

1 *Consortium Specification*

2 **Interconnect Transport API (IT-API)**

3 **Issue 1.0**

4 **The Interconnect Software Consortium**

5 in association with

6  THE *Open* GROUP

7 Copyright © 2004, 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 any
14 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 NONINFRINGEMENT. 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 THE
25 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 Consortium Specification

32 **Interconnect Transport API (IT-API) Issue 1.0**

33 ISBN: 1-931624-37-2

34 Document Number: C040

35 Published by The Open Group, February 2004.

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

37 The Open Group
38 Apex Plaza, Forbury Road
39 Reading
40 Berkshire, RG1 1AX
41 United Kingdom

42 or by electronic mail to:

43 ogspecs@opengroup.org

44

Contents

45 1 Introduction..... 1

46 1.1 Interface Adapters..... 1

47 1.2 Memory Management..... 2

48 1.3 Communication Endpoints 2

49 1.4 Data Transfer Operations..... 3

50 1.5 Events 3

51 1.6 Event Notification..... 4

52 2 Definitions..... 5

53 3 Global Behavior 15

54 3.1 Non-Blocking APIs..... 15

55 3.2 Thread Safety..... 15

56 3.3 Signal Handlers..... 21

57 3.4 Fork Semantics 21

58 3.5 Exec Semantics 21

59 3.6 Exit Semantics 21

60 3.7 Error Handling 21

61 3.8 IT Handle Management 22

62 4 API Manual Pages..... 23

63 it_address_handle_create()..... 26

64 it_address_handle_free()..... 29

65 it_address_handle_modify()..... 30

66 it_address_handle_query() 32

67 it_convert_net_addr()..... 34

68 it_ep_accept()..... 36

69 it_ep_connect()..... 39

70 it_ep_disconnect()..... 44

71 it_ep_free()..... 47

72 it_ep_modify() 49

73 it_ep_query()..... 51

74 it_ep_rc_create() 53

75 it_ep_reset() 57

76 it_ep_ud_create() 58

77 it_evd_create() 61

78 it_evd_dequeue()..... 72

79 it_evd_free()..... 74

80 it_evd_modify() 76

81 it_evd_post_se() 80

82 it_evd_query()..... 82

83 it_evd_wait() 85

84	it_get_consumer_context().....	89
85	it_get_handle_type()	91
86	it_get_pathinfo().....	92
87	it_handoff()	95
88	it_hton64().....	97
89	it_ia_create()	98
90	it_ia_free().....	100
91	it_ia_info_free()	101
92	it_ia_query().....	102
93	it_interface_list().....	103
94	it_listen_create().....	106
95	it_listen_free()	108
96	it_listen_query()	109
97	it_lmr_create().....	111
98	it_lmr_free()	116
99	it_lmr_modify().....	117
100	it_lmr_query()	119
101	it_lmr_sync_rdma_read().....	121
102	it_lmr_sync_rdma_write().....	123
103	it_make_rdma_addr().....	125
104	it_post_rdma_read()	126
105	it_post_rdma_write().....	129
106	it_post_recv()	133
107	it_post_recvfrom()	136
108	it_post_send().....	139
109	it_post_sendto().....	142
110	it_pz_create()	145
111	it_pz_free().....	146
112	it_pz_query().....	147
113	it_reject()	149
114	it_rmr_bind().....	151
115	it_rmr_create()	155
116	it_rmr_free().....	156
117	it_rmr_query().....	157
118	it_rmr_unbind().....	159
119	it_set_consumer_context()	162
120	it_ud_service_reply()	163
121	it_ud_service_request()	166
122	it_ud_service_request_handle_create().....	169
123	it_ud_service_request_handle_free()	172
124	it_ud_service_request_handle_query()	173
125	5 Data Type Manual Pages	176
126	it_aevd_notification_event_t.....	177
127	it_affiliated_event_t	178
128	it_boolean_t.....	182
129	it_cm_msg_events.....	183
130	it_cm_req_events	193

131	it_conn_qual_t.....	196
132	it_context_t.....	198
133	it_dg_remote_ep_addr_t.....	199
134	it_dto_cookie_t.....	201
135	it_dto_events.....	202
136	it_dto_flags_t.....	205
137	it_dto_status_t.....	210
138	it_ep_attributes_t.....	217
139	it_ep_state_t.....	222
140	it_event_t.....	232
141	it_handle_t.....	236
142	it_ia_info_t.....	238
143	it_lmr_triplet_t.....	244
144	it_net_addr_t.....	245
145	it_path_t.....	247
146	it_software_event_t.....	250
147	it_status_t.....	251
148	it_unaffiliated_event_t.....	254
149	A. Implementer’s Guide.....	256
150	B. Header Files.....	267
151	B.1 it_api.h.....	267
152	B.2 it_api_os_specific.h.....	300
153		
154		
155	Figures	
156	Figure 1 : Three Way Passive RC Endpoint State Diagram.....	227
157	Figure 2 : Three Way Active RC Endpoint State Diagram.....	228
158	Figure 3 : Two Way Active RC Endpoint State Diagram.....	229
159	Figure 4 : Two Way Passive RC Endpoint State Diagram.....	230
160	Figure 5 : Unreliable Datagram Endpoint State Diagram.....	230
161		
162		
163	Tables	
164	Table 1: Thread Safety Models.....	16
165	Table 2: Thread-Safety Models Applied to IT-APIs.....	21
166	Table 3: Connection Management Event Definitions.....	186
167	Table 4: Event Management Event Fields.....	187
168	Table 5: reject_reason_code Descriptions.....	188
169	Table 6: UD Service Resolution Reply Event Definitions.....	188

170	Table 7: Service Resolution Reply Status	188
171	Table 8: InfiniBand reject_reason_code Mapping	189
172	Table 9: VIA reject_reason_code Mapping.....	190
173	Table 10: ep_info Element Mapping	190
174	Table 11: Communication Management Request Event Definitions	195
175		

176

Preface

177

The Interconnect Software Consortium

178

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.

179

180

181

The Open Group

182

The Open Group, a vendor and technology-neutral consortium, has a vision of Boundaryless Information Flow achieved through global interoperability in a secure, reliable, and timely manner. The Open Group mission is to drive the creation of Boundaryless Information Flow by:

183

184

185

Working with customers to capture, understand, and address current and emerging requirements, establish policies, and share best practices

186

187

Working with suppliers, consortia, and standards bodies to develop consensus and facilitate interoperability, to evolve and integrate open specifications and open source technologies

188

189

Offering a comprehensive set of services to enhance the operational efficiency of consortia

190

Developing and operating the industry's premier certification service and encouraging procurement of certified products

191

192

The Open Group provides opportunities to exchange information and shape the future of IT. The Open Group members include some of the largest and most influential organizations in the world. The flexible structure of The Open Group membership allows for almost any organization, no matter what their size, to join and have a voice in shaping the future of the IT world.

193

194

195

196

197

More information is available at www.opengroup.org.

198

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.

199

200

201

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

202

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/pubs.

203

204

205

206

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

207

208

209 A new *Version* indicates there is no change to the definitive information contained in the
210 previous publication of that title, but additions/extensions are included. As such, it
211 *replaces* the previous publication.

212 A new *Issue* indicates there is substantive change to the definitive information contained in the
213 previous publication of that title, and there may also be additions/extensions. As such,
214 both previous and new documents are maintained as current publications.

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

217 **This Document**

218 This document is the Consortium Specification for the Interconnect Transport API. It has been
219 developed and approved by The Interconnect Software Consortium in association with The Open
220 Group.

