311	RI_RET_SUCCESS	Successful completion. The list of Work
3312		Requests is successfully posted on the specified
3313		QP.
3314	RI_RET_QP_INVALID_HANDLE	The QP handle passed is invalid.
3315	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
3316	RI_RET_WR_SG_INVALID_FORMAT	Invalid Scatter/Gather list format.
3317	RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ handle.
3318	RI_RET_WR_INVALID_FLAGS	Invalid flags specified.
3319	RI_RET_QP_INVALID_STATE	Invalid QP state. Note that this error is returned
3320		only when the QP is in the Idle or Init state. It is
3321		not returned when the QP is in the Error state
3322		due to race conditions that could result in
3323		indeterminate behavior. Work Requests that are
3324		posted to the Receive Queue while the QP is in
3325		the Error state, are completed with a flush error.
3326	RI_RET_QP_CAPACITY_EXCEEDED	Too many Work Requests posted.
3327	RI_RET_RNIC_SG_LIMIT_EXCEEDED	Invalid Scatter/Gather list length.
3328	RI_RET_QP_TIED_TO_SRQ	Attempt to post to a QP that is tied to an SRQ.
3329	RI_RET_SRQ_UNSUPPORTED	SRQs not supported in the RNIC.
3330	RI_RET_FAIL	Unknown/fatal error.

3331 CONSUMER USAGE

3332 None.

3333 IMPLEMENTATION NOTES

3334 None.

3335 **SEE ALSO**

3336 *ri_cq_create(), ri_cq_poll(), ri_cq_req_notification(), ri_mr_reg(), ri_qp_create(),*
3337 *ri_rnic_open(), ri_qp_postsq()*

ri_srq_create()

3338

3339 NAME

3340 ri_srq_create – create a Shared Receive Queue object

3341 SYNOPSIS

```
3342 ri_api_return_t ri_srq_create(  
3343     IN     ri_rnic_handle_t rnic,  
3344     IN OUT ri_srq_attr_t   *srq_attrs,  
3345     OUT    ri_srq_handle_t *srq,  
3346     IN OUT ri_vp_data_t    vp_data,  
3347     IN OUT ri_os_data_t    os_data,  
3348     OUT    ri_err_data_t   *err_data  
3349 );
```

3350 DESCRIPTION

3351 *rnic* RNIC handle.

3352 *srq_attrs* Desired SRQ attributes on input. Actual SRQ attributes on output.

3353 *srq* Returned SRQ handle of the newly created SRQ.

3354 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3355 information gets communicated from uVP to the “syscall” version of this function which in
3356 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3357 when passed by uAL to uVP and uVPs must ignore this parameter.

3358 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3359 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3360 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3361 get passed unchanged to appropriate calls back into the OS.

3362 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3363 RNICPI. The relation between the value received and later passed back is also not defined.
3364 Some interested OSVs and IHVs may use it for private communication.

3365 *ri_srq_create()* creates a Shared Receive Queue for the specified RNIC. A set of initial SRQ
3366 attributes listed as IN parameters in *ri_srq_attr_t* must be specified by the Consumer. If the
3367 RNIC supports SRQ and any of the required initial attributes are illegal or missing, an error shall
3368 be returned and the SRQ shall not be created. On success, a handle to the newly created SRQ is
3369 returned.

3370 RETURN VALUE

3371 RI_RET_SUCCESS Successful completion.

3372 RI_RET_RESOURCE_SHORTAGE Adequate resources not available to complete
3373 the call.

3374 RI_RET_RNIC_INVALID_HANDLE Invalid RNIC handle.

3375 RI_RET_RNIC_INVALID_PARAMETER NULL supplied for mandatory OUT parameter.

3376	RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests exceeds RNIC's capability.
3377		
3378	RI_RET_RNIC_SG_LIMIT_EXCEEDED	Maximum number of scatter/gather elements exceeds RNIC's capability.
3379		
3380	RI_RET_PD_INVALID	Invalid PD handle.
3381	RI_RET_SRQ_LIMIT_OUT_OF_RANGE	<i>lowat</i> supplied in <i>srq_attrs</i> exceeds RNIC's capability.
3382		
3383	RI_RET_SRQ_UNSUPPORTED	SRQ not supported in the RNIC.
3384	RI_RET_FAIL	Unknown/fatal error.
3385	CONSUMER USAGE	
3386	None.	
3387	IMPLEMENTATION NOTES	
3388	None.	
3389	SEE ALSO	
3390	<i>ri_srq_destroy()</i> , <i>ri_srq_modify()</i> , <i>ri_srq_query()</i>	

ri_srq_query()

3391

3392 **NAME**

3393 `ri_srq_query` – query a Shared Receive Queue object

3394 **SYNOPSIS**

```
3395 ri_api_return_t ri_srq_query(  
3396     IN  ri_rnic_handle_t  rnic,  
3397     IN  ri_srq_handle_t   srq,  
3398     OUT ri_srq_attr_t     *srq_attrs,  
3399     OUT ri_err_data_t     *err_data  
3400 );
```

3401 **DESCRIPTION**

3402 `rnic` RNIC handle.

3403 `srq` SRQ handle of the queried SRQ.

3404 `srq_attrs` SRQ attributes of `srq`.

3405 `err_data` Data passed in all calls. How the recipient of the information uses it is outside the scope of
3406 RNICPI. The relation between the value received and later passed back is also not defined.
3407 Some interested OSVs and IHVs may use it for private communication.

3408 `ri_srq_query()` returns the attributes of the specified SRQ.

3409 **RETURN VALUE**

3410 `RI_RET_SUCCESS` Successful completion.

3411 `RI_RET_RNIC_INVALID_HANDLE` Invalid RNIC handle.

3412 `RI_RET_SRQ_INVALID_HANDLE` Invalid SRQ handle.

3413 `RI_RET_INVALID_PARAMETER` NULL supplied for mandatory OUT parameter.

3414 `RI_RET_SRQ_UNSUPPORTED` SRQ not supported in the RNIC.

3415 `RI_RET_SRQ_IN_ERROR` SRQ is in error state.

3416 `RI_RET_FAIL` Unknown/fatal error.

3417 **CONSUMER USAGE**

3418 None.

3419 **IMPLEMENTATION NOTES**

3420 None.

3421 **SEE ALSO**

3422 `ri_srq_create()`, `ri_srq_destroy()`, `ri_srq_modify()`

ri_srq_modify()

3423

3424 NAME

3425 ri_srq_modify – modify the attributes of a Shared Receive Queue

3426 SYNOPSIS

```
3427 ri_api_return_t ri_srq_modify(  
3428     IN  ri_rnic_handle_t    rnic,  
3429     IN  ri_srq_handle_t    srq,  
3430     IN  ri_srq_attr_mask_t srqmask,  
3431     OUT ri_srq_attr_t      *attrs,  
3432     OUT ri_err_data_t      *err_data  
3433 );  
3434  
3435 enum ri_srq_attr_mask {  
3436     RI_SRQ_ATTR_MAX_WRS = 1 << 0,  
3437     /* max_sgl attribute is NOT modifiable */  
3438     RI_SRQ_ATTR_LOWAT = 1 << 1,  
3439     /* pd is NOT modifiable *  
3440     * modifying lowat implicitly arms the alarm.*/  
3441 };  
3442 typedef enum ri_srq_attr_mask ri_srq_attr_mask_t;
```

3443 DESCRIPTION

3444 *rnic* RNIC handle.

3445 *srq* SRQ handle of the queried SRQ.

3446 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3447 RNICPI. The relation between the value received and later passed back is also not defined.
3448 Some interested OSVs and IHVs may use it for private communication.

3449 *ri_srq_modify()* modifies the attributes of an SRQ for the specified RNIC. If any of the modify
3450 attributes are invalid, none of the attributes shall be modified. The SRQ attributes that can be
3451 modified after the SRQ has been created are:

3452 *max_wrs* If resizing of the SRQ is supported by the RNIC.

3453 *lowat* If the SRQ Limit is greater than zero, then it shall be armed upon returning from this verb.

3454 The actual number of outstanding Work Requests supported on the Shared Receive Queue is
3455 returned in *max_wrs*. If an error is not returned, this is guaranteed to be greater than or equal to
3456 the number requested. (This may require the Consumer to increase the size of the CQ.)

3457 Note that there is no mechanism to modify an SRQ to disarm the limit alarm. However, the
3458 threshold can be changed even when the alarm is already enabled.

3459 RETURN VALUE

3460 RI_RET_SUCCESS Successful completion.

3461 RI_RET_RESOURCE_SHORTAGE Adequate resources not available to complete the
3462 call.

3463	RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
3464	RI_RET_SRQ_INVALID_HANDLE	Invalid SRQ Handle.
3465 3466	RI_RET_RNIC_INVALID_PARAMETER	NULL supplied for mandatory OUT parameter or <i>srqmask</i> is invalid.
3467 3468	RI_RET_RNIC_WR_LIMIT_EXCEEDED	Maximum number of Work Requests requested exceeds RNIC capability.
3469	RI_RET_SRQ_IN_ERROR	SRQ is in the Error State.
3470	RI_RET_SRQ_RESIZE_UNSUPPORTED	RNIC does not support resizing SRQ.
3471 3472	RI_RET_SRQ_LIMIT_OUT_OF_RANGE	SRQ Limit exceeds maximum number of Work Requests allowed on the SRQ.
3473 3474	RI_RET_SRQ_CANNOT_SHRINK	More outstanding entries on WQ than size specified.
3475	RI_RET_SRQ_UNSUPPORTED	RNIC does not support SRQ.
3476	RI_RET_FAIL	Unknown/fatal error.
3477	CONSUMER USAGE	
3478	None.	
3479	IMPLEMENTATION NOTES	
3480	None.	
3481	SEE ALSO	
3482	<i>ri_srq_create()</i> , <i>ri_srq_destroy()</i> , <i>ri_srq_query()</i>	

ri_srq_destroy()

3483

3484 **NAME**

3485 `ri_srq_destroy` – destroy a Shared Receive Queue object

3486 **SYNOPSIS**

```
3487 ri_api_return_t ri_srq_destroy(  
3488     IN ri_rnic_handle_t rnic,  
3489     IN ri_srq_handle_t srq,  
3490     OUT ri_err_data_t *err_data  
3491 );
```

3492 **DESCRIPTION**

3493 *rnic* RNIC handle.

3494 *srq* SRQ handle.

3495 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3496 RNICPI. The relation between the value received and later passed back is also not defined.
3497 Some interested OSVs and IHVs may use it for private communication.

3498 *ri_srq_destroy()* destroys the specified SRQ. An SRQ can be destroyed only after all the QPs
3499 associated with the SRQ have been destroyed. However, an SRQ can be destroyed even if there
3500 are Work Requests posted to the SRQ. It is the Consumer's responsibility to know what Work
3501 Requests were still in the SRQ when it was destroyed.

3502 The only way to obtain a list of Work Requests still in an SRQ is to have a list of all buffers ever
3503 posted to the SRQ and then remove those that have completed. The Work Requests in the SRQ
3504 will not be flushed to any CQ when the SRQ is destroyed.

3505 This limitation is not as severe as it might appear at first. The Consumer may know the entire set
3506 of buffers posted to the SRQ because they all come from a single resource that is about to be
3507 freed. Certainly that applies when SRQs are only destroyed just before the application exits. If
3508 the Consumer posts buffers from a variety of sources to an SRQ, and will attempt to continue
3509 operating after destroying that SRQ, they are strongly advised to maintain their own list of
3510 posted but not yet returned buffers.

3511 Upon successful return from this call, there will be no async errors/events in progress for this
3512 SRQ and no further async errors/events invoked for this SRQ.

3513 **RETURN VALUE**

3514 `RI_RET_SUCCESS` Successful completion.

3515 `RI_RET_RNIC_INVALID_HANDLE` Invalid RNIC handle.

3516 `RI_RET_SRQ_INVALID_HANDLE` Invalid SRQ handle.

3517 `RI_RET_SRQ_INUSE` SRQ still has QPs associated with it.

3518 `RI_RET_SRQ_UNSUPPORTED` RNIC does not support SRQ.

3519 `RI_RET_FAIL` Unknown/fatal error.

3520 **CONSUMER USAGE**

3521 None.

3522 **IMPLEMENTATION NOTES**

3523 None.

3524 **SEE ALSO**

3525 *ri_srq_create()*, *ri_srq_modify()*, *ri_srq_query()*

ri_ltag_alloc()

3526

3527 NAME

3528 `ri_ltag_alloc` – allocate a Memory Region Stag

3529 SYNOPSIS

```
3530 ri_api_return_t ri_ltag_alloc(  
3531     IN     ri_rnic_handle_t rnic,  
3532     IN OUT uint_t          *address_list_length,  
3533     IN     ri_pd_handle_t  pd,  
3534     IN     ri_mem_attr_t   attr,  
3535     OUT    ri_mr_handle_t  *mr,  
3536     IN OUT ri_ltag_t       *ltag,  
3537     OUT    ri_rtag_t       *rtag,  
3538     IN OUT ri_vp_data_t    vp_data,  
3539     IN OUT ri_os_data_t    os_data,  
3540     OUT    ri_err_data_t   *err_data  
3541 );
```

3542 DESCRIPTION

3543 *rnic* *rnic* handle. Must support fast register.

3544 *address_list_length*

3545 Non-null pointer to length of the address list. On input: minimum length desired. On output:
3546 if successful, the actual length allocated (which is greater than or equal to the length
3547 requested).

3548 *pd* Handle to the Protection Domain.

3549 *attr* Set of permissions ever to be enabled for the created Memory Region. The ability to later
3550 grant local permissions is assumed. The ability to later grant remote permissions is only
3551 granted if at least one remote permission is specified here. Flags not related to access
3552 permission are not relevant for this call.

3553 *mr* Non-null pointer to mem handle to be created.

3554 *ltag* Non-null pointer to output location for STag to be created for local use. The key portion of
3555 the LTag will be taken from the input value.

3556 *rtag* Pointer to output location for STag to be created for remote use. It must be non-NULL if
3557 remote permissions are requested. See [ri_rtag_t](#).

3558 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3559 information gets communicated from uVP to the “syscall” version of this function which in
3560 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3561 when passed by uAL to uVP and uVPs must ignore this parameter.

3562 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3563 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3564 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3565 get passed unchanged to appropriate calls back into the OS.

3566 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3567 RNICPI. The relation between the value received and later passed back is also not defined.
3568 Some interested OSVs and IHVs may use it for private communication.

3569 This API allocates a Memory Region STag without specifying the memory mapped. This will
3570 allocate or reserve physical buffer list resources for use by this Memory Region. This call either
3571 allocates the entire capacity requested or returns an error without modifying the state of the *rnic*.
3572 The actual size of the PBL resources allocated is returned in *address_list_length*. If the call is
3573 successful, the actual size allocated will be equal to or greater than the size requested. The
3574 allocated memory registration handle is released by invoking *ri_mr_dereg()*.

3575 RETURN VALUE

3576	RI_RET_SUCCESS	Successful completion.
3577	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the
3578		call.
3579	RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
3580	RI_RET_PD_INVALID	Invalid PD handle.
3581	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
3582	RI_RET_EXT_BMM_UNSUPPORTED	Base memory extensions not supported on the
3583		RNIC.
3584	RI_RET_NO_PERMISSION	Caller is not privileged to perform this
3585		operation.
3586	RI_RET_FAIL	Unknown/fatal error.

3587 CONSUMER USAGE

3588 The returned *ltag*, *rtag*, and memory handle may be interchangeable on many RNICs,
3589 particularly iWARP RNICs. To maintain portability the Consumer should take care to avoid
3590 confusing these returns, as that some RNICs will not detect using the “wrong” value in later calls
3591 while others will rely upon these values being distinct.

3592 IMPLEMENTATION NOTES

3593 The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will
3594 return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform
3595 physical registrations. This should not be the default.

3596 SEE ALSO

3597 *ri_mr_dereg()*, *ri_qp_postsq()*, *ri_ltag_t*, *ri_mem_attr_t*

ri_mr_reg()

3598

3599 **NAME**

3600 `ri_mr_reg` – register a Memory Region for direct access by the RNIC

3601 **SYNOPSIS**

```
3602 ri_api_return_t ri_mr_reg(  
3603     IN      ri_rnic_handle_t rnic,  
3604     IN const ri_mr_reg_attr_t *mr_reg_attr,  
3605     OUT     ri_mr_handle_t *mr,  
3606     IN OUT  ri_ltag_t *ltag,  
3607     OUT     ri_rtag_t *rtag,  
3608     IN OUT  ri_vp_data_t vp_data,  
3609     IN OUT  ri_os_data_t os_data,  
3610     OUT     ri_err_data_t *err_data  
3611 ) ;
```

3612 **DESCRIPTION**

3613 *rnic* Handle to the RNIC on which to register the region.

3614 *mr_reg_attr* Attributes of the Memory Region to be created.

3615 *mr* Handle to the Memory Region returned.

3616 *ltag* On output, *Ltag* for local access. See *ri_ltag_t*. On input, the “key” portion of the *ltag* when
3617 the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified (and the RNIC
3618 indicates support for “extended” key splitting through the
3619 RI_RNIC_EXT_INDEX_KEY_SPLIT flag). Otherwise, this argument is ignored on input.

3620 *rtag* *Rtag* for remote access. This should be non-NULL and points to a valid value only if
3621 RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is
3622 requested by the caller.

3623 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3624 information gets communicated from uVP to the “syscall” version of this function which in
3625 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3626 when passed by uAL to uVP and uVPs must ignore this parameter.

3627 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3628 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3629 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3630 get passed unchanged to appropriate calls back into the OS.

3631 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3632 RNICPI. The relation between the value received and later passed back is also not defined.
3633 Some interested OSVs and IHVs may use it for private communication.

3634 This API allows the caller to prepare a Memory Region for use by the specified RNIC. The
3635 memory registered is specified as a virtual address range. This call uses OS-specific methods to
3636 pin memory pages and create the virtual to physical translations that represent the Memory
3637 Region. The RI_MEM_FLAGS_USER_VA in *mr_reg_attr* (in the *mem_attr* field) indicates
3638 whether the virtual address supplied should be interpreted in a user process address space or in
3639 the kernel virtual address space.

3640 **RETURN VALUE**

3641	RI_RET_SUCCESS	Successful completion. A valid region handle is returned.
3642	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
3643	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
3644	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
3645	RI_RET_MEM_INVALID_ADDRESS	Invalid virtual address passed.
3646	RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.
3647	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
3648	RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
3649	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA
3650	RI_RET_FAIL	Unknown/fatal error.

3651 **CONSUMER USAGE**

3652 To allow later use of *ri_mr_rereg_phys()* or the fast register Work Request, set the
3653 RI_MEM_CONSUMER_OWNS_KEY *mem_attr*.

3654 Translation of virtual addresses to RNIC usable addresses must be performed using
3655 *ri_mem_map()*.

3656 **IMPLEMENTATION NOTES**

3657 None.

3658 **SEE ALSO**

3659 *ri_mem_map()*, *ri_mr_dereg()*, *ri_mr_reg_phys()*, *ri_mr_reg_shared()*, *ri_mr_rereg()*, *ri_ltag_t*,
3660 *ri_mr_reg_attr_t*

ri_mr_query()

3661

3662 NAME

3663 ri_mr_query --retrieve information about the specified Memory Region

3664 SYNOPSIS

```
3665 ri_api_return_t ri_mr_query(  
3666     IN ri_rnic_handle_t rnic,  
3667     IN ri_mr_handle_t mr,  
3668     OUT ri_mr_info_t *mr_info,  
3669     OUT ri_err_data_t *err_data  
3670 );  
3671  
3672 typedef struct ri_mr_info ri_mr_info_t;  
3673 struct ri_mr_info {  
3674     ri_mr_reg_attr_t attr;  
3675     ri_ltag_t ltag;  
3676     ri_rtag_t rtag;  
3677     ri_boolean_t is_shared_region;  
3678     ri_addr_t local_lb;  
3679     ri_addr_t local_ub;  
3680     ri_addr_t remote_lb;  
3681     ri_addr_t remote_ub;  
3682     ri_stag_state_t stag_state;  
3683 };
```

3684 DESCRIPTION

3685 *rnic* Handle to the RNIC to which the region belongs.

3686 *mr* Handle to the Memory Region to query.

3687 *mr_info* Holding buffer for Memory Region information.

3688 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3689 RNICPI. The relation between the value received and later passed back is also not defined.
3690 Some interested OSVs and IHVs may use it for private communication.

3691 *ri_mr_info_t* fields:

3692 *attr* Memory Region attributes also used for create/modify.

3693 *ltag* *Ltag* issued at the time of registration.

3694 *rtag* *Rtag* issued at the time of registration.

3695 *is_shared_region*
3696 Flag indicating whether the region is currently a shared region. If false, the Memory Region
3697 does not share page table resources with any other Memory Region. Invalidates and fast
3698 memory registers can only be performed on non-shared regions.

3699 *local_lb,local_ub*
3700 Actual local protection bounds enforced for the region.

3701 *remote_lb,remote_ub*
3702 Actual remote protection bounds enforced for the region.

3703 *stag_state* Stag state (valid/invalid).

3704 This API retrieves the keys, protection domain and bounds, and access control settings for the
3705 specified region. The remote protection bounds *remote_lb* and *remote_ub* contain valid values
3706 only if the region is enabled with `RI_MEM_ACCESS_REMOTE_WRITE` or
3707 `RI_MEM_ACCESS_REMOTE_READ` access.

3708 **RETURN VALUE**

3709	<code>RI_RET_SUCCESS</code>	Successful completion.
3710	<code>RI_RET_RNIC_INVALID_HANDLE</code>	The RNIC handle passed is invalid.
3711	<code>RI_RET_MR_INVALID_HANDLE</code>	The region handle passed is invalid.
3712	<code>RI_RET_INVALID_PARAMETER</code>	NULL pointer for mandatory OUT parameter.
3713	<code>RI_RET_FAIL</code>	Unknown/fatal error.

3714 **CONSUMER USAGE**

3715 InfiniBand 1.1 adapters may enable a larger region than requested. Consumers should use
3716 *ri_mr_query()* to determine the actual region enabled if adjacent memory structures must be
3717 protected from RNIC access. The boundaries returned are in the same address domain as those
3718 originally specified. The “actual” boundaries only differ from the input boundaries because of
3719 “rounding out” to page boundaries. They are not “actual” in the sense that they have been
3720 translated to physical addresses or from zero-based addresses.

3721 **IMPLEMENTATION NOTES**

3722 None.

3723 **SEE ALSO**

3724 *ri_mr_dereg()*, *ri_mr_reg()*, *ri_mr_rereg()*, *ri_ltag_t*, *ri_mem_attr_t*, *ri_stag_state_t*

3725

ri_mr_rereg()

3726

3727 NAME

3728 ri_mr_rereg –modify the attributes of an existing Memory Region

3729 SYNOPSIS

```
3730 ri_api_return_t ri_mr_rereg(  
3731     IN     ri_rnic_handle_t rnic,  
3732     IN     ri_mr_handle_t in_mr,  
3733     IN     ri_mr_rereg_t change_mask,  
3734     IN const ri_mr_reg_attr_t *mr_attr,  
3735     OUT    ri_mr_handle_t *out_mr,  
3736     IN OUT ri_ltag_t *ltag,  
3737     OUT    ri_rtag_t *rtag,  
3738     IN OUT ri_vp_data_t vp_data,  
3739     IN OUT ri_os_data_t os_data,  
3740     OUT    ri_err_data_t *err_data  
3741 );
```

3742 DESCRIPTION

3743 *rnic* Handle to the RNIC the region belongs to.

3744 *in_mr* Handle to the Memory Region originally registered.

3745 *change_mask* Set of Memory Region Attributes to be changed: address translation, protection domain,
3746 and/or access rights.

3747 *mr_attr* Attributes of the Memory Region to be created.

3748 *out_mr* Handle to the modified Memory Region. This may or may not be the same as mem passed.

3749 *ltag* On output, the new *Ltag* assigned for local access. See *ri_ltag_t*. On input, the “key” portion
3750 of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified (and
3751 the RNIC indicates support for “extended” key splitting through the
3752 RI_RNIC_EXT_INDEX_KEY_SPLIT flag). Otherwise, this argument is ignored on input.

3753 *rtag* New *Rtag* assigned for remote access. This should be non-NULL and points to a valid value
3754 only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ
3755 is requested by the caller.

3756 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3757 information gets communicated from uVP to the “syscall” version of this function which in
3758 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3759 when passed by uAL to uVP and uVPs must ignore this parameter.

3760 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3761 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3762 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3763 get passed unchanged to appropriate calls back into the OS.

3764 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3765 RNICPI. The relation between the value received and later passed back is also not defined.
3766 Some interested OSVs and IHVs may use it for private communication.

3767 This API modifies the attributes of an existing Memory Region. Any existing region owned by
 3768 the caller can be modified, regardless of whether *ri_mr_reg()*, *ri_mr_reg_shared()*, or
 3769 *ri_mr_rereg()* has been previously called on the region. The *ltag* returned is a handle for the
 3770 Memory Region suitable for use in Work Requests to describe locally accessible memory. When
 3771 *perms* requested contains `RI_MEM_ACCESS_REMOTE_WRITE` or
 3772 `RI_MEM_ACCESS_REMOTE_READ`, and the region is not already enabled with the access
 3773 control, a remote handle (*rtag*) for the Memory Region suitable for use by inbound RDMA
 3774 and/or atomic operations (*Rtag*) is returned. The `RI_MEM_FLAGS_USER_VA` in *mr_attr* (in
 3775 the *mem_attr* field) indicates whether the virtual address supplied should be interpreted in a user
 3776 process address space or in the kernel virtual address space.

3777 This call conceptually performs the functions *ri_mr_dereg()* followed by *ri_mr_reg()*. If the
 3778 caller attempts to reregister a Memory Region while the region still has any Memory Windows
 3779 bound to it, an `RI_RET_WINDOWS_BOUND` error is returned. If the original region is sharing
 3780 the translation tables in the RNIC with other region(s) through *ri_mr_reg_shared()*, this call will
 3781 not invalidate those registrations. No change will be made to the existing registration if the
 3782 following errors are returned:

3783 `RI_RET_WINDOWS_BOUND`
 3784 `RI_RET_RNIC_INVALID_HANDLE`
 3785 `RI_RET_MR_INVALID_HANDLE`
 3786 `RI_RET_INVALID_PARAMETER`

3787 If the call returns any other error, both “old” and “new” registrations are invalidated and any
 3788 associated resources are freed.

3789 If a remote agent is accessing a Memory Region while it is in the process of being reregistered,
 3790 the behavior of a deregistration operation followed by a separate registration operation would be
 3791 exhibited.

3792 To allow later use of *ri_mr_rereg_phys()* or the fast register Work Request, set the
 3793 `RI_MEM_CONSUMER_OWNS_KEY` *mem_attr*.

3794 RETURN VALUE

3795	<code>RI_RET_SUCCESS</code>	Successful completion. A valid region handle is returned in <i>out_mem</i> .
3796		
3797	<code>RI_RET_RESOURCE_SHORTAGE</code>	Adequate resources not available to reregister the Memory Region.
3798		
3799	<code>RI_RET_RNIC_INVALID_HANDLE</code>	The RNIC handle passed is invalid.
3800	<code>RI_RET_MR_INVALID_HANDLE</code>	The region handle passed is invalid.
3801	<code>RI_RET_INVALID_PARAMETER</code>	NULL supplied for mandatory OUT parameter.
3802	<code>RI_RET_MEM_INVALID_ADDRESS</code>	Invalid virtual address passed.
3803	<code>RI_RET_MEM_INVALID_LENGTH</code>	Invalid length of Memory Region passed.
3804	<code>RI_RET_PD_INVALID</code>	Invalid Protection Domain.
3805	<code>RI_RET_MEM_INVALID_ACCESS</code>	Invalid Access Control Specifier.

3806	RI_RET_WINDOWS_BOUND	Operation Denied; region still has bound
3807		Memory Window(s).
3808	RI_RET_EXT_BMM_UNSUPPORTED	Base Memory management extensions not
3809		supported.
3810	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
3811	RI_RET_FAIL	Unknown/fatal error.
3812	CONSUMER USAGE	
3813	None.	
3814	IMPLEMENTATION NOTES	
3815	Where possible, resources are expected to be reused instead of deallocated and reallocated.	
3816	Resources should not be available for other callers to claim between “deallocation” and	
3817	“reallocation”.	
3818	SEE ALSO	
3819	<i>ri_mr_dereg(), ri_mr_reg(), ri_mr_reg_shared(), ri_mr_rereg(), ri_mr_rereg_phys(), ri_ltag_t,</i>	
3820	<i>ri_mr_reg_attr_t</i>	

ri_mr_dereg()

3821

3822 NAME

3823 ri_mr_dereg – deregister the specified Memory Region

3824 SYNOPSIS

```
3825 ri_api_return_t ri_mr_dereg(  
3826     IN ri_rnic_handle_t rnic,  
3827     IN ri_mr_handle_t mr,  
3828     OUT ri_err_data_t *err_data  
3829 );
```

3830 DESCRIPTION

3831 *rnic* Handle to the RNIC on which to register the region.

3832 *mr* Handle to a registered Memory Region.

3833 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3834 RNICPI. The relation between the value received and later passed back is also not defined.
3835 Some interested OSVs and IHVs may use it for private communication.

3836 This API removes a Memory Region that was previously allocated by *ri_ltag_alloc()*,
3837 *ri_mr_reg_phys()*, or *ri_mr_reg()* from the RNIC translation table. The region is unpinned by
3838 using OSV services if it was previously pinned in the associated registration call. If the caller
3839 attempts to deregister a Memory Region while it still has any Memory Windows bound to it, an
3840 RI_RET_WINDOWS_BOUND error is returned.

3841 Work Requests or remote operation requests that reference the deregistered Memory Region will
3842 fail with a completion status of RI_COMPLT_STATUS_LOCAL_PROTECTION_ERROR. On
3843 successful return from this call, all access to local memory enabled by this Memory Region will
3844 have been completed or terminated.

3845 RETURN VALUE

3846 RI_RET_SUCCESS Successful completion. The specified Memory Region is
3847 successfully destroyed.

3848 RI_RET_RNIC_INVALID_HANDLE The RNIC handle passed is invalid.

3849 RI_RET_MR_INVALID_HANDLE The Memory Region handle passed is invalid.

3850 RI_RET_WINDOWS_BOUND Operation denied; region still has bound Memory
3851 Window(s).

3852 RI_RET_FAIL Unknown/fatal error.

3853 CONSUMER USAGE

3854 This function deallocates a Memory Region. Use *ri_mw_dealloc()* to deallocate a Memory
3855 Window.

3856 IMPLEMENTATION NOTES

3857 None.

3858 **SEE ALSO**
3859 *ri_ltag_alloc(), ri_mr_reg(), ri_mr_reg_phys(), ri_mr_reg_shared()*

ri_mr_reg_phys()

3860

3861 NAME

3862 `ri_mr_reg_phys` – register a set of physically addressed buffers as a single Memory Region for
3863 direct access by RNIC

3864 SYNOPSIS

```
3865 ri_api_return_t ri_mr_reg_phys(  
3866     IN     ri_rnic_handle_t    rnic,  
3867     IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,  
3868     OUT    ri_mr_handle_t      *mr,  
3869     IN OUT ri_ltag_t           *ltag,  
3870     OUT    ri_rtag_t           *rtag,  
3871     IN OUT ri_vp_data_t        vp_data,  
3872     IN OUT ri_os_data_t        os_data,  
3873     OUT    ri_err_data_t       *err_data  
3874 );
```

3875 DESCRIPTION

3876 *rnic* Handle to the RNIC on which to register the region.

3877 *mr_phys_attr*
3878 Attributes requested for the Memory Region to be created on input. Actual attributes on
3879 output.

3880 *mr* Handle to the Memory Region returned.

3881 *ltag* On output, *Ltag* for local access. See *ri_ltag_t*. On input, the “key” portion of the *ltag* when
3882 the `RI_MEM_FLAGS_CONSUMER_OWNS_KEY` flag is specified. Otherwise, this
3883 argument is ignored on input.

3884 *rtag* *Rtag* for remote access. This should be non-NULL and point to a valid value only if
3885 `RI_MEM_ACCESS_REMOTE_WRITE` or `RI_MEM_ACCESS_REMOTE_READ` is
3886 requested by the caller.

3887 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3888 information gets communicated from uVP to the “syscall” version of this function which in
3889 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3890 when passed by uAL to uVP and uVPs must ignore this parameter.

3891 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3892 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3893 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3894 get passed unchanged to appropriate calls back into the OS.

3895 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
3896 RNICPI. The relation between the value received and later passed back is also not defined.
3897 Some interested OSVs and IHVs may use it for private communication.

3898 This API prepares a Memory Region for use by an RNIC based upon a physical addresses. This
3899 call differs from *ri_mr_reg()* primarily in the type of memory range that is registered. The
3900 additional difference is that *ri_mr_reg_phys()* also outputs the actual Memory Region attributes,

3901 which are typically unchanged from a successful request. When the call completes, the region
3902 handle returned can be used just as a handle obtained from the *ri_mr_reg()* call.

3903 The virtual address (in *mr_phys_attr->common.va*) actually assigned may differ from that
3904 specified on input for InfiniBand adapters. The virtual address field is updated to reflect the
3905 actual value used in these cases.

3906 RETURN VALUE

3907	RI_RET_SUCCESS	Successful completion. A valid region handle is returned.
3908	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
3909	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
3910	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter..
3911	RI_RET_MEM_INVALID_ADDRESS	At least one of the physically addressed buffers is
3912		specified incorrectly.
3913	RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.
3914	RI_RET_MEM_INVALID_PBLE_LENGTH	Invalid PBLE length passed.
3915	RI_RET_MEM_INVALID_PBLE	Invalid PBLE entry supplied.
3916	RI_RET_MEM_INVALID_OFFSET	The offset passed is invalid.
3917	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
3918	RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
3919	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
3920	RI_RET_NO_PERMISSION	Caller does not have permission to register physical
3921		pages.
3922	RI_RET_FAIL	Unknown/fatal error.

3923 CONSUMER USAGE

3924 None.

3925 IMPLEMENTATION NOTES

3926 The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will
3927 return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform
3928 physical registrations. This should not be the default.

3929 SEE ALSO

3930 *ri_mr_dereg()*, *ri_mr_reg()*, *ri_mr_reg_shared()*, *ri_mr_rereg()*, *ri_mr_rereg_phys()*, *ri_ltag_t*,
3931 *ri_mr_reg_phys_attr_t*

ri_mr_rereg_phys()

3932

3933 NAME

3934 ri_mr_rereg_phys – modify the attributes of an existing non-shared Memory Region with the
3935 translation changes, if any, specified as a list of physically addressed buffers

3936 SYNOPSIS

```
3937 ri_api_return_t ri_mr_rereg_phys(  
3938     IN     ri_rnic_handle_t      rnic,  
3939     IN     ri_mr_handle_t        in_mr,  
3940     IN     ri_mr_rereg_t         change_mask,  
3941     IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,  
3942     OUT    ri_mr_handle_t        *out_mr,  
3943     IN OUT ri_ltag_t             *ltag,  
3944     IN OUT ri_rtag_t             *rtag,  
3945     IN OUT ri_vp_data_t          vp_data,  
3946     IN OUT ri_os_data_t          os_data,  
3947     OUT    ri_err_data_t         *err_data  
3948 );
```

3949 DESCRIPTION

3950 *rnic* Handle to the RNIC the region belongs to.

3951 *in_mem* Handle to the non-shared Memory Region originally registered.

3952 *change_mask*
3953 Set of Memory Region Attributes to be changed: address translation, protection domain,
3954 and/or access rights.

3955 *mr_phys_attr*
3956 Attributes desired for the Memory Region. This is an IN OUT parameter; some attributes
3957 may be modified by call completion to indicate actual resources/addresses granted/assigned.

3958 *out_mem* Handle to the modified Memory Region. This may or may not be same as *in_mem*.

3959 *ltag* On output, the new *Ltag* assigned for local access. See *ri_ltag_t*. On input, the “key” portion
3960 of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is specified.
3961 Otherwise, this argument is ignored on input.

3962 *rtag* New *Rtag* assigned for remote access. This should be non-NULL and point to a valid value
3963 only if RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ
3964 is requested by the caller in perms.

3965 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
3966 information gets communicated from uVP to the “syscall” version of this function which in
3967 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
3968 when passed by uAL to uVP and uVPs must ignore this parameter.

3969 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
3970 information gets communicated from uAL to uVP which in turn gets passed unchanged to
3971 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
3972 get passed unchanged to appropriate calls back into the OS.

3973 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
 3974 RNICPI. The relation between the value received and later passed back is also not defined.
 3975 Some interested OSVs and IHVs may use it for private communication.

3976 This API modifies the attributes of an existing Memory Region. Any non-shared existing region
 3977 owned by the caller can be modified, regardless of whether it was created using *ri_mr_reg()*, or
 3978 *ri_mr_reg_phys()* has been previously called on the region. A description of the Memory Region
 3979 suitable for use in Work Requests to describe locally accessible memory locations (*Ltag*) is
 3980 returned. When the access permissions requested contains
 3981 RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ, and the
 3982 region is not already enabled with the access control, a description of the Memory Region
 3983 suitable for use by inbound RDMA and/or atomic operations (*Rtag*) is returned.

3984 The virtual address (in *mr_phys_attr->common.va*) actually assigned may differ from that
 3985 specified on input for InfiniBand adapters. The virtual address field is updated to reflect the
 3986 actual value used in these cases.

3987 This call conceptually performs the functions *ri_mr_dereg()* followed by *ri_mr_reg_phys()*. If
 3988 the caller attempts to reregister a Memory Region while the region still has any Memory
 3989 Windows bound to it, an RI_RET_WINDOWS_BOUND error is returned. If the original region
 3990 is sharing the translation tables in the RNIC with other region(s) through *ri_mr_reg_shared()*,
 3991 this call will not invalidate those registrations. No change will be made to the existing
 3992 registration if the following errors are returned:

- 3993 RI_RET_WINDOWS_BOUND
- 3994 RI_RET_RNIC_INVALID_HANDLE
- 3995 RI_RET_MR_INVALID_HANDLE
- 3996 RI_RET_INVALID_PARAMETER

3997 If the call returns any other error, both “old” and “new” registrations are invalidated and any
 3998 associated resources are freed.

3999 If a remote agent is accessing a Memory Region while it is in the process of being reregistered,
 4000 the behavior of a deregistration operation followed by a separate registration operation would be
 4001 exhibited.

4002 RETURN VALUE

4003	RI_RET_SUCCESS	Successful completion. A valid region handle is returned in <i>outMem</i> .
4004		
4005	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to reregister the Memory Region.
4006		
4007	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4008	RI_RET_MR_INVALID_HANDLE	The Memory Region handle passed is invalid, or belongs to a shared Memory Region.
4009		
4010	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
4011	RI_RET_MEM_INVALID_ADDRESS	At least one of the physically addressed buffers is specified incorrectly.
4012		
4013	RI_RET_MEM_INVALID_LENGTH	Invalid length of Memory Region passed.

4014	RI_RET_MEM_INVALID_PBLE_LENGTH	Invalid PBL Length provided.
4015	RI_RET_MEM_INVALID_OFFSET	The offset passed is invalid.
4016	RI_RET_PD_INVALID_HANDLE	Invalid Protection Domain.
4017	RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
4018 4019	RI_RET_WINDOWS_BOUND	Operation Denied; region still has bound Memory Window(s).
4020	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
4021	RI_RET_EXT_BMM_UNSUPPORTED	RNIC does not support memory extensions.
4022 4023	RI_RET_NO_PERMISSION	Caller does not have permission for physical registers.
4024 4025	RI_RET_FAIL	Unknown/fatal error. <i>ri_get_err_detail</i> can be used to get detailed error message.

4026 **CONSUMER USAGE**
4027 None.

4028 **IMPLEMENTATION NOTES**

4029 Where possible, resources are expected to be reused instead of deallocated and reallocated.
4030 Resources should not be available for other callers to claim between “deallocation” and
4031 “reallocation”.

4032 The kAL, or its equivalent, is responsible for ensuring that user mode calls to this routine will
4033 return RI_RET_NO_PERMISSION unless the caller actually does have permission to perform
4034 physical registrations. This should not be the default.

4035 **SEE ALSO**

4036 *ri_mr_dereg()*, *ri_mr_reg()*, *ri_mr_reg_shared()*, *ri_mr_rereg()*, *ri_mr_rereg_phys()*, *ri_ltag_t*,
4037 *ri_mr_reg_phys_attr_t*

ri_mr_reg_shared()

4038

4039 NAME

4040 ri_mr_reg_shared – register a Memory Region that shares the physical pages with an existing
4041 Memory Region

4042 SYNOPSIS

```
4043 ri_api_return_t ri_mr_reg_shared(  
4044     IN     ri_rnic_handle_t  rnic,  
4045     IN     ri_mr_handle_t    in_mr,  
4046     IN OUT ri_mr_reg_attr_t  *mr_reg_attr,  
4047     OUT    ri_mr_handle_t    *out_mr,  
4048     IN OUT ri_ltag_t         *ltag,  
4049     OUT    ri_rtag_t         *rtag,  
4050     IN OUT ri_vp_data_t      vp_data,  
4051     IN OUT ri_os_data_t      os_data,  
4052     OUT    ri_err_data_t     *err_data  
4053 );
```

4054 DESCRIPTION

4055 *rnic* Handle to the RNIC the region belongs to.

4056 *in_mr* Handle to the Memory Region originally registered.

4057 *mr_reg_attr* Attributes desired for the new Memory Region. Length is ignored on input, and matches the
4058 original region's length. RI_MEM_FLAGS_NONCOHERENT is ignored on input, and is
4059 only meaningful on output.

4060 *out_mr* Handle to the cloned Memory Region.

4061 *ltag* On output, *Ltag* for local access for the new region. See *ri_ltag_t*. On input, the “key”
4062 portion of the *ltag* when the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag is
4063 specified. Otherwise, this argument is ignored on input.

4064 *rtag* *rtag* assigned for remote access. This should be non-NULL and point to a valid value only if
4065 RI_MEM_ACCESS_REMOTE_WRITE or RI_MEM_ACCESS_REMOTE_READ is
4066 requested by the caller in perms.

4067 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
4068 information gets communicated from uVP to the “syscall” version of this function which in
4069 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
4070 when passed by uAL to uVP and uVPs must ignore this parameter.

4071 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
4072 information gets communicated from uAL to uVP which in turn gets passed unchanged to
4073 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
4074 get passed unchanged to appropriate calls back into the OS.

4075 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4076 RNICPI. The relation between the value received and later passed back is also not defined.
4077 Some interested OSVs and IHVs may use it for private communication.

4078 Given an existing Memory Region, this API creates a new Memory Region associated with the
4079 same physical memory locations, with the intention that the new Memory Region share RNIC
4080 mapping resources to the extent possible.

4081 The virtual address, Protection Domain, and Access Rights specified for the new Memory
4082 Region need not be the same as those of the existing Memory Region. The size of the new
4083 region is, by definition, the same as that of the original.

4084 The caller supplies a requested virtual address to be associated with the first page in the new
4085 Memory Region, and the implementation returns the virtual address that is actually assigned. If
4086 the RI_MEM_FLAGS_SHARED_REG_IOVA flag is set in *mr_reg_attr* (in the *mem_attr* field),
4087 then the virtual address supplied is an I/O virtual address (as in physical memory registration).
4088 When interpreted as an IOVA, the virtual address (in *mr_reg_attr->va*) actually assigned may
4089 differ from that specified on input for InfiniBand adapters. The virtual address field is updated to
4090 reflect the actual value used in these cases.

4091 If the RI_MEM_FLAGS_SHARED_REG_IOVA flag is not set, the virtual address should be
4092 interpreted relative to the appropriate address space (as in virtual memory registration). The
4093 RI_MEM_FLAGS_USER_VA indicates whether the address space to use is a user process
4094 address space or the kernel address space.

4095 RETURN VALUE

4096	RI_RET_SUCCESS	Successful completion. A valid region handle is returned 4097 in <i>outMem</i> .
4098	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to create a shared 4099 Memory Region.
4100	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4101	RI_RET_MR_INVALID_HANDLE	The region handle passed (<i>mr</i>) is invalid.
4102	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
4103	RI_RET_PD_INVALID	Invalid Protection Domain.
4104	RI_RET_MEM_INVALID_ACCESS	Invalid Access Control Specifier.
4105	RI_RET_EXT_ZBVA_UNSUPPORTED	RNIC does not support ZBVA.
4106	RI_RET_FAIL	Unknown/fatal error. <i>ri_get_err_detail</i> can be used to get 4107 detailed error message.

4108 CONSUMER USAGE

4109 *ri_mr_reg_shared()* can be used to register the same memory with different access permissions
4110 and/or for a different Protection Domain without requiring redundant mapping resources.

4111 IMPLEMENTATION NOTES

4112 None.

4113 SEE ALSO

4114 *ri_mr_dereg()*, *ri_mr_reg()*, *ri_mr_reg_shared()*, *ri_mr_rereg()*, *ri_mr_rereg_phys()*, *ri_ltag_t*,
4115 *ri_mr_reg_attr_t*

ri_mw_alloc()

4116

4117 NAME

4118 ri_mw_alloc – allocate a Memory Window on the specified RNIC

4119 SYNOPSIS

```
4120 ri_api_return_t ri_mw_alloc(  
4121     IN     ri_rnic_handle_t  rnic,  
4122     IN     ri_pd_handle_t    pd,  
4123     IN     ri_mw_type_t      mw_type,  
4124     OUT    ri_mw_handle_t    *mw,  
4125     OUT    ri_rtag_t         *rtag,  
4126     IN OUT ri_vp_data_t      vp_data,  
4127     IN OUT ri_os_data_t      os_data,  
4128     OUT    ri_err_data_t     *err_data  
4129 );
```

4130 DESCRIPTION

4131 *rnic* Handle to the RNIC on which to allocate a window.

4132 *pd* Handle to Protection Domain the window is to be associated with.

4133 *mw_type* Type of the window – Type 1 (wide) or Type 2 (narrow).

4134 *mw* Handle to window returned by the call.

4135 *rtag* *Rtag* returned for remote access of the window.

4136 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
4137 information gets communicated from uVP to the “syscall” version of this function which in
4138 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
4139 when passed by uAL to uVP and uVPs must ignore this parameter.

4140 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
4141 information gets communicated from uAL to uVP which in turn gets passed unchanged to
4142 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
4143 get passed unchanged to appropriate calls back into the OS.

4144 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4145 RNICPI. The relation between the value received and later passed back is also not defined.
4146 Some interested OSVs and IHVs may use it for private communication.

4147 This API allocates a Memory Window which is associated with the specified protection domain.
4148 It is not inherently associated with any Memory Region at the time of allocation. The *rtag*
4149 returned is not yet in a valid state and must be bound using *ri_qp_postsq()*.

4150 RETURN VALUE

4151 RI_RET_SUCCESS Successful completion. A valid window handle is
4152 returned.

4153 RI_RET_RESOURCE_SHORTAGE Adequate resources not available to complete the call.

4154	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4155	RI_RET_INVALID_PARAMETER	NULL pointer for mandatory OUT parameter.
4156	RI_RET_PD_INVALID	Invalid Protection Domain.
4157	RI_RET_WINDOW_TYPE_UNSUPPORTED	The RNIC does not support the requested window type.
4158	RI_RET_FAIL	Unknown/fatal error.

4159 **CONSUMER USAGE**

4160 None.

4161 **IMPLEMENTATION NOTES**

4162 None.

4163 **SEE ALSO**

4164 *ri_mw_dealloc(), ri_mw_query(), ri_qp_postsq(), ri_ltag_t*

ri_mw_query()

4165
4166 **NAME**
4167 `ri_mw_query` – return the attributes associated with the specified Memory Window

4168 **SYNOPSIS**
4169

```
ri_api_return_t ri_mw_query(  
4170     IN    ri_rnic_handle_t  rnic,  
4171     IN    ri_mw_handle_t   mw,  
4172     OUT   ri_mw_type_t     *mw_type,  
4173     OUT   ri_stag_state_t  *stag_state,  
4174     OUT   ri_pd_handle_t   *pd,  
4175     OUT   ri_mem_attr_t    *attr,  
4176     OUT   ri_rtag_t        *rtag,  
4177     OUT   ri_err_data_t    *err_data  
4178 );
```

4179 DESCRIPTION

4180 *rnic* Handle to the RNIC the window belongs to.
4181 *mw* Handle to the window.
4182 *mw_type* Type 1 (wide) or Type 2 (narrow) window.
4183 *stag_state* Stag state – valid/invalid.
4184 *pd* Handle to Protection Domain the Memory Window is associated with.
4185 *attr* Attributes of the Memory Window, including permissions and ZBVA. Note that a Memory
4186 Window has no permissions before it is bound to a Memory Region.
4187 *rtag* *Rtag* for remote access of the window.
4188 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4189 RNICPI. The relation between the value received and later passed back is also not defined.
4190 Some interested OSVs and IHVs may use it for private communication.
4191 This API returns the window type, state, Protection Domain, permissions, and *Rtag* of the
4192 Memory Window specified. The caller may supply NULL pointers for any data that is not
4193 desired.
4194 **Note:** The window handle is passed in, not the current *Rtag*. This is contrary to what might
4195 be expected from the RDMAC verbs, but is required to also support InfiniBand verbs.

4196 RETURN VALUE

4197 `RI_RET_SUCCESS` Successful completion. *Rtag* and *pd* returned are valid.
4198 `RI_RET_RNIC_INVALID_HANDLE` The RNIC handle passed is invalid.
4199 `RI_RET_MW_INVALID_HANDLE` The Memory Window handle passed is invalid.
4200 `RI_RET_FAIL` Unknown/fatal error.

4201 **CONSUMER USAGE**

4202 None.

4203 **IMPLEMENTATION NOTES**

4204 None.

4205 **SEE ALSO**

4206 *ri_mw_alloc(), ri_mw_dealloc(), ri_ltag_t, ri_stag_state_t*

ri_mw_dealloc()

4207

4208 NAME

4209 `ri_mw_dealloc` – deallocate the specified Memory Window

4210 SYNOPSIS

```
4211 ri_api_return_t ri_mw_dealloc(  
4212     IN  ri_rnic_handle_t rnic,  
4213     IN  ri_mw_handle_t  mw,  
4214     OUT ri_err_data_t   *err_data  
4215 );
```

4216 DESCRIPTION

4217 *rnic* Handle of the RNIC the window belongs to.

4218 *mw* Handle to the window to be deallocated.

4219 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4220 RNICPI. The relation between the value received and later passed back is also not defined.
4221 Some interested OSVs and IHVs may use it for private communication.

4222 Any remote operation requests that are in process and actively using a Memory Window while
4223 and after it is deallocated will fail with `RI_COMPLT_STATUS_LOCAL_`
4224 `PROTECTION_ERROR`. Any API calls involving window handles but for `ri_mw_alloc()` will
4225 fail with `RI_RET_MW_INVALID_HANDLE` if a deallocated window handle is specified.

4226 RETURN VALUE

4227 `RI_RET_SUCCESS` Successful completion. The specified Memory Window
4228 is successfully deallocated.

4229 `RI_RET_RNIC_INVALID_HANDLE` The RNIC handle passed is invalid.

4230 `RI_RET_MW_INVALID_HANDLE` The window handle passed is invalid.

4231 `RI_RET_FAIL` Unknown/fatal error.

4232 CONSUMER USAGE

4233 None.

4234 IMPLEMENTATION NOTES

4235 None.

4236 SEE ALSO

4237 `ri_mw_alloc()`

4238

ri_mem_pin(), ri_mem_unpin()4239 **NAME**4240 `ri_mem_pin` – pin memory through the RNIC services4241 `ri_mem_unpin` – unpin memory through the RNIC services4242 **SYNOPSIS**

```

4243 ri_api_return_t ri_mem_pin (
4244     IN      ri_rnic_handle_t  rnic,
4245     IN      ri_vaddr_t       vaddr,
4246     IN      ri_length_t      len,
4247     IN OUT  ri_os_data_t      os_data,
4248     OUT     ri_err_data_t     *err_data
4249 );
4250
4251 ri_api_return_t ri_mem_unpin (
4252     IN      ri_rnic_handle_t  rnic,
4253     IN      ri_vaddr_t       vaddr,
4254     IN      ri_length_t      len,
4255     IN OUT  ri_os_data_t      os_data,
4256     OUT     ri_err_data_t     *err_data
4257 );

```

4258 **DESCRIPTION**4259 `rnic` Handle of RNIC, returned from `ri_rnic_get_hdl()`.4260 `vaddr` Virtual address.4261 `len` Length of the virtual address range in bytes.

4262 `os_data` Opaque data owned by OS modules. In general, IHV modules should not try to interpret it.
 4263 This information gets communicated from uAL to uVP which in turn gets passed unchanged
 4264 via “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged
 4265 to appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for
 4266 details.

4267 `err_data` Data passed in all calls. How the recipient of the information uses it is outside the scope of
 4268 RNICPI. The relation between the value received and later passed on is also not defined.
 4269 Some interested OSVs and IHVs may use it for private communication.

4270 These functions allow the Consumer to pin and unpin memory using RNIC services. The range
 4271 of memory is specified in terms of a virtual address and length in bytes using `vaddr` and `len`. The
 4272 address range of an `ri_mem_unpin()` call must be the same as the `ri_mem_pin()` call originally
 4273 used to pin the memory. Multiple `ri_mem_pin()` calls can be made on the same or overlapping
 4274 memory ranges with all pin reference counting issues properly handled. Some implementations
 4275 may also understand when a memory range need not be pinned and automatically optimize for
 4276 this case, so that a Consumer does not have to distinguish between different pinning
 4277 requirements of various memory resource pools.

4278 If `os_data` is being used to manage the state of the virtual memory range `ri_mem_pin()`, then the
 4279 same `os_data` value must be passed into `ri_mem_unpin()`. Further, it is required to use the same
 4280 `os_data` for subsequent `ri_mem_map()` and `ri_mem_unmap()` calls on this memory range. In

4281 some OSs, it is required to call this entry point instead of calling the device-independent memory
4282 pinning service. (Mostly because these OSs need to know which device will “own” this memory
4283 during subsequent operations.) For these OSs, the VP implementation of *ri_mem_pin()* (and
4284 *ri_mem_unpin()*) has to modify the *os_data* to include the proper device context information.
4285 The exact method of doing this is OS-dependent (since it is related to the format of *os_data* for
4286 that OS).

4287 RNICPI supports two different styles of pinning: in one, Consumers pin memory using an
4288 RNIC-independent service; in the other, pinning requires device-dependent context and uses the
4289 RNIC facility provided by *ri_mem_pin()* and *ri_mem_unpin()*.

4290 These functions apply only to privileged callers. If an attempt is made by a non-privileged caller
4291 to invoke these functions, *RI_RET_NO_PERMISSION* is returned.

4292 RETURN VALUE

4293	<i>RI_RET_SUCCESS</i>	Successful completion.
4294	<i>RI_RET_RESOURCE_SHORTAGE</i>	Not enough resources to complete the call.
4295	<i>RI_RET_NO_PERMISSION</i>	Caller is not privileged.
4296	<i>RI_RET_RNIC_INVALID_HANDLE</i>	The RNIC handle passed is invalid.
4297	<i>RI_RET_INVALID_PARAMETER</i>	One of the parameters other than the RNIC handle is 4298 invalid.
4299	<i>RI_RET_FAIL</i>	Unknown/fatal error.

4300 CONSUMER USAGE

4301 None.

4302 IMPLEMENTATION NOTES

4303 Two different styles of pinning are supported in RNICPI. One style uses a model independent of
4304 the RNIC. An OS using this first style will not have to call *ri_mem_pin()* and *ri_mem_unpin()*.
4305 The other style of pinning needs device-dependent context to efficiently pin. The calls
4306 *ri_mem_pin()* and *ri_mem_unpin()* aim at supporting this latter model. As a matter of
4307 compatibility, these entry points can be implemented for the first style as:

```
4308 ri_api_return_t ri_mem_pin (  
4309     ri_rnic_handle_t rnic,  
4310     ri_vaddr_t vaddr,  
4311     ri_length_t len,  
4312     ri_os_data_t os_data,  
4313     ri_err_data_t *err_data)  
4314 {  
4315     return ri_svc_mem_pin(vaddr, len, os_data, err_data);  
4316 }  
4317  
4318 ri_api_return_t ri_mem_unpin (  
4319     ri_rnic_handle_t rnic,  
4320     ri_vaddr_t vaddr,  
4321     ri_length_t len,  
4322     ri_os_data_t os_data,
```

```
4323         ri_err_data_t    *err_data)
4324     {
4325         return ri_svc_mem_unpin(vaddr, len, os_data, err_data);
4326     }
```

4327 **SEE ALSO**

4328 *ri_mem_map(), ri_mem_unmap(), ri_svc_mem_pin(), ri_svc_mem_unpin()*

4329

ri_mem_map(), ri_mem_unmap()

4330 NAME

4331 **ri_mem_map** – map a virtual memory range to a physical memory range directly accessible by
4332 an interface adapter

4333 **ri_mem_unmap** – remove a mapping of a virtual memory range to a physical memory range
4334 directly accessible by an interface adapter

4335 SYNOPSIS

```

4336 ri_api_return_t ri_mem_map (
4337     IN     ri_rnic_handle_t  rnic,
4338     IN     ri_vaddr_t       vaddr,
4339     IN     ri_length_t      len,
4340     OUT    ri_phys_buf_t    *io,
4341     IN OUT ri_os_data_t     os_data,
4342     OUT    ri_err_data_t    *err_data
4343 );
4344
4345 ri_api_return_t ri_mem_unmap (
4346     IN     ri_rnic_handle_t  rnic,
4347     IN     ri_phys_buf_t     *io,
4348     IN OUT ri_os_data_t     os_data,
4349     OUT    ri_err_data_t    *err_data
4350 );

```

4351 DESCRIPTION

4352 *rnic* Handle to the RNIC on which to map the memory range.

4353 *vaddr* Virtual address.

4354 *len* Length of the virtual address range in bytes.

4355 *io* Address and length of the physical memory range.

4356 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
4357 information gets communicated from uAL to uVP which in turn gets passed unchanged to
4358 *ri_syscall*. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
4359 appropriate calls back into the OS.

4360 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4361 RNICPI. The relation between the value received and later passed back is also not defined.
4362 Some interested OSVs and IHVs may use it for private communication.

4363 *ri_mem_map()* attempts to map a virtual memory range defined by *vaddr* and *len* to a physical
4364 memory range *io* addressable by the specified interface adapter. The size of the memory
4365 successfully mapped is returned in the *phys_len* field of *io* and it is safe to assume that the entire
4366 address range mapped is contiguous for the interface adapter. A Consumer is not guaranteed to
4367 get the entire address range mapped by the implementation. Before calling *ri_mem_map()*, a
4368 Consumer must pin the virtual address range either by using *ri_mem_pin()* or some other
4369 mechanism outside the scope of this specification.

4370 *ri_mem_unmap()* unmaps the mapping of a virtual memory range to a physical memory range *io*
4371 addressable by the specified interface adapter. *os_data* is an opaque parameter used exclusively
4372 by the OSV code to manage the state of the mapping represented by *io*. The opaque *os_data*
4373 used to unmap *io* with *ri_mem_unmap()* must be the same opaque used when *io* was previously
4374 mapped with *ri_mem_map()*.

4375 *ri_mem_map()* and *ri_mem_unmap()* can be invoked from an interrupt context.

4376 **RETURN VALUE**

4377	RI_RET_SUCCESS	Successful completion.
4378	RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.
4379	RI_RET_NO_PERMISSION	Caller is not privileged.
4380	RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
4381	RI_RET_INVALID_PARAMETER	One of the parameters other than the RNIC handle is 4382 invalid.
4383	RI_RET_FAIL	Unknown/fatal error.

4384 **CONSUMER USAGE**

4385 If the call does not entirely map the virtual address range, the Consumer can call again to obtain
4386 a mapping of the remaining part. In this case, the caller should offset *vaddr* by the amount of
4387 bytes that was successfully mapped in the last call to *ri_mem_map()* and subtract *len* by the same
4388 amount.

4389 **IMPLEMENTATION NOTES**

4390 kAL and uAL can use *os_data* to pass state information that characterizes a particular virtual
4391 memory range being mapped. Examples of state information are the process context or the
4392 device.

4393 **SEE ALSO**

4394 *ri_mem_map()*, *ri_mem_unmap()*, *ri_svc_mem_pin()*, *ri_svc_mem_unpin()*

ri_mem_sync_rread()

4395

4396 NAME

4397 `ri_mem_sync_rread` – make memory changes visible to an incoming RDMA read operation

4398 SYNOPSIS

```
4399 ri_api_return_t ri_mem_sync_rread(  
4400     IN   ri_rnic_handle_t rnic,  
4401     IN   ri_data_seg_t   *local_seg_list,  
4402     IN   uint_t          num_seg,  
4403     OUT  ri_err_data_t   *err_data  
4404 );
```

4405 DESCRIPTION

4406 *rnic* Handle to the RNIC where the memory segments are registered.

4407 *local_seg_list*
4408 Pointer to array of memory segments.

4409 *num_seg* Number of segments in the array.

4410 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4411 RNICPI. The relation between the value received and later passed on is also not defined.
4412 Some interested OSVs and IHVs may use it for private communication.

4413 The `ri_mem_sync_rread()` function is used with non-coherent Memory Regions to make
4414 memory updates visible to subsequent incoming RDMA read operations. The `ri_sync_rread`
4415 function is needed if and only if a Memory Region was created in non-coherent mode using the
4416 `RI_MEM_FLAGS_NONCOHERENT` flag. After modifications are made to a non-coherent
4417 Memory Region, `ri_mem_sync_rread()` must be called to ensure the updates will be visible
4418 before incoming RDMA read operations on that memory start. It is permissible to batch
4419 synchronizations for multiple subsequent RDMA reads in a single operation by passing a
4420 *local_seg_list* containing all the modified memory ranges.

4421 If the `ri_mem_sync_rread()` function is not called as described above, the results of an RDMA
4422 read operation on a non-coherent Memory Region are undefined.

4423 The *local_seg_list* argument contains a list of memory segments of length *num_seg*. Only
4424 segments with *ltags* for non-coherent regions will be synchronized. Segments of coherent
4425 Memory Regions (i.e., created without the `RI_MEM_FLAGS_NONCOHERENT` flag) in the list
4426 are unaffected.

4427 This call is a no-op and always returns successfully if the implementation does not support non-
4428 coherent mode or if none of the memory segments reference non-coherent Memory Regions.

4429 The kernel version of this function is suitable for interrupt usage.

4430 The definitions for *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

4431 RETURN VALUE

4432 `RI_RET_SUCCESS` Successful completion.

4433 `RI_RET_RNIC_INVALID_HANDLE` RNIC handle passed is invalid.

4434	RI_RET_INVALID_PARAMETER	<i>local_seg_list</i> is NULL, when <i>num_seg</i> is non-zero.
4435	RI_RET_MEM_INVALID_LTAG	Invalid LTAG in memory segment list.
4436	RI_RET_MEM_INVALID_ADDRESS	Invalid address in memory segment list.
4437	RI_RET_MEM_INVALID_LENGTH	Invalid length in memory segment list.
4438	RI_RET_FAIL	Unknown/fatal error.

4439 **CONSUMER USAGE**

4440 This function is described in terms of RDMA usage for non-coherent memory. While this is the
4441 common usage, it is possible to use it to support other operations using non-coherent memory.

4442 **IMPLEMENTATION NOTES**

4443 None.

4444 **SEE ALSO**

4445 [*ri_mem_sync_rwrite\(\)*](#)

ri_mem_sync_rwrite()

4446

4447 NAME

4448 `ri_mem_sync_rwrite` – make results of incoming RDMA write operation visible to Consumers

4449 SYNOPSIS

```
4450 ri_api_return_t ri_mem_sync_rwrite(  
4451     IN    ri_rnic_handle_t  rnic,  
4452     IN    ri_data_seg_t    *local_seg_list,  
4453     IN    uint_t           num_seg,  
4454     OUT   ri_err_data_t    *err_data  
4455 );
```

4456 DESCRIPTION

4457 *rnic* Handle to the RNIC where the memory segments are registered.

4458 *local_seg_list*
4459 Pointer to array of memory segments.

4460 *num_seg* Number of segments in the array.