221 **Typographical Conventions**

222 The following typographical conventions are used throughout this document:

223 Bold font is used in text for filenames, type names, and data structures

224 Italic strings are used for emphasis. Italics in text also denote variable names and functions.

225 Normal font is used for the names of constants and literals.

226 Syntax and code examples are shown in fixed width font.

227 Bold Italic is used for all terms defined in the Definitions section when they first appear in
228 Chapter 1. IT-API objects are capitalized throughout the document (e.g. Interface Adapter,
229 Endpoint, etc).

230

Trademarks

231

The Open Group® is a registered trademark of The Open Group in the United States and other countries

232

233

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

234

POSIX® is a registered Trademark of The IEEE.

235

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.

236

237

238

239

Acknowledgements

240
241

The Interconnect Software Consortium gratefully acknowledges the contribution of the following people in the development of this document:

Caitlin Bestler	Matthew Pearson
Edward Chang	Todd Pisek
Joe Cowan	Sherman Pun
Ellen Delegates	Ashok Raj
David Ford	Kevin Reilly
Rama Govindaraju	Jim Roberts
Jim Hamrick	Jay Rosser
Al Hartmann	Sridharan Sakthivelu
Yaron Haviv	Heidi Scott
Carl Hensler	Steve Sistare
Jimmy Hill	Rajeev Sivaram
Peter Hochschild	Raja Srinivasan
Nobutaka Imamura	Tom Talpey
Arkady Kanevsky	Robert Teisberg
Ted Kim	Anthony Topper
John Kingman	Richard Treumann
Martin Kirk	Tom Tucker
Michael Krause	Andrew Twigger
Mike Moretti	Mark Wittle
Neil Moses	Fred Worley
Peter Ogilvie	Hanhong Xue

242

243

Referenced Documents

244

The following documents are referenced in this document:

245

Infiniband Architecture Release 1.1 specification

246

Infiniband Trade Association

247

248 1 Introduction

249 The IT-API defines interfaces for direct interaction with RDMA-capable transports. The Phase
 250 1 Specification covers VIA networks and the *Reliable Connection* and *Unreliable Datagram*
 251 services of InfiniBand networks. The IT-API Phase 1 Specification documentation set includes
 252 this introduction, a glossary, a global behaviors section, manual pages for 62 APIs and their
 253 supporting data type definitions, an implementation guide section, and two sample header files.
 254 The introduction and implementation guide and the sample header files are informative only; the
 255 remaining sections are the normative sections of the specification.

256 This overview describes the general architecture presented by the IT-API, reviews the significant
 257 data structures that implement the architecture, and introduces key terminology used throughout
 258 the API man pages. It is not a complete description of all supporting interfaces provided by the
 259 IT-API, nor does it include the level of descriptive detail provided by the man pages. It is an
 260 introduction to how to use the API. A separate Implementation Guide discusses issues related to
 261 implementing the API.

262 1.1 Interface Adapters

263 RDMA-capable transports are implemented in a number of ways, on various hardware
 264 platforms, and within different transport layering architectures. A vendor who provides the
 265 hardware and software components that make up an RDMA transport implementation, also
 266 called the *Implementation*, will see to it that the named instances of RDMA-capable transports
 267 available within a system can be listed using the IT-API interface [it interface list](#). The
 268 application program that uses the IT-API to access an RDMA-capable transport is called the
 269 *Consumer*. The Consumer may use the information returned by [it interface list](#) to identify an
 270 appropriate transport resource. The Consumer then uses the [it ia create](#) call to create and
 271 associate an IT *Interface Adapter* instance with the specified transport resource. The Interface
 272 Adapter, also called an IA, is used to access the underlying RDMA transport.

273 When the Consumer creates an IA using the [it ia create](#) call, an [it ia handle](#) is returned. The
 274 [it ia handle](#) is an opaque type reference *Handle* used by the Consumer to refer to a specific
 275 instance of an Implementation created *IT Object*. The [it ia handle](#) is used as a parameter to
 276 subsequent IT-API calls involving the IA. All IT-API interfaces that create an IT Object return
 277 an opaque type reference Handle that the Consumer can use in subsequent IT-API calls. It is the
 278 Consumer's responsibility to track these Handles, and use them appropriately.

279 The [it ia handle](#) is used both to query IA attributes and to create additional IT Objects used for
 280 communication on the Interface Adapter. The Consumer can call [it ia query](#) to retrieve
 281 attributes and transport-specific parameters associated with the IA; [it ia info free](#) to release the
 282 buffers allocated by [it ia query](#), and [it ia free](#) to release the [it ia handle](#) and all IT Objects
 283 associated with it. Most IT Objects follow the basic pattern of support for a standard set of
 284 create, query, modify, and free interfaces that are used to manage the object. Additional
 285 interfaces make use of each object's specific capabilities.

286 1.2 Memory Management

287 One of the key advantages of RDMA-capable transports is the ability for the transport
 288 Implementation to directly access Consumer defined message buffers. The IT-API provides
 289 interfaces to manage the Interface Adapter's use of the Consumer's memory.

290 The Consumer creates a **Local Memory Region**, also called an **LMR**, which defines a region of
 291 local memory to be used for message buffers. The Consumer defines the LMR and associates it
 292 with an Interface Adapter using the [it_lmr_create](#) call. The [it_lmr_create](#) call returns an
 293 [it_lmr_handle](#) that is used in subsequent IT-API calls to manage the IA's use of the LMR.
 294 Remote access privileges for the LMR can be set when the LMR is created. Attributes of the
 295 LMR can be queried and modified using the [it_lmr_query](#) and [it_lmr_modify](#) calls, respectively.
 296 The LMR is released using the [it_lmr_free](#) call.

297 A **Protection Zone**, also called a **PZ**, is used to control access to memory when messages are
 298 transferred. Many IT-API objects are associated with a PZ when they are created. IT-API
 299 objects involved in a Data Transfer Operation are required to have the same Protection Zone for
 300 the operation to succeed. A Protection Zone is created using the [it_pz_create](#) call, which returns
 301 an [it_pz_handle](#). Attributes of the PZ can be queried using the [it_pz_query](#) call. A PZ is released
 302 using the [it_pz_free](#) call.

303 The Consumer may create a Remote Memory Region, also called an RMR, using the
 304 [it_rmr_create](#) call, which returns an [it_rmr_handle](#). An RMR can be used in subsequent RDMA
 305 **Data Transfer Operations** to describe a local data **Destination** buffer. RMR attributes can be
 306 queried using the [it_rmr_query](#) call, and can be released using the [it_rmr_free](#) call.

307 1.3 Communication Endpoints

308 In order to communicate using an Interface Adapter, the Consumer must create a communication
 309 **Endpoint**, also called an **EP**. An Endpoint is used to issue requests on the IA. The Endpoint
 310 also provides a target for establishing connected communications, and can be associated with an
 311 address for use with datagram communications.

312 The Consumer creates an Endpoint by calling either [it_ep_rc_create](#), for use with Reliable
 313 Connection communications, or [it_ep_ud_create](#), for use with Unreliable Datagram
 314 communications. The EP can be queried and modified using the [it_ep_query](#) and [it_ep_modify](#)
 315 calls, respectively. It can be released using the [it_ep_free](#) call.