4461 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4462 RNICPI. The relation between the value received and later passed on is also not defined.
4463 Some interested OSVs and IHVs may use it for private communication.

4464 The `ri_mem_sync_rwrite()` function is used with non-coherent Memory Regions to make the
4465 results of an incoming RDMA write operation visible to Consumers. The `ri_mem_sync_rwrite()`
4466 function is needed if and only if a Memory Region was created in non-coherent mode using the
4467 `RI_MEM_FLAGS_NONCOHERENT` flag. After an RDMA write operation has completed on a
4468 non-coherent Memory Region, `ri_mem_sync_rwrite()` must be called to ensure the updates will
4469 be visible in memory before Consumers attempt to read the affected memory. It is permissible to
4470 batch synchronizations for multiple RDMA writes in a single operation by passing a
4471 *local_seg_list* containing all the modified memory ranges.

4472 If the `ri_mem_sync_rwrite()` function is not called as described above, the results of an RDMA
4473 write operation on a non-coherent Memory Region are undefined.

4474 The *local_seg_list* argument contains a list of memory segments of length *num_seg*. Only
4475 segments with *ltags* for non-coherent regions will be synchronized. Segments of coherent
4476 Memory Regions (i.e., created without the `RI_MEM_FLAGS_NONCOHERENT` flag) in the list
4477 are unaffected.

4478 This call is a no-op and always returns successfully if the implementation does not support non-
4479 coherent mode or if none of the memory segments reference non-coherent Memory Regions.

4480 The kernel version of this function is suitable for interrupt usage.

4481 The definitions for *ri_rnic_handle_t* and *ri_err_data_t* are OS-dependent.

4482 RETURN VALUE

4483 `RI_RET_SUCCESS` Successful completion.

4484 `RI_RET_RNIC_INVALID_HANDLE` RNIC handle passed is invalid.

4485	RI_RET_INVALID_PARAMETER	<i>local_seg_list</i> is NULL, when <i>num_seg</i> is non-zero.
4486	RI_RET_MEM_INVALID_LTAG	Invalid LTAG in memory segment list.
4487	RI_RET_MEM_INVALID_ADDRESS	Invalid address in memory segment list.
4488	RI_RET_MEM_INVALID_LENGTH	Invalid length in memory segment list.
4489	RI_RET_FAIL	Unknown/fatal error.

4490 **CONSUMER USAGE**

4491 This function is described in terms of RDMA usage for non-coherent memory. While this is the
4492 common usage, it is possible to use it to support other operations using non-coherent memory.

4493 **IMPLEMENTATION NOTES**

4494 None.

4495 **SEE ALSO**

4496 [*ri_mem_sync_rread\(\)*](#)

ri_ah_create()

4497

4498 NAME

4499 `ri_ah_create` – create a handle for the specified address vector to be used in all IB Unreliable
4500 Datagram (UD) communication

4501 SYNOPSIS

```
4502 ri_api_return_t ri_ah_create(  
4503     IN     ri_rnic_handle_t      rnic,  
4504     IN     ri_pd_handle_t       pd,  
4505     IN     ri_ud_address_vector_t *av,  
4506     OUT    ri_address_handle_t   *ah,  
4507     IN OUT ri_vp_data_t         vp_data,  
4508     IN OUT ri_os_data_t         os_data,  
4509     OUT    ri_err_data_t        *err_data  
4510 );  
4511  
4512 typedef struct ri_ud_address_vector ri_ud_address_vector_t;  
4513 struct ri_ud_address_vector {  
4514     ri_gen_address_vector_t  av_common;  
4515     uint8_t                  physical_port;  
4516     /* number of the local physical port */  
4517 };
```

4518 DESCRIPTION

4519 *rnic* Handle to RNIC on which to create address handle.

4520 *pd* Handle to Protection Domain the address handle is to be associated with.

4521 *av* Address vector describing the remote node address for UD sends.

4522 *ah* Address handle returned.

4523 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
4524 information gets communicated from uVP to the “syscall” version of this function which in
4525 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
4526 when passed by uAL to uVP and uVPs must ignore this parameter.

4527 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
4528 information gets communicated from uAL to uVP which in turn gets passed unchanged to
4529 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
4530 get passed unchanged to appropriate calls back into the OS.

4531 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4532 RNICPI. The relation between the value received and later passed on is also not defined.
4533 Some interested OSVs and IHVs may use it for private communication.

4534 `ri_ah_create()` creates an address handle corresponding to the address vector passed. The
4535 address handle returned can be used by the caller to reference local and global destinations in all
4536 UD QP Post Sends.

4537 **RETURN VALUE**

4538 4539	RI_RET_SUCCESS	Successful completion. A valid address handle is returned.
4540	RI_RET_RESOURCE_SHORTAGE	Adequate resources not available to complete the call.
4541	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4542	RI_RET_PD_INVALID	The Protection Domain passed is invalid.
4543 4544	RI_RET_INVALID_PARAMETER	<i>av</i> or <i>ah</i> passed is NULL or <i>av</i> points to an invalid address vector.
4545	RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device
4546	RI_RET_FAIL	Unknown/fatal error.

4547 **CONSUMER USAGE**

4548 None.

4549 **IMPLEMENTATION NOTES**

4550 None.

4551 **SEE ALSO**

4552 [*ri_ah_destroy\(\)*](#), [*ri_ah_modify\(\)*](#), [*ri_ah_query\(\)*](#)

ri_ah_query()

4553

4554 NAME

4555 ri_ah_query – return the address vector and Protection Domain associated with the specified
4556 address handle

4557 SYNOPSIS

```
4558 ri_api_return_t ri_ah_query(  
4559     IN    ri_rnic_handle_t      rnic,  
4560     IN    ri_address_handle_t   ah,  
4561     OUT   ri_ud_address_vector_t *av,  
4562     OUT   ri_pd_handle_t        *pd,  
4563     OUT   ri_err_data_t         *err_data  
4564 );
```

4565 DESCRIPTION

4566 *rnic* Handle to RNIC on which to create address handle.

4567 *ah* Address handle.

4568 *av* Address vector describing the remote node address.

4569 *pd* Handle to Protection Domain the address handle is associated with.

4570 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4571 RNICPI. The relation between the value received and later passed on is also not defined.
4572 Some interested OSVs and IHVs may use it for private communication.

4573 *ri_ah_query()* queries the specified address handle and returns the address vector and Protection
4574 Domain it was created with through the *ri_ah_create()* call.

4575 RETURN VALUE

4576 RI_RET_SUCCESS Successful completion. Valid address vector and
4577 Protection Domain are returned.

4578 RI_RET_RNIC_INVALID_HANDLE The RNIC handle passed is invalid.

4579 RI_RET_ADDRESS_HANDLE_INVALID The address handle passed is invalid.

4580 RI_RET_INVALID_PARAMETER *av* or *pd* passed is NULL.

4581 RI_RET_XPORT_UNSUPPORTED The operation is done on a non-InfiniBand device.

4582 RI_RET_FAIL Unknown/fatal error.

4583 CONSUMER USAGE

4584 None.

4585 IMPLEMENTATION NOTES

4586 None.

4587 **SEE ALSO**
4588 *ri_ah_create(), ri_ah_destroy(), ri_ah_modify()*
4589

ri_ah_modify()

4590
4591 **NAME**
4592 `ri_ah_modify` – associate a new address vector with the specified address handle

4593 **SYNOPSIS**
4594

```
ri_api_return_t ri_ah_modify(  
4595     IN    ri_rnic_handle_t    rnic,  
4596     IN    ri_address_handle_t ah,  
4597     IN    ri_ud_address_vector_t *av,  
4598     OUT   ri_err_data_t      *err_data  
4599 );
```

4600 DESCRIPTION

4601 *rnic* Handle to RNIC on which to create address handle.
4602 *ah* Address handle.
4603 *av* Address vector describing the remote node address for UD sends.
4604 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4605 RNICPI. The relation between the value received and later passed on is also not defined.
4606 Some interested OSVs and IHVs may use it for private communication.
4607 *ri_ah_modify()* changes the address vector associated with the address handle passed in by the
4608 caller.

4609 RETURN VALUE

4610	RI_RET_SUCCESS	Successful completion. The address vector associated with the specified address handle is successfully modified.
4611		
4612		
4613	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4614	RI_RET_ADDRESS_HANDLE_INVALID	The address handle passed is invalid.
4615	RI_RET_INVALID_PARAMETER	<i>av</i> passed is NULL or <i>av</i> points to an invalid address vector.
4616		
4617	RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
4618	RI_RET_FAIL	Unknown/fatal error.

4619 CONSUMER USAGE

4620 None.

4621 IMPLEMENTATION NOTES

4622 None.

4623 SEE ALSO

4624 *ri_ah_create()*, *ri_ah_destroy()*, *ri_ah_query()*

ri_ah_destroy()

4625

4626 NAME

4627 `ri_ah_destroy` – destroy the specified address handle used for Unreliable Datagram (UD)
4628 communication

4629 SYNOPSIS

```
4630 ri_api_return_t ri_ah_destroy(  
4631     IN    ri_rnic_handle_t    rnic,  
4632     IN    ri_address_handle_t ah,  
4633     OUT   ri_err_data_t      *err_data  
4634 );
```

4635 DESCRIPTION

4636 *rnic* Handle to RNIC the address handle belongs to.

4637 *ah* Address handle to be destroyed.

4638 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4639 RNICPI. The relation between the value received and later passed on is also not defined.
4640 Some interested OSVs and IHVs may use it for private communication.

4641 `ri_ah_destroy()` invalidates an address vector and its associated address handle. After the address
4642 handle is invalidated, it can no longer be used to reference any destination.

4643 RETURN VALUE

4644	RI_RET_SUCCESS	Successful completion. The address handle is
4645		successfully destroyed.
4646	RI_RET_RNIC_INVALID_HANDLE	The RNIC handle passed is invalid.
4647	RI_RET_ADDRESS_HANDLE_INVALID	The address handle passed is invalid.
4648	RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand device.
4649	RI_RET_FAIL	Unknown/fatal error.

4650 CONSUMER USAGE

4651 None.

4652 IMPLEMENTATION NOTES

4653 None.

4654 SEE ALSO

4655 [*ri_ah_create\(\)*](#), [*ri_ah_modify\(\)*](#), [*ri_ah_query\(\)*](#)

4656

ri_mcast_attach()

4657

4658 **NAME**

4659 `ri_mcast_attach` – attach a QP to a multicast group

4660 **SYNOPSIS**

```
4661 ri_api_return_t ri_mcast_attach(  
4662     IN      ri_rnic_handle_t  rnic,  
4663     IN      ri_lid_t          mlid,  
4664     IN      ri_gid_t          *mgid,  
4665     IN      ri_qp_handle_t    qp,  
4666     IN OUT  ri_vp_data_t      vp_data,  
4667     IN OUT  ri_os_data_t      os_data,  
4668     OUT     ri_err_data_t     *err_data  
4669 );
```

4670 **DESCRIPTION**

4671 *rnic* RNIC handle.

4672 *mlid* Multicast group MLID.

4673 *mgid* Multicast group MGID.

4674 *qp* QP handle of the QP to be attached to the specified multicast group.

4675 *vp_data* Opaque data owned by IHV modules; OS modules should not try to interpret it. This
4676 information gets communicated from uVP to the “syscall” version of this function which in
4677 turn gets passed unchanged by kAL to kVP while processing this call. This must be zero
4678 when passed by uAL to uVP and uVPs must ignore this parameter.

4679 *os_data* Opaque data owned by OS modules; IHV modules should not try to interpret it. This
4680 information gets communicated from uAL to uVP which in turn gets passed unchanged to
4681 the “syscall” version of this function. Similarly, when conveyed from kAL to kVP, it should
4682 get passed unchanged to appropriate calls back into the OS.

4683 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4684 RNICPI. The relation between the value received and later passed on is also not defined.
4685 Some interested OSVs and IHVs may use it for private communication.

4686 Attaches the QP to the specified multicast group. The only function of this Verb is to assign the
4687 Receive Work Queue of this QP to the specified multicast group; after the attachment completes,
4688 this QP will be provided with a copy of every multicast message addressed to the group
4689 specified by the MGID and received on the RNIC port with which the QP is associated. Creation
4690 of the multicast group, and reconfiguration of the fabric such that packets addressed to that
4691 group are routed to a local RNIC port, is described in Section 7.10, *IBA and Raw Packet*
4692 *Multicast* of the InfiniBand Architecture Release 1.2, Volume 1 [IB-R1.2]. The Service Type of
4693 the specified QP must be Unreliable Datagram. It is an error to specify a QP with any other
4694 Service Type. One or more QPs are allowed to be attached to a multicast group on the RNIC. If
4695 the maximum number of multicast group attachments has already been reached for the RNIC
4696 when a QP attempts to attach to the multicast group, an error is returned. The input modifier
4697 which determines the multicast group to attach to are the MGID and MLID. Both input

4698 modifiers must be supplied. The IB unreliable multicast feature is optional. This Verb is required
4699 only if IB unreliable multicast is supported by the RNIC.

4700 **RETURN VALUE**

4701	RI_RET_SUCCESS	Operation completed successfully.
4702	RI_RET_RESOURCE_SHORTAGE	Insufficient resources to complete request.
4703	RI_RET_RNIC_INVALID_HANDLE	Invalid RNIC handle.
4704	RI_RET_MGID_INVALID	Invalid multicast MLID.
4705	RI_RET_MLID_INVALID	Invalid Multicast group MGID.
4706	RI_RET_QP_INVALID_HANDLE	Invalid QP handle.
4707	RI_RET_QP_INVALID_SERVICE_TYPE	Invalid Service Type for this QP.
4708	RI_RET_RNIC_QPS_PER_MCAST_PORT_EXCEEDED	Number of QPs attached to multicast groups
4709		exceeded.
4710	RI_RET_OP_UNSUPPORTED	Multicast not supported by RNIC.
4711	RI_RET_XPORT_UNSUPPORTED	The operation is done on a non-InfiniBand
4712		device.
4713	RI_RET_FAIL	Unknown/fatal error.

4714 **CONSUMER USAGE**

4715 None.

4716 **IMPLEMENTATION NOTES**

4717 None.

4718 **SEE ALSO**

4719 [*ri_mcast_detach\(\)*](#)

ri_mcast_detach()

4720

4721 NAME

4722 ri_mcast_detach – detach a QP from the multicast group

4723 SYNOPSIS

```
4724 ri_api_return_t ri_mcast_detach(  
4725     IN    ri_rnic_handle_t  rnic,  
4726     IN    ri_lid_t          mlid,  
4727     IN    ri_gid_t          *mgid,  
4728     IN    ri_qp_handle_t    qp,  
4729     OUT   ri_err_data_t     *err_data  
4730 );
```

4731 DESCRIPTION

4732 *rnic* RNIC handle.

4733 *mlid* Multicast group MLID.

4734 *mgid* Multicast group MGID.

4735 *qp* QP handle.

4736 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4737 RNICPI. The relation between the value received and later passed on is also not defined.
4738 Some interested OSVs and IHVs may use it for private communication.

4739 Detaches the specified QP from a multicast group. The only function of this Verb is to detach the
4740 Receive Work Queue of this QP from the specified multicast group. All of the input modifiers
4741 must be correct for the QP to be detached. If the QP is attached to a different multicast group or
4742 port, an error will be returned. This Verb is required only if IB unreliable multicast is supported
4743 by the RNIC. The IB unreliable multicast feature is optional.

4744 RETURN VALUE

4745 RI_RET_SUCCESS Operation completed successfully.

4746 RI_RET_RNIC_INVALID_HANDLE Invalid RNIC handle.

4747 RI_RET_MLID_INVALID Invalid multicast MLID.

4748 RI_RET_MGID_INVALID Invalid Multicast group MGID.

4749 RI_RET_QP_INVALID_HANDLE Invalid QP handle.

4750 RI_RET_XPORT_UNSUPPORTED The operation is done on a non-InfiniBand device.

4751 RI_RET_OP_UNSUPPORTED Multicast not supported by RNIC.

4752 CONSUMER USAGE

4753 None.

4754 **IMPLEMENTATION NOTES**
4755 None.

4756 **SEE ALSO**
4757 *ri_mcast_attach()*

ri_process_sma_mad()

4758

4759 NAME

4760 ri_process_sma_mad – process the InfiniBand SMA MAD request

4761 SYNOPSIS

```
4762 ri_api_return_t ri_process_sma_mad(  
4763     IN    ri_rnic_handle_t  rnic,  
4764     IN    uint8_t          port_num,  
4765     IN    char              *in_mad,  
4766     OUT   char              *out_mad,  
4767     OUT   ri_err_data_t     *err_data  
4768 );
```

4769 DESCRIPTION

4770 *rnic* RNIC handle.

4771 *port_num* Port number on which the MAD is received.

4772 *in_mad* Request MAD.

4773 *out_mad* Response MAD.

4774 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4775 RNICPI. The relation between the value received and later passed on is also not defined.
4776 Some interested OSVs and IHVs may use it for private communication.

4777 *ri_process_mad* can be used to build IHV-independent InfiniBand Subnet Management Agent.
4778 The SMA can pass the MAD message received on its UD QPs and get the response MAD built
4779 by the IHV code. The response MAD can then be sent back using the hardware-independent
4780 *ri_qp_postsq()* or its equivalent call. If *in_mad* is a *SubnGet/SubnSet* request, MAD and
4781 *out_mad* would be the corresponding response *SubnGetResp* MAD. If a *SubnTrap* message is
4782 originated in the CI without the knowledge of the Consumer, the message will be made available
4783 on the Receive Queue of QP0. Upon receiving the message, the Consumer (SMA) could forward
4784 the message to the SM and get back a *SubnTrapRepress* MAD. The SMA later passes on
4785 *SubnTrapRepress* MAD through *ri_process_sma_mad()* just to notify the HCA of SM
4786 acknowledgement. *out_mad* is undefined if *in_mad* contains *SubnTrapRepress*. If the trap is
4787 initiated by the Consumer, *SubnTrapRepress* MAD shall not be passed to
4788 *ri_process_sma_mad()*.

4789 This call can be invoked from an interrupt context.

4790 RETURN VALUE

4791 RI_RET_SUCCESS Successful completion.

4792 RI_RET_XPORT_UNSUPPORTED Attempt to do the operation on a non-InfiniBand device.

4793 RI_RET_FAIL The request MAD could not be processed.

4794 CONSUMER USAGE

4795 None.

4796 **IMPLEMENTATION NOTES**
4797 None.

4798 **SEE ALSO**
4799 None.

ri_svc_attach()

4800

4801 **NAME**

4802 `ri_svc_attach` – attach an RNIC to the RDMA access layer

4803 **SYNOPSIS: KERNEL**

```
4804 ri_api_return_t ri_svc_attach(  
4805     IN    ri_vp_handle_t    vp_hdl,  
4806     IN    ri_dev_info_t    dev_info,  
4807     IN    ri_svcs_dev_arg_t *svcs,  
4808     IN    ri_ops_t         *al_locking_ops,  
4809     IN    ri_ops_t         *vp_locking_ops,  
4810     OUT   ri_err_data_t    *err_data  
4811 );  
4812  
4813 typedef struct ri_svcs_dev_arg ri_svcs_dev_arg_t;  
4814 struct ri_svcs_dev_arg {  
4815     /* Major version of RNICPI implemented */  
4816     uint32_t rnicpi_major_version;  
4817  
4818     /* Minor version of RNICPI implemented */  
4819     uint32_t rnicpi_minor_version;  
4820  
4821     /* Major version of RNIC firmware */  
4822     uint32_t fw_major_version;  
4823  
4824     /* Minor version of RNIC firmware */  
4825     uint32_t fw_minor_version;  
4826  
4827     /* Sub minor version of RNIC firmware */  
4828     uint32_t fw_sub_minor_version;  
4829  
4830     /* size of buffer to allocate for query_rnic in bytes */  
4831     uint_t query_size;  
4832  
4833     /*  
4834     * NULL terminated string pathname of the uVP shared  
4835     * library. This will be used later to load the  
4836     * appropriate uVP library.  
4837     */  
4838     char lib_path[RI_MAX_LIB_PATH_LEN];  
4839  
4840     /*  
4841     * NULL terminated string of hardware path of the device  
4842     * on the OSV environment. This could be used for  
4843     * diagnostics or for persistent identification.  
4844     */  
4845     char hw_path[RI_MAX_HW_PATH_LEN];  
4846 };
```

4847 **DESCRIPTION**

4848 `vp_hdl` Verb provider handle to the RNIC. This handle must be unique for each RNIC.

4849 *dev_info* OS-specific device identifier.

4850 *svcs* High-level information about the device, its interfaces, and its compatible software.

4851 *al_locking_ops*
4852 Set of pointers to functions implementing the RNICPI operations using access layer
4853 enforced serialization.

4854 *vp_locking_ops*
4855 Set of pointers to functions implementing the RNICPI operations using verb provider layer
4856 enforced serialization. This vector is optional and, if not supported, this argument will be
4857 NULL.

4858 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4859 RNICPI. The relation between the value received and later passed on is also not defined.
4860 Some interested OSVs and IHVs may use it for private communication.

4861 The *ri_svc_attach()* function is an upcall from the RNIC driver to the OS RDMA access layer.
4862 The function “attaches” or registers the device with the RDMA access layer. Without calling this
4863 function, the OS may be unaware of a device's RDMA ability. The *vp_hdl* is an opaque handle
4864 used to uniquely identify the RNIC device. It is required that this handle be unique for each
4865 RNIC. The *vp_hdl* is also used in subsequent calls such as *ri_rnic_open()* and *ri_svc_detach()*.
4866 The *dev_info* argument is used by the OS to identify the device or network interface.
4867 Necessarily, it is an OS-dependent type. The *svcs* argument describes the version of the device
4868 and its driver interfaces. The RDMA access layer may use this information passed for diagnostic
4869 purposes. The values passed for the *rnicpi_major_version* and *rnicpi_minor_version* fields of
4870 *svcs* correspond to version of the RNICPI specification that the VP implements. The device
4871 driver passes the set of operation functions (serialized through the AL) in the *al_locking_ops*
4872 argument. Another vector of operations using VP enforced serialization is optionally provided
4873 in the *vp_locking_ops* argument (NULL if not provided). The *query_size* field of *svcs* is the size of
4874 a single buffer capable of containing all the information returned from *ri_rnic_query()*, including
4875 the main *ri_rnic_attr_t* struct, the transport-specific struct, and any related port information
4876 structs.

4877 The RDMA infrastructure may start invoking the operation entry points even from within an
4878 invocation of *ri_svc_attach()*. In particular, *ri_svc_attach()* should be prepared for the possibility
4879 that *ri_rnic_open()* will be called from within the invocation of this function. The
4880 RI_RET_RNIC_INUSE error is returned if an attempt it made to attach the same RNIC (as
4881 identified through the *vp_rnic_handle*) more than once without first detaching it through an
4882 *ri_svc_detach()* call.

4883 This function only exists in the kernel. There is no user-level version of this function. User-level
4884 initialization uses the *ri_svc_lib_reg()* function.

4885 Definitions for *ri_vp_handle_t*, *ri_dev_info_t*, *ri_err_data_t*, RI_MAX_HW_PATH_LEN, and
4886 RI_MAX_LIB_PATH_LEN are OS-dependent.

4887 **RETURN VALUE**

4888 RI_RET_SUCCESS Device attached to RDMA access layer successfully.

4889 RI_RET_RESOURCE_SHORTAGE No resources available to attach the device.

4890	RI_RET_VP_INVALID_HANDLE	Verb provider handle passed is invalid.
4891	RI_RET_INVALID_DEV_INFO	<i>dev_info</i> parameter is invalid.
4892	RI_RET_INVALID_PARAMETER	<i>svcs</i> , and/or <i>al_locking_ops</i> parameter is NULL.
4893	RI_RET_INVALID_HW_PATH	<i>hw_path</i> in <i>svcs</i> is invalid.
4894	RI_RET_INVALID_LIB_PATH	<i>lib_path</i> in <i>svcs</i> is invalid.
4895 4896	RI_RET_INVALID_VP_VERSION	Verb provider version is incompatible with that of the OS infrastructure.
4897	RI_RET_RNIC_INUSE	RNIC is already attached.
4898	RI_RET_FAIL	Unknown/fatal error.

4899 **CONSUMER USAGE**

4900 None.

4901 **IMPLEMENTATION NOTES**

4902 As stated above, it is required that *ri_svc_attach()* can uniquely distinguish each RNIC through
4903 the VP handle. This requirement means there is an agreement between AL (and/or OS) and the
4904 IHVs about the verb provider (VP) handle implementation, so that each IHV can supply a VP
4905 handle independently that will not clash with other IHV VP handles. Typically, this is
4906 accomplished by the VP returning either an identifier previously given to it by the AL and/or OS
4907 (e.g., OS-specific device identifier) or by using a pointer to memory that has been privately
4908 allocated for each VP. Because there is an agreement about the definition of the VP RNIC
4909 handle, it may be possible that an incorrect one could be recognized, which will cause the
4910 RI_RET_VP_INVALID_HANDLE error.

4911 Another area requiring an agreement with the OS is the *dev_info* argument which is meant to be
4912 an OS-specific identifier/handle for the device or network interface. This allows the OS to
4913 properly coordinate the usage of the RNIC as a device and network interface as well.

4914 **SEE ALSO**

4915 *ri_async_set_event_handler()*, *ri_rnic_close()*, *ri_rnic_free_hdl()*, *ri_rnic_get_hdl()*,
4916 *ri_rnic_open()*, *ri_svc_detach()*, *ri_svc_lib_reg()*, *ri_ops_t*

ri_svc_detach()

4917

4918 **NAME**

4919 `ri_svc_detach` – detach an RNIC from the RDMA access layer

4920 **SYNOPSIS: KERNEL**

```
4921 ri_api_return_t ri_svc_detach(  
4922     IN ri_vp_handle_t vp_hdl,  
4923     OUT ri_err_data_t *err_data  
4924 );
```

4925 **DESCRIPTION**

4926 `vp_hdl` Verb provider handle to the RNIC. This is the same handle passed in the `ri_svc_attach()`
4927 function.

4928 `err_data` Data passed in all calls. How the recipient of the information uses it is outside the scope of
4929 RNICPI. The relation between the value received and later passed on is also not defined.
4930 Some interested OSVs and IHVs may use it for private communication.

4931 The `ri_svc_detach()` function is an upcall from the RNIC driver to the RDMA access layer. This
4932 function “detaches” or deregisters the RNIC from the RDMA access layer. The `vp_hdl` argument
4933 uniquely specifies the RNIC device being detached from the RDMA access layer.

4934 Upon successful return of `ri_svc_detach()`, the caller should not attempt to use the `vp_hdl` for any
4935 other function call. This function should only be called after a successful invocation of
4936 `ri_rnic_close()`. Otherwise, the `RI_RET_RNIC_INUSE` error is returned. The RDMA access
4937 layer shall be able to handle this call anytime after `ri_rnic_close()` returns.

4938 This function only exists in the kernel. Similar functionality at user level uses other mechanisms.

4939 The definitions for `ri_vp_handle_t` and `ri_err_data_t` are OS-dependent.

4940 **RETURN VALUE**

4941 `RI_RET_SUCCESS` Successful completion.

4942 `RI_RET_VP_INVALID_HANDLE` Verb provider handle to RNIC is invalid.

4943 `RI_RET_RNIC_INUSE` RNIC is still open.

4944 `RI_RET_FAIL` Unknown/fatal error.

4945 **CONSUMER USAGE**

4946 None.

4947 **IMPLEMENTATION NOTES**

4948 None.

4949 **SEE ALSO**

4950 `ri_async_set_event_handler()`, `ri_rnic_close()`, `ri_rnic_free_hdl()`, `ri_rnic_get_hdl()`,
4951 `ri_rnic_open()`, `ri_svc_attach()`

ri_svc_lib_reg()

4952
4953 **NAME**
4954 `ri_svc_lib_reg` – register a user verb provider library with the user access layer

4955 **SYNOPSIS: USER**

```
4956 ri_api_return_t ri_svc_lib_reg(  
4957     IN     char      *lib_path,  
4958     IN     ri_ops_t  *al_locking_ops,  
4959     IN     ri_ops_t  *vp_locking_ops,  
4960     IN OUT ri_os_data_t os_data,  
4961     OUT    ri_err_data_t *err_data  
4962 );
```

4963 **DESCRIPTION**

4964 *lib_path* Library pathname used by the uAL to identify the devices of this type. The pathname
4965 pointed to by this argument should be no longer than RI_MAX_LIB_PATH chars
4966 (including terminating NULL).

4967 *al_locking_ops*
4968 Set of pointers to functions implementing the RNICPI operations using access layer
4969 enforced serialization.

4970 *vp_locking_ops*
4971 Set of pointers to functions implementing the RNICPI operations using verb provider layer
4972 enforced serialization. This vector is optional and if not supported, this argument will be
4973 NULL.

4974 *os_data* Opaque data owned by OS modules. IHV modules should not try to interpret it. This
4975 information gets communicated from uAL to uVP which in turn gets passed unchanged via
4976 “syscall”. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
4977 appropriate calls back into the OS. Check the OS-specific supplement to RNICPI for details.

4978 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
4979 RNICPI. The relation between the value received and later passed on is also not defined.
4980 Some interested OSVs and IHVs may use it for private communication.

4981 The `ri_svc_lib_reg()` function is an upcall used to register the user-level verb provider library
4982 with the user access layer. Once this function has been called, the uAL may call
4983 `ri_rnic_get_hdl()` on available RNICs that use this library. The *lib_path* argument identifies the
4984 library being loaded to the uAL. The *al_locking_ops* parameter is the operations vector exported
4985 by the library that implements AL enforced locking. An optional second VP enforced locking
4986 operations vector may be present in *vp_locking_ops*.

4987 This function is called once for each library loaded. If there are several RNIC instances that use
4988 the same library, this function is still only called once for that library.

4989 The *lib_path* should match a library name provided to the kAL by at least one currently open
4990 RNIC (see `ri_rnic_open()`). Library names are passed to the kAL through the *lib_path* field in
4991 the *svcs* argument of `ri_svc_attach()`. If the library path does not match a known library name,
4992 the RI_RET_INVALID_LIB_PATH error is returned.

4993 This function is only available at user level. The kernel uses a different initialization mechanism
4994 (see *ri_svc_attach()*).

4995 Definitions for *ri_os_data_t*, *ri_err_data_t*, and *RI_MAX_LIB_PATH_LEN* are OS-dependent.

4996 **RETURN VALUE**

4997 *RI_RET_SUCCESS* Library registered with RDMA access layer successfully.

4998 *RI_RET_RESOURCE_SHORTAGE* No resources available to complete the operation.

4999 *RI_RET_INVALID_LIB_PATH* Library pathname does not match known library.

5000 *RI_RET_INVALID_PARAMETER* *al_locking_ops* parameter is NULL.

5001 *RI_RET_FAIL* Unknown/fatal error.

5002 **CONSUMER USAGE**

5003 Typically, the behavior of being called once per library is accomplished by calling this function
5004 from the “initializer” entry point of the library being dynamically loaded.

5005 **IMPLEMENTATION NOTES**

5006 How the uAL is informed of available RNICs which have successfully called *ri_svc_attach()*
5007 and have been opened (via *ri_rnic_open()*) in the kernel is outside the scope of RNICPI.
5008 Typically, there will be private syscall communication between uAL and kAL to facilitate this.

5009 **SEE ALSO**

5010 *ri_rnic_get_hdl()*, *ri_rnic_open()*, *ri_svc_attach()*

5011 **ri_svc_mem_pin(), ri_svc_mem_unpin()**

5012 **NAME**

5013 `ri_svc_mem_pin` – pin a range of virtual memory

5014 `ri_svc_mem_unpin` – unpin a range of virtual memory

5015 **SYNOPSIS: KERNEL**

```
5016 ri_api_return_t ri_svc_mem_pin (  
5017     IN      ri_vaddr_t      vaddr,  
5018     IN      ri_length_t     len,  
5019     IN OUT  ri_os_data_t     os_data,  
5020     OUT     ri_err_data_t    *err_data  
5021 );  
5022  
5023 ri_api_return_t ri_svc_mem_unpin (  
5024     IN      ri_vaddr_t      vaddr,  
5025     IN      ri_length_t     len,  
5026     IN OUT  ri_os_data_t     os_data,  
5027     OUT     ri_err_data_t    *err_data  
5028 );
```

5029 **DESCRIPTION**

5030 *vaddr* Virtual address.