316 For Reliable Connection communications, a Consumer may issue a request to connect a local
 317 Endpoint to a remote Endpoint using the [it_ep_connect](#) call. In order to receive a **Connection**
 318 **Request**, a Consumer creates an IT Listen Handle object that is used to await Connection
 319 Requests. The Listen Handle is created using the [it_listen_create](#) call, which returns an
 320 [it_listen_handle](#). Attributes of the Listen Handle can be queried using the [it_listen_query](#) and
 321 the Listen Handle can be released using the [it_listen_free](#) call.

322 When a Connection Request is received it can be accepted or rejected using the [it_ep_accept](#) and
 323 [it_ep_reject](#) calls, respectively. An existing Connection can be terminated using the
 324 [it_ep_disconnect](#) call. During the lifetime of a connected communication session, an EP

325 proceeds through successive stages of Connection establishment via state transitions. These
326 states and transitions are described in [it_ep_state](#).

327 For Unreliable Datagram communications, an IT **Address Handle** object can be created for use
328 in defining and targeting specific remote Endpoints. An Address Handle is created using the
329 [it_address_handle_create](#) call, which returns an [it_address_handle](#). Attributes of the Address
330 Handle can be queried and modified using the [it_address_handle_query](#) and
331 [it_address_handle_modify](#) calls. The Address Handle can be released using the
332 [it_address_handle_free](#) call.

333 For Unreliable Datagram communications, the Consumer can create an IT Service Request
334 Handle that is used to store Destination address information. A Service Request Handle is
335 created using the [it_ud_service_request_handle_create](#) call, which returns an
336 [it_ud_svc_req_handle](#). Attributes of the Service Request Handle can be queried using the
337 [it_ud_service_request_handle_query](#) and the Service Request Handle can be released using the
338 [it_ud_service_request_handle_free](#) call. A Service Request Handle is used in the
339 [it_ud_service_request](#) to provide addressing information for use in sending the reply message
340 sent by the [it_ud_service_reply](#) call.

341 1.4 Data Transfer Operations

342 The Consumer can queue different kinds of Data Transfer Operations, also called **DTOs**, to an
343 Endpoint. DTOs include sending and receiving messages, issuing RDMA requests, and
344 associating and disassociating Remote Memory Regions with Local Memory Regions.

345 The Consumer can associate an RMR with a sub-region of memory within an LMR using the
346 [it_rmr_bind](#) call. The [it_rmr_bind](#) call returns an [it_rmr_context](#) identifier, which can be used in
347 subsequent RDMA transfer requests to define the message buffer to be used for the RDMA
348 operation. The [it_rmr_unbind](#) call removes the binding currently associated with a RMR.
349 These calls provide request-specific access control for IA memory accesses, in addition to the
350 region-based access control offered by LMRs.

351 The Consumer may issue requests to send messages using either the [it_post_send](#) or
352 [it_post_sendto](#) interfaces, depending on whether the Endpoint used for communication is of the
353 Connected or Datagram type, respectively. The Consumer issues requests to receive messages
354 using either the [it_post_recv](#) or [it_post_recvfrom](#) calls. RDMA operations are initiated using the
355 [it_post_rdma_read](#) and [it_post_rdma_write](#) calls. Completions of the posted operations are
356 reported to Consumers asynchronously via **Events**.

357 1.5 Events

358 IT-API calls normally return program control immediately to the issuing Consumer. The call
359 return value indicates either success or immediate failure through an error indication. For some
360 calls, a successful return value means that the request has been executed successfully, while for
361 other calls it indicates only that the request has been accepted by the Implementation for later
362 execution. The Consumer is notified of the completion of a request via an asynchronous event
363 mechanism. Each type of Event generated by the IT-API has an associated IT Event object that
364 contains information about the Event. Common IT Event types include DTO completion,

365 Connection establishment-related Events, and transport error conditions. Event objects are
 366 created by the Implementation and made available to the Consumer when an Event occurs. The
 367 Implementation enqueues these Event objects on an *Event Dispatcher*, also called an *EVD*.

368 The Consumer creates an Event Dispatcher by calling the *it_evd_create* call, which returns an
 369 *it_evd_handle*. Attributes of an EVD can be queried and modified using the *it_evd_query* and
 370 *it_evd_modify* calls, respectively. An EVD is released using the *it_evd_free* call. The Consumer
 371 may access a queued Event object using the *it_evd_dequeue* call, or by using the *it_evd_wait* call
 372 which provides a blocking interface for awaiting Events.

373 The IT-API supports two types of EVDs. A *Simple EVD*, also called an *SEVD*, enqueues only
 374 Events of a single type. This simplifies the implementation of an SEVD and enhances its
 375 performance characteristics. For many communication scenarios, this provides a Consumer with
 376 the best performance option. An *Aggregate EVD*, also called an *AEVD*, can be used to collect
 377 Events from a set of SEVDs. This allows the Consumer to create a single *Notification*
 378 mechanism that will enqueue many different types of Events. In addition to these IT-API
 379 Notification mechanisms, *it_evd_create* allows an Implementation-defined *file descriptor* to be
 380 associated with each EVD. This allows the Consumer to use a File Descriptor-based
 381 Notification mechanism provided by the Implementation (e.g. POSIX *poll*) to collect both non-
 382 IT-API Events and IT-API Events from multiple IAs.

383 Most Events are associated with Endpoints. These Events correspond to DTO completions or
 384 other communication Events related to the Endpoint. When an Endpoint is created, the
 385 Consumer associates EVDs with it. These EVDs collect a specific type of Event. One EVD
 386 enqueues Events associated with the completion of Consumer initiated DTO requests. This
 387 includes RMR association and disassociation, messages sent, and RDMA requests completed. A
 388 second EVD enqueues Events generated as a result of messages being received at the Endpoint.
 389 For EPs using connected communications, a third EVD enqueues Events related to Connection
 390 management. EVDs may be shared across multiple Endpoints.

391 Some Events are not associated with a particular Endpoint, but are associated with the specific
 392 Interface Adapter. Consumers can receive Notification of these Events by creating a separate
 393 EVD and specifying that it should enqueue this type of Event.

394 1.6 Event Notification

395 The IT-API provides the Consumer with control over a number of aspects of Event Notification.
 396 IT-API interfaces used to initiate Data Transfer Operations include flags to support Event and
 397 Event Notification Suppression, and to request remote-side Endpoint Notification of message
 398 delivery. The details of these features are described in *it_dto_flags*. The *it_evd_wait* call
 399 provides a blocking interface for waiting for the next Event to occur, along with a timeout value.
 400 EVDs provide a thresholding attribute used to batch the delivery of Event Notification.

401 **2 Definitions**

402 **Address Handle**

403 An object that contains the information necessary to transmit messages to a remote port over
404 Unreliable Datagram service. (It should be noted that an Address Handle is an IT Object, not a
405 Handle as defined later on in this section.)

406 **AEVD**

407 See Aggregation Event Dispatcher.

408 **Affiliated Asynchronous Event**

409 An Event associated with a specific Endpoint or EVD.

410 **Affiliated Event**

411 See Affiliated Asynchronous Event.

412 **Aggregation Event Dispatcher (AEVD)**

413 An IT Object that conceptually merges Event completion Notifications from one or more Simple
414 Event Dispatchers. This provides the Consumer with a single point to receive Notification of
415 Event completions across multiple Event Streams.

416 **Bind**

417 See RMR Bind.

418 **Communication Management Message Events**

419 The set of Event types related to the sequence of messages involved in RC Connection
420 establishment, normal disconnect, Connection error conditions, and Unreliable Datagram
421 Service Resolution Replies.

422 **Communication Management Request Events**

423 The set of Event types that result from messages received requesting RC Connection
424 establishment or Unreliable Datagram service. Normally these Events trigger state changes at
425 the receiving Endpoint.

426 **Completion Event**

427 The set of Event types that indicate that a previously posted operation has completed.

428 **Completion Suppression**

- 429 An optional DTO behavior specifying that no Event is to be generated upon successful
430 completion of the operation.
- 431 **Connection**
- 432 An association between a pair of Endpoints such that data posted via Data Transfer Operations
433 of either Endpoint arrives at the other Endpoint of the Connection.
- 434 **Connection Qualifier**
- 435 A value that allows an incoming Connection Request or Unreliable Datagram Service Resolution
436 Request to be associated with an entity that can provide that service.
- 437 **Connection Request**
- 438 A message that requests RC Connection establishment.
- 439 **Consumer**
- 440 An application that utilizes the IT-API.
- 441 **Context**
- 442 A Consumer-supplied value that can be associated with an instance of an IT Object.
- 443 **Data Transfer Operation (DTO)**
- 444 A request submitted by the Consumer to the Implementation to move data between two
445 Endpoints.
- 446 **Destination**
- 447 The Endpoint where a message is received.
- 448 **DTO**
- 449 See Data Transfer Operation.
- 450 **DTO Cookie**
- 451 A Consumer-supplied identifier for a Data Transfer Operation, Bind, or Unbind operation that
452 allows the Consumer to uniquely identify the operation when it completes.
- 453 **Endpoint (EP)**
- 454 The object to which DTOs and RMR operations are posted. An Endpoint is associated with a
455 single Spigot.
- 456 **Endpoint ID**
- 457 An identifier for an Endpoint on a given Interface Adapter. This is used to help identify the
458 particular Endpoint where a datagram is to be delivered.

459 Endpoint Key

460 A construct that some transports require to be associated with an outgoing datagram to allow the
461 Receiver to validate that the sender of the datagram has permission to access the Receiver's
462 Endpoint.

463 Endpoint Protection Zone

464 The Protection Zone associated with an Endpoint.

465 EP

466 See Endpoint.

467 EVD

468 See Event Dispatcher.

469 Event

470 A structure or record that is delivered to the Consumer through an Event Dispatcher to provide
471 notice of some kind. Types of Events include DTO completions, Connection state changes,
472 asynchronous errors, and information passed through the [it_evd_post_se](#) interface that is
473 generated by the Consumer.

474 Event Dispatcher (EVD)

475 An IT Object that conceptually merges Event completion Notifications for the Consumer. The
476 IT-API defines two types of EVDs: a Simple Event Dispatcher, and an Aggregation Event
477 Dispatcher.

478 Event Stream

479 A source of Events for the Simple Event Dispatcher: DTO completions, Connection Requests,
480 Connection reject Notifications, Connection establishment completion Notifications, disconnect
481 Notifications, Connection errors, Connection Request timeouts, asynchronous errors, RMR Bind
482 and Unbind completion Notifications, and Consumer-generated Notifications. An Event Stream
483 is the conduit between IT-API objects that generate Events and Simple Event Dispatchers that
484 consume Events.

485 Handle

486 An opaque data type used to reference an object.

487 IA

488 See Interface Adapter.

489 IANA

490 See Internet Address Naming Authority.

491 IANA Port Number

- 492 A specific port address as defined by IANA.
- 493 **IB**
- 494 See InfiniBand.
- 495 **ICSC**
- 496 See Interconnect Software Consortium.
- 497 **IETF**
- 498 See Internet Engineering Task Force.
- 499 **Implementation**
- 500 The collection of software and hardware that combine to provide the service exported by the IT-
501 API.
- 502 **Implementer's Guide**
- 503 A non-normative section of the IT-API documentation set that contains information provided to
504 assist implementers of the IT-API.
- 505 **InfiniBand (IB)**
- 506 One of the transports that the IT-API supports. The host interface portion of InfiniBand is
507 defined in "InfiniBand Architecture Specification Volume 1", available at
508 <http://www.infinibandta.org>.
- 509 **InfiniBand Global Routing Header**
- 510 A routing header that may be present in the first 40 bytes of a completed Unreliable Datagram
511 Receive operation. See the InfiniBand specification for a description of the format of this
512 routing header.
- 513 **InfiniBand Native Transport**
- 514 Transport services defined by InfiniBand Architecture.
- 515 **Interconnect Software Consortium (ICSC)**
- 516 Standards organization that includes the ITWG. The [Interconnect Software Consortium](#) is
517 affiliated with [The Open Group](#).
- 518 **Interconnect Transport Working Group (ITWG)**
- 519 The [ICSC working group](#) that created the IT-API.
- 520 **Internet Engineering Task Force (IETF)**
- 521 [Internet Engineering Task Force](#).

522	Internet Address Naming Authority (IANA)
523	IETF Network Address naming authority.
524	Interface
525	A host resident device that transfers data to and from the host memory to which it is attached.
526	Interface Adapter (IA)
527	An instance of an Interface that is created by the it_ia_create call. An Interface Adapter may
528	contain one or more Spigots.
529	IP
530	The IETF Internet Protocol.
531	IPv4
532	The IETF Internet Protocol version 4.
533	IPv6
534	The IETF Internet Protocol version 6.
535	IT-API
536	The data structures and routines that make up the Interconnect Transport Application
537	Programming Interface.
538	IT Handle
539	An opaque reference to an IT Object. An IT Handle is returned to the Consumer whenever an IT
540	Object is created for the Consumer's use. The IT Handle can be used to reference the IT Object
541	in subsequent calls into the IT-API.
542	IT Object
543	A software object created by the IT-API Implementation as a result of a Consumer call into the
544	IT-API, used to satisfy subsequent Consumer requests. When the IT Object is created, an
545	opaque reference to the object, called a Handle, is returned to the Consumer for use in
546	subsequent calls into the IT-API.
547	ITWG
548	See Interconnect Transport Working Group
549	LMR
550	See Local Memory Region.
551	LMR Triplet

552	A type used to specify a section of a Local Memory Region. Each LMR Triplet specifies the
553	LMR Handle, the LMR virtual address, and a length.
554	Local Memory Region (LMR)
555	A virtually contiguous area of arbitrary size within a Consumer's address space that has been
556	registered using the it_lmr_create routine, enabling local access and optional remote access.
557	Network Address
558	An identifier that can be used to reach a particular Spigot attached to a network.
559	Notification
560	An asynchronous mechanism for providing the Consumer with information about the completion
561	of a previously posted operation.
562	Notification Event
563	An Event in an Event Stream whose arrival triggers the Notification of the Event to a waiting
564	Consumer via either a wakeup from it_evd_wait , or via a higher-level Notification mechanism.
565	Notification Suppression
566	A Consumer-specified option for Data Transfer Operations that informs the Implementation that
567	no Notification Event should be created if the DTO completes successfully. Notification
568	Suppression has no effect on operations that complete in error – in this case the completion will
569	generate an error Event.
570	Organization Unique Identifier (OUI)
571	An OUI is a 24-bit globally unique number assigned by the Institute of Electrical and Electronics
572	Engineers (IEEE). The IT-API uses an OUI to map IETF IANA Port Numbers into the IB
573	Service ID space for use within the IT-API.
574	OUI
575	See Organization Unique Identifier.
576	Outstanding Operation
577	An operation is "Outstanding" until the Event for the operation completes, or for an operation
578	whose completion has been suppressed, until an operation posted subsequent to it completes.
579	Path
580	The collection of links, switches, and routers a message traverses from a Source Spigot to a
581	Destination Spigot. This is represented in the IT-API by the it_path_t structure.
582	Port Number
583	See IANA Port Number.