5031 *len* Length of the virtual address range in bytes.

5032 *os_data* Opaque data owned by OS modules, IHV modules should not try to interpret it. This
5033 information gets communicated from uAL to uVP which in turn gets passed unchanged to
5034 *ri_syscall*. Similarly, when conveyed from kAL to kVP, it should get passed unchanged to
5035 appropriate calls back into the OS.

5036 *err_data* Data passed in all calls. How the recipient of the information uses it is outside the scope of
5037 RNICPI. The relation between the value received and later passed back is also not defined.
5038 Some interested OSVs and IHVs may use it for private communication.

5039 These functions allow the verbs provider to pin and unpin a virtual memory range *vaddr* with
5040 length *len*.

5041 Multiple *ri_svc_mem_pin()* calls can be made on the same or overlapping memory ranges with
5042 all pin reference counting issues properly handled.

5043 The address range of an *ri_svc_mem_unpin()* call must be the same as the *ri_svc_mem_pin()* call
5044 originally used to pin the memory.

5045 *os_data* is an opaque parameter used exclusively by the OSV code to manage the state of the
5046 virtual memory range *vaddr*. The opaque *os_data* used to unpin *vaddr* with *ri_svc_mem_unpin()*
5047 must be the same *os_data* opaque used when *vaddr* was previously pinned with
5048 *ri_svc_mem_pin()*. An OSV may also require that the *os_data* field used in these functions be
5049 the same as the *os_data* opaque passed by the RNICPI downcall that uses this service (e.g., the
5050 *os_data* passed by *ri_mr_reg()*).

5051 These functions apply only to privileged callers and should only be invoked by the kVP.

5052 **RETURN VALUE**

5053	RI_RET_SUCCESS	Successful completion.
5054	RI_RET_RESOURCE_SHORTAGE	Not enough resources to complete the call.
5055	RI_RET_NO_PERMISSION	Caller is not privileged.
5056	RI_RET_INVALID_PARAMETER	One of the parameters is invalid.
5057	RI_RET_FAIL	Unknown/fatal error.

5058 **CONSUMER USAGE**

5059 None.

5060 **IMPLEMENTATION NOTES**

5061 None.

5062 **SEE ALSO**

5063 *ri_mem_map(), ri_mem_pin(), ri_mem_unmap(), ri_mem_unpin()*

5064

5066

ri_api_return_t

5067 **NAME**

5068 ri_api_return_t – function call status return codes

5069 **SYNOPSIS**

```
5070 typedef enum ri_api_return ri_api_return_t
5071 enum ri_api_return {
5072     RI_RET_SUCCESS = 0,
5073
5074     /* generic errors */
5075     RI_RET_RESOURCE_SHORTAGE,
5076     RI_RET_INVALID_PARAMETER,
5077     RI_RET_NO_PERMISSION,
5078     RI_RET_XPORT_UNSUPPORTED,
5079     RI_RET_OP_UNSUPPORTED,
5080     RI_RET_BUFFER_TOO_SMALL,
5081     RI_RET_FAIL,
5082
5083     /* options */
5084     RI_RET_QP_RESIZE_UNSUPPORTED,
5085     RI_RET_SRQ_UNSUPPORTED,
5086     RI_RET_SRQ_RESIZE_UNSUPPORTED,
5087     RI_RET_EXT_BMM_UNSUPPORTED,
5088     RI_RET_EXT_ZBVA_UNSUPPORTED,
5089     RI_RET_EXT_LIF_UNSUPPORTED,
5090     RI_RET_RNIC_ATOMICS_UNSUPPORTED,
5091
5092     /* VP */
5093     RI_RET_VP_INVALID_HANDLE,
5094     RI_RET_INVALID_DEV_INFO,
5095     RI_RET_INVALID_HW_PATH,
5096     RI_RET_INVALID_LIB_PATH,
5097     RI_RET_INVALID_PBL_MODE,
5098     RI_RET_INVALID_VP_VERSION,
5099
5100     /* RNIC */
5101     RI_RET_RNIC_INVALID_HANDLE,
5102     RI_RET_RNIC_INUSE,
5103     RI_RET_RNIC_NOT_OPEN,
5104
5105     /* PD */
5106     RI_RET_PD_INVALID_HANDLE,
5107     RI_RET_PD_INUSE,
5108
5109     /* CQ */
5110     RI_RET_CQ_INVALID_HANDLE,
5111     RI_RET_CQ_CAPACITY_EXCEEDED,
5112     RI_RET_CQ_EMPTY,
5113     RI_RET_CQ_ERROR,
5114     RI_RET_CQ_INUSE,
5115     RI_RET_CQ_INVALID_EVENT_HANDLER,
5116     RI_RET_CQ_NO_EVENT_HANDLER,
5117     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,

```

```

5172         RI_RET_MGID_INVALID,
5173         RI_RET_MLID_INVALID,
5174         RI_RET_MULTICAST_ATTACHED,
5175         RI_RET_RNIC_QPS_PER_MCAST_PORT_EXCEEDED,
5176
5177         /* LLP */
5178         RI_RET_INVALID_LL_HANDLE,
5179         RI_RET_INVALID_DDP_RDMA_VERSION,
5180         RI_RET_LL_BAD_STATE,
5181         RI_RET_MPA_ATTR_UNSUPPORTED
5182     };

```

5183 **DESCRIPTION**

5184 The *ri_api_return_t* definition includes all function return values used in the RNICPI
5185 specification.

5186 **CONSUMER USAGE**

5187 None.

5188 **IMPLEMENTATION NOTES**

5189 None.

5190 **SEE ALSO**

5191 None.

5192

5193 **NAME**

5194 ri_ops_t – RNICPI operations vector

5195 **SYNOPSIS**

```

5196 /* Definition of the vector of function pointers. */
5197 typedef struct ri_ops ri_ops_t;
5198 struct ri_ops {
5199     ri_op_rnic_open_t           ri_op_rnic_open;
5200     ri_op_rnic_close_t         ri_op_rnic_close;
5201     ri_op_async_set_event_handler_t ri_op_async_set_event_handler;
5202     ri_op_rnic_get_hdl_t       ri_op_rnic_get_hdl;
5203     ri_op_rnic_query_t         ri_op_rnic_query;
5204     ri_op_rnic_modify_t        ri_op_rnic_modify;
5205     ri_op_rnic_free_hdl_t      ri_op_rnic_free_hdl;
5206     ri_op_rnic_diag_t          ri_op_rnic_diag;
5207     ri_op_pd_alloc_t           ri_op_pd_alloc;
5208     ri_op_pd_dealloc_t         ri_op_pd_dealloc;
5209     ri_op_cq_create_t          ri_op_cq_create;
5210     ri_op_cq_query_t           ri_op_cq_query;
5211     ri_op_cq_modify_t          ri_op_cq_modify;
5212     ri_op_cq_destroy_t         ri_op_cq_destroy;
5213     ri_op_special_qp_get_t     ri_op_special_qp_get;
5214     ri_op_cq_set_event_handler_t ri_op_cq_set_event_handler;
5215     ri_op_cq_req_notification_t ri_op_cq_req_notification;
5216     ri_op_cq_poll_t            ri_op_cq_poll;
5217     ri_op_qp_create_t          ri_op_qp_create;
5218     ri_op_qp_query_t           ri_op_qp_query;
5219     ri_op_qp_modify_t          ri_op_qp_modify;
5220     ri_op_qp_destroy_t         ri_op_qp_destroy;
5221     ri_op_qp_postsq_t          ri_op_qp_postsq;
5222     ri_op_qp_post_send_t       ri_op_qp_post_send;
5223     ri_op_qp_post_rdma_read_t  ri_op_qp_post_rdma_read;
5224     ri_op_qp_post_rdma_write_t ri_op_qp_post_rdma_write;
5225     ri_op_qp_post_bind_t       ri_op_qp_post_bind;
5226     ri_op_qp_post_fmr_t        ri_op_qp_post_fmr;
5227     ri_op_qp_post_local_invalidate_t ri_op_qp_post_local_invalidate;
5228     ri_op_qp_post_atomic_t     ri_op_qp_post_atomic;
5229     ri_op_qp_post_datagram_t    ri_op_qp_post_datagram;
5230     ri_op_qp_postrq_t          ri_op_qp_postrq;
5231     ri_op_qp_post_recv_t       ri_op_qp_post_recv;
5232     ri_op_srq_create_t         ri_op_srq_create;
5233     ri_op_srq_query_t          ri_op_srq_query;
5234     ri_op_srq_modify_t         ri_op_srq_modify;
5235     ri_op_srq_destroy_t        ri_op_srq_destroy;
5236     ri_op_srq_post_t           ri_op_srq_post;
5237     ri_op_srq_post_recv_t      ri_op_srq_post_recv;
5238     ri_op_ltag_alloc_t         ri_op_ltag_alloc;
5239     ri_op_mr_reg_t             ri_op_mr_reg;
5240     ri_op_mr_query_t           ri_op_mr_query;
5241     ri_op_mr_rereg_t           ri_op_mr_rereg;
5242     ri_op_mr_dereg_t           ri_op_mr_dereg;

```

```

5243         ri_op_mr_reg_phys_t           ri_op_mr_reg_phys;
5244         ri_op_mr_rereg_phys_t         ri_op_mr_rereg_phys;
5245         ri_op_mr_reg_shared_t         ri_op_mr_reg_shared;
5246         ri_op_mw_alloc_t              ri_op_mw_alloc;
5247         ri_op_mw_dealloc_t            ri_op_mw_dealloc;
5248         ri_op_mw_query_t              ri_op_mw_query;
5249         ri_op_mem_pin_t               ri_op_mem_pin;
5250         ri_op_mem_unpin_t             ri_op_mem_unpin;
5251         ri_op_mem_map_t               ri_op_mem_map;
5252         ri_op_mem_unmap_t             ri_op_mem_unmap;
5253         ri_op_mem_sync_rread_t        ri_op_mem_sync_rread;
5254         ri_op_mem_sync_rwrite_t       ri_op_mem_sync_rwrite;
5255         ri_op_ah_create_t             ri_op_ah_create;
5256         ri_op_ah_query_t              ri_op_ah_query;
5257         ri_op_ah_modify_t             ri_op_ah_modify;
5258         ri_op_ah_destroy_t            ri_op_ah_destroy;
5259         ri_op_mcast_attach_t          ri_op_mcast_attach;
5260         ri_op_mcast_detach_t          ri_op_mcast_detach;
5261         ri_op_process_sma_mad_t       ri_op_process_sma_mad;
5262     };

```

5263 **DESCRIPTION**

5264 The *ri_ops_t* structure defines the operations that a verb provider implementation supplies. For
5265 each “downcall” from AL to VP, there is a corresponding function pointer. This function call
5266 vector definition is used for both uVP and kVP. There are some functions which are not intended
5267 to be used at both kernel and user level. In these cases, a stub function is provided for the
5268 inappropriate context that always returns an error.

5269 **CONSUMER USAGE**

5270 None.

5271 **IMPLEMENTATION NOTES**

5272 None.

5273 **SEE ALSO**

5274 [ri_svc_attach\(\)](#), [ri_svc_lib_reg\(\)](#)

5275

ri_rnic_attr_t

5276 **NAME**

5277 ri_rnic_attr_t – RNIC attribute data structures

5278 **SYNOPSIS**

```
5279 typedef enum ri_rdma_transport_type ri_rdma_transport_type_t;
5280 enum ri_rdma_transport_type {
5281     RI_RDMA_TRANSPORT_TYPE_IB,
5282     RI_RDMA_TRANSPORT_TYPE_IWARP
5283 };
5284
5285 typedef union ri_transport_attr_ptr ri_transport_attr_ptr_t;
5286 union ri_transport_attr_ptr {
5287     ri_ib_rnic_attr_t *ib;
5288     ri_iwarp_rnic_attr_t *iwarp;
5289 };
5290
5291 typedef enum ri_rnic_flags ri_rnic_flags_t;
5292 enum ri_rnic_flags {
5293     /* iWARP will always have BMM, ZBVA & LIF set */
5294     RI_RNIC_EXT_SRQ = 1 << 0, /* IB 1.1: not set */
5295     RI_RNIC_EXT_BMM = 1 << 1, /* IB 1.1: not set */
5296
5297     /* note that ZBVA, LIF & BLOCK_MODE require BMM */
5298     RI_RNIC_EXT_ZBVA = 1 << 2, /* IB 1.1: not set */
5299     RI_RNIC_EXT_LIF = 1 << 3, /* IB 1.1: not set */
5300
5301     /* if BLOCK_MODE not set, RNIC is in page mode */
5302     RI_RNIC_EXT_BLOCK_MODE = 1 << 4, /* IB 1.1: not set */
5303
5304     RI_RNIC_RQ_OVERFLOW_CHECKING = 1 << 5, /* IB: set */
5305     RI_RNIC_CQ_OVERFLOW_CHECKING = 1 << 6, /* IB: set */
5306
5307     RI_RNIC_CAN_RESIZE_QP = 1 << 7,
5308
5309     /* if RI_RNIC_EXT_SRQ not set, then SRQ stuff not defined */
5310     RI_RNIC_CAN_RESIZE_SRQ = 1 << 8,
5311     RI_RNIC_SRQ_SEQ_DEQUEUE = 1 << 9, /* IB SRQ: not set */
5312     RI_RNIC_SRQ_LAST_WQE_ASYNC = 1 << 10, /* IB SRQ: set */
5313
5314     /*
5315      * safe to post on passive side after successful modify
5316      * to RTS
5317      */
5318     RI_RNIC_SAFE_POST_ON_RTS = 1 << 11, /* IB: set */
5319
5320     /* narrow windows support is part of RI_RNIC_EXT_BMM */
5321     RI_RNIC_WIDE_WINDOWS_SUPPORTED = 1 << 12, /* IB: set */
5322     /*
5323      * extended key splitting
5324      * wide windows and virtual MRs also use index/key splitting
5325      */
5326 }
```

```

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

```

```

5432         /* Additional Vendor Info (NULL terminated) */
5433         char addl_vendor_info[RI_MAX_VENDOR_STRING_LEN];
5434     };

```

5435 DESCRIPTION

5436 The *ri_rnic_attr_t* struct defines the transport-independent attributes of an RNIC. This structure
5437 is returned in *ri_rnic_query()*. Transport-specific attributes are in substructures accessed through
5438 the *trans_attr* pointer field. The type of the transport-specific struct will be *ri_ib_rnic_attr_t* if
5439 the value of *transport_type* is RI_RDMA_TRANSPORT_TYPE_IB. For
5440 RI_RDMA_TRANSPORT_TYPE_IWARP, the type of the transport-specific structure is
5441 *ri_iwarp_rnic_attr_t*.

5442 The maximum values defined in this section are guaranteed not-to-exceed values. It is possible
5443 for an implementation to allocate various RNIC resources from a shared pool. In that case, all
5444 maximum values cannot be achieved simultaneously.

5445 A number of optional features are flagged in *rnic_flags*. Several of the flags are named with the
5446 RI_RNIC_EXT_ prefix indicating that they are regarded as “verb extensions” in InfiniBand
5447 Version 1.2. RI_RNIC_EXT_SRQ indicates support for Shared Receive Queues.
5448 RI_RNIC_EXT_BMM indicates support for the “Memory Management” extensions including
5449 the use of *ri_ltag_alloc()*, fast register WRs, local invalidation WRs, remote invalidation WRs,
5450 *reserved_lkey*/Stag0, Narrow Memory Windows, and the explicit designation of whether a
5451 Memory Region is shared or not. RI_RNIC_EXT_ZBVA indicates support for “Zero Based”
5452 virtual addresses. RI_RNIC_EXT_LIF indicates support for Local Invalidate Fencing.
5453 RI_RNIC_BLOCK_MODE indicates support for “Block mode” physical buffer lists. The
5454 following RI_RNIC_EXT_ items are dependent on RI_RNIC_EXT_BMM being supported:
5455 ZBVA, LIF, and BLOCK_MODE. A strictly 1.1 implementation of InfiniBand will not have
5456 support for any of the RI_RNIC_EXT_ items. An iWARP implementation will always indicate
5457 support for the following RI_RNIC_EXT_ items: BMM, ZBVA, and LIF.

5458 Note that there is no flag for the InfiniBand 1.2 “Queue Management” extension. If the
5459 underlying RNIC is not capable of posting more than one WR at a time, then the posting
5460 functions are required to iterate over the list of WRs posting one at a time. If the RNIC does not
5461 support multiple completion event handlers, then the number of supported completion handlers
5462 is one.

5463 If the RI_RNIC_SAFE_POST_ON_RTS flag is set, then it is safe for the passive side to do an
5464 SQ post immediately after transitioning the QP to RTS. If not set, the Consumer must refrain
5465 from posting to the SQ until after a receive operation completes on the QP.

5466 “Narrow windows” are RDMAC or IB type2 style windows which are associated with a single
5467 connection when bound. “Wide windows” are IB type1 windows which are access checked
5468 against a PD.

5469 If the RI_RNIC_EXT_INDEX_KEY_SPLIT flag is set, then *LTags* and *RTags* can always use
5470 the index/key split scheme even when not otherwise required for virtually registered Memory
5471 Regions (through the RI_MEM_FLAGS_CONSUMER_OWNS_KEY flag) and “wide”
5472 Memory Windows. Even when not set, *Ltags* associated with fast memory registers and *RTags*
5473 associated with Narrow Memory Windows use the index/key split scheme.

5474 If the RI_RNIC_CONSUMER_x_OPAQUE flags are set, the corresponding pass-through flags
5475 from the WR are returned in the WC. Note that RI_RNIC_CONSUMER2_OPAQUE is used to

5476 support local solicited behavior and has the same definition as
5477 RI_RNIC_LOCAL_SOLICITED_OPAQUE.

5478 If the RI_RNIC_LOCAL_SOLICITED_OPAQUE flag is set, then a pass-through flag from the
5479 WR is returned in the WC. This pass-through flag is to be used by the AL to emulate local
5480 solicited behavior (i.e., generate an event). If the RI_RNIC_LOCAL_SOLICITED_EVENT flag
5481 is set, then the RNIC actually generates a completion event when completions with the local
5482 solicited flag are generated. If neither flag is set, then local solicited behavior is not supported.

5483 InfiniBand does not allow for out-of-order receives for the Reliable Connected (RC) transport
5484 service type, so strictly speaking SRQ sequential *versus* arrival dequeuing order does not directly
5485 apply to RC SRQs. An out-of-order receive for an RC SRQ does not dequeue WQEs for either
5486 the arriving message (unlike iWARP) or for messages in the sequence gap. InfiniBand also
5487 allows UD SRQs, which don't have any ordering guarantees due to the nature of UD. So an out-
5488 of-order receive in this case acts more like arrival ordering, though there is no concept of a
5489 sequence gap for UD. Because of these two behaviors, InfiniBand RNICs supporting SRQ
5490 always clear the RI_RNIC_SEQ_DEQUEUE flag to indicate that out-of-order receives do not
5491 cause the dequeue of additional WQEs in the sequence gap.

5492 An iWARP implementation that does not go beyond the RDMAC verbs specification will have a
5493 *max_sgl_rdma_read* value of one.

5494 Inline data support is indicated by a non-zero *max_unregistered_inline* value. Data may be
5495 inlined if the WR has the inline flag on (it's only a hint) and the message is less than or equal to
5496 *max_unregistered_inline* bytes in length.

5497 The definition of RI_MAX_VENDOR_STRING_LEN is OS-dependent.

5498 **CONSUMER USAGE**

5499 None.

5500 **IMPLEMENTATION NOTES**

5501 None.

5502 **SEE ALSO**

5503 [*ri_rnic_query\(\)*](#), [*ri_ib_rnic_attr_t*](#), [*ri_iwarp_rnic_attr_t*](#)

ri_iwarp_rnic_attr_t

5504

5505 **NAME**

5506 ri_iwarp_rnic_attr_t – iWARP-specific RNIC attributes data structures

5507 **SYNOPSIS**

```
5508 typedef enum ri_iwarp_rnic_flags ri_iwarp_rnic_flags_t;
5509 enum ri_iwarp_rnic_flags {
5510     RI_IWARP_CAN_MODIFY_IRD = 1 << 0,
5511     RI_IWARP_CAN_INCREASE_ORD = 1 << 1,
5512
5513     /* supported IP transports */
5514     RI_IWARP_TCP_SUPPORTED = 1 << 2,
5515     RI_IWARP_SCTP_SUPPORTED = 1 << 3,
5516
5517     /* if IETF and !PERMISSIVE, RNIC is non-permissive */
5518     RI_IWARP_IETF_PERMISSIVE = 1 << 4,
5519
5520     /* marker & CRC preference for MPA negotiation on TCP */
5521     RI_IWARP_PREFER_MARKERS = 1 << 5,
5522     RI_IWARP_PREFER_CRC = 1 << 6,
5523
5524     /* modify/query QP has KEEPALIVE option support */
5525     RI_IWARP_KEEPALIVE_SUPPORTED = 1 << 7
5526 };
5527
5528 typedef struct ri_iwarp_port_attr ri_iwarp_port_attr_t;
5529 struct ri_iwarp_port_attr {
5530     struct in6_addr *ipv6_addrs;
5531     struct in_addr *ipv4_addrs;
5532     uint_t port_num;
5533     uint_t num_ipv4_addrs;
5534     uint_t num_ipv6_addrs;
5535 };
5536
5537 typedef struct ri_iwarp_rnic_attr ri_iwarp_rnic_attr_t;
5538 struct ri_iwarp_rnic_attr {
5539     ri_iwarp_port_attr_t *port_attrs;
5540     uint_t protocol_version; /* RDMAC = 0, IETF = 1 */
5541     uint_t num_ports;
5542     ri_iwarp_rnic_flags_t iwarp_rnic_flags;
5543 };
```

5544 **DESCRIPTION**

5545 The *ri_iwarp_rnic_attr_t* structure defines the iWARP transport-specific RNIC attributes based
5546 on the RDMAC 1.0 verb specification [VERBS-RDMAC] with small additions made to account
5547 for IETF differences and RNICPI additions.

5548 The *protocol_version* field is zero for an RDMAC RNIC and one for an RNIC implementing the
5549 current IETF specifications. If the RNIC is IETF-compliant, then the
5550 RI_IWARP_IETF_PERMISSIVE flag distinguishes between a permissive and non-permissive
5551 implementation. The IP transports that an RNIC supports are signaled in the
5552 RI_IWARP_TCP_SUPPORTED and RI_IWARP_SCTP_SUPPORTED flags. For TCP, the

5553 MPA marker and CRC preferences are signaled in the `RI_IWARP_PREFER_MARKERS` and
5554 `RI_IWARP_PREFER_CRC`; if a given flag is not set, then that item (markers or CRC) is not
5555 preferred.

5556 If port information has not been requested in `ri_rnic_query()`, the `port_attrs` field will be set by
5557 the VP to NULL. However, the `num_ports` field is still filled in. If port information has been
5558 requested, then the `port_attrs` field points to an array of `ri_iwarp_port_attr_t` structs. That struct
5559 in turn describes the port number as well as having lists of the IPv4 and IPv6 addresses assigned
5560 to that port. If no addresses of a particular type are assigned to the port, then the pointer to the
5561 list of addresses will be NULL and the count of those addresses will be zero.

5562 **CONSUMER USAGE**

5563 None.

5564 **IMPLEMENTATION NOTES**

5565 None.

5566 **SEE ALSO**

5567 `ri_rnic_query()`, `ri_rnic_attr_t`

5568

ri_ib_rnic_attr_t

5569 **NAME**

5570 ri_ib_rnic_attr_t – InfiniBand-specific RNIC attributes data structures

5571 **SYNOPSIS**

```
5572 typedef uint8_t ri_sl_t; /* Service Level (4 bits) */
5573 typedef uint8_t ri_lmc_t; /* LID Mask Control (7 bits) */
5574 typedef uint16_t ri_lid_t; /* LID address */
5575 typedef uint16_t ri_pkey_t; /* P_Key */
5576
5577 typedef struct gid ri_gid_t;
5578 struct ri_gid { /* GID address */
5579     uint64_t gid_prefix;
5580     uint64_t gid_guid;
5581 };
5582
5583 typedef enum ri_ib_port_state ri_ib_port_state_t;
5584 enum ri_ib_port_state { /* link/port states */
5585     RI_IB_LINK_DOWN = 1 << 1,
5586     RI_IB_LINK_INITIALIZE = 1 << 2,
5587     RI_IB_LINK_ARM = 1 << 3,
5588     RI_IB_LINK_ACT_DEFER = 1 << 4, /* transient */
5589     RI_IB_LINK_ACTIVE = 1 << 5 /* "up" state */
5590 };
5591 /* how IB verbs define port up and down */
5592 #define RI_IB_PORT_DOWN \
5593     (RI_IB_LINK_DOWN|RI_IB_LINK_INITIALIZE|RI_IB_LINK_ARM)
5594 #define RI_IB_PORT_UP \
5595     (RI_IB_LINK_ACT_DEFER|RI_IB_LINK_ACTIVE)
5596
5597 typedef enum ri_ib_mtu ri_ib_mtu_t;
5598 enum ri_ib_mtu { /* MTU sizes */
5599     RI_IB_MTU_256 = 1,
5600     RI_IB_MTU_512,
5601     RI_IB_MTU_1K,
5602     RI_IB_MTU_2K,
5603     RI_IB_MTU_4K
5604 };
5605
5606 /*
5607  * port capability flags
5608  * bit positions match portinfo mask
5609  */
5610 typedef enum ri_ib_port_flags ri_ib_port_flags_t;
5611 enum ri_ib_port_flags {
5612     RI_IB_PORT_SM = 1 << 0,
5613     RI_IB_PORT_SM_DISABLED = 1 << 10,
5614     RI_IB_PORT_SNMP_TUNNEL = 1 << 17,
5615     RI_IB_PORT_DEV_MGT = 1 << 19,
5616     RI_IB_PORT_VENDOR_CLASS = 1 << 20,
5617     RI_IB_PORT_CLIENT_REREG = 1 << 25
5618 };
```

```

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 */

```

```

5673     uint64_t      node_guid; /* Node GUID */
5674     uint64_t      system_image_guid; /* zero if not supported */
5675     ri_ib_port_attr_t *port_attrs;
5676     ri_ib_rnic_flags_t ib_rnic_flags;
5677     uint_t        max_addr_hdls; /* max UD Addr Handles */
5678
5679     /* multicast fields are zero, if multicast unsupported */
5680     uint_t        max_mcgs; /* max num multicast groups */
5681     uint32_t      max_multicast_qps; /* max MCG attachable QPs */
5682     uint32_t      max_qps_per_mcg; /* max num of QPs per MCG */
5683
5684     uint16_t      partition_table_size; /* size must be >= 1 */
5685     uint8_t       num_ports;
5686     uint8_t       local_ack_delay; /* 5 bits */
5687 };

```

5688 **DESCRIPTION**

5689 The *ri_iwarp_rnic_attr_t* structure defines the InfiniBand transport-specific RNIC attributes
5690 based on the 1.1 and 1.2 IBTA verb specifications. No attempt is made to distinguish earlier
5691 versions of the IBTA specifications. Further, no information is returned about the Reliable
5692 Datagram, Unreliable Connected, or Raw transport service types, which are all outside the scope
5693 of RNICPI.

5694 The *port_attrs* field is a pointer to an array (*num_ports* long) of *ri_ib_port_attr_t* structs. An
5695 *ri_rnic_query()* invocation which does not retrieve port information will return *port_attrs* set to
5696 NULL by the VP. However, *num_ports* will still be valid to indicate how many ports are on the
5697 IB RNIC. The *ri_ib_port_attr_t* structure contains pointers to the source GID table
5698 (*source_gid_table* field) and the *P_Key* table (*partition_table* field). Note that the *P_Key* table
5699 for each port is of identical size which is given in the *partition_table_size* field of the
5700 *ri_iwarp_rnic_attr_t* structure.

5701 **CONSUMER USAGE**

5702 None.

5703 **IMPLEMENTATION NOTES**

5704 None.

5705 **SEE ALSO**

5706 *ri_rnic_query()*, *ri_rnic_attr_t*

ri_ltag_t, ri_rtag_t

5707

5708 NAME

5709 ri_ltag_t, ri_rtag_t – *LTag/Rtag* type definitions

5710 SYNOPSIS

```
5711 typedef uint32_t ri_ltag_t;  
5712 typedef uint32_t ri_rtag_t;
```

5713 DESCRIPTION

5714 These types describe the keys issued by the IHV to enable local (*LTag*) and remote (*RTag*)
5715 memory access. These map to InfiniBand *L_Keys* and *R_Keys*, and to iWARP STags.

5716 Selection of the values for an *LTag* is frequently divided between the Consumer and the RNIC.
5717 The RNIC controlled portion is referred to as the “index”, and the Consumer controlled portion
5718 as the “key”. This is always the case for iWARP providers, and for all InfiniBand Memory
5719 Regions that can be fast registered. It may also be true for other InfiniBand Memory Regions
5720 and Memory Windows.

5721 iWARP IHVs will use the same underlying type and value for the “*LTag*” and “*RTag*” of a
5722 Memory Region, and even potentially for the Memory Region/Window handle. These are
5723 defined as distinct types to support mapping to IB. Additionally, even for iWARP, Memory
5724 Region STags and Memory Window STags are clearly distinct types, even though they are both
5725 referred to as STags. An *LTag* maps to a Memory Region STag. An *RTag* maps either to a
5726 Memory Window STag or to the STag of a Memory Region that has been enabled for remote
5727 access. With iWARP the usage of an *LTag* and *RTag* may be different, but they are numbered as
5728 though they were a uniform resource. The index portion identifies the Memory Region or
5729 Memory Window for either. This is not necessarily true for IB.

5730 Consumer editing of the key portion should be made to the *LTag* for Memory Regions, and to
5731 the *RTag* for Memory Windows.

5732 CONSUMER USAGE

5733 None.

5734 IMPLEMENTATION NOTES

5735 None.

5736 SEE ALSO

5737 None.

ri_stag_state_t

5738

5739 **NAME**

5740 ri_stag_state_t – state of an *Ltag* or *Rtag*

5741 **SYNOPSIS**

```
5742 enum ri_stag_state {  
5743     RI_STAG_STATE_INVALID,  
5744     RI_STAG_STATE_VALID  
5745 };  
5746 typedef enum ri_stag_state ri_stag_state_t;
```

5747 **DESCRIPTION**

5748 Both *Ltags* and *Rtags* can be in invalid or valid states.

5749 **CONSUMER USAGE**

5750 None.

5751 **IMPLEMENTATION NOTES**

5752 None.

5753 **SEE ALSO**

5754 None.

ri_phys_buf_t

5755

5756 NAME

5757 ri_phys_buf_t – descriptor of a physical memory block

5758 SYNOPSIS

```
5759 struct ri_phys_buf {  
5760     ri_phys_addr_t    phys_address;  
5761     ri_phys_buf_len_t phys_len;  
5762 };  
5763 typedef struct ri_phys_buf ri_phys_buf_t;
```

5764 DESCRIPTION

5765 An *ri_phys_buf_t* defines a single entry in a physical buffer list when the length is supplied on a
5766 per-entry basis. Typically, this is when the associated parameter/field for the *pbl* entry length is
5767 zero. When there is a non-zero *pble_len* the *ri_phys_buf_t* pointer is actually an *ri_phys_addr_t*
5768 pointer.

5769 Both *ri_phys_addr_t* and *ri_phys_buf_len_t* are OS-dependent types that may vary in size.

5770 CONSUMER USAGE

5771 None.

5772 IMPLEMENTATION NOTES

5773 None.

5774 SEE ALSO

5775 None.

5776

ri_mem_attr_t

5777 **NAME**

5778 ri_mem_attr_t – boolean attributes for Memory Regions/Windows

5779 **SYNOPSIS**

```

5780 enum ri_mem_attr {
5781     RI_MEM_FLAGS_LOCAL_READ           = 1 << 0,
5782     RI_MEM_FLAGS_LOCAL_WRITE        = 1 << 1,
5783     RI_MEM_FLAGS_LOCAL              = (RI_MEM_FLAGS_LOCAL_READ |
5784                                         RI_MEM_FLAGS_LOCAL_WRITE),
5785
5786     RI_MEM_FLAGS_REMOTE_WRITE       = 1 << 2,
5787     RI_MEM_FLAGS_REMOTE_READ        = 1 << 3,
5788     RI_MEM_FLAGS_REMOTE             = (RI_MEM_FLAGS_REMOTE_WRITE |
5789                                         RI_MEM_FLAGS_REMOTE_READ),
5790     RI_MEM_FLAGS_ATOMICS            = 1 << 4,
5791     RI_MEM_FLAGS_BINDING             = 1 << 5,
5792     RI_MEM_FLAGS_CONSUMER_OWNS_KEY = 1 << 6,
5793     RI_MEM_FLAGS_ZBVA               = 1 << 7,
5794     RI_MEM_FLAGS_NONCOHERENT        = 1 << 8,
5795     RI_MEM_FLAGS_USER_VA            = 1 << 9,
5796     RI_MEM_FLAGS_SHARED_REG_IOVA    = 1 << 10
5797 };
5798 typedef ri_mem_attr ri_mem_attr_t;

```

5799 **DESCRIPTION**

5800 This type provides a set of flags that specify boolean characteristics of Memory Regions and/or
5801 Memory Windows.