584	Private Data
585	Consumer data that is opaque to the Implementation and is passed between the local and remote
586	Consumers by the Implementation's Connection establishment and UD service resolution
587	routines.
588	Protection Zone (PZ)
589	A mechanism for associating Endpoints and registered LMR and RMR memory of an Interface
590	Adapter that defines protection for local and remote memory accesses by DTO operations.
591	PZ
592	See Protection Zone.
593	RC
594	See Reliable Connected.
595	RDMA
596	See Remote Direct Memory Access.
597	RDMA Read
598	The Data Transfer Operation (DTO) that is initiated by the <i>it_post_rdma_read</i> routine.
599	RDMA Write
600	The Data Transfer Operation (DTO) that is initiated by the <i>it_post_rdma_write</i> routine.
601	Receive
602	The Data Transfer Operation (DTO) that is initiated by the <i>it_post_recv</i> or <i>it_post_recvfrom</i>
603	routine.
604	Receive Queue
605	An internal queue associated with an Endpoint on which Receive DTOs are posted.
606	Reliable Connected (RC)
607	A Transport Service Type in which an Endpoint is associated with only one other Endpoint, such
608	that messages transmitted from one Endpoint are reliably delivered to the other Endpoint,
609	uncorrupted in the absence of errors and in the order defined by the Reliable Connection
610	ordering rules. As such, each Endpoint is said to be "connected" to the opposite Endpoint.
611	Reliable Connection
612	A Connection type such that data of posted DTOs of either Endpoint of the Connection reliably
613	arrives at the other Endpoint of the Connection uncorrupted in the absence of errors and in the
614	order defined by the Reliable Connection ordering rules.

- 615 **Remote Direct Memory Access (RDMA)**
- 616 A method of accessing memory on a remote system without interrupting the processing of the
617 CPU(s) on that system.
- 618 **Remote Memory Region (RMR)**
- 619 A window that can be bound to a section of a Local Memory Region to enable remote accesses.
- 620 **Request Queue**
- 621 An internal queue of an Endpoint on which DTOs and RMR Binds and Unbinds are posted. The
622 Request Queue to which RMR Bind, RMR Unbind, Send, RDMA Read, and RDMA Write
623 operations are posted is commonly called the Send Queue. The Request Queue to which
624 Receive operations are posted is commonly called the Receive Queue.
- 625 **RMR**
- 626 See Remote Memory Region.
- 627 **RMR Bind**
- 628 An operation that associates an RMR with a section of an LMR and thereby enables remote
629 access to that section.
- 630 **RMR Context**
- 631 An opaque identifier generated by the Implementation to represent a contiguous memory region.
632 Used by remote Consumers in RDMA operations that target this region.
- 633 **RMR Unbind**
- 634 An operation that destroys the association of an RMR with a section of an LMR.
- 635 **SE**
- 636 See Software Event.
- 637 **Send**
- 638 The Data Transfer Operation (DTO) that is initiated by the [*it_post_send*](#) or [*it_post_sendto*](#)
639 routine.
- 640 **Send Queue**
- 641 An internal queue of an Endpoint on which Receive DTOs are posted.
- 642 **Service Reply**
- 643 See UD Service Reply.
- 644 **Service Request**

645 See UD Service Request.

646 **Service Type**

647 A class of transport service defining basic attributes of the communication, e.g., connected or
648 unconnected, reliable or unreliable.

649 **SEVD**

650 See Simple Event Dispatcher.

651 **Simple Event Dispatcher (SEVD)**

652 An IT Object that conceptually merges Events from one or more Event Streams. These Events
653 can be dequeued by the Consumer directly. The Consumer is notified that Events are available
654 through the [it_evd_wait](#) interface, or through higher-level Notification mechanisms, such as the
655 Aggregation Event Dispatcher. The Simple Event Dispatcher is responsible for completion of
656 transport-specific fetching and handshaking for the Events it collects. Each Event is delivered to
657 the Consumer exactly once.

658 **Software Event (SE)**

659 An Event generated for a Simple Event Dispatcher by the Consumer, as opposed to those
660 generated by the Interface Adapter.

661 **Solicited Wait**

662 A modifier for Send DTOs submitted to an Endpoint of the Connection. It specifies that the
663 completion of matching Receive DTOs on the remote side of the Connection generate
664 Notification Receive DTO Completion Events.

665 **Source**

666 The Endpoint where a message originates.

667 **Spigot**

668 A host resident device that transfers data to and from the host memory to which it is attached. A
669 Spigot is associated with a single Interface. One or more Spigots may be associated with the
670 same Interface.

671 **The Open Group**

672 [The Open Group](#) standards organization.

673 **Transport Service Type**

674 See Service Type.

675 **UD**

676 See Unreliable Datagram.

677	Unaffiliated Asynchronous Event
678	An Event that is not associated with a specific Endpoint or EVD, but is only associated with an
679	Interface Adapter.
680	Unbind
681	See RMR Unbind.
682	Unreliable Datagram (UD)
683	A Transport Service Type in which an Endpoint may transmit and Receive single-packet
684	messages to/from any other Endpoint that supports that Service Type. Ordering and delivery are
685	not guaranteed, and the Receiver may drop delivered packets.
686	UD Service Reply
687	A reply message sent via the Unreliable Datagram service in response to a UD Service Request.
688	UD Service Request
689	A request message sent via the Unreliable Datagram service requesting service resolution.
690	VIA
691	See Virtual Interface Architecture.
692	Virtual Interface Architecture (VIA)
693	One of the transports that the IT-API supports. VIA is defined by "The
694	Virtual Interface Architecture Specification", which is available at
695	http://developer.intel.com/design/servers/vi/the_spec/specification.htm .

696 **3 Global Behavior**

697 The IT-API Global Behavior section describes certain general aspects of the behavior of the IT-
698 API. Behavior described in this section is applicable to all IT-API interfaces except where noted
699 explicitly in individual manual pages.

700 **3.1 Non-Blocking APIs**

701 Nearly all IT-API routine invocations return program control to the issuing Consumer without
702 blocking the caller's execution indefinitely. The call may return success or failure, with call
703 success indicating that the request was completed successfully, or that the request has been
704 accepted for later execution. The Consumer is notified of the subsequent completion of a
705 request via an asynchronous Event mechanism. Each Event generated by the IT-API has an
706 associated IT Event object that contains information about the disposition and status of the
707 Event.

708 A few IT-API interfaces may block the caller's execution pending some Event or condition.
709 These exceptions are noted explicitly in the appropriate manual pages.

710 **3.2 Thread Safety**

711 The IT-API supports multi-threaded applications through a variety of different thread safety
712 models. The basic issue in thread-safety is to provide mutually exclusive access to a shared
713 resource being accessed by multiple threads executing in parallel. Within a multi-threaded
714 application, it is common to share data resources. A common solution to provide mutual
715 exclusion is to serialize potentially conflicting accesses into a well-ordered succession of
716 executions. For example, if two callers make a call at the same time the results of the two
717 executions are as if the two calls were serialized in arbitrary order. Normally ensuring mutual
718 exclusion introduces some performance reduction, and so is only desirable when needed.

719 To support multi-threaded applications, the IT-API defines three models of thread safety.
720 Briefly, these three models are described as:

- 721 - Strongly Thread-Safe.
- 722 - Efficiently Thread-Safe.
- 723 - Not Thread-Safe.

724 While none of the three models is completely thread-safe, each provides a different degree of
725 thread-safety. Each of the models is appropriate for a different Consumer programming model.
726 The thread safety models are described in more detail below.

727
728
729
730
731
732

Implementations of the IT-API may support one or more thread safety models. Which model or models a particular Implementation supports, and how that thread safety support is communicated to the Consumer, is beyond the scope of the IT-API. One potential mechanism would have the Implementation vendor associate a specific thread safety model with a particular library Implementation of the IT-API. In this scheme, different libraries would support different thread safety models. The IT-API defines the thread safety models described in Table 1.

Model	Description
Strongly Thread Safe	<p>Nearly all routines are thread-safe. This model assumes that the Consumer wants the Implementation to provide considerable thread-safety. This thread-safety model may introduce some performance cost.</p> <p>All IT-API routines are thread-safe except for object destruction routines. The Consumer is required to ensure that an IT Object is not in use when a call is made to free or destroy that Object.</p>
Efficiently Thread Safe	<p>Some routines are thread-safe, and some are not thread-safe. This model assumes that the Consumer primarily wants the Implementation to provide high performance, and the Consumer is willing to take some responsibility for providing thread-safety.</p> <p>The Implementation provides thread-safety for routines that are not critical to the performance of the I/O code paths, and for routines where thread-safety can not be managed by the Consumer. This includes thread-safety for routines that harvest and post Events.</p> <p>Routines that are critical to the performance of the I/O code paths, and object management routines for objects that are central to the function of the I/O code paths are not thread-safe. The Consumer is required to ensure that I/O operations are suspended when IT Object management routines are invoked on objects associated with the I/O code path. In addition, object destruction routines are not thread-safe. The Consumer is required to ensure that an IT Object is not in use when a call is made to free or destroy that Object.</p>
Not Thread-Safe	<p>No routines are thread-safe. This model assumes that the Consumer is single-threaded, or that the Consumer is managing mutual exclusion access control to IT Objects.</p>

733

Table 1: Thread Safety Models

734
735
736

For the Strongly Thread-Safe and Efficiently Thread-Safe models, the IT-API defines thread safety on a routine-by-routine basis, and applies thread safety to IT Objects according to specific rules.

737
738
739
740

Thread-safety means that the routine 1) provides well-defined results without imposing any restrictions on other IT-API routines called by other threads in the system, and 2) provides well-defined results without regard to which other IT-API routines currently have threads of execution within them.

741
742

Not thread-safe means that the results of the routine can possibly be NOT well-defined if another in-progress not thread-safe routine is called with the same primary (i.e., first) call argument, and

743 none of the calls is an object destructor routine. (For [rnr_bind](#) the *lmr_handle*, *rnr_handle*, and
 744 *ep_handle* arguments must be treated as primary call arguments for thread-safety purposes. For
 745 [rnr_unbind](#) the *rnr_handle* and *ep_handle* arguments, and the *lmr_handle* that is bound to the
 746 *rnr* must be treated as primary call arguments for thread-safety purposes.)

747 This definition of thread-safe and not thread-safe routines allow simultaneous execution of not
 748 thread-safe calls involving different instances of an IT Object (e.g., two different EPs) or
 749 involving IT Objects that are related (e.g., an EVD and an EP), so long as the primary call
 750 argument is not the same for the two routines.

751 For the Not Thread-Safe model, the Implementation does not provide thread-safety for any data
 752 structures whatsoever.

753 The IT-API applies these thread safety models on a routine-by-routine basis. IT-API routines
 754 can be classified into five groups according to their basic function. These groups determine their
 755 thread-safety under each thread safety model:

- 756 - Non-performance critical routines. These routines create objects and manage and query
 757 the state of objects that do interact with the I/O code paths.
- 758 - Event harvesting and posting routines. These routines retrieve Events from the
 759 Implementation, and invoke software Events.
- 760 - Performance critical routines. These routines invoke Data Transfer Operations and RMR
 761 Operations.
- 762 - Object management routines. These routines modify and query the state of objects that
 763 interact with the I/O code paths.
- 764 - Object destructor routines. These routines free and/or destroy IT Objects.

765 The thread-safety of each IT-API routine is given in Table 2.

766

Non-Performance Critical Routines	Strongly Thread-Safe Model	Efficiently Thread-Safe Model	Not Thread-Safe Model
<u>it_address_handle_create</u> <u>it_convert_net_addr</u> <u>it_ep_rc_create</u> <u>it_ep_ud_create</u> <u>it_evd_create</u> <u>it_get_handle_type</u> <u>it_get_pathinfo</u> <u>it_hton64, it_ntoh64</u> <u>it_ia_create</u> <u>it_ia_query</u> <u>it_interface_list</u> <u>it_listen_create</u> <u>it_listen_query</u> <u>it_lmr_create</u> <u>it_make_rdma_addr</u> <u>it_pz_create</u> <u>it_pz_query</u> <u>it_rmr_create</u> <u>it_ud_service_request_handle_create</u>	Thread-safe		Not thread-safe
Event Harvesting and Posting Routines	Strongly Thread-Safe Model	Efficiently Thread-Safe Model	Not Thread-Safe Model
<u>it_evd_dequeue</u> <u>it_evd_post_se</u> <u>it_evd_wait</u>	Thread-safe		Not thread-safe
Performance Critical Routines	Strongly Thread-Safe Model	Efficiently Thread-Safe Model	Not Thread-Safe Model
<u>it_post_rdma_read</u> <u>it_post_rdma_write</u> <u>it_post_recv</u> <u>it_post_recvfrom</u>	Thread-safe		Not thread-safe

<u>it_post_send</u> <u>it_post_sendto</u> <u>it_rmr_bind</u> <u>it_rmr_unbind</u> <u>it_ud_service_reply</u> <u>it_ud_service_request</u>		
--	--	--

767

Object Management Routines	Strongly Thread-Safe Model	Efficiently Thread-Safe Model	Not Thread-Safe Model
<u>it_address_handle_modify</u> <u>it_address_handle_query</u> <u>it_ep_connect</u> <u>it_ep_modify</u> <u>it_ep_query</u> <u>it_ep_reset</u> <u>it_evd_modify</u> <u>it_evd_query</u> <u>it_get_consumer_context</u> <u>it_lmr_modify</u> <u>it_lmr_query</u> <u>it_lmr_sync_rdma_read</u> <u>it_lmr_sync_rdma_write</u> <u>it_rmr_query</u> <u>it_set_consumer_context</u> <u>it_ud_service_request_handle_query</u>	Thread-safe	Not thread-safe	
Object Destructor Routines	Strongly Thread-Safe Model	Efficiently Thread-Safe Model	Not Thread-Safe Model
<u>it_address_handle_free</u> <u>it_ep_accept</u> <u>it_ep_disconnect</u> <u>it_ep_free</u> <u>it_evd_free</u> <u>it_handoff</u> <u>it_ia_free</u> <u>it_ia_info_free</u> <u>it_listen_free</u> <u>it_lmr_free</u> <u>it_reject</u>	Not thread-safe		

<u>it_ud_service_request_handle_free</u> <u>it_pz_free</u> <u>it_rmr_free</u>	
---	--

769 **Table 2: Thread-Safety Models Applied to IT-APIs**

770 **3.3 Signal Handlers**

771 IT-API interfaces are not required to be safely executable from within a signal handler
772 invocation.

773 **3.4 Fork Semantics**

774 Use of the POSIX *fork* family of calls is supported within the IT-API with the following
775 semantics. After a fork call, the parent process' references to IT Objects are unchanged, and it
776 may continue to use the references as it had before the fork call.