5802 Most of the flags specify the access permissions for the Memory Region/Window:

5803	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.
5804		
5805		
5806		
5807		
5808	RI_MEM_FLAGS_LOCAL_WRITE	Permission flag that enables local write access.
5809	RI_MEM_FLAGS_REMOTE_WRITE	Permission flag that enables remote write access.
5810	RI_MEM_FLAGS_REMOTE_READ	Permission flag that enables remote read access.
5811	RI_MEM_FLAGS_ATOMIC	Permission flag that enables remote atomic access. This flag is irrelevant if the transport does not support atomic operations.
5812		
5813		
5814	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.
5815		
5816		
5817		

5818	RI_MEM_FLAGS_CONSUMER_OWNS_KEY	The <i>LTag/RTags</i> 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.
5819		
5820		
5821		
5822	RI_MEM_FLAGS_ZBVA	Flag that indicates that the Memory Region (or Window) uses zero-based addressing.
5823		
5824	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.
5825		
5826		
5827		
5828		
5829		
5830		
5831		
5832		
5833	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.
5834		
5835		
5836		
5837		
5838		
5839	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 <i>ri_mr_reg_shared()</i> .
5840		
5841		
5842		
5843		
5844		
5845		
5846	CONSUMER USAGE	
5847		The Consumer creates the settings desired for a Memory Region/Window by bitwise OR'ing the entire set of desired permissions and attributes.
5848		
5849	IMPLEMENTATION NOTES	
5850		None.
5851	SEE ALSO	
5852		None.

ri_mr_reg_attr_t, ri_mr_rereg_t

5853

5854 NAME

5855 ri_mr_reg_attr_t – Memory Region creation attributes

5856 ri_mr_rereg_t – Reregistration request types

5857 SYNOPSIS

```
5858 typedef struct ri_mr_reg_attr ri_mr_reg_attr_t;
5859 struct ri_mr_reg_attr {
5860     ri_addr_t      va;
5861     ri_length_t    length;
5862     ri_mem_attr_t  mem_attr;
5863     ri_pd_handle_t pd;
5864 };
5865 typedef enum ri_mr_rereg ri_mr_rereg_t;
5866 enum ri_mr_rereg {
5867     RI_MR_REREG_TRANSLATE = 1<<0,
5868     RI_MR_REREG_PD       = 1<<1,
5869     RI_MR_REREG_ACCESS   = 1<<2
5870 };
```

5871 DESCRIPTION

5872 The attributes required when creating a Memory Region.

5873 *va* Virtual Address of the region and/or the address of the memory to be registered in the
5874 Consumer's current virtual memory map.

5875 *length* Length in bytes of the Memory Region.

5876 *attr* Boolean attributes for the Memory Region. See [ri_mem_attr_t](#).

5877 *pd* Handle of the Protection Domain.

5878 The *ri_mr_rereg* attributes are used on reregister calls to indicate which Memory Region
5879 attributes are being changed:

5880 RI_MR_REREG_TRANSLATE The address translation for the Memory Region may be
5881 changed by this reregister call.

5882 RI_MR_REREG_PD The Protection Domain for the Memory Region may be
5883 changed by this reregister call.

5884 RI_MR_REREG_ACCESS The access rights for the Memory Region may be changed by
5885 this reregister call.

5886 CONSUMER USAGE

5887 InfiniBand 1.1 adapters may enable a larger region than requested. Consumers should use
5888 [ri_mr_query\(\)](#) to determine the actual region enabled if adjacent memory structures must be
5889 protected from RNIC access.

5890 IMPLEMENTATION NOTES

5891 None.

5892 **SEE ALSO**
5893 *ri_mr_query(), ri_mem_attr_t*

5894

ri_mr_reg_phys_attr_t

5895 **NAME**

5896 ri_mr_reg_phys_attr_t – Memory Region physical creation attributes

5897 **SYNOPSIS**

```
5898 typedef struct ri_mr_reg_phys_attr ri_mr_reg_phys_attr_t;
5899
5900 struct ri_mr_reg_phys_attr {
5901     ri_mr_reg_attr_t    common;
5902     void                *phys_buf_list;
5903     uint_t              nbufs;
5904     uint_t              pble_size;
5905     uint_t              address_offset;
5906 };
```

5907 **DESCRIPTION**

5908 The attributes required when creating a Memory Region using physical buffer lists.

5909 *common* Attributes that apply to creating any Memory Region.

5910 *phys_buf_list*

5911 If *pble_size* is non-zero, this is an *ri_phys_addr_t* pointer to an array of physical addresses
5912 for blocks/pages that are all *pble_size* bytes long. If *pble_size* is zero, then this is a
5913 *ri_phys_buf_t* pointer to an array of block specifications that each contain the physical
5914 address and block size. This field is not returned on a query.

5915 *nbufs* Number of buffers in *phys_buf_list*.

5916 *pble_size* Size of each buffer in *phys_buf_list*, if non-zero. If zero, indicates that the lengths are in the
5917 *phys_buf_list*.

5918 *offset* Offset within the first physical buffer corresponding to the starting virtual address.

5919 **CONSUMER USAGE**

5920 Consumers are advised to use *ri_mr_query()* to determine the actual boundaries of the Memory
5921 Region created.

5922 **IMPLEMENTATION NOTES**

5923 None.

5924 **SEE ALSO**

5925 *ri_mr_reg_attr_t*

ri_mw_type_t

5926

5927 NAME

5928 ri_mw_type_t – type of Memory Window: narrow/wide

5929 SYNOPSIS

```
5930 enum ri_mw_type {  
5931     RI_MW_TYPE_1, RI_MW_TYPE_WIDE=RI_MW_TYPE_1,  
5932     RI_MW_TYPE_2, RI_MW_TYPE_NARROW=RI_MW_TYPE_2  
5933 };  
5934 typedef ri_mw_type ri_mw_type_t;
```

5935 DESCRIPTION

5936 Memory Windows must be created to support either type 1 (wide) or type 2 (narrow) binding.

5937 Once bound, a wide-bind window is valid on all QPs with the matching protection domain.
5938 Narrow-bind windows are valid only on the QP where the bind operation was performed. The
5939 RNIC attributes indicate which types are supported. Generally, wide-binding **MUST** be
5940 supported on InfiniBand and **MAY** be supported on iWARP. Narrow binding **MAY** be
5941 supported on InfiniBand and **MUST** be supported on iWARP.

5942 Making a window narrow-bound typically simplifies internal processing for the RNIC,
5943 particularly when dealing with invalidations. Therefore, when both types are supported the
5944 Consumer is advised to use narrow-bind windows unless there is a specific application need for
5945 wide-bind windows.

5946 CONSUMER USAGE

5947 None.

5948 IMPLEMENTATION NOTES

5949 None.

5950 SEE ALSO

5951 None.

ri_qp_attr_t

5952

5953 **NAME**

5954 ri_qp_attr_t – QP attributes

5955 **SYNOPSIS**

```
5956 enum ri_svc_type
5957 {
5958     RI_SERVICE_RELIABLE_CONNECTION,
5959     RI_SERVICE_UNRELIABLE_DATAGRAM
5960 };
5961 typedef enum ri_svc_type ri_svc_type_t;
5962
5963 /* Some states are not valid for RDMAC/IB NICs. Attempts to modify the
5964 state attribute will fail with RI_RET_INVALID_STATE if not appropriate
5965 for the technology */
5966
5967 enum ri_qp_state
5968 {
5969     RI_QP_STATE_IDLE,
5970     RI_QP_STATE_READY_TO_SEND,
5971     RI_QP_STATE_ERROR,
5972
5973     /* The following states are valid only for iWARP */
5974     RI_QP_STATE_TERMINATE,
5975     RI_QP_STATE_CLOSING,
5976
5977     /* The following states are valid only for IB */
5978     RI_QP_STATE_SEND_QUEUE_ERROR,
5979     RI_QP_STATE_INITIALIZED,
5980     RI_QP_STATE_READY_TO_RECEIVE,
5981     RI_QP_STATE_SEND_QUEUE_DRAINING,
5982     RI_QP_STATE_SEND_QUEUE_DRAINED,
5983     RI_QP_STATE_TIMEWAIT
5984 };
5985 typedef enum ri_qp_state ri_qp_state_t;
5986
5987 struct ri_qp_attr
5988 {
5989     uint32_t qp_id;
5990     union {
5991         ri_cq_handle_t scq_handle;
5992         ri_os_data_t scq_os_data;
5993     } scq;
5994     union {
5995         ri_cq_handle_t rcq_handle;
5996         ri_os_data_t rcq_os_data;
5997     } rcq;
5998     union {
5999         ri_srq_handle_t srq_handle;
6000         ri_os_data_t srq_os_data;
6001     } srq;
6002     ri_pd_handle_t pd;
```

```

6003         uint32_t         sq_size;
6004         uint32_t         rq_size;
6005         ri_boolean_t     srq_valid;
6006         uint32_t         sq_max_sges;
6007         uint32_t         sq_max_sges_rdmaw;
6008         uint32_t         rq_max_sges;
6009         ri_qp_state_t    state;
6010         ri_boolean_t     enable_rdma_read;
6011         ri_boolean_t     enable_rdma_write;
6012         uint32_t         max_rdma_read_outgoing;
6013         uint32_t         max_rdma_read_incoming;
6014         ri_boolean_t     enable_stag0_fastreg;
6015         union {
6016             ri_iwarp_qp_attr_t    qp_iwarp;
6017             ri_ib_qp_attr_t       qp_ib;
6018         } qp_xport;
6019     };
6020     typedef struct ri_qp_attr ri_qp_attr_t;
6021
6022     struct ri_mpa_attr
6023     {
6024         ri_boolean_t     rcv_marker_enabled;
6025         ri_boolean_t     xmit_marker_enabled;
6026         ri_boolean_t     crc_enabled;
6027         uint32_t         ddp_rdma_version; /* 0 or 1 */
6028     };
6029     typedef struct ri_mpa_attr ri_mpa_attr_t;
6030
6031     struct ri_iwarp_qp_attr
6032     {
6033         ri_boolean_t         enable_bind;
6034         char                 terminate_buffer[52];
6035         uint32_t             terminate_msg_length;
6036         ri_boolean_t         is_terminate_local;
6037         ri_llp_stream_handle_t llp_stream_handle;
6038         ri_mpa_attr_t        mpa_attributes;
6039         char                 *stream_msg_buf;
6040         uint32_t             stream_msg_buf_length;
6041         uint32_t             rq_hiwat;
6042         ri_boolean_t         rq_armed;
6043     };
6044     typedef struct ri_iwarp_qp_attr ri_iwarp_qp_attr_t;
6045
6046     struct ri_gen_address_vector
6047     {
6048         uint8_t             sl;
6049         uint8_t             send_global_routing_hdr_flg;
6050         uint8_t             hop_limit;
6051         uint8_t             source_gid_index;
6052         uint32_t            flow_label;
6053         ri_gid_t            dgid;
6054         ri_lid_t            dlid;
6055         uint8_t             traffic_class;
6056         uint8_t             max_static_rate;

```

```

6057     uint8_t    source_path_bits;
6058 };
6059 typedef struct ri_gen_address_vector ri_gen_address_vector_t;
6060
6061 struct ri_path_address_info
6062 {
6063     /* For UD QPs, pkey_index and physical_port are alone valid */
6064     uint16_t    pkey_index;
6065     uint8_t     physical_port;
6066     uint16_t    path_mtu;
6067     uint8_t     local_ack_timeout;
6068     uint8_t     retry_count;
6069     uint8_t     rnr_retry_count;
6070     ri_gen_address_vector_t    av;
6071 };
6072 typedef struct ri_path_address_info ri_path_address_info_t;
6073
6074 enum ri_path_migration_state
6075 {
6076     RI_MIG_STATE_MIGRATED,
6077     RI_MIG_STATE_REARM,
6078     RI_MIG_STATE_ARMED
6079 };
6080 typedef enum ri_path_migration_state ri_path_migration_state_t;
6081
6082 struct ri_ib_qp_attr
6083 {
6084     /* Common to RC and UD */
6085     uint32_t    send_psn;
6086     ri_boolean_t    signal_completions;
6087     ri_svc_type_t    trans_type;
6088     uint32_t    enable_async_notify_for_sqd;
6089     ri_path_address_info_t    primary_av;
6090     /* RC-specific attributes */
6091     uint32_t    recv_psn;
6092     uint32_t    dest_qpnr;
6093     uint32_t    min_rnrnak_timer;
6094     ri_path_address_info_t    alternate_av;
6095     ri_path_migration_state_t    mig_state;
6096     ri_boolean_t    enable_atomic;
6097     /* UD-specific attributes */
6098     ri_qp_state_t    current_state;
6099     uint32_t    recv_qkey;
6100 };
6101 typedef struct ri_ib_qp_attr ri_ib_qp_attr_t;

```

6102 DESCRIPTION

6103 The following table outlines the validity of each attribute in *ri_qp_create()* and *ri_qp_modify()*.
6104 All attributes are valid in *ri_qp_query()*. If an attribute is not defined as valid in *ri_qp_create()*,
6105 it must not be passed in the call and will be ignored by the implementation. If an attribute is
6106 defined as modifiable, it may be accepted by the implementation depending on the state of the
6107 QP.

QP Attributes	Description	Valid in <i>ri_qp_create()</i>	Modifiable?
Common Transport-Neutral QP Attributes			
<i>qp_id</i>	Local QP ID or number.	No	No
<i>scq</i>	Handle to the CQ associated with the Send Queue when passed in <i>ri_qp_create()</i> . Contains <i>os_data</i> of the CQ upon return from <i>ri_qp_query()</i> .	Yes	No
<i>rcq</i>	Handle to the CQ associated with the Receive Queue when passed in <i>ri_qp_create()</i> . Contains <i>os_data</i> of the CQ upon return from <i>ri_qp_query()</i> .	Yes	No
<i>sq_size</i>	Number of outstanding requests supported on the Send Queue.	Yes	Yes
<i>rq_size</i>	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
<i>srq</i>	Handle to the SRQ associated with this QP when passed in <i>ri_qp_create()</i> . Contains <i>os_data</i> of the SRQ upon return from <i>ri_qp_query()</i> . This field is valid only if <i>srq_valid</i> is TRUE.	Yes	No
<i>srq_valid</i>	TRUE if QP is associated with an SRQ. This must be set in <i>ri_qp_create()</i> if the QP needs to be associated with an SRQ.	Yes	No
<i>sq_max_sges</i>	Number of scatter/gather entries supported on Work Requests submitted to the Send Queue.	Yes	No

QP Attributes	Description	Valid in <i>ri_qp_create()</i>	Modifiable?
<i>sq_max_sges_rdmaw</i>	Number of scatter/gather entries supported on RDMA-write Work Requests submitted to the Send Queue. IB implementations must return the same value as <i>sq_max_sges</i> in <i>ri_qp_query()</i> and may need to allocate more resources to make this equal to <i>sq_max_sges</i> upon create and modify.	Yes	No
<i>rq_max_sges</i>	Number of scatter/gather entries supported on Work Requests submitted to the Receive Queue.	Yes	No
<i>state</i>	Current QP state (see <i>ri_qp_state_t</i>). If passed to <i>ri_qp_modify()</i> , this is the next desired state for the QP.	No	Yes
<i>enable_rdma_read</i>	Enable RDMA read operations on the QP.	Yes (iWARP) No (IB)	No (iWARP) Yes (IB)
<i>enable_rdma_write</i>	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)
<i>max_rdma_read_outgoing</i>	Number of outgoing RDMA reads (and atomic operations for IB implementations) outstanding on the QP.	Yes (iWARP) No (IB)	Yes
<i>max_rdma_read_incoming</i>	Number of incoming RDMA reads (and atomic operations for IB implementations) outstanding on the QP.	Yes (iWARP) No (IB)	Yes
<i>pd</i>	Handle to the PD associated with the QP.	Yes	No
<i>enable_stag0_fastreg</i>	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			
<i>enable_bind</i>	Enable Memory Window bind operations.	Yes	No
<i>terminate_buffer</i>	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
<i>terminate_msg_length</i>	Number of bytes in <i>terminate_buffer</i> that contains a valid terminate message.	No	No
<i>is_terminate_local</i>	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 <i>ri_qp_create()</i>	Modifiable?
<i>llp_stream_handle</i>	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 <i>ri_qp_query()</i> and it may be different from what was passed by the Consumer in <i>ri_qp_modify()</i> . If this attribute has never been modified, <i>ri_qp_query()</i> must return NULL for this attribute. After passing this to a successful <i>ri_qp_modify()</i> operation, the Consumer must not use it to perform send/receive operations directly on the TCP connection, instead <i>ri_qp_postsq()</i> , <i>ri_qp_postrq()</i> , 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 <i>ri_qp_modify()</i> and are implementation-dependent.	No	Yes
<i>recv_marker_enable</i>	Enable/disable accepting MPA frames with markers on the connection associated with the QP. <i>ri_qp_query()</i> reports the preference (on/off). The Consumer may attempt to override the preference through <i>ri_qp_modify()</i> 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
<i>xmit_marker_enable</i>	Enable/disable send MPA frames with markers on the connection associated with the QP. <i>ri_qp_query()</i> reports the preference (on/off). The Consumer may attempt to override the preference through <i>ri_qp_modify()</i> 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
<i>crc_enable</i>	Enable/disable transmitting CRCs in outgoing MPA frames and checking CRCs in incoming MPA frames.	Yes	Yes

QP Attributes	Description	Valid in <i>ri_qp_create()</i>	Modifiable?
<i>stream_msg_buf</i>	Pointer to the last streaming mode message on the connection. Valid only for <i>ri_qp_modify()</i> if moving from Idle to RTS. A value of NULL is returned for this attribute in <i>ri_qp_query()</i> . 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
<i>stream_msg_buf_length</i>	Number of bytes in <i>stream_msg_buf</i> containing the last streaming mode message.	No	Yes
<i>ddp_rdma_version</i>	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
<i>rq_hiwat</i>	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 <i>rq_hiwat</i> in <i>ri_qp_modify()</i> means that the asynchronous event is triggered when the QP consumes its first buffer from SRQ since last arm. If zero <i>rq_hiwat</i> is passed in <i>ri_qp_create()</i> , RQ limit arming must be disabled.	Yes	Yes
<i>rq_armed</i>	RQ limit armed indicator.	No	Yes
IB-Specific QP Attributes			
<i>current_state</i>	Current state of the QP. Valid only if <i>state</i> is RI_QP_STATE_READY_TO_SEND.	No	Yes
<i>recv_qkey</i>	<i>Q_Key</i> for the UD Receive Queue.	No	Yes
<i>dest_qpn</i>	Destination QP number for RC and UC QPs.	No	Yes
<i>min_rnrnak_timer</i>	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 <i>ri_qp_create()</i>	Modifiable?
<i>signal_completions</i>	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
<i>trans_type</i>	The transport service type requested for this QP. See <i>ri_svc_type_t</i> .	Yes	No
<i>enable_async_notify_for_sqd</i>	Enable or disable Send Queue Drained Asynchronous Affiliated Event Notification. This modifier is only applicable when the QP state chosen is SQD.	No	Yes
<i>mig_state</i>	Path migration state. See <i>ri_path_migration_state_t</i> .	No	Yes
<i>send_psn</i>	Packet Sequence Number on the Send Queue.	No	Yes
<i>recv_psn</i>	Packet Sequence Number on the Receive Queue.	No	Yes
<i>enable_atomic</i>	Enable atomic operations on the QP.	No	Yes
<i>pkey_index</i>	Primary Key Index.	No	Yes
<i>physical_port</i>	Number of the local port to put the QP on.	No	Yes
<i>The following attributes are valid only for RC.</i>			
<i>sl</i>	Service Level.	No	Yes
<i>send_global_routing_hdr_flg</i>	Send Global Routing Header Flag.	No	Yes
<i>hop_limit</i>	Hop Limit.	No	Yes
<i>source_gid_index</i>	Source GID Index.	No	Yes
<i>flow_label</i>	Flow Label.	No	Yes
<i>dgid</i>	Destination GID.	No	Yes
<i>Dlid</i>	Destination LID.	No	Yes
<i>traffic_class</i>	Traffic Class.	No	Yes
<i>max_static_rate</i>	Maximum Static Rate.	No	Yes
<i>path_mtu</i>	Path MTU.	No	Yes
<i>local_ack_timeout</i>	Local Ack Timeout.	No	Yes
<i>retry_count</i>	Retry Count.	No	Yes
<i>rnr_retry_count</i>	RNR Retry Count.	No	Yes
<i>source_path_bits</i>	Source Path Bits.	No	Yes

6109 **CONSUMER USAGE**

6110 None.

6111 **IMPLEMENTATION NOTES**

6112 *rq_size* can be used to guide allocation of per-QP resources to track receive Work Requests
6113 allocated to the QP. For example, an RNIC might allocate an RQ even when an SRQ is assigned,
6114 and copy the receive Work Request (or a pointer to it) when it is allocated to the QP.

6115 The RNIC MAY simply ignore this field when an SRQ is specified.

6116 **SEE ALSO**

6117 [*ri_qp_create\(\)*](#), [*ri_qp_modify\(\)*](#)

ri_srq_attr_t

6118

6119 NAME

6120 ri_srq_attr_t – SRQ attributes

6121 SYNOPSIS

```
6122 struct ri_srq_attr {
6123     uint32_t      max_wrs;
6124     uint32_t      max_sgl;
6125     uint32_t      lowat;
6126     ri_pd_handle_t pd;
6127     ri_boolean_t  srq_armed;
6128 };
6129 typedef struct ri_srq_attr ri_srq_attr_t;
```

6130 DESCRIPTION

6131 *max_wrs* On input, maximum number of Work Requests the Consumer expects to submit. On output,
6132 maximum number the Consumer is allowed to submit.

6133 *max_sgl* On input, maximum number of scatter/gather elements the Consumer expects to submit. On
6134 output, maximum number the Consumer is allowed to submit. This will be greater than or
6135 equal to the number requested for a successful creation. This field is not valid for
6136 *ri_srq_modify()*.

6137 *lowat* A low water mark of available number of posted WQEs. When the number available is
6138 lower than or equal to this, it will trigger an asynchronous event to the Consumer. The
6139 Consumer will have to re-arm explicitly for subsequent notifications. At most one event is
6140 generated per arming. The SRQ is automatically armed if *lowat* is non-zero in
6141 *ri_srq_create()*.

6142 *pd* Handle of the PD this SRQ is to be associated with. This is not valid for *ri_srq_modify()*.

6143 *srq_armed* TRUE when SRQ Limit is armed. This is an output only field that is ignored on input.

6144 Attributes of a Shared Receive Queue that can be used as an IN OUT or OUT parameter.

6145 It is used as an IN OUT parameter when created or modifying an SRQ. The desired attributes are
6146 passed in. On successful completion, the actual results are passed out.

6147 CONSUMER USAGE

6148 None.

6149 IMPLEMENTATION NOTES

6150 None.

6151 SEE ALSO

6152 *ri_srq_create()*, *ri_srq_modify()*

6153

ri_wr_t

6154 **NAME**

6155 ri_wr_t – Work Request structures for the Send Queue

6156 **SYNOPSIS**

```
6157 enum ri_wr_type{
6158     RI_WR_TYPE_SEND,
6160     RI_WR_TYPE_SEND_INVALIDATE,
6162     RI_WR_TYPE_RDMA_WRITE,
6163     RI_WR_TYPE_RDMA_READ,
6164     RI_WR_TYPE_BIND,
6165     RI_WR_TYPE_FAST_REGISTER,
6166     RI_WR_TYPE_LOCAL_INVALIDATE_MR,
6167     RI_WR_TYPE_LOCAL_INVALIDATE_MW,
6168
6169     /* iWarp-specific opcodes */
6170     RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE,
6171
6172     /* IB-specific opcodes */
6173     RI_WR_TYPE_COMPARE_AND_SWAP,
6174     RI_WR_TYPE_FETCH_AND_ADD,
6175     RI_WR_TYPE_SEND_IMMEDIATE,
6176     RI_WR_TYPE_RDMA_WRITE_IMMEDIATE
6177 };
6178 typedef enum ri_wr_type ri_wr_type_t;
6180 enum ri_wr_flags{
6181     /* Generate completion event */
6182     RI_WR_SINGALED_FLAG = 0x01,
6184     /* Generate notification for completion */
6185     RI_WR_LOCAL_SOLICITED_FLAG = 0x02,
6187     /* Notify the remote on completion */
6188     RI_WR_SOLICITED_EVENT_FLAG = 0x04,
6190     /* Block until all previous RDMA reads are completed */
6191     RI_WR_READ_FENCE_FLAG = 0x08,
6193     /* Block until all previous Work Requests are completed */
6194     RI_WR_LOCAL_FENCE_FLAG = 0x10,
6195     /* Coalesce Work Requests to the Send Queue */
6196     RI_WR_COALESCE_FLAG = 0x20,
6198     /* Hint to allow inlining of data */
6199     RI_WR_INLINE_OK_FLAG = 0x40,
6201     /* Pass-through in the absence of notification */
6202     RI_WR_CONSUMER_FLAG2 = RI_WR_LOCAL_SOLICITED_FLAG,
6203
```



```

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_sgls;
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_sgls;
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_sgls;
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_mr_handle_t   mrw_handle;
6299     };
6300     typedef struct ri_wr_local_invalidate ri_wr_local_invalidate_t;
6301
6302     union ri_mr_handle{
6303         ri_mr_handle_t   mr_handle;
6304         ri_mw_handle_t   mw_handle;
6305     };
6306     typedef union ri_mr_handle ri_mr_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;

```

```

6312         ri_rtag_t      rtag;
6313         uint64_t       atomic_operand1;
6314         uint64_t       atomic_operand2;
6315         ri_data_seg_t  atomic_result;
6316     };
6317     typedef struct ri_wr_atomic ri_wr_atomic_t;
6318
6319     struct ri_wr_datagram {
6320         ri_wr_type_t     type;
6321         ri_wr_flags_t    flags;
6322         ri_wr_id_t       id;
6323         ri_data_seg_t    *sg_list;
6324         uint_t           num_sgls;
6325         uint32_t         imm_data;
6326         ri_address_handle_t remote_node_address;
6327         uint32_t         dest_qp;
6328         uint32_t         dest_qkey;
6329         uint8_t          max_static_rate;
6330     };
6331     typedef struct ri_wr_datagram ri_wr_datagram_t;
6332
6333     union ri_wr{
6334         ri_wr_common_t     wr_common;
6335         ri_wr_send_t       wr_send;
6336         ri_wr_rdma_read_t  wr_rdma_read;
6337         ri_wr_rdma_write_t wr_rdma_write;
6338         ri_wr_bind_t       wr_bind;
6339         ri_wr_fmr_t        wr_fmr;
6340         ri_wr_local_invalidate_t wr_li;
6341         ri_wr_atomic_t     wr_atomic;
6342         ri_wr_datagram_t   wr_datagram;
6343     };
6344     typedef union ri_wr ri_wr_t;

```

6345 **DESCRIPTION**

6346

Work Request Field	Description
<i>type</i>	Type of Work Request. See <i>ri_wr_type_t</i> . 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
<i>flags</i>	<p>Modifier for a Work Request. See <i>ri_wr_flags_t</i>. <i>RI_WR_SIGNED_FLAG</i> and <i>RI_WR_LOCAL_SOLICITED_FLAG</i> are valid for all send Work Request types. <i>RI_WR_SOLICITED_EVENT_FLAG</i> is valid only for Send Work Request types and RDMA write with immediate data. <i>RI_WR_READ_FENCE_FLAG</i> is valid only for Send and RDMA-writes on iWarp transport. It is extended to RDMA read, atomic, and Bind on IB transport. <i>RI_WR_LOCAL_FENCE_FLAG</i> is valid only for <i>RI_WR_TYPE_LOCAL_INVALIDATE</i> Work Request type. <i>RI_WR_COALESCE_FLAG</i> has the following characteristics:</p> <ul style="list-style-type: none"> • Consumers typically set <i>RI_WR_COALESCE_FLAG</i> for post calls and eventually make a post call with <i>RI_WR_COALESCE_FLAG</i> cleared. If <i>RI_WR_COALESCE_FLAG</i> 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 <i>RI_WR_COALESCE_FLAG</i> is cleared. • Implementation could post pending Work Requests to RNIC at any time even if <i>RI_WR_COALESCE_FLAG</i> 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 <i>RI_WR_COALESCE_FLAG</i>. • Consumers MUST clear the flag before causing the Send Queue to be full. • <i>RI_WR_CONSUMER_FLAG0</i>, <i>RI_WR_CONSUMER_FLAG1</i>, and <i>RI_WR_CONSUMER_FLAG2</i> 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 <i>ri_rnic_query()</i>. Note that <i>RI_WR_CONSUMER_FLAG2</i> is meant to be used by the Consumer when RNIC does not support generation of notification upon completion of a specific Work Request. • If <i>RI_WR_INLINE_OK_FLAG</i> 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 <i>LTag</i> provided. If RNIC supports a non-zero <i>max_unregistered_inline</i> attribute, RNIC will not access the first <i>max_unregistered_inline</i> bytes in the Work Request through <i>Ltag</i>; 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 <i>max_unregistered_inline</i>, the Consumer must supply valid <i>Ltags</i> and RNIC may still inline some data if <i>RI_WR_INLINE_OK_FLAG</i> 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 <i>Ltags</i> 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
<i>sg_list</i>	A triplet containing address, length, and STag for local use. <i>ri_data_seg_t</i> 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. <i>addr</i> can be a virtual address for virtually addressed regions or windows. For STag0 operations, <i>tag</i> must be set to <i>stag0_value</i> reported by <i>ri_rnic_query()</i> . If <i>tag</i> refers to ZBVA region or window, <i>addr.offset</i> 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 <i>tag.rtag</i> must be set since Memory Window does not have an <i>Ltag</i> .
<i>id</i>	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.
<i>num_sgle</i>	Number of valid scatter/gather elements in <i>sg_list</i> .
<i>imm_data</i>	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.
<i>rtag</i>	<p>RTag of the remote buffer for RDMA write/read and atomic operations.</p> <p>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 <i>rtag</i> of the region to be re-registered or unmapped RTag obtained through <i>ri_ltag_alloc()</i>.</p> <p>For local invalidate Work Request type, this represents the <i>rtag</i> of the region to be invalidated. For RI_WR_TYPE_RDMA_READ_LOCAL_INVALIDATE, this represents the <i>rtag</i> of the remote region that is targeted. For RI_WR_TYPE_SEND_INVALIDATE, this represents the <i>rtag</i> 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.</p> <p><i>rtag</i> is undefined and ignored by the implementation for fast register and local invalidate Work Request types if the region or window identified by <i>mr_handle/mw_handle</i> does not have remote access enabled.</p>
<i>rdma_addr</i>	Target virtual address of the remote buffer for RDMA write/read and atomic operations.
<i>mw_handle</i>	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
<i>mr_handle</i>	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 <i>ri_ltag_alloc()</i> . For local invalidate Work Request type, this represents the handle to the region to be invalidated.
<i>ltag</i>	<i>Ltag</i> (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 <i>Ltag</i> of the region to be re-registered or the unmapped <i>Ltag</i> obtained through <i>ri_ltag_alloc()</i> . The “key” portion of this field contains the new value for the <i>Ltag</i> of the region being registered. For local invalidate Work Request types, this represents the <i>Ltag</i> of the region to be invalidated. It is undefined if a Memory Window is invalidated.
<i>vaddr</i>	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.
<i>length</i>	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.
<i>mem_attr</i>	Attributes for a fast registered Memory Region or a bound Memory Window. See <i>ri_mem_attr_t</i> .
<i>phys_buf_list, nbufs</i> <i>pble_size</i>	See <i>ri_mr_reg_phys()</i> for description of these fields.
IB-Specific Parameters	
<i>atomic_operand1</i>	Quantity used to compare against (RI_WR_TYPE_COMPARE_AND_SWAP) or add to (RI_WR_TYPE_FETCH_AND_ADD).
<i>atomic_operand2</i>	Quantity that is swapped with the remote if compare operation succeeds. Ignored for RI_WR_TYPE_FETCH_AND_ADD.
<i>atomic_result</i>	Original value of the remote data prior to the atomic operation is deposited here.
<i>remote_node_address</i>	Valid only for UD. Represents the address handle.
<i>dest_qp</i>	Remote QP number/ID to be targeted.
<i>dest_qkey</i>	Destination QKey.
<i>max_static_rate</i>	Maximum static rate.