777 The child process' IT Object references are invalid following the fork call (with one exception
778 for file descriptors discussed below).

779 The one exception to this behavior for the child is for file descriptors that were associated with
780 EVDs before the fork call occurred. The Implementation supports the child's use of the *close*
781 call to close the file descriptor.

782 **3.5 Exec Semantics**

783 The process' IT Object references are invalid following the POSIX *exec* family of calls.

784 **3.6 Exit Semantics**

785 Following an implicit or explicit call to the POSIX *exit*, all IT Objects associated with the
786 process are destroyed and all references to them are invalid.

787 **3.7 Error Handling**

788 Error Notification is provided to Consumers of the IT-API in two ways: as error return values to
789 interface calls, and as asynchronous Events containing error status information for the earlier
790 request. In general, interface calls return an error when a call argument is invalid or
791 incompatible with a condition relevant to the request. However, some errors of this type are
792 determined by the transport layer stack executing below the IT-API, and in this case the
793 Consumer is notified of call parameter-related errors through an asynchronous Event.

794 **3.8 IT Handle Management**

795 IT-API interfaces that create an IT Object return an opaque type reference Handle that the
796 Consumer can use in subsequent IT-API calls. It is the Consumer's responsibility to track these
797 Handles, and use them appropriately. The Implementation will make its best effort to detect
798 improper use of Handles by the Consumer and will return an invalid Handle error whenever
799 possible. However, it may not always be possible for the Implementation to detect improper use
800 of a Handle, and improper use may result in data corruption or fatal errors for the Consumer.

801 **4 API Manual Pages**

- 802 [it_address_handle_create](#) – create an Address Handle
- 803 [it_address_handle_free](#) – free an Address Handle
- 804 [it_address_handle_modify](#) – modify an Address Handle
- 805 [it_address_handle_query](#) – query an Address Handle
- 806 [it_convert_net_addr](#) – convert a Network Address from one format to another.
- 807 [it_ep_accept](#) – accept an incoming Connection establishment request or reply.
- 808 [it_ep_connect](#) – Initiate an Endpoint Connection establishment request.
- 809 [it_ep_disconnect](#) - disconnects an existing Endpoint-to-Endpoint Connection.
- 810 [it_ep_free](#) – destroys an RC or UD Endpoint
- 811 [it_ep_modify](#) – modify parameters of an existing Endpoint
- 812 [it_ep_query](#) – query an existing Endpoint
- 813 [it_ep_rc_create](#) – Create an Endpoint for Reliable Connection.
- 814 [it_ep_reset](#) – resets a Reliable Connected Endpoint into the initial state
- 815 [it_ep_ud_create](#) – Create an Endpoint for Unreliable Datagram.
- 816 [it_evd_create](#) – create Simple or Aggregate Event Dispatcher
- 817 [it_evd_dequeue](#) – Dequeue Events from Event Dispatcher
- 818 [it_evd_free](#) – destroys an Event Dispatcher
- 819 [it_evd_modify](#) – modify an existing Event Dispatcher
- 820 [it_evd_post_se](#) – Post Software Event on Simple Event Dispatcher
- 821 [it_evd_query](#) – query an existing Simple or Aggregate Event Dispatcher
- 822 [it_evd_wait](#) – Wait for Events on Event Dispatcher
- 823 [it_get_consumer_context](#) – get Consumer Context associated with an IT Object Handle
- 824 [it_get_handle_type](#) – return the Handle type value associated with an IT Object Handle
- 825 [it_get_pathinfo](#) – retrieve Paths used to communicate with a remote Network Address

826 [it_handoff](#) - hands-off an incoming Connection Request to another Connection Qualifier.

827 [it_hton64](#), [it_ntoh64](#) – convert 64-bit integers between host and network byte order

828 [it_ia_create](#) – create an Interface Adapter

829 [it_ia_free](#) – destroy an Interface Adapter Handle

830 [it_ia_info_free](#) – free an [it_ia_info_t](#) structure that was returned by [it_ia_query](#)

831 [it_ia_query](#) – retrieve attributes of given Interface Adapter and its Spigots

832 [it_interface_list](#) – retrieve information about the available Interface Adapters

833 [it_listen_create](#) – listen for an incoming Connection Request for a Connection Qualifier.

834 [it_listen_free](#) – destroys a listening point for a Connection Qualifier.

835 [it_listen_query](#) – query parameters associated with a listening point.

836 [it_lmr_create](#) – create a Local Memory Region and register with an Interface Adapter.

837 [it_lmr_free](#) – destroy a Local Memory Region

838 [it_lmr_modify](#) – modify selected attributes of a Local Memory Region

839 [it_lmr_query](#) – get attributes of a Local Memory Region

840 [it_lmr_sync_rdma_read](#) – make memory changes visible to an incoming RDMA Read op

841 [it_lmr_sync_rdma_write](#) – make effects of an incoming RDMA Write operation visible

842 [it_make_rdma_addr](#) – make a platform independent RDMA address

843 [it_post_rdma_read](#) – post an RDMA Read DTO to a Reliable Connected Endpoint

844 [it_post_rdma_write](#) – post an RDMA Write DTO to a connected Endpoint

845 [it_post_recv](#) – post a Receive DTO to a connected Endpoint

846 [it_post_recvfrom](#) – post a Receive DTO to a datagram Endpoint

847 [it_post_send](#) – post a Send DTO to a connected Endpoint

848 [it_post_sendto](#) – post a Send DTO to a datagram Endpoint

849 [it_pz_create](#) – create a new Protection Zone

850 [it_pz_free](#) – destroy a Protection Zone.

851 [it_pz_query](#) – get attributes of a Protection Zone

852 [it_reject](#) - reject an incoming Connection establishment request or reply.

- 853 [*it_rmr_bind*](#) – post request to Bind a Remote Memory Region to a memory range
- 854 [*it_rmr_create*](#) – create a Remote Memory Region (RMR)
- 855 [*it_rmr_free*](#) – destroy a Remote Memory Region
- 856 [*it_rmr_query*](#) – get attributes of a Remote Memory Region
- 857 [*it_rmr_unbind*](#) – post operation to Unbind a Remote Memory Region from its memory range
- 858 [*it_set_consumer_context*](#) – associate a Consumer Context with an IT Object Handle
- 859 [*it_ud_service_reply*](#) – returns UD communication information
- 860 [*it_ud_service_request*](#) – request the recipient to return UD communication information.
- 861 [*it_ud_service_request_handle_create*](#) – creates a UD Service Request Handle.
- 862 [*it_ud_service_request_handle_free*](#) – free a previously created [*it_ud_svc_req_handle_t*](#)
- 863 [*it_ud_service_request_handle_query*](#) – returns [*it_ud_svc_req_handle_t*](#) information.
- 864

865 **it_address_handle_create()**

866 **NAME**

867 `it_address_handle_create` – create an Address Handle

868 **SYNOPSIS**

```
869 #include <it_api.h>
870
871 it_status_t it_address_handle_create(
872     IN          it_pz_handle_t          pz_handle,
873     IN    const it_path_t              *destination_path,
874     IN          it_ah_flags_t          ah_flags,
875     OUT         it_addr_handle_t       *addr_handle
876 );
877
878 typedef enum {
879     IT_AH_PATH_COMPLETE = 0x1
880 } it_ah_flags_t;
```

881 **DESCRIPTION**

882 *pz_handle* Handle for the Protection Zone to be associated with the created
883 Address Handle. This implicitly identifies the Interface Adapter that
884 the Address Handle will be associated with.