6347 **CONSUMER USAGE**
6348 None.

6349 **IMPLEMENTATION NOTES**
6350 None.

6351 **SEE ALSO**
6352 None.

6353

6354 9 Implementers Guide

6355 9.1 RNIC Management

6356 Figure 3 shows a possible scenario of complete RNICPI Consumer registrations in both
6357 privileged and non-privileged domains initiated by IT-API style interface creation calls. It
6358 assumes demand-loaded kVP and uVP components which are matching Consumer-required
6359 RNIC attributes. Using the example sequence chart of Figure 3, this section discusses the main
6360 activities during RNIC provider modules initialization and Consumer registration. Following the
6361 implementation model depicted in Figure 2, the RNIC-specific components of the end system's
6362 RDMA stack comprise uVP and kVP; OS generic modules are uAL and kAL.

6363 9.1.1 Registration of an RNIC with the Operating System

6364 In a typical OS environment, some initial mechanism outside the scope of RNICPI may load
6365 and/or initialize the “device driver” code module which implements the vendor-specific
6366 hardware control functions. In the described scenario it is assumed that module loading is
6367 triggered by a kernel Consumer's request to open an RDMA interface (*it_ia_create*). As a result
6368 of the module initialization it will have called the *ri_svc_attach()* interface to represent itself as
6369 the kernel RDMA verbs provider (kVP) to the kAL. This call provides the mandatory P-RNICPI
6370 function vector *ri_ops_t* implementing Consumer (e.g., kAL) controlled thread safety and may
6371 also provide the optional provider controlled thread safety interface (see Section 4.3).
6372 Furthermore, it provides to the kAL a Verb Provider handle to the RNIC (*vp_handle_t*) and the
6373 *ri_svcs_dev_arg_t* parameter which contains among others information about the supported
6374 RNICPI version and the load path of the user space library (uVP).

6375 An RNIC is now unambiguously referenced by the combination of *vp_handle_t* and the current
6376 *ri_ops_t* interface function vector. Note that only the combination of both parameters can make
6377 the selection of an RNIC unambiguous, if potentially multiple kVPs and/or RNICs are available
6378 in the system. Point (1) in Figure 3 represents this status of the kAL.

6379 9.1.2 Opening an RNIC

6380 Before any RDMA service can be used, the RNIC device must be configured regarding its
6381 physical buffer list mode: the *ri_rnic_open()* call requests either Block List Mode or Page List
6382 Mode as RNIC global behavior (2). This mode remains selected for all offered RDMA services
6383 until the RNIC gets closed.

6384 9.1.3 Registering a Consumer with the P-RNICPI

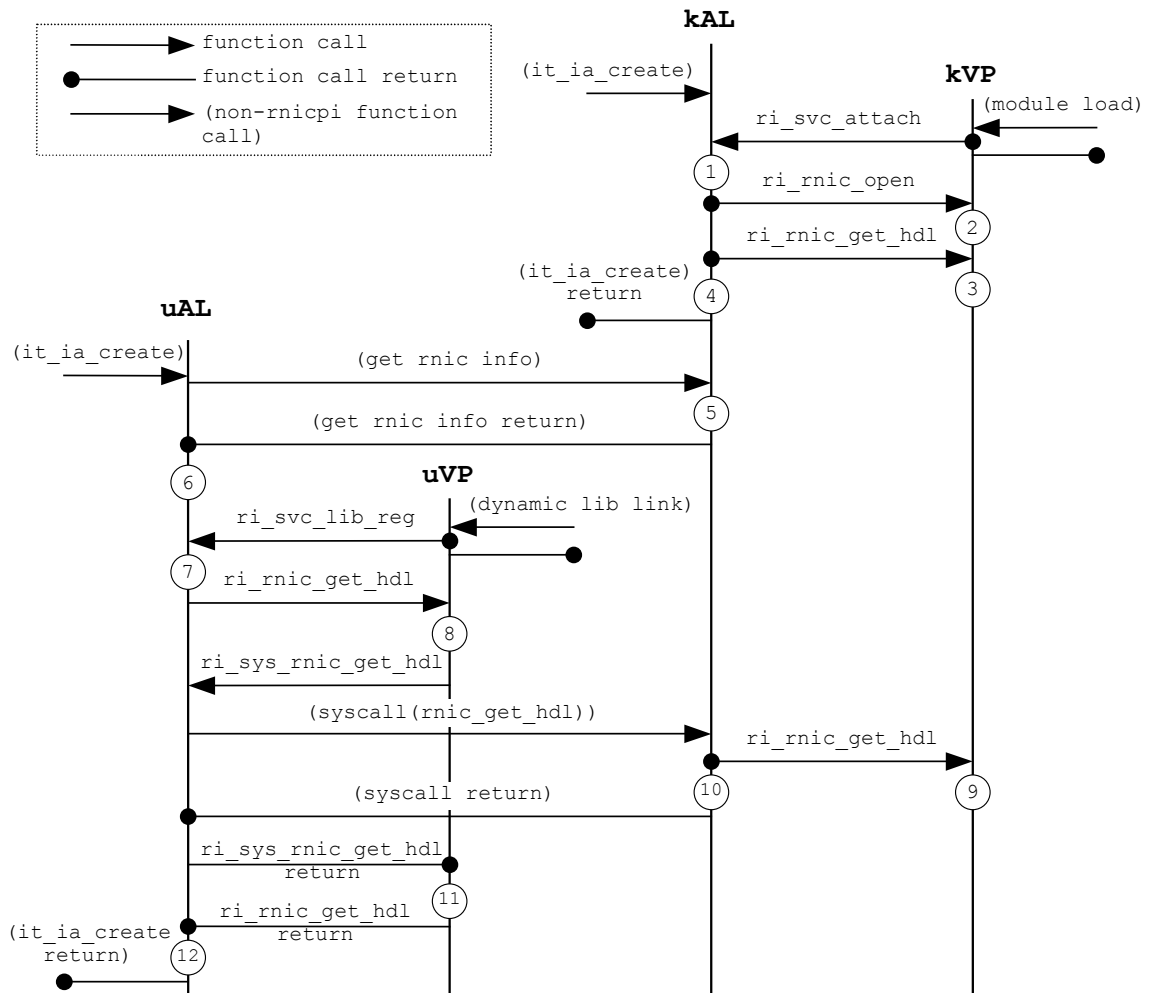
6385 Before using RDMA services, each RNICPI Consumer must first obtain its private RNIC
6386 handle. The Consumer private RNIC handle is obtained via the *ri_rnic_get_hdl()* interface.

6387
 6388
 6389
 6390
 6391
 6392
 6393
 6394

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.

6395
 6396

After a successful call to *ri_rnic_get_hdl()*, the privileged Consumer may access RDMA services using its private *rnic* handle (4).



6397
 6398

Figure 3: RNIC Management and Consumer Registration

6399 9.1.4 Selection and Registration of an uVP with the uAL

6400 The RNICPI assumes some private communication between uAL and kAL to allow the uAL
6401 selecting the correct uVP library. In a typical scenario, selection and loading of the uVP may be
6402 triggered by a Consumer's RDMA service request qualified by a remote IP address and a
6403 requested RNICPI interface version. By private uAL/kAL communication and relying on the
6404 kAL interface mapping service (see Section 3.2.3), the uAL will receive the library loading path
6405 for the uVP. This may imply that the kAL must load the corresponding kVP before providing
6406 this information back to the uAL (the library load path gets provided by the *ri_svc_attach()*
6407 routine; see above). It is expected that the kAL will also establish a unique mapping of the kVPs
6408 *vp_handle_t* to the application-level Consumer (5) and provide the mapped *vp_handle_t* together
6409 with the uVP library load path back to the uAL (6).

6410 The uAL now can dynamically link the uVP code. Similar to the *ri_svc_attach()* operation
6411 described for the privileged domain, during linking and initializing the user-level module will
6412 call the *ri_svc_lib_reg()* interface to register itself with the uAL as uVP (7).

6413 In a next step, the Consumer will request its RNIC handle via *ri_rnic_get_hdl()* and thereby
6414 implicitly select the desired level of thread safety provision (8). The *ri_rnic_get_handle* call will
6415 result in the uVP's call to the SYS-RNICPI (*ri_sys_rnic_get_hdl*) which is routed to the kAL
6416 and then mapped into a P-RNICPI call to the appropriate kVP. After the successful P-RNICPI
6417 call, the kVP will have established context information related to this new Consumer (9) and the
6418 kAL will have created the necessary state information to police and forward all further slow path
6419 activities of the new user space RNICPI Consumer (10). The complete call stack of the
6420 *ri_sys_rnic_get_hdl*, uAL/kAL communication, and *ri_rnic_get_hdl()* at the P-RNICPI
6421 transports opaque *vp_data* to synchronize uVP and kVP during resource setup. After returning
6422 from the SYS-RNICPI call, the uVP will have initialized its *rnic* handle-related structures (11)
6423 and the *rnic* handle is valid to be used for requesting further RDMA services (12).

6424 Note that the RNICPI does not allow the setting of the PBL mode from the NP-RNICPI. The
6425 *ri_rnic_open()* call is available only at the P-RNICPI.

6427 10.1 <rnicipi.h>

```
6428 /*
6429  * rnicpi.h
6430  *
6431  * Copyright © 2005 The Open Group
6432  * All rights reserved.
6433  * The copyright owner hereby grants permission for all or part of this
6434  * publication to be reproduced, stored in a retrieval system, or
6435  * transmitted, in any form or by any means, electronic, mechanical,
6436  * photocopying, recording, or otherwise, provided that it remains
6437  * unchanged and that this copyright statement is included in all
6438  * copies or substantial portions of the publication.
6439  *
6440  * For any software code contained within this specification,
6441  * permission is hereby granted, free-of-charge, to any person
6442  * obtaining a copy of this specification (the "Software"), to deal in
6443  * the Software without restriction, including without limitation the
6444  * rights to use, copy, modify, merge, publish, distribute, sublicense,
6445  * and/or sell copies of the Software, and to permit persons to whom
6446  * the Software is furnished to do so, subject to the above copyright
6447  * notice and this permission notice being included in all copies or
6448  * substantial portions of the Software.
6449  *
6450  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
6451  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
6452  * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
6453  * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
6454  * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
6455  * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
6456  * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
6457  * SOFTWARE.
6458  *
6459  * Permission is granted for implementers to use the names, labels,
6460  * etc. contained within the specification. The intent of publication
6461  * of the specification is to encourage implementations of the
6462  * specification. This specification has not been verified for
6463  * avoidance of possible third-party proprietary rights. In
6464  * implementing this specification, usual procedures to ensure the
6465  * respect of possible third-party intellectual property rights should
6466  * be followed.
6467  */
6468
6469 #ifndef _RNICPI_H_
6470 #define _RNICPI_H_
6471 #include <rnicipi_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_LL_HANDLE,
6582         RI_RET_INVALID_DDP_RDMAP_VERSION,
6583         RI_RET_LL_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

```

```

6904     );
6905
6906     typedef ri_api_return_t (*ri_op_rnic_close_t) (
6907         IN    ri_vp_handle_t  vp_hdl,
6908         OUT   ri_err_data_t   *err_data
6909     );
6910
6911     typedef enum ri_consumer_domain ri_consumer_domain_t;
6912     enum ri_consumer_domain {
6913         RI_CONSUMER_LOCATION_KERNEL,
6914         RI_CONSUMER_LOCATION_USER
6915     };
6916
6917     typedef uint32_t ri_ltag_t;
6918     typedef uint32_t ri_rtag_t;
6919
6920     typedef enum ri_stag_state ri_stag_state_t;
6921     enum ri_stag_state {
6922         RI_STAG_STATE_INVALID,
6923         RI_STAG_STATE_VALID
6924     };
6925
6926     typedef struct ri_phys_buf ri_phys_buf_t;
6927     struct ri_phys_buf {
6928         ri_phys_addr_t    phys_address;
6929         ri_phys_buf_len_t phys_len;
6930     };
6931
6932     typedef enum ri_mr_rereg ri_mr_rereg_t;
6933     enum ri_mr_rereg {
6934         RI_MR_REREG_TRANSLATE    = 1 << 0,
6935         RI_MR_REREG_PD           = 1 << 1,
6936         RI_MR_REREG_ACCESS       = 1 << 2
6937     };
6938
6939     typedef enum ri_mem_attr ri_mem_attr_t;
6940     enum ri_mem_attr {
6941         RI_MEM_FLAGS_LOCAL_READ    = 1 << 0,
6942         RI_MEM_FLAGS_LOCAL_WRITE   = 1 << 1,
6943         RI_MEM_FLAGS_LOCAL         = (RI_MEM_FLAGS_LOCAL_READ |
6944                                     RI_MEM_FLAGS_LOCAL_WRITE),
6945
6946         RI_MEM_FLAGS_REMOTE_WRITE  = 1 << 2,
6947         RI_MEM_FLAGS_REMOTE_READ   = 1 << 3,
6948         RI_MEM_FLAGS_REMOTE        = (RI_MEM_FLAGS_REMOTE_WRITE |
6949                                     RI_MEM_FLAGS_REMOTE_READ),
6950
6951         RI_MEM_FLAGS_ATOMICS       = 1 << 4,
6952         RI_MEM_FLAGS_BINDING       = 1 << 5,
6953         RI_MEM_FLAGS_CONSUMER_OWNS_KEY = 1 << 6,
6954         RI_MEM_FLAGS_ZBVA         = 1 << 7,
6955         RI_MEM_FLAGS_NONCOHERENT   = 1 << 8,
6956         RI_MEM_FLAGS_USER_VA      = 1 << 9,
6957         RI_MEM_FLAGS_SHARED_REG_IOVA = 1 << 10
6958     };

```

```

6958
6959     typedef uint64_t ri_addr_t;
6960
6961     typedef struct ri_mr_reg_attr ri_mr_reg_attr_t;
6962     struct ri_mr_reg_attr {
6963         ri_addr_t      va;
6964         ri_length_t    length;
6965         ri_mem_attr_t  mem_attr;
6966         ri_pd_handle_t pd;
6967     };
6968
6969     typedef struct ri_mr_reg_phys_attr ri_mr_reg_phys_attr_t;
6970     struct ri_mr_reg_phys_attr {
6971         ri_mr_reg_attr_t common;
6972         void              *phys_buf_list;
6973         uint_t            nbufs;
6974         uint_t            pble_size;
6975         uint_t            address_offset;
6976     };
6977
6978     /*
6979     * alternate: type of phys_buf_t could be union of
6980     * ri_phys_addr_t * and ri_phys_buf_t *
6981     */
6982
6983     typedef enum ri_mw_type ri_mw_type_t;
6984     enum ri_mw_type {
6985         RI_MW_TYPE_1, RI_MW_TYPE_WIDE=RI_MW_TYPE_1,
6986         RI_MW_TYPE_2, RI_MW_TYPE_NARROW=RI_MW_TYPE_2
6987     };
6988
6989     typedef struct ri_srq_attr ri_srq_attr_t;
6990     struct ri_srq_attr {
6991         uint32_t      max_wrs;
6992         uint32_t      max_sgl;
6993         uint32_t      lowat;
6994         ri_pd_handle_t pd;
6995         ri_boolean_t  srq_armed;
6996     };
6997
6998     typedef enum ri_svc_type ri_svc_type_t;
6999     enum ri_svc_type {
7000         RI_SERVICE_RELIABLE_CONNECTION,
7001         RI_SERVICE_UNRELIABLE_DATAGRAM
7002     };
7003
7004     /* Some states are not valid for iWARP/IB NICs.
7005     * Attempts to modify the state attribute will fail
7006     * with RI_RET_INVALID_STATE if not appropriate
7007     * for the technology.
7008     */
7009
7010     typedef enum ri_qp_state ri_qp_state_t;
7011     enum ri_qp_state {

```

```

7012     RI_QP_STATE_IDLE,
7013     RI_QP_STATE_READY_TO_SEND,
7014     RI_QP_STATE_ERROR,
7015
7016     /* The following states are valid only for iWARP */
7017     RI_QP_STATE_TERMINATE,
7018     RI_QP_STATE_CLOSING,
7019
7020     /* The following states are valid only for IB */
7021     RI_QP_STATE_SEND_QUEUE_ERROR,
7022     RI_QP_STATE_INITIALIZED,
7023     RI_QP_STATE_READY_TO_RECEIVE,
7024     RI_QP_STATE_SEND_QUEUE_DRAINING,
7025     RI_QP_STATE_SEND_QUEUE_DRAINED,
7026     RI_QP_STATE_TIMEWAIT
7027 };
7028
7029 typedef struct ri_mpa_attr ri_mpa_attr_t;
7030 struct ri_mpa_attr {
7031     ri_boolean_t  recv_marker_enabled;
7032     ri_boolean_t  xmit_marker_enabled;
7033     ri_boolean_t  crc_enabled;
7034     uint32_t      ddp_rdma_version; /* 0 or 1 */
7035 };
7036
7037 typedef struct ri_iwarp_qp_attr ri_iwarp_qp_attr_t;
7038
7039 struct ri_iwarp_qp_attr {
7040     ri_boolean_t  enable_bind;
7041     char          terminate_buffer[52];
7042     uint32_t      terminate_msg_length;
7043     ri_boolean_t  is_terminate_local;
7044     ri_llp_stream_handle_t llp_stream_handle;
7045     ri_mpa_attr_t mpa_attrs;
7046     char          *stream_msg_buf;
7047     uint32_t      stream_msg_buf_length;
7048     uint32_t      rq_hiwat;
7049     ri_boolean_t  rq_armed;
7050 };
7051
7052 typedef struct ri_gen_address_vector ri_gen_address_vector_t;
7053 struct ri_gen_address_vector {
7054     uint8_t      sl;
7055     uint8_t      send_global_routing_hdr_flg;
7056     uint8_t      hop_limit;
7057     uint8_t      source_gid_index;
7058     uint32_t     flow_label;
7059     ri_gid_t     dgid;
7060     ri_lid_t     dlid;
7061     uint8_t      traffic_class;
7062     uint8_t      max_static_rate;
7063     uint8_t      source_path_bits;
7064 };
7065

```

```

7066 typedef struct ri_path_address_info ri_path_address_info_t;
7067 struct ri_path_address_info {
7068     uint16_t                pkey_index;
7069     uint8_t                 physical_port;
7070     uint16_t                path_mtu;
7071     uint8_t                 local_ack_timeout;
7072     uint8_t                 retry_count;
7073     uint8_t                 rnr_retry_count;
7074     ri_gen_address_vector_t av;
7075 };
7076
7077 typedef enum ri_path_migration_state ri_path_migration_state_t;
7078 enum ri_path_migration_state {
7079     RI_MIG_STATE_MIGRATED,
7080     RI_MIG_STATE_REARM,
7081     RI_MIG_STATE_ARMED
7082 };
7083
7084 typedef struct ri_ib_qp_attr ri_ib_qp_attr_t;
7085 struct ri_ib_qp_attr {
7086     /* common to RC and UD */
7087     uint32_t                send_psn;
7088     ri_boolean_t            signal_completions;
7089     ri_svc_type_t           trans_type;
7090     uint32_t                enable_async_notify_for_sqd;
7091     ri_path_address_info_t  primary_av;
7092     /* RC-specific attributes */
7093     uint32_t                rcv_psn;
7094     uint32_t                dest_qpn;
7095     uint32_t                min_rnrnak_timer;
7096     ri_path_address_info_t  alternate_av;
7097     ri_path_migration_state_t mig_state;
7098     ri_boolean_t            enable_atomic;
7099     /* UD-specific attributes */
7100     ri_qp_state_t           current_state;
7101     uint32_t                rcv_qkey;
7102 };
7103
7104 typedef struct ri_qp_attr ri_qp_attr_t;
7105 struct ri_qp_attr {
7106     uint32_t                qp_id;
7107     union {
7108         ri_cq_handle_t      scq_handle;
7109         ri_os_data_t        scq_os_data;
7110     } scq;
7111     union {
7112         ri_cq_handle_t      rcq_handle;
7113         ri_os_data_t        rcq_os_data;
7114     } rcq;
7115     union {
7116         ri_srq_handle_t     srq_handle;
7117         ri_os_data_t        srq_os_data;
7118     } srq;
7119     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 */

```

```

7174     RI_WR_LOCAL_FENCE_FLAG = 0x10,
7175
7176     /* Coalesce Work Requests to the Send queue */
7177     RI_WR_COALESCE_FLAG = 0x20,
7178
7179     /* Hint to allow inlining of data */
7180     RI_WR_INLINE_OK_FLAG = 0x40
7181
7182     RI_WR_CONSUMER_FLAG2 = RI_WR_LOCAL_SOLICITED_FLAG,
7183     RI_WR_CONSUMER_FLAG1 = 1 << 29,
7184     RI_WR_CONSUMER_FLAG0 = 1 << 30
7185 };
7186
7187 typedef struct ri_data_seg ri_data_seg_t;
7188 struct ri_data_seg {
7189     union {
7190         ri_addr_t      vaddr;
7191         ri_phys_addr_t paddr;
7192         ri_length_t    offset;
7193     } addr;
7194     union {
7195         ri_ltag_t      ltag;
7196         ri_rtag_t      rtag;
7197     } tag;
7198     uint32_t len;
7199 };
7200
7201 typedef uint64_t ri_wr_id_t;
7202
7203 typedef struct ri_wr_common ri_wr_common_t;
7204 struct ri_wr_common {
7205     ri_wr_type_t      type;
7206     ri_wr_flags_t     flags;
7207     ri_wr_id_t        id;
7208 };
7209
7210 typedef struct ri_wr_send ri_wr_send_t;
7211 struct ri_wr_send {
7212     ri_wr_type_t      type;
7213     ri_wr_flags_t     flags;
7214     ri_wr_id_t        id;
7215     ri_data_seg_t     *sg_list;
7216     ri_rtag_t          rtag;
7217     uint_t             num_sgls;
7218     uint32_t           imm_data;
7219 };
7220
7221 typedef struct ri_wr_rdma_write ri_wr_rdma_write_t;
7222 struct ri_wr_rdma_write {
7223     ri_wr_type_t      type;
7224     ri_wr_flags_t     flags;
7225     ri_wr_id_t        id;
7226     ri_data_seg_t     *sg_list;
7227     uint64_t           rdma_addr;

```



```

7228         ri_rtag_t      rtag;
7229         uint_t         num_sgls;
7230         uint32_t       imm_data;
7231     };
7232
7233     typedef struct ri_wr_rdma_read ri_wr_rdma_read_t;
7234     struct ri_wr_rdma_read {
7235         ri_wr_type_t    type;
7236         ri_wr_flags_t   flags;
7237         ri_wr_id_t      id;
7238         ri_data_seg_t   *sg_list;
7239         uint64_t        rdma_addr;
7240         uint_t          num_sgls;
7241         ri_rtag_t       rtag;
7242     };
7243
7244     typedef struct ri_wr_bind ri_wr_bind_t;
7245     struct ri_wr_bind {
7246         ri_wr_type_t    type;
7247         ri_wr_flags_t   flags;
7248         ri_wr_id_t      id;
7249         ri_mw_handle_t  mw_handle;
7250         ri_rtag_t       rtag;
7251         ri_mr_handle_t  mr_handle;
7252         ri_ltag_t       ltag;
7253         ri_addr_t       vaddr;
7254         ri_length_t     length;
7255         ri_mem_attr_t   mem_attr;
7256     };
7257
7258     typedef struct ri_wr_fmr ri_wr_fmr_t;
7259     struct ri_wr_fmr {
7260         ri_wr_type_t    type;
7261         ri_wr_flags_t   flags;
7262         ri_wr_id_t      id;
7263         ri_mr_reg_phys_attr_t attr;
7264         ri_ltag_t       ltag;
7265         ri_rtag_t       rtag;
7266         ri_mr_handle_t  mr_handle;
7267     };
7268
7269     typedef union ri_mr_w_handle ri_mr_w_handle_t;
7270     union ri_mr_w_handle {
7271         ri_mr_handle_t  mr_handle;
7272         ri_mw_handle_t  mw_handle;
7273     };
7274
7275     typedef struct ri_wr_local_invalidate ri_wr_local_invalidate_t;
7276     struct ri_wr_local_invalidate {
7277         ri_wr_type_t    type;
7278         ri_wr_flags_t   flags;
7279         ri_wr_id_t      id;
7280         ri_ltag_t       ltag;
7281         ri_rtag_t       rtag;

```

```

7282         ri_mrwr_handle_t  mrwr_handle;
7283     };
7284
7285     typedef struct ri_wr_atomic ri_wr_atomic_t;
7286     struct ri_wr_atomic {
7287         ri_wr_type_t      type;
7288         ri_wr_flags_t     flags;
7289         ri_wr_id_t        id;
7290         ri_rtag_t         rtag;
7291         uint64_t          atomic_operand1;
7292         uint64_t          atomic_operand2;
7293         ri_data_seg_t     atomic_result;
7294     };
7295
7296     typedef struct ri_wr_datagram ri_wr_datagram_t;
7297     struct ri_wr_datagram {
7298         ri_wr_type_t      type;
7299         ri_wr_flags_t     flags;
7300         ri_wr_id_t        id;
7301         ri_data_seg_t     *sg_list;
7302         uint_t            num_sgls;
7303         uint32_t          imm_data;
7304         ri_address_handle_t remote_node_address;
7305         uint32_t          dest_qp;
7306         uint32_t          dest_qkey;
7307         uint8_t           max_static_rate;
7308     };
7309
7310     typedef union ri_wr ri_wr_t;
7311     union ri_wr {
7312         ri_wr_common_t      wr_common;
7313         ri_wr_send_t        wr_send;
7314         ri_wr_rdma_read_t   wr_rdma_read;
7315         ri_wr_rdma_write_t  wr_rdma_write;
7316         ri_wr_bind_t        wr_bind;
7317         ri_wr_fmr_t         wr_fmr;
7318         ri_wr_local_invalidate_t wr_li;
7319         ri_wr_atomic_t      wr_atomic;
7320         ri_wr_datagram_t    wr_datagram;
7321     };
7322
7323     typedef ri_api_return_t (*ri_op_rnic_get_hdl_t) (
7324         IN      ri_vp_handle_t      vp_hdl,
7325         IN      ri_consumer_domain_t location,
7326         OUT     ri_rnic_handle_t     *rnic,
7327         IN OUT  ri_vp_data_t         vp_data,
7328         IN OUT  ri_os_data_t         os_data,
7329         OUT     ri_err_data_t        *err_data
7330     );
7331
7332     typedef ri_api_return_t (*ri_op_rnic_free_hdl_t) (
7333         IN      ri_vp_handle_t      vp_hdl,
7334         IN      ri_rnic_handle_t    rnic,
7335         OUT     ri_err_data_t        *err_data

```

```

7336 );
7337
7338 typedef enum ri_rnic_query_flags ri_rnic_query_flags_t;
7339 enum ri_rnic_query_flags {
7340     RI_QUERY_RNIC_PORT_INFO = 1 << 0
7341 };
7342
7343 typedef ri_api_return_t (*ri_op_rnic_query_t) (
7344     IN     ri_rnic_handle_t      rnic,
7345     IN     ri_rnic_query_flags_t flags,
7346     OUT    ri_rnic_attr_t        *attrs,
7347     IN OUT uint_t                 *attr_size,
7348     OUT    ri_err_data_t         *err_data
7349 );
7350
7351 typedef enum ri_rnic_attr_mask ri_rnic_attr_mask_t;
7352 enum ri_rnic_attr_mask {
7353     RI_IB_RNIC_ATTR_SI_GUID = 1 << 0
7354 };
7355
7356 typedef enum ri_port_attr_mask ri_port_attr_mask_t;
7357 enum ri_port_attr_mask {
7358     RI_IB_PORT_ATTR_SHUTDOWN = 1 << 0,
7359     RI_IB_PORT_ATTR_INIT_TYPE = 1 << 1,
7360     RI_IB_PORT_ATTR_QKEY_RESET = 1 << 2,
7361     RI_IB_PORT_ATTR_CAP_SET = 1 << 3,
7362     RI_IB_PORT_ATTR_CAP_FLAG_SM = 1 << 4,
7363     RI_IB_PORT_ATTR_CAP_FLAG_SNMP = 1 << 5,
7364     RI_IB_PORT_ATTR_CAP_FLAG_DEVMGT = 1 << 6,
7365     RI_IB_PORT_ATTR_CAP_FLAG_VENDOR = 1 << 7
7366 };
7367
7368 #define RI_IB_ALL_PORTS (0)
7369
7370 typedef struct ri_port_modify_req ri_port_modify_req_t;
7371 struct ri_port_modify_req {
7372     ri_port_attr_mask_t port_attr_mask;
7373     uint8_t              port_number;
7374     uint8_t              port_init_type; /* 4 bits */
7375 };
7376
7377 typedef struct ri_rnic_modify_req ri_rnic_modify_req_t;
7378 struct ri_rnic_modify_req {
7379     uint64_t             si_guid;
7380     ri_port_modify_req_t *port_attrs;
7381     uint8_t              port_list_size;
7382 };
7383
7384 typedef ri_api_return_t (*ri_op_rnic_modify_t) (
7385     IN     ri_rnic_handle_t      rnic,
7386     IN     ri_rnic_attr_mask_t   attr_mask,
7387     IN     ri_rnic_modify_req_t  *attrs,
7388     OUT    ri_err_data_t         *err_data
7389 );

```

```

7390
7391 typedef ri_api_return_t (*ri_op_pd_alloc_t) (
7392     IN    ri_rnic_handle_t  rnic,
7393     OUT    ri_pd_handle_t   *pd,
7394     IN OUT ri_vp_data_t     vp_data,
7395     IN OUT ri_os_data_t     os_data,
7396     OUT    ri_err_data_t    *err_data
7397 );
7398
7399 typedef ri_api_return_t (*ri_op_pd_dealloc_t) (
7400     IN    ri_rnic_handle_t  rnic,
7401     IN    ri_pd_handle_t   pd,
7402     OUT    ri_err_data_t    *err_data
7403 );
7404
7405 typedef uint_t ri_cq_event_handler_id_t;
7406
7407 typedef struct ri_cq_attr ri_cq_attr_t;
7408 struct ri_cq_attr {
7409     uint32_t          num_cqe;
7410     ri_cq_event_handler_id_t handler_id;
7411 };
7412
7413 typedef ri_api_return_t (*ri_op_cq_create_t) (
7414     IN    ri_rnic_handle_t  rnic,
7415     IN OUT ri_cq_attr_t     *cq_attr,
7416     OUT    ri_cq_handle_t   *cq,
7417     IN OUT ri_vp_data_t     vp_data,
7418     IN OUT ri_os_data_t     os_data,
7419     OUT    ri_err_data_t    *err_data
7420 );
7421
7422 typedef ri_api_return_t (*ri_op_cq_destroy_t) (
7423     IN    ri_rnic_handle_t  rnic,
7424     IN    ri_cq_handle_t   cq,
7425     OUT    ri_err_data_t    *err_data
7426 );
7427
7428 typedef ri_api_return_t (*ri_op_cq_query_t) (
7429     IN    ri_rnic_handle_t  rnic,
7430     IN    ri_cq_handle_t   cq,
7431     OUT    ri_cq_attr_t     *cq_attr,
7432     OUT    ri_err_data_t    *err_data
7433 );
7434
7435 typedef enum ri_cq_modify_mask ri_cq_modify_mask_t;
7436 enum ri_cq_modify_mask {
7437     RI_CQ_MODIFY_MASK_NUM_CQE = 0x1
7438 };
7439
7440 typedef ri_api_return_t (*ri_op_cq_modify_t) (
7441     IN    ri_rnic_handle_t  rnic,
7442     IN    ri_cq_handle_t   cq,
7443     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_LLQ_CLOSE_COMPLETE,
7452         RI_EVENT_QP_TERMINATE_MESSAGE_RECEIVED,
7453         RI_EVENT_QP_LLQ_CONNECTION_RESET,
7454         RI_EVENT_QP_LLQ_CONNECTION_LOST,
7455         RI_EVENT_QP_LLQ_INTEGRITY_ERROR_SIZE,
7456         RI_EVENT_QP_LLQ_INTEGRITY_ERROR_CRC,
7457         RI_EVENT_QP_LLQ_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_LLQ_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 */

```

```

7498
7499     /* IB-specific events */
7500     RI_EVENT_TYPE_QP_PATH_MIGRATED,
7501     RI_EVENT_TYPE_QP_COMMUNICATION_ESTABLISHED,
7502     RI_EVENT_TYPE_QP_SEND_QUEUE_DRAINED,
7503     RI_EVENT_CQ_ERROR,
7504     RI_EVENT_QP_LOCAL_WQ_CATASTROPHIC_ERROR,
7505     RI_EVENT_QP_INVALID_REQ_LOCAL_WQ_ERROR,
7506     RI_EVENT_QP_LOCAL_ACCESS_VIOLATION_WQ_ERROR,
7507     RI_EVENT_QP_PATH_MIG_REQ_ERROR,
7508     RI_EVENT_PORT_EVENT,
7509     RI_EVENT_CLIENT_REG_EVENT
7510 };
7511
7512 typedef struct ri_ib_port_event ri_ib_port_event_t;
7513 struct ri_ib_port_event {
7514     /* IB verbs do not specify that the port
7515      * number has to be returned. Port number 0
7516      * indicates that there is no knowledge of
7517      * which port this event is associated with.
7518      */
7519     uint8_t          port_number;
7520     ri_ib_port_state_t port_state;
7521 };
7522
7523 typedef union ri_event_data ri_event_data_t;
7524 union ri_event_data {
7525     ri_ib_port_event_t  port_event;
7526     ri_os_data_t        resource_os_data;
7527 };
7528
7529 typedef struct ri_event_record ri_event_record_t;
7530 struct ri_event_record {
7531     ri_async_event_type_t event_type;
7532     ri_event_data_t       event_data;
7533     ri_err_data_t        err_data;
7534 };
7535
7536 typedef void (*ri_async_event_callback_t)(
7537     IN OUT ri_os_data_t      vp_hdl_os_data,
7538     IN      ri_event_record_t *event
7539 );
7540
7541 typedef ri_api_return_t (*ri_op_async_set_event_handler_t)(
7542     IN ri_vp_handle_t      vp_hdl,
7543     IN ri_async_event_callback_t async_event_callback,
7544     OUT ri_err_data_t      *err_data
7545 );
7546
7547 typedef void (*ri_cq_event_callback_t)(
7548     IN OUT ri_os_data_t vp_hdl_os_data,
7549     IN OUT ri_os_data_t cq_os_data
7550 );
7551

```

```

7552 typedef ri_api_return_t (*ri_op_cq_set_event_handler_t) (
7553     IN     ri_vp_handle_t          vp_hdl,
7554     IN OUT ri_cq_event_handler_id_t *handler_id,
7555     IN     ri_cq_event_callback_t  cq_callback,
7556     OUT    ri_err_data_t           *err_data
7557 );
7558
7559 typedef enum ri_cmplt_notify_type ri_cmplt_notify_type_t;
7560 enum ri_cmplt_notify_type {
7561     RI_CMPLT_NOTIFY_SOLICITED,
7562     RI_CMPLT_NOTIFY_ANY
7563 };
7564
7565 typedef ri_api_return_t (*ri_op_cq_req_notification_t) (
7566     IN  ri_rnic_handle_t          rnic,
7567     IN  ri_cq_handle_t           cq,
7568     IN  ri_cmplt_notify_type_t   type,
7569     OUT ri_err_data_t            *err_data
7570 );
7571
7572 /* Performance Note: An attempt to align the wc types with IT API's
7573  * event types has been done for the first five types. It is highly
7574  * desirable that the definitions do not change with future versions
7575  * of IT API.
7576  */
7577 typedef enum ri_wc_type ri_wc_type_t;
7578 enum ri_wc_type {
7579     RI_WC_TYPE_SEND = 0x1,
7580     RI_WC_TYPE_DATA_RECEIVED = 0x2,
7581     RI_WC_TYPE_RDMA_WRITE = 0x4,
7582     RI_WC_TYPE_RDMA_READ = 0x5,
7583     RI_WC_TYPE_BIND = 0x6,
7584     RI_WC_TYPE_FAST_REGISTER,
7585     RI_WC_TYPE_LOCAL_INVALIDATE,
7586     RI_WC_TYPE_RDMA_READ_LOCAL_INVALIDATE,
7587     RI_WC_TYPE_COMPARE_AND_SWAP,
7588     RI_WC_TYPE_FETCH_AND_ADD,
7589     RI_WC_TYPE_RDMA_IMMEDIATE_RECEIVED
7590 };
7591
7592 typedef enum ri_wc_flags ri_wc_flags_t;
7593 enum ri_wc_flags {
7594     RI_WC_FLAG_SOLICITED_EVENT = 1 << 0,
7595     RI_WC_FLAG_LOCAL_SOLICITED = 1 << 1,
7596     RI_WC_FLAG_RTAG_INVALIDATE = 1 << 2,
7597     RI_WC_FLAG_IMMEDIATE_DATA = 1 << 3,
7598     /* Pass through if there is no local solicited notification */
7599     RI_WC_FLAG_CONSUMER2 = RI_WC_FLAG_LOCAL_SOLICITED,
7600     RI_WC_FLAG_CONSUMER1 = 1 << 29,
7601     RI_WC_FLAG_CONSUMER0 = 1 << 30
7602 };
7603
7604 /* Performance Note: An attempt to align the completion status with
7605  * IT API's dto_status_t has been done for the first five types.

```

```

7606     * It is highly desirable that the definitions do not change with
7607     * future versions of IT API.
7608     */
7609     typedef enum ri_wc_status ri_wc_status_t;
7610     enum ri_wc_status {
7611         RI_WC_STATUS_SUCCESS = 0,
7612         RI_WC_STATUS_WR_FLUSHED_ERROR = 1,
7613         RI_WC_STATUS_LOCAL_QP_CATASTROPHIC = 2,
7614         RI_WC_STATUS_LOCAL_QP_OP_ERROR = 2,
7615         /* same as LOCAL_QP_CATASTROPHIC */
7616
7617         /* iWarp-specific status */
7618         RI_WC_STATUS_WQE_FORMAT_ERROR,
7619         RI_WC_STATUS_REMOTE_TERMINATION_ERROR,
7620         RI_WC_STATUS_INVALID_STAG_ERROR,
7621         RI_WC_STATUS_BASE_BOUNDS_VIOLATION_ERROR,
7622         RI_WC_STATUS_ACCESS_VIOLATION_ERROR,
7623         RI_WC_STATUS_INVALID_PD_ERROR,
7624         RI_WC_STATUS_ADDRESS_WRAP_ERROR,
7625         RI_WC_STATUS_STAG_INVALIDATE_ERROR,
7626         RI_WC_STATUS_ZERO_ORD_ERROR,
7627         RI_WC_STATUS_FMR_QP_NOT_PRIVILEGED_ERROR,
7628         RI_WC_STATUS_STAG_NOT_IN_INVALID_STATE_ERROR,
7629         RI_WC_STATUS_INVALID_FMR_PAGE_SIZE_ERROR,
7630         RI_WC_STATUS_INVALID_FMR_BLOCK_SIZE_ERROR,
7631         RI_WC_STATUS_INVALID_FMR_PBL_ENTRY_ERROR,
7632         RI_WC_STATUS_INVALID_FMR_FBO_ERROR,
7633         RI_WC_STATUS_INVALID_FMR_LENGTH_ERROR,
7634         RI_WC_STATUS_INVALID_FMR_PERMS_ERROR,
7635         RI_WC_STATUS_FMR_PBL_TOO_LONG_ERROR,
7636         RI_WC_STATUS_INVALID_FMR_VA_ERROR,
7637         RI_WC_STATUS_INVALID_BIND_MR_ERROR,
7638         RI_WC_STATUS_INVALID_BIND_MW_ERROR,
7639         RI_WC_STATUS_HUGE_PAYLOAD_ERROR,
7640
7641         /* IB-specific status */
7642         RI_WC_STATUS_LOCAL_LENGTH_ERROR,
7643         RI_WC_STATUS_LOCAL_PROTECTION_ERROR,
7644         RI_WC_STATUS_MEMORY_MGMT_ERROR,
7645         RI_WC_STATUS_BAD_RESPONSE_ERROR,
7646         RI_WC_STATUS_LOCAL_ACCESS_ERROR,
7647         RI_WC_STATUS_REMOTE_INVALID_REQ_ERROR,
7648         RI_WC_STATUS_REMOTE_ACCESS_ERROR,
7649         RI_WC_STATUS_REMOTE_OP_ERROR,
7650         RI_WC_STATUS_TRANSPORT_RETRY_EXCEEDED,
7651         RI_WC_STATUS_RNR_RETRY_EXCEEDED,
7652         RI_WC_STATUS_REMOTE_ABORTED_ERROR
7653     };
7654
7655     /* Performance Note: An attempt to align RC (non-immediate data) work
7656     * completion structure with IT API's dto_cmpl_event_t structure has
7657     * been made. It is highly desirable that the definitions do not change
7658     * with future versions of IT API.
7659     */

```



```

7660
7661 typedef struct ri_rc_wc ri_rc_wc_t;
7662 struct ri_rc_wc {
7663     /* op_type should be the first field in the structure */
7664     ri_wc_type_t  op_type;
7665
7666     /* Placeholder for it_evd_handle_t in ITAPI. OSVs may need to
7667     * define flags and invalidated_rtag to take the size of
7668     * it_evd_handle_t on their platforms.
7669     */
7670     ri_os_data_t  cq_os_data;
7671     ri_wc_flags_t  flags;
7672     ri_rtag_t     invalidated_rtag;
7673
7674     /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
7675     * ri_os_data_t identical to it_ep_handle_t on their platforms
7676     */
7677     ri_os_data_t  qp_os_data;
7678
7679     ri_wr_id_t id;
7680     ri_wc_status_t  op_status;
7681     uint32_t      bytes_transferred; /* valid only for receive */
7682     uint32_t      immediate_data; /* valid only if immediate data
7683     * is received along with incoming
7684     * Send or RDMA Write (IB only)
7685     */
7686     ri_err_data_t  err_data;
7687 };
7688
7689 typedef struct ri_gen_wc ri_gen_wc_t;
7690 struct ri_gen_wc {
7691     /* op_type should be the first field in the structure */
7692     ri_wc_type_t  op_type;
7693
7694     /* Placeholder for it_evd_handle_t in ITAPI. OSVs may need to pad
7695     * this appropriately to take the size of it_evd_handle_t on their
7696     * platforms.
7697     */
7698     ri_os_data_t  cq_os_data;
7699     ri_wc_flags_t  flags;
7700     uint32_t      qpn;
7701
7702     /* Placeholder for it_ep_handle_t in ITAPI. OSVs may need to define
7703     * ri_handle_t identical to it_ep_handle_t on their platforms.
7704     */
7705     ri_os_data_t  qp_os_data;
7706     ri_wr_id_t     id;
7707     ri_wc_status_t  op_status;
7708     uint32_t      bytes_transferred;
7709     uint32_t      immediate_data;
7710     ri_lid_t      remote_lid;
7711     uint32_t      remote_qpn;
7712     uint8_t       remote_sl;
7713     uint8_t       local_lid_pathbits;

```

```

7714         uint16_t         ether_type;
7715         uint32_t         grh_present;
7716         uint32_t         pkey_index;
7717         uint32_t         freed_resource_count;
7718         ri_err_data_t    err_data;
7719     };
7720
7721     typedef union ri_wc ri_wc_t;
7722     union ri_wc {
7723         ri_wc_type_t     type_wc;
7724         ri_rc_wc_t       rc_wc;
7725         ri_gen_wc_t      ud_wc;
7726     };
7727
7728     typedef ri_api_return_t (*ri_op_cq_poll_t) (
7729         IN         ri_rnic_handle_t  rnic,
7730         IN         ri_cq_handle_t     cq,
7731         OUT        ri_wc_t            *wcs,
7732         IN OUT     uint_t              *num_wcs,
7733         OUT        ri_err_data_t      *err_data
7734     );
7735
7736     typedef ri_api_return_t (*ri_op_qp_create_t) (
7737         IN         ri_rnic_handle_t  rnic,
7738         IN OUT     ri_qp_attr_t      *attrs,
7739         OUT        ri_qp_handle_t    *qp,
7740         IN OUT     ri_vp_data_t      vp_data,
7741         IN OUT     ri_os_data_t      os_data,
7742         OUT        ri_err_data_t      *err_data
7743     );
7744
7745     typedef ri_api_return_t (*ri_op_qp_query_t) (
7746         IN         ri_rnic_handle_t  rnic,
7747         IN         ri_qp_handle_t    qp,
7748         OUT        ri_qp_attr_t      *attrs,
7749         OUT        ri_err_data_t      *err_data
7750     );
7751
7752     typedef enum ri_qp_attr_mask ri_qp_attr_mask_t;
7753     enum ri_qp_attr_mask {
7754         RI_QP_ATTR_NEXT_STATE = 1 << 0,
7755         RI_QP_ATTR_SQ_SIZE = 1 << 1,
7756         RI_QP_ATTR_RQ_SIZE = 1 << 2,
7757         RI_QP_ATTR_MAX_RDMA_READ_OUTGOING = 1 << 3,
7758         RI_QP_ATTR_MAX_RDMA_READ_INCOMING = 1 << 4,
7759
7760         /* iWARP-specific masks */
7761         RI_QP_ATTR_RQ_HIWAT = 1 << 5,
7762         RI_QP_ATTR_RQ_ARMED = 1 << 6,
7763         RI_QP_ATTR_LLQ_STREAM_HANDLE = 1 << 7,
7764         RI_QP_ATTR_STREAM_MSG_BUFFER = 1 << 8,
7765         RI_QP_ATTR_MPA_ATTR = 1 << 9,
7766
7767         /* 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,

```

```

7822         OUT  ri_err_data_t    *err_data
7823     );
7824
7825     typedef ri_api_return_t (*ri_op_srq_modify_t) (
7826         IN    ri_rnic_handle_t  rnic,
7827         IN    ri_srq_handle_t   srq,
7828         IN    ri_srq_attr_mask_t srqmask,
7829         OUT   ri_srq_attr_t     *attrs,
7830         OUT   ri_err_data_t     *err_data
7831     );
7832
7833     typedef struct ri_wr_recv ri_wr_recv_t;
7834     struct ri_wr_recv {
7835         ri_wr_flags_t  flags;
7836         ri_wr_id_t     id;
7837         ri_data_seg_t  *sg_list;
7838         uint_t         num_sgls;
7839     };
7840
7841     typedef ri_api_return_t (*ri_op_srq_post_t) (
7842         IN    ri_rnic_handle_t  rnic,
7843         IN    ri_srq_handle_t   srq_handle,
7844         IN    ri_wr_recv_t      *wr,
7845         IN OUT uint_t           *num_wrs,
7846         OUT   ri_err_data_t     *err_data
7847     );
7848
7849     typedef ri_api_return_t (*ri_op_srq_post_recv_t) (
7850         IN    ri_rnic_handle_t  rnic,
7851         IN    ri_srq_handle_t   srq_handle,
7852         IN    ri_wr_flags_t     flags,
7853         IN    ri_wr_id_t        id,
7854         IN    ri_data_seg_t     *sg_list,
7855         IN    uint_t            num_sgls,
7856         OUT   ri_err_data_t     *err_data
7857     );
7858
7859     typedef ri_api_return_t (*ri_op_srq_destroy_t) (
7860         IN    ri_rnic_handle_t  rnic,
7861         IN    ri_srq_handle_t   srq,
7862         OUT   ri_err_data_t     *err_data
7863     );
7864
7865     typedef ri_api_return_t (*ri_op_qp_postsq_t) (
7866         IN    ri_rnic_handle_t  rnic,
7867         IN    ri_qp_handle_t    qp,
7868         IN    ri_wr_t           *wr,
7869         IN OUT uint_t           *num_wrs,
7870         OUT   ri_err_data_t     *err_data
7871     );
7872
7873     typedef ri_api_return_t (*ri_op_qp_post_send_t) (
7874         IN    ri_rnic_handle_t  rnic,
7875         IN    ri_qp_handle_t    qp,

```

```

7876     IN    ri_wr_type_t      type,
7877     IN    ri_wr_flags_t     flags,
7878     IN    ri_wr_id_t        id,
7879     IN    ri_data_seg_t     *sg_list,
7880     IN    ri_rtag_t         rtag,
7881     IN    uint_t            num_sgle,
7882     IN    uint32_t          imm_data,
7883     OUT   ri_err_data_t     *err_data
7884 );
7885
7886 typedef ri_api_return_t (*ri_op_qp_post_rdma_write_t) (
7887     IN    ri_rnic_handle_t  rnic,
7888     IN    ri_qp_handle_t    qp,
7889     IN    ri_wr_type_t      type,
7890     IN    ri_wr_flags_t     flags,
7891     IN    ri_wr_id_t        id,
7892     IN    ri_data_seg_t     *sg_list,
7893     IN    uint64_t          rdma_addr,
7894     IN    ri_rtag_t         rtag,
7895     IN    uint_t            num_sgle,
7896     IN    uint32_t          imm_data,
7897     OUT   ri_err_data_t     *err_data
7898 );
7899
7900 typedef ri_api_return_t (*ri_op_qp_post_rdma_read_t) (
7901     IN    ri_rnic_handle_t  rnic,
7902     IN    ri_qp_handle_t    qp,
7903     IN    ri_wr_type_t      type,
7904     IN    ri_wr_flags_t     flags,
7905     IN    ri_wr_id_t        id,
7906     IN    ri_data_seg_t     *sg_list,
7907     IN    uint64_t          rdma_addr,
7908     IN    uint_t            num_sgle,
7909     IN    ri_rtag_t         rtag,
7910     OUT   ri_err_data_t     *err_data
7911 );
7912
7913 typedef ri_api_return_t (*ri_op_qp_post_bind_t) (
7914     IN    ri_rnic_handle_t  rnic,
7915     IN    ri_qp_handle_t    qp,
7916     IN    ri_wr_type_t      type,
7917     IN    ri_wr_flags_t     flags,
7918     IN    ri_wr_id_t        id,
7919     IN    ri_mw_handle_t    mw_handle,
7920     IN    ri_rtag_t         rtag,
7921     IN    ri_mr_handle_t    mr_handle,
7922     IN    ri_ltag_t         ltag,
7923     IN    ri_addr_t         vaddr,
7924     IN    ri_length_t       length,
7925     IN    ri_mem_attr_t     mem_attr,
7926     OUT   ri_err_data_t     *err_data
7927 );
7928
7929 typedef ri_api_return_t (*ri_op_qp_post_fmr_t) (

```

```

7930         IN    ri_rnic_handle_t    rnic,
7931         IN    ri_qp_handle_t      qp,
7932         IN    ri_wr_type_t        type,
7933         IN    ri_wr_flags_t       flags,
7934         IN    ri_wr_id_t          id,
7935         IN    ri_mr_reg_phys_attr_t *attr,
7936         IN    ri_ltag_t           ltag,
7937         IN    ri_rtag_t           rtag,
7938         IN    ri_mr_handle_t      mr_handle,
7939         OUT   ri_err_data_t       *err_data
7940     );
7941
7942     typedef ri_api_return_t (*ri_op_qp_post_local_invalidate_t) (
7943         IN    ri_rnic_handle_t    rnic,
7944         IN    ri_qp_handle_t      qp,
7945         IN    ri_wr_type_t        type,
7946         IN    ri_wr_flags_t       flags,
7947         IN    ri_wr_id_t          id,
7948         IN    ri_rtag_t           rtag,
7949         IN    ri_ltag_t           ltag,
7950         IN    ri_mr_handle_t      mrw_handle,
7951         OUT   ri_err_data_t       *err_data
7952     );
7953
7954     typedef ri_api_return_t (*ri_op_qp_post_atomic_t) (
7955         IN    ri_rnic_handle_t    rnic,
7956         IN    ri_qp_handle_t      qp,
7957         IN    ri_wr_type_t        type,
7958         IN    ri_wr_flags_t       flags,
7959         IN    ri_wr_id_t          id,
7960         IN    ri_rtag_t           rtag,
7961         IN    uint64_t            atomic_operand1,
7962         IN    uint64_t            atomic_operand2,
7963         IN    ri_data_seg_t       *sg_list,
7964         OUT   ri_err_data_t       *err_data
7965     );
7966
7967     typedef ri_api_return_t (*ri_op_qp_post_datagram_t) (
7968         IN    ri_rnic_handle_t    rnic,
7969         IN    ri_qp_handle_t      qp,
7970         IN    ri_wr_type_t        type,
7971         IN    ri_wr_flags_t       flags,
7972         IN    ri_wr_id_t          id,
7973         IN    ri_data_seg_t       *sg_list,
7974         IN    uint_t              num_sgle,
7975         IN    uint32_t            imm_data,
7976         IN    ri_address_handle_t  remote_node_address,
7977         IN    uint32_t            dest_qp,
7978         IN    uint32_t            dest_qkey,
7979         IN    uint8_t             max_static_rate,
7980         OUT   ri_err_data_t       *err_data
7981     );
7982
7983     typedef ri_api_return_t (*ri_op_qp_postrq_t) (

```

```

7984         IN        ri_rnic_handle_t  rnic,
7985         IN        ri_qp_handle_t    qp_handle,
7986         IN        ri_wr_rcv_t       *wr,
7987         IN OUT    uint_t             *num_wrs,
7988         OUT        ri_err_data_t     *err_data
7989     );
7990
7991     typedef ri_api_return_t (*ri_op_qp_post_rcv_t) (
7992         IN        ri_rnic_handle_t  rnic,
7993         IN        ri_qp_handle_t    qp_handle,
7994         IN        ri_wr_flags_t     flags,
7995         IN        ri_wr_id_t        id,
7996         IN        ri_data_seg_t     *sg_list,
7997         IN        uint_t            num_sgls,
7998         OUT        ri_err_data_t     *err_data
7999     );
8000
8001     typedef ri_api_return_t (*ri_op_ltag_alloc_t) (
8002         IN        ri_rnic_handle_t  rnic,
8003         IN OUT    uint_t            *address_list_length,
8004         IN        ri_pd_handle_t    pd,
8005         IN        ri_mem_attr_t     attr,
8006         OUT        ri_mr_handle_t   *mr,
8007         IN OUT    ri_ltag_t         *ltag,
8008         OUT        ri_rtag_t         *rtag,
8009         IN OUT    ri_vp_data_t      vp_data,
8010         IN OUT    ri_os_data_t      os_data,
8011         OUT        ri_err_data_t     *err_data
8012     );
8013
8014     typedef ri_api_return_t (*ri_op_mr_reg_t) (
8015         IN        ri_rnic_handle_t  rnic,
8016         IN const  ri_mr_reg_attr_t  *mr_reg_attr,
8017         OUT        ri_mr_handle_t   *mr,
8018         IN OUT    ri_ltag_t         *ltag,
8019         OUT        ri_rtag_t         *rtag,
8020         IN OUT    ri_vp_data_t      vp_data,
8021         IN OUT    ri_os_data_t      os_data,
8022         OUT        ri_err_data_t     *err_data
8023     );
8024
8025     typedef ri_api_return_t (*ri_op_mr_dereg_t) (
8026         IN        ri_rnic_handle_t  rnic,
8027         IN        ri_mr_handle_t    mr,
8028         OUT        ri_err_data_t     *err_data_t
8029     );
8030
8031     typedef struct ri_mr_info ri_mr_info_t;
8032     struct ri_mr_info {
8033         ri_mr_reg_attr_t  attr;
8034         ri_ltag_t         ltag;
8035         ri_rtag_t         rtag;
8036         ri_boolean_t     is_shared_region;
8037         ri_addr_t         local_lb;

```

```

8038         ri_addr_t         local_ub;
8039         ri_addr_t         remote_lb;
8040         ri_addr_t         remote_ub;
8041         ri_stag_state_t   stag_state;
8042     };
8043
8044     typedef ri_api_return_t (*ri_op_mr_query_t) (
8045         IN    ri_rnic_handle_t   rnic,
8046         IN    ri_mr_handle_t     mr,
8047         OUT   ri_mr_info_t       *mr_info,
8048         OUT   ri_err_data_t      *err_data
8049     );
8050
8051     typedef ri_api_return_t (*ri_op_mr_rereg_t) (
8052         IN    ri_rnic_handle_t   rnic,
8053         IN    ri_mr_handle_t     in_mr,
8054         IN    ri_mr_rereg_t      change_mask,
8055         IN const ri_mr_reg_attr_t *mr_attr,
8056         OUT   ri_mr_handle_t     *out_mr,
8057         IN OUT ri_ltag_t         *ltag,
8058         OUT   ri_rtag_t         *rtag,
8059         IN OUT ri_vp_data_t      vp_data,
8060         IN OUT ri_os_data_t      os_data,
8061         OUT   ri_err_data_t      *err_data
8062     );
8063
8064     typedef ri_api_return_t (*ri_op_mr_reg_phys_t) (
8065         IN    ri_rnic_handle_t   rnic,
8066         IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,
8067         OUT   ri_mr_handle_t     *mr,
8068         IN OUT ri_ltag_t         *ltag,
8069         OUT   ri_rtag_t         *rtag,
8070         IN OUT ri_vp_data_t      vp_data,
8071         IN OUT ri_os_data_t      os_data,
8072         OUT   ri_err_data_t      *err_data
8073     );
8074
8075     typedef ri_api_return_t (*ri_op_mr_rereg_phys_t) (
8076         IN    ri_rnic_handle_t   rnic,
8077         IN    ri_mr_handle_t     in_mr,
8078         IN    ri_mr_rereg_t      change_mask,
8079         IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,
8080         OUT   ri_mr_handle_t     *out_mr,
8081         IN OUT ri_ltag_t         *ltag,
8082         IN OUT ri_rtag_t         *rtag,
8083         IN OUT ri_vp_data_t      vp_data,
8084         IN OUT ri_os_data_t      os_data,
8085         OUT   ri_err_data_t      *err_data
8086     );
8087
8088     typedef ri_api_return_t (*ri_op_mr_reg_shared_t) (
8089         IN    ri_rnic_handle_t   rnic,
8090         IN    ri_mr_handle_t     in_mr,
8091         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     );

```

```

8146
8147 typedef ri_api_return_t (*ri_op_ah_modify_t) (
8148     IN    ri_rnic_handle_t    rnic,
8149     IN    ri_address_handle_t  ah,
8150     IN    ri_ud_address_vector_t *av,
8151     OUT   ri_err_data_t       *err_data
8152 );
8153
8154 typedef ri_api_return_t (*ri_op_ah_destroy_t) (
8155     IN    ri_rnic_handle_t    rnic,
8156     IN    ri_address_handle_t  ah,
8157     OUT   ri_err_data_t       *err_data
8158 );
8159
8160 typedef enum ri_sqp_type ri_sqp_type_t;
8161 enum ri_sqp_type {
8162     RI_SQP_TYPE_SMI,
8163     RI_SQP_TYPE_GSI
8164 };
8165
8166 typedef ri_api_return_t (*ri_op_special_qp_get_t) (
8167     IN    ri_rnic_handle_t    rnic,
8168     IN    uint8_t             port,
8169     IN    ri_sqp_type_t       qp_type,
8170     IN OUT ri_qp_attr_t       *attrs,
8171     OUT   ri_qp_handle_t      *qp,
8172     IN OUT ri_vp_data_t       vp_data,
8173     IN OUT ri_os_data_t       os_data,
8174     OUT   ri_err_data_t       *err_data
8175 );
8176
8177 typedef ri_api_return_t (*ri_op_process_sma_mad_t) (
8178     IN    ri_rnic_handle_t    rnic,
8179     IN    uint8_t             port_num,
8180     IN    char                 *in_mad,
8181     OUT   char                 *out_mad,
8182     OUT   ri_err_data_t       *err_data
8183 );
8184
8185 typedef ri_api_return_t (*ri_op_mcast_attach_t) (
8186     IN    ri_rnic_handle_t    rnic,
8187     IN    ri_lid_t            mlid,
8188     IN    ri_gid_t            *mgid,
8189     IN    ri_qp_handle_t      qp,
8190     IN OUT ri_vp_data_t       vp_data,
8191     IN OUT ri_os_data_t       os_data,
8192     OUT   ri_err_data_t       *err_data
8193 );
8194
8195 typedef ri_api_return_t (*ri_op_mcast_detach_t) (
8196     IN    ri_rnic_handle_t    rnic,
8197     IN    ri_lid_t            mlid,
8198     IN    ri_gid_t            *mgid,
8199     IN    ri_qp_handle_t      qp,

```

```

8200         OUT  ri_err_data_t    *err_data
8201     );
8202
8203     typedef ri_api_return_t (*ri_op_mw_dealloc_t) (
8204         IN    ri_rnic_handle_t  rnic,
8205         IN    ri_mw_handle_t    mw,
8206         OUT  ri_err_data_t    *err_data
8207     );
8208
8209     typedef ri_api_return_t (*ri_op_mem_sync_rread_t) (
8210         IN    ri_rnic_handle_t  rnic,
8211         IN    ri_data_seg_t     *local_seg_list,
8212         IN    uint_t            num_seg,
8213         OUT  ri_err_data_t    *err_data
8214     );
8215
8216     typedef ri_api_return_t (*ri_op_mem_sync_rwrite_t) (
8217         IN    ri_rnic_handle_t  rnic,
8218         IN    ri_data_seg_t     *local_seg_list,
8219         IN    uint_t            num_seg,
8220         OUT  ri_err_data_t    *err_data
8221     );
8222
8223     typedef ri_api_return_t (*ri_op_mem_pin_t) (
8224         IN    ri_rnic_handle_t  rnic,
8225         IN    ri_vaddr_t        vaddr,
8226         IN    ri_length_t       len,
8227         IN OUT ri_os_data_t     os_data,
8228         OUT  ri_err_data_t     *err_data
8229     );
8230
8231     typedef ri_api_return_t (*ri_op_mem_unpin_t) (
8232         IN    ri_rnic_handle_t  rnic,
8233         IN    ri_vaddr_t        vaddr,
8234         IN    ri_length_t       len,
8235         IN OUT ri_os_data_t     os_data,
8236         OUT  ri_err_data_t     *err_data
8237     );
8238
8239     typedef ri_api_return_t (*ri_op_mem_map_t) (
8240         IN    ri_rnic_handle_t  rnic,
8241         IN    ri_vaddr_t        vaddr,
8242         IN    ri_length_t       len,
8243         OUT  ri_phys_buf_t     *io,
8244         IN OUT ri_os_data_t     os_data,
8245         OUT  ri_err_data_t     *err_data
8246     );
8247
8248     typedef ri_api_return_t (*ri_op_mem_unmap_t) (
8249         IN    ri_rnic_handle_t  rnic,
8250         IN    ri_phys_buf_t     *io,
8251         IN OUT ri_os_data_t     os_data,
8252         OUT  ri_err_data_t     *err_data
8253     );

```

```

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;

```

```

8362         ri_op_mem_unmap_t           ri_op_mem_unmap;
8363         ri_op_mem_sync_rread_t      ri_op_mem_sync_rread;
8364         ri_op_mem_sync_rwrite_t     ri_op_mem_sync_rwrite;
8365         ri_op_ah_create_t           ri_op_ah_create;
8366         ri_op_ah_query_t            ri_op_ah_query;
8367         ri_op_ah_modify_t           ri_op_ah_modify;
8368         ri_op_ah_destroy_t          ri_op_ah_destroy;
8369         ri_op_mcast_attach_t        ri_op_mcast_attach;
8370         ri_op_mcast_detach_t        ri_op_mcast_detach;
8371         ri_op_process_sma_mad_t      ri_op_process_sma_mad;
8372     };
8373
8374     /*
8375     * the actual svc routines
8376     */
8377
8378     ri_api_return_t ri_svc_attach(
8379         IN    ri_vp_handle_t          vp_hdl,
8380         IN    ri_dev_info_t           dev_info,
8381         IN    ri_services_dev_arg_t   *svcs,
8382         IN    ri_ops_t                *al_locking_ops,
8383         IN    ri_ops_t                *vp_locking_ops,
8384         OUT   ri_err_data_t           *err_data
8385     );
8386
8387     ri_api_return_t ri_svc_detach(
8388         IN    ri_vp_handle_t          vp_hdl,
8389         OUT   ri_err_data_t          *err_data
8390     );
8391
8392     ri_api_return_t ri_svc_lib_reg(
8393         IN    char                    *lib_path,
8394         IN    ri_ops_t                *al_locking_ops,
8395         IN    ri_ops_t                *vp_locking_ops,
8396         IN OUT ri_os_data_t           os_data,
8397         OUT   ri_err_data_t          *err_data
8398     );
8399
8400     ri_api_return_t ri_svc_mem_pin(
8401         IN    ri_vaddr_t              vaddr,
8402         IN    ri_length_t             len,
8403         IN    ri_os_data_t            os_data,
8404         OUT   ri_err_data_t          *err_data
8405     );
8406
8407     ri_api_return_t ri_svc_mem_unpin(
8408         IN    ri_vaddr_t              vaddr,
8409         IN    ri_length_t             len,
8410         IN    ri_os_data_t            os_data,
8411         OUT   ri_err_data_t          *err_data
8412     );
8413
8414     #include "rnicpi_osdfunc.h"
8415     #endif

```

8416 10.2 <rnicipi_osd.h>

```
8417 /*
8418 * rnicipi_osd.h -- prototype of OS-specific file to support rnicipi.h
8419 *
8420 * Copyright © 2005 The Open Group
8421 * All rights reserved.
8422 * The copyright owner hereby grants permission for all or part of this
8423 * publication to be reproduced, stored in a retrieval system, or
8424 * transmitted, in any form or by any means, electronic, mechanical,
8425 * photocopying, recording, or otherwise, provided that it remains
8426 * unchanged and that this copyright statement is included in all
8427 * copies or substantial portions of the publication.
8428 *
8429 * For any software code contained within this specification,
8430 * permission is hereby granted, free-of-charge, to any person
8431 * obtaining a copy of this specification (the "Software"), to deal in
8432 * the Software without restriction, including without limitation the
8433 * rights to use, copy, modify, merge, publish, distribute, sublicense,
8434 * and/or sell copies of the Software, and to permit persons to whom
8435 * the Software is furnished to do so, subject to the above copyright
8436 * notice and this permission notice being included in all copies or
8437 * substantial portions of the Software.
8438 *
8439 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
8440 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
8441 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
8442 * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
8443 * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
8444 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
8445 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
8446 * SOFTWARE.
8447 *
8448 * Permission is granted for implementers to use the names, labels,
8449 * etc. contained within the specification. The intent of publication
8450 * of the specification is to encourage implementations of the
8451 * specification. This specification has not been verified for
8452 * avoidance of possible third-party proprietary rights. In
8453 * implementing this specification, usual procedures to ensure the
8454 * respect of possible third-party intellectual property rights should
8455 * be followed.
8456 */
8457
8458 #ifndef _RNICPI_OSD_H_
8459 #define _RNICPI_OSD_H_
8460
8461 #define IN
8462 #define OUT
8463
8464 typedef int ri_llp_stream_handle_t;
8465 typedef unsigned int uint_t;
8466 typedef unsigned short ri_boolean_t;
8467 typedef unsigned int ri_length_t;
8468
```

```

8469 struct ri_handle_struct {
8470     const struct ri_ops *ops;
8471     /* implementation and type-dependent fields follow
8472      * This format MUST be used for vp_handle and ri_rnic_handle_t.
8473      * It MAY be used for all other handles.
8474      *
8475      * Other implementations may vary, but must provide a mechanism to
8476      * translate from a vp_handle or rnic_handle to a struct ri_ops. */
8477     */
8478 };
8479
8480 typedef void          *ri_handle_t;
8481 typedef ri_handle_t  ri_vp_handle_t;
8482 typedef ri_handle_t  ri_rnic_handle_t;
8483 typedef ri_handle_t  ri_pd_handle_t;
8484 typedef ri_handle_t  ri_qp_handle_t;
8485 typedef ri_handle_t  ri_cq_handle_t;
8486 typedef ri_handle_t  ri_mr_handle_t;
8487 typedef ri_handle_t  ri_mw_handle_t;
8488 typedef ri_handle_t  ri_srq_handle_t;
8489 typedef ri_handle_t  ri_address_handle_t;
8490 typedef void          *ri_vp_data_t;
8491 typedef void          *ri_os_data_t;
8492 typedef uint64_t      ri_err_data_t;
8493
8494 typedef char          *ri_dev_info_t;
8495
8496 typedef void          *ri_vaddr_t;
8497
8498 static __inline__ const struct ri_ops *ri_svc_ops(
8499     IN void *handle
8500 )
8501 {
8502     return ((struct ri_handle_struct *)handle)->ops;
8503 }
8504
8505 static __inline__ void *ri_svc_handle_ptr(
8506     IN ri_handle_t handle
8507 )
8508 {
8509     return (void *)handle;
8510 }
8511
8512 static __inline__ ri_handle_t ri_svc_make_handle(
8513     IN struct ri_handle_struct *p
8514 )
8515 {
8516     return (ri_handle_t)p;
8517 }
8518
8519 typedef uint64_t      ri_phys_addr_t;
8520 typedef uint64_t      ri_phys_buf_len_t;
8521
8522 #define RI_MAX_LIB_PATH_LEN 64

```



```
8523 #define RI_MAX_HW_PATH_LEN 64
8524 #define RI_MAX_VENDOR_STRING_LEN 64
8525
8526 #endif
```

8527 **10.3** <rnici_osdfunc.h>

```
8528 /*
8529  * rnicpi_osdfunc.h - Platform-specific function declarations
8530  *
8531  * Copyright © 2005 The Open Group
8532  * All rights reserved.
8533  * The copyright owner hereby grants permission for all or part of this
8534  * publication to be reproduced, stored in a retrieval system, or
8535  * transmitted, in any form or by any means, electronic, mechanical,
8536  * photocopying, recording, or otherwise, provided that it remains
8537  * unchanged and that this copyright statement is included in all
8538  * copies or substantial portions of the publication.
8539  *
8540  * For any software code contained within this specification,
8541  * permission is hereby granted, free-of-charge, to any person
8542  * obtaining a copy of this specification (the "Software"), to deal in
8543  * the Software without restriction, including without limitation the
8544  * rights to use, copy, modify, merge, publish, distribute, sublicense,
8545  * and/or sell copies of the Software, and to permit persons to whom
8546  * the Software is furnished to do so, subject to the above copyright
8547  * notice and this permission notice being included in all copies or
8548  * substantial portions of the Software.
8549  *
8550  * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
8551  * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
8552  * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND
8553  * NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
8554  * BE LIABLE FOR ANY CLAIM, DAMAGES, OR OTHER LIABILITY, WHETHER IN AN
8555  * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF, OR IN
8556  * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
8557  * SOFTWARE.
8558  *
8559  * Permission is granted for implementers to use the names, labels,
8560  * etc. contained within the specification. The intent of publication
8561  * of the specification is to encourage implementations of the
8562  * specification. This specification has not been verified for
8563  * avoidance of possible third-party proprietary rights. In
8564  * implementing this specification, usual procedures to ensure the
8565  * respect of possible third-party intellectual property rights should
8566  * be followed.
8567  */
8568 /*
8569  * Inline re-direction functions. Each platform is encouraged
8570  * to provide these named symbols, using these inline definitions
8571  * or by other means. If the symbols are not provided, platform-
8572  * specific instructions on how to invoke the correct function
8573  * through the provider's ri_ops_t should be provided.
```

```

8574     */
8575
8576 #ifndef _RNICPI_OSDFUNC_H_
8577 #define _RNICPI_OSDFUNC_H_
8578
8579 static __inline__ ri_api_return_t ri_async_set_event_handler(
8580     IN    ri_vp_handle_t    vp_hdl,
8581     IN    ri_async_event_callback_t  async_event_callback,
8582     OUT   ri_err_data_t    *err_data
8583 )
8584 {
8585     return ri_svc_ops(vp_hdl)->ri_op_async_set_event_handler(vp_hdl,
8586         async_event_callback,err_data);
8587 }
8588
8589 static __inline__ ri_api_return_t ri_rnic_open(
8590     IN    ri_vp_handle_t    vp_hdl,
8591     IN    ri_pbl_mode_t    pbl_mode,
8592     IN OUT ri_os_data_t    os_data,
8593     OUT   ri_err_data_t    *err_data)
8594 {
8595     return ri_svc_ops(vp_hdl)->ri_op_rnic_open(vp_hdl,pbl_mode,
8596         os_data,err_data);
8597 }
8598
8599 static __inline__ ri_api_return_t ri_rnic_close(
8600     IN    ri_vp_handle_t    vp_hdl,
8601     OUT   ri_err_data_t    *err_data)
8602 {
8603     return ri_svc_ops(vp_hdl)->ri_op_rnic_close(vp_hdl,err_data);
8604 }
8605
8606 static __inline__ ri_api_return_t ri_rnic_free_hdl(
8607     IN    ri_vp_handle_t    vp_hdl,
8608     IN    ri_rnic_handle_t  rnic,
8609     OUT   ri_err_data_t    *err_data)
8610 {
8611     return ri_svc_ops(vp_hdl)->ri_op_rnic_free_hdl(vp_hdl,
8612         rnic,err_data);
8613 }
8614
8615 static __inline__ ri_api_return_t ri_rnic_query(
8616     IN    ri_rnic_handle_t  rnic,
8617     IN    ri_rnic_query_flags_t  flags,
8618     OUT   ri_rnic_attr_t    *attrs,
8619     IN OUT uint_t           *attr_size,
8620     OUT   ri_err_data_t    *err_data)
8621 {
8622     return ri_svc_ops(rnic)->ri_op_rnic_query(rnic,flags,
8623         attrs,attr_size,err_data);
8624 }
8625
8626 static __inline__ ri_api_return_t ri_modify_rnic(
8627     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,

```

```

8790         srq_attrs,err_data);
8791     }
8792
8793     static __inline__ ri_api_return_t ri_srq_modify(
8794         IN    ri_rnic_handle_t    rnic,
8795         IN    ri_srq_handle_t     srq,
8796         IN    ri_srq_attr_mask_t  srqmask,
8797         OUT   ri_srq_attr_t       *attrs,
8798         OUT   ri_err_data_t       *err_data)
8799     {
8800         return ri_svc_ops(rnic)->ri_op_srq_modify(rnic,srq,srqmask,attrs,
8801             err_data);
8802     }
8803
8804     static __inline__ ri_api_return_t ri_srq_destroy(
8805         IN    ri_rnic_handle_t    rnic,
8806         IN    ri_srq_handle_t     srq,
8807         OUT   ri_err_data_t*      err_data)
8808     {
8809         return ri_svc_ops(rnic)->ri_op_srq_destroy(rnic,srq,err_data);
8810     }
8811
8812     static __inline__ ri_api_return_t ri_qp_postsq(
8813         IN    ri_rnic_handle_t    rnic,
8814         IN    ri_qp_handle_t      qp,
8815         IN    ri_wr_t             *wr,
8816         IN OUT uint_t             *num_wrs,
8817         OUT   ri_err_data_t       *err_data)
8818     {
8819         return ri_svc_ops(rnic)->ri_op_qp_postsq(rnic,qp,wr,
8820             num_wrs,err_data);
8821     }
8822
8823     static __inline__ ri_api_return_t ri_qp_post_send(
8824         IN    ri_rnic_handle_t    rnic,
8825         IN    ri_qp_handle_t      qp,
8826         IN    ri_wr_type_t        type,
8827         IN    ri_wr_flags_t       flags,
8828         IN    ri_wr_id_t          id,
8829         IN    ri_data_seg_t       *sg_list,
8830         IN    ri_rtag_t           rtag,
8831         IN    uint_t              num_sgls,
8832         IN    uint32_t            imm_data,
8833         OUT   ri_err_data_t       *err_data
8834     )
8835     {
8836         return ri_svc_ops(rnic)->ri_op_qp_post_send(rnic,qp,type,flags,id,
8837             sg_list,rtag,num_sgls,imm_data,err_data);
8838     }
8839
8840     static __inline__ ri_api_return_t ri_qp_post_rdma_write(
8841         IN    ri_rnic_handle_t    rnic,
8842         IN    ri_qp_handle_t      qp,
8843         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,

```

```

8898     IN    ri_wr_type_t          type,
8899     IN    ri_wr_flags_t        flags,
8900     IN    ri_wr_id_t           id,
8901     IN    ri_mr_reg_phys_attr_t *attr,
8902     IN    ri_ltag_t            ltag,
8903     IN    ri_rtag_t            rtag,
8904     IN    ri_mr_handle_t       mr_handle,
8905     OUT   ri_err_data_t        *err_data
8906 )
8907 {
8908     return ri_svc_ops(rnic)->ri_op_qp_post_fmr(rnic,qp,type,flags,
8909         id,attr,ltag,rtag,mr_handle,err_data);
8910 }
8911
8912 static __inline__ ri_api_return_t ri_qp_post_local_invalidate(
8913     IN    ri_rnic_handle_t  rnic,
8914     IN    ri_qp_handle_t    qp,
8915     IN    ri_wr_type_t      type,
8916     IN    ri_wr_flags_t    flags,
8917     IN    ri_wr_id_t       id,
8918     IN    ri_rtag_t        rtag,
8919     IN    ri_ltag_t        ltag,
8920     IN    ri_mr_handle_t   mrw_handle,
8921     OUT   ri_err_data_t    *err_data
8922 )
8923 {
8924     return ri_svc_ops(rnic)->ri_op_qp_post_local_invalidate(rnic,qp,
8925         type,flags,id,rtag,ltag,mw_handle,err_data);
8926 }
8927
8928 static __inline__ ri_api_return_t ri_qp_post_atomic(
8929     IN    ri_rnic_handle_t  rnic,
8930     IN    ri_qp_handle_t    qp,
8931     IN    ri_wr_type_t      type,
8932     IN    ri_wr_flags_t    flags,
8933     IN    ri_wr_id_t       id,
8934     IN    ri_rtag_t        rtag,
8935     IN    uint64_t          atomic_operand1,
8936     IN    uint64_t          atomic_operand2,
8937     IN    ri_data_seg_t     *atomic_result,
8938     OUT   ri_err_data_t    *err_data
8939 )
8940 {
8941     return ri_svc_ops(rnic)->ri_op_qp_post_atomic(rnic,qp,type,flags,
8942         id,rtag,atomic_operand1,atomic_operand2,atomic_result,
8943         err_data);
8944 }
8945
8946 static __inline__ ri_api_return_t ri_qp_postrq(
8947     IN    ri_rnic_handle_t  rnic,
8948     IN    ri_qp_handle_t    qp_handle,
8949     IN    ri_wr_recv_t       *wr,
8950     IN OUT uint_t           *num_wrs,
8951     OUT   ri_err_data_t     *err_data)

```



```

8952     {
8953         return ri_svc_ops(rnic)->ri_op_qp_postrq(rnic,qp_handle,wr,
8954             num_wrs,err_data);
8955     }
8956
8957     static __inline__ ri_api_return_t ri_qp_post_recv(
8958         IN    ri_rnic_handle_t  rnic,
8959         IN    ri_qp_handle_t    qp_handle,
8960         IN    ri_wr_flags_t     flags,
8961         IN    ri_wr_id_t        id,
8962         IN    ri_data_seg_t     *sg_list,
8963         IN    uint_t            num_sgls,
8964         OUT   ri_err_data_t     *err_data
8965     )
8966     {
8967         return ri_svc_ops(rnic)->ri_op_qp_post_recv(rnic,qp_handle,flags,
8968             id,sg_list,num_sgls,err_data);
8969     }
8970
8971     static __inline__ ri_api_return_t ri_srq_post(
8972         IN    ri_rnic_handle_t  rnic,
8973         IN    ri_srq_handle_t   srq_handle,
8974         IN    ri_wr_recv_t      *wr,
8975         IN OUT uint_t           *num_wrs,
8976         OUT   ri_err_data_t     *err_data)
8977     {
8978         return ri_svc_ops(rnic)->ri_op_srq_post(rnic,srq_handle,wr,num_wrs,
8979             err_data);
8980     }
8981
8982     static __inline__ ri_api_return_t ri_srq_post_recv(
8983         IN    ri_rnic_handle_t  rnic,
8984         IN    ri_handle_t       srq_handle,
8985         IN    ri_wr_flags_t     flags,
8986         IN    ri_wr_id_t        wrid,
8987         IN    ri_data_seg_t     *sg_list,
8988         IN    uint_t            num_sgls,
8989         OUT   ri_err_data_t     *err_data)
8990     {
8991         return ri_svc_ops(rnic)->ri_op_srq_post_recv(rnic,srq_handle,
8992             flags,wrid,sg_list,num_sgls,err_data);
8993     }
8994
8995     static __inline__ ri_api_return_t ri_ltag_alloc(
8996         IN    ri_rnic_handle_t  rnic,
8997         IN OUT uint_t           *address_list_length,
8998         IN    ri_pd_handle_t    pd,
8999         IN    ri_mem_attr_t     attr,
9000         OUT   ri_mr_handle_t    *mr,
9001         IN OUT ri_ltag_t        *ltag,
9002         IN OUT ri_rtag_t        *rtag,
9003         IN    ri_vp_data_t      vp_data,
9004         IN    ri_os_data_t      os_data,
9005         OUT   ri_err_data_t     *err_data)

```

```

9006     {
9007         return ri_svc_ops(rnic)->ri_op_ltag_alloc(rnic,
9008             address_list_length,pd,attr,mr,ltag,rtag,vp_data,
9009             os_data,err_data);
9010     }
9011
9012     static __inline__ ri_api_return_t ri_mr_reg(
9013         IN      ri_rnic_handle_t    rnic,
9014         IN const ri_mr_reg_attr_t   *mr_reg_attr,
9015         OUT     ri_mr_handle_t      *mem,
9016         OUT     ri_ltag_t           *ltag,
9017         OUT     ri_rtag_t           *rtag,
9018         IN      ri_vp_data_t        vp_data,
9019         IN      ri_os_data_t        os_data,
9020         OUT     ri_err_data_t       *err_data)
9021     {
9022         return ri_svc_ops(rnic)->ri_op_mr_reg(rnic,mr_reg_attr,mem,ltag,
9023             rtag,vp_data,os_data,err_data);
9024     }
9025
9026     static __inline__ ri_api_return_t ri_mr_dereg(
9027         IN      ri_rnic_handle_t    rnic,
9028         IN      ri_mr_handle_t      mem,
9029         OUT     ri_err_data_t       *err_data)
9030     {
9031         return ri_svc_ops(rnic)->ri_op_mr_dereg(rnic,mem,err_data);
9032     }
9033
9034     static __inline__ ri_api_return_t ri_mr_query(
9035         IN      ri_rnic_handle_t    rnic,
9036         IN      ri_mr_handle_t      mr,
9037         OUT     ri_mr_info_t        *mr_info,
9038         OUT     ri_err_data_t       *err_data)
9039     {
9040         return ri_svc_ops(rnic)->ri_op_mr_query(rnic,mr,mr_info,
9041             err_data);
9042     }
9043
9044     static __inline__ ri_api_return_t ri_mr_rereg(
9045         IN      ri_rnic_handle_t    rnic,
9046         IN      ri_mr_handle_t      in_mr,
9047         IN      ri_mr_rereg_t       change_mask,
9048         IN const ri_mr_reg_attr_t   *mem_attr,
9049         OUT     ri_mr_handle_t      *out_mr,
9050         IN OUT  ri_ltag_t           *ltag,
9051         OUT     ri_rtag_t           *rtag,
9052         IN      ri_vp_data_t        vp_data,
9053         IN      ri_os_data_t        os_data,
9054         OUT     ri_err_data_t       *err_data)
9055     {
9056         return ri_svc_ops(rnic)->ri_op_mr_rereg(rnic,in_mr,change_mask,
9057             mem_attr,out_mr,ltag,rtag,vp_data,os_data,err_data);
9058     }
9059

```

```

9060 static __inline__ ri_api_return_t ri_mr_reg_phys(
9061     IN ri_rnic_handle_t rnic,
9062     IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,
9063     OUT ri_mr_handle_t *mr,
9064     IN OUT ri_ltag_t *ltag,
9065     OUT ri_rtag_t *rtag,
9066     IN ri_vp_data_t vp_data,
9067     IN ri_os_data_t os_data,
9068     OUT ri_err_data_t *err_data)
9069 {
9070     return ri_svc_ops(rnic)->ri_op_mr_reg_phys(rnic,mr_phys_attr,mr,
9071         ltag,rtag,vp_data,os_data,err_data);
9072 }
9073
9074 static __inline__ ri_api_return_t ri_mr_rereg_phys(
9075     IN ri_rnic_handle_t rnic,
9076     IN ri_mr_handle_t in_mr,
9077     IN ri_mr_rereg_t change_mask,
9078     IN OUT ri_mr_reg_phys_attr_t *mr_phys_attr,
9079     OUT ri_mr_handle_t *out_mr,
9080     IN OUT ri_ltag_t *ltag,
9081     IN OUT ri_rtag_t *rtag,
9082     IN OUT ri_vp_data_t vp_data,
9083     IN OUT ri_os_data_t os_data,
9084     OUT ri_err_data_t *err_data)
9085 {
9086     return ri_svc_ops(rnic)->ri_op_mr_rereg_phys(rnic,in_mr,
9087         change_mask,mr_phys_attr,out_mr,ltag,rtag,vp_data,
9088         os_data,err_data);
9089 }
9090
9091 static __inline__ ri_api_return_t ri_mr_reg_shared(
9092     IN ri_rnic_handle_t rnic,
9093     IN ri_mr_handle_t in_mr,
9094     IN OUT ri_mr_reg_attr_t *mem_reg_attr,
9095     OUT ri_mr_handle_t *out_mr,
9096     IN OUT ri_ltag_t *ltag,
9097     OUT ri_rtag_t *rtag,
9098     IN ri_vp_data_t vp_data,
9099     IN ri_os_data_t os_data,
9100     OUT ri_err_data_t *err_data)
9101 {
9102     return ri_svc_ops(rnic)->ri_op_mr_reg_shared(rnic,in_mr,
9103         mem_reg_attr,out_mr,ltag,rtag,vp_data,os_data,err_data);
9104 }
9105
9106 static __inline__ ri_api_return_t ri_mw_alloc(
9107     IN i_rnic_handle_t rnic,
9108     IN i_pd_handle_t pd,
9109     IN i_mw_type_t mw_type,
9110     OUT i_mw_handle_t *mw,
9111     OUT i_rtag_t *rtag,
9112     IN i_vp_data_t vp_data,
9113     IN i_os_data_t os_data,

```

```

9114     OUT i_err_data_t    *err_data)
9115 {
9116     return ri_svc_ops(rnic)->ri_op_mw_alloc(rnic,pd,mw_type,mw,rtag,
9117         vp_data,os_data,err_data);
9118 }
9119
9120 static __inline__ ri_api_return_t ri_mw_query(
9121     IN  ri_rnic_handle_t  rnic,
9122     IN  ri_mw_handle_t   mw,
9123     OUT ri_mw_type_t     *mw_type,
9124     OUT ri_stag_state_t  *stag_state,
9125     OUT ri_pd_handle_t   *pd,
9126     OUT ri_mem_attr_t    *attr,
9127     OUT ri_rtag_t        *rtag,
9128     OUT ri_err_data_t    *err_data)
9129 {
9130     return ri_svc_ops(rnic)->ri_op_mw_query(rnic,mw,mw_type,
9131         stag_state,pd,attr,rtag,err_data);
9132 }
9133
9134 static __inline__ ri_api_return_t ri_mw_dealloc(
9135     IN  ri_rnic_handle_t  rnic,
9136     IN  ri_mw_handle_t   mw,
9137     OUT ri_err_data_t    *err_data)
9138 {
9139     return ri_svc_ops(rnic)->ri_op_mw_dealloc(rnic,mw,err_data);
9140 }
9141
9142 static __inline__ ri_api_return_t ri_ah_create(
9143     IN  ri_rnic_handle_t  rnic,
9144     IN  ri_pd_handle_t    pd,
9145     IN  ri_ud_address_vector_t *av,
9146     OUT ri_address_handle_t *ah,
9147     IN OUT ri_vp_data_t   vp_data,
9148     IN OUT ri_os_data_t   os_data,
9149     OUT  ri_err_data_t    *err_data)
9150 {
9151     return ri_svc_ops(rnic)->ri_op_ah_create(rnic,pd,av,ah,vp_data,
9152         os_data,err_data);
9153 }
9154
9155 static __inline__ ri_api_return_t ri_ah_query(
9156     IN  ri_rnic_handle_t  rnic,
9157     IN  ri_address_handle_t ah,
9158     OUT ri_ud_address_vector_t *av,
9159     OUT ri_pd_handle_t    *pd,
9160     OUT ri_err_data_t     *err_data)
9161 {
9162     return ri_svc_ops(rnic)->ri_op_ah_query(rnic,ah,av,pd,err_data);
9163 }
9164
9165 static __inline__ ri_api_return_t ri_ah_modify(
9166     IN  ri_rnic_handle_t  rnic,
9167     IN  ri_address_handle_t ah,

```

```

9168     IN   ri_ud_address_vector_t  *av,
9169     OUT  ri_err_data_t           *err_data)
9170 {
9171     return ri_svc_ops(rnic)->ri_op_ah_modify(rnic,ah,av,err_data);
9172 }
9173
9174 static __inline__ ri_api_return_t ri_ah_destroy(
9175     IN   ri_rnic_handle_t    rnic,
9176     IN   ri_address_handle_t ah,
9177     OUT  ri_err_data_t       *err_data)
9178 {
9179     return ri_svc_ops(rnic)->ri_op_ah_destroy(rnic,ah,err_data);
9180 }
9181
9182 static __inline__ ri_api_return_t ri_special_qp_get(
9183     IN   ri_rnic_handle_t    rnic,
9184     IN   uint8_t             port,
9185     IN   ri_sqp_type_t       qp_type,
9186     IN OUT ri_qp_attr_t      *attrs,
9187     OUT  ri_qp_handle_t      *qp,
9188     IN OUT ri_vp_data_t      vp_data,
9189     IN OUT ri_os_data_t      os_data,
9190     OUT  ri_err_data_t       *err_data)
9191 {
9192     return ri_svc_ops(rnic)->ri_op_special_qp_get(rnic,port,qp_type,
9193         attrs,qp,vp_data,os_data,err_data);
9194 }
9195
9196 static __inline__ ri_api_return_t ri_process_sma_mad(
9197     IN   ri_rnic_handle_t    rnic,
9198     IN   uint8_t             port_num,
9199     IN   char                 *in_mad,
9200     OUT  char                 *out_mad,
9201     OUT  ri_err_data_t       *err_data)
9202 {
9203     return ri_svc_ops(rnic)->ri_op_process_sma_mad(rnic,port_num,
9204         in_mad,out_mad,err_data);
9205 }
9206
9207 static __inline__ ri_api_return_t ri_mcast_attach(
9208     IN   ri_rnic_handle_t    rnic,
9209     IN   ri_lid_t            dlid,
9210     IN   ri_gid_t            *mgid,
9211     IN   ri_qp_handle_t      qp,
9212     IN OUT ri_vp_data_t      vp_data,
9213     IN OUT ri_os_data_t      os_data,
9214     OUT  ri_err_data_t       *err_data)
9215 {
9216     return ri_svc_ops(rnic)->ri_op_mcast_attach(rnic,dlid,mgid,qp,
9217         vp_data,os_data,err_data);
9218 }
9219
9220 static __inline__ ri_api_return_t ri_mcast_detach(
9221     IN   ri_rnic_handle_t    rnic,

```

```

9222         IN   uint32_t          dlid,
9223         IN   ri_gid_t          *gid,
9224         IN   ri_qp_handle_t    qp,
9225         OUT  ri_err_data_t     *err_data)
9226     {
9227         return ri_svc_ops(rnic)->ri_op_mcast_detach(rnic,dlid,gid,qp,
9228             err_data);
9229     }
9230
9231     static __inline__ ri_api_return_t ri_mem_sync_rread(
9232         IN   ri_rnic_handle_t    rnic,
9233         IN   ri_data_seg_t       *local_seg_list,
9234         IN   uint_t              num_seg,
9235         OUT  ri_err_data_t       *err_data
9236     )
9237     {
9238         return ri_svc_ops(rnic)->ri_op_mem_sync_rread(rnic,local_seg_list,
9239             num_seg,err_data);
9240     }
9241
9242     static __inline__ ri_api_return_t ri_mem_sync_rwrite(
9243         IN   ri_rnic_handle_t    rnic,
9244         IN   ri_data_seg_t       *local_seg_list,
9245         IN   uint_t              num_seg,
9246         OUT  ri_err_data_t       *err_data
9247     )
9248     {
9249         return ri_svc_ops(rnic)->ri_op_mem_sync_rwrite(rnic,local_seg_list,
9250             num_seg,err_data);
9251     }
9252
9253     static __inline__ ri_api_return_t ri_mem_pin(
9254         IN   ri_rnic_handle_t    rnic,
9255         IN   ri_vaddr_t          vaddr,
9256         IN   ri_length_t         len,
9257         IN   ri_os_data_t        os_data,
9258         OUT  ri_err_data_t       *err_data)
9259     {
9260         return ri_svc_ops(rnic)->ri_op_mem_pin(rnic,vaddr,len,os_data,
9261             err_data);
9262     }
9263
9264     static __inline__ ri_api_return_t ri_mem_unpin(
9265         IN   ri_rnic_handle_t    rnic,
9266         IN   ri_vaddr_t          vaddr,
9267         IN   ri_length_t         len,
9268         IN OUT ri_os_data_t       os_data,
9269         OUT  ri_err_data_t       *err_data)
9270     {
9271         return ri_svc_ops(rnic)->ri_op_mem_unpin(rnic,vaddr,len,os_data,
9272             err_data);
9273     }
9274
9275     static __inline__ ri_api_return_t ri_mem_map(

```

```

9276         IN      ri_rnic_handle_t  rnic,
9277         IN      ri_vaddr_t        vaddr,
9278         IN      ri_length_t       len,
9279         OUT     ri_phys_buf_t     *io,
9280         IN OUT  ri_os_data_t       os_data,
9281         OUT     ri_err_data_t     *err_data)
9282     {
9283         return ri_svc_ops(rnic)->ri_op_mem_map(rnic,vaddr,len,io,os_data,
9284         err_data);
9285     }
9286
9287     static __inline__ ri_api_return_t ri_mem_unmap(
9288         IN      ri_rnic_handle_t  rnic,
9289         IN      ri_phys_buf_t     *io,
9290         IN      ri_os_data_t       os_data,
9291         OUT     ri_err_data_t     *err_data)
9292     {
9293         return ri_svc_ops(rnic)->ri_op_mem_unmap(rnic,io,os_data,
9294         err_data);
9295     }
9296
9297     static __inline__ ri_api_return_t ri_op_rnic_diag(
9298         IN      ri_rnic_handle_t  rnic,
9299         IN      ri_diag_op_type_t op,
9300         IN      void              *arg,
9301         IN OUT  unsigned char     *opval,
9302         IN OUT  ri_length_t       *oplen,
9303         OUT     ri_err_data_t     *err_data)
9304     {
9305         return ri_svc_ops(rnic)->ri_op_rnic_diag(rnic,op,arg,opval,oplen,
9306         err_data);
9307     }
9308
9309     /* get_hdl must be an actual function
9310     */
9311
9312     extern ri_api_return_t ri_rnic_get_hdl(
9313         IN      ri_vp_handle_t     vp_hdl,
9314         IN      ri_consumer_domain_t location,
9315         OUT     ri_rnic_handle_t   *rnic,
9316         IN      ri_vp_data_t       vp_data,
9317         IN      ri_os_data_t       os_data,
9318         OUT     ri_err_data_t     *err_data);
9319
9320 #endif
9321

```