885 *destination_path* The Path to use to create the Address Handle.

886 *ah_flags* The logical OR of the set of operation modifier flags specified
887 below.

888 *addr_handle* Returned datagram Address Handle.

889 *it_address_handle_create* creates an Address Handle, which is used when performing a Send
890 DTO on an Unreliable Datagram Endpoint.

891 The Protection Zone to associate with the newly created Address Handle is specified by
892 *pz_handle*. An Address Handle can only be used to post a Send DTO on an Unreliable
893 Datagram Endpoint that has a matching Protection Zone.

894 The Source and Destination address information necessary to create the Address Handle are
895 specified in the *destination_path* parameter. The Path can either be completely or incompletely
896 specified. A completely specified Path is one that contains all the necessary information to
897 create the Address Handle without the Implementation needing to consult a database of Path
898 records. An incompletely specified Path does not contain enough information to create the
899 Address Handle directly, but does contain enough information that the Implementation can
900 determine the rest of the information needed by consulting a database of Path records. The
901 Consumer should set the IT_AH_PATH_COMPLETE bit in *ah_flags* if the Path is completely
902 specified, or clear it if it is incompletely specified.

903 A completely specified Path that the Consumer can use to access a given remote network
904 Endpoint can be obtained using the [it_get_pathinfo](#) routine. A Path returned from
905 [it_get_pathinfo](#) can be used without modification by [it_address_handle_create](#). If the Consumer

906 wishes to have full control over the Path that datagrams sent using the created Address Handle
907 will take, they should furnish a completely specified Path.

908 An incompletely specified Path is obtained from the Completion Event for a Receive operation
909 on a datagram Endpoint. (See [it_dto_events](#) for details.) If an incompletely specified Path is
910 supplied to *it_address_handle_create*, the routine will automatically choose the unspecified
911 components of the Path required in order to reach the intended Destination.

912 The Consumer may also directly format the *destination_path* if they so desire. The
913 *destination_path* actually contains more information than is necessary to create an Address
914 Handle. The members of the [it_path_t](#) structure that are pertinent for creating an Address
915 Handle using a completely specified Path are listed in the table below. For each member,
916 whether the member is needed for an incompletely specified Path and the input modifier for the
917 Infiniband “Create Address Handle” verb that the member corresponds to are also identified.
918 For a detailed explanation of the semantics associated with each input modifier, see the “Create
919 Address Handle” section in chapter 11 of the Infiniband specification.

it_path_t member	Needed for incomplete Path?	IB Create Address Handle Input Modifier
ib.sl	Yes	Service level
ib.remote_port_lid	Yes	Destination LID
ib.flow_label	No	Flow label
ib.hop_limit	No	Hop limit
ib.traffic_class	No	Traffic class
ib.local_port_gid	No	Source GID index. (The Implementation uses the local_port_gid to determine the appropriate Source GID index.)
ib.remote_port_gid	No	Destination’s GID
ib.packet_rate	No	Maximum Static Rate
ib.local_port_lid	No	Source Path Bits. (The low order bits of the supplied local_port_lid are used as the Source Path Bits.)
ib.subnet_local	No	Send InfiniBand Global Routing Header flag
spigot_id	No	Physical Port

920

921 If the Consumer chooses to directly format the Path, it is possible that the Implementation will
922 decide that the resulting Path is one that the Consumer should not have access to. If so, a
923 permission violation error will be returned. The Implementation will generally not return such a
924 permission violation error if the Consumer instead uses a Path returned by [it_get_pathinfo](#) or
925 from the Completion Event for a Receive operation. (It is still possible that a permission
926 violation error could be returned if the network were reconfigured after the Path was returned
927 but before *it_address_handle_create* was furnished with that Path.)

928 **RETURN VALUE**

929 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
930 below.

931	IT_ERR_RESOURCES	The requested operation failed due to insufficient resources.
932		
933	IT_ERR_INVALID_PZ	The Protection Zone Handle (<i>pz_handle</i>) was invalid.
934	IT_ERR_INVALID_FLAGS	The flags (<i>ah_flags</i>) value was invalid.
935	IT_ERR_NO_PERMISSION	The Consumer did not have the proper permissions to perform the requested operation.
936		
937	IT_ERR_INVALID_SOURCE_PATH	One of the components of the Source portion of the supplied Path was invalid.
938		
939	IT_ERR_INVALID_SPIGOT	An invalid Spigot ID was specified.
940	IT_ERR_IA_CATASTROPHE	The Interface Adapter has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
941		
942		
943		

944 **ERRORS**

945 None.

946 **SEE ALSO**

947 [it_get_pathinfo\(\)](#), [it_path_t](#), [it_address_handle_query\(\)](#), [it_address_handle_modify\(\)](#),
948 [it_address_handle_free\(\)](#)

it_address_handle_free()

949

NAME950 `it_address_handle_free` – free an Address Handle
951**SYNOPSIS**

```

952 #include <it_api.h>
953
954 it_status_t it_address_handle_free(
955     IN it_addr_handle_t addr_handle
956 );
957

```

DESCRIPTION958 `addr_handle` Address Handle to free.

959 `it_address_handle_free` removes an existing Address Handle and frees all associated underlying resources. Once `it_address_handle_free` returns, `addr_handle` can no longer be used in DTO operations. If an Address Handle is freed while there is still a Send DTO outstanding that references the Address Handle, whether that Send completes successfully is implementation-dependent. Consumers that wish to write code that is independent of the Implementation are therefore advised to allow all outstanding Send operations that reference an Address Handle to complete before freeing the Address Handle.

RETURN VALUE

967 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described
968 below.
969
970

971 `IT_ERR_INVALID_AH` The Address Handle (`addr_handle`) was invalid.

972 `IT_ERR_IA_CATASTROPHE` The Interface Adapter has experienced a catastrophic error
973 and is in the disabled state. None of the output parameters
974 from this routine are valid. See [it_ia_info_t](#) for a description
975 of the disabled state.

ERRORS976 None.
977**SEE ALSO**978 [it_address_handle_create\(\)](#), [it_address_handle_query\(\)](#), [it_address_handle_modify\(\)](#)
979

it_address_handle_modify()

980

981 **NAME**

982 `it_address_handle_modify` – modify an Address Handle

983 **SYNOPSIS**

```

984 #include <it_api.h>
985
986 it_status_t it_address_handle_modify(
987     IN          it_addr_handle_t      addr_handle,
988     IN          it_addr_param_mask_t  mask,
989     IN  const   it_addr_param_t      *params
990 );
    
```

991 **DESCRIPTION**

- 992 *addr_handle* Address Handle to modify.
- 993 *mask* Logical OR of flags for desired parameters to be modified.
- 994 *params* Structure whose members contain the new parameter values.

995 *it_address_handle_modify* changes selected attributes of the Address Handle *addr_handle*. If
 996 this routine returns success, all requested attributes are modified. If it does not return success,
 997 none of the requested attributes are modified.

998 The Consumer should avoid calling this routine while a DTO that references this Address
 999 Handle is in progress. If the Consumer fails to abide by this restriction, the Destination that the
 1000 DTO is sent to is undefined.

1001 The attributes to be modified are specified by the flags in *mask*. New values for the attributes
 1002 are specified by the corresponding fields in the structure pointed to by *params*. Each field and
 1003 the corresponding flag name that must appear in *mask* to modify the given field are shown
 1004 below. (The flag name appears in a comment to the right of the field.) Note that attributes
 1005 represented by fields of [it_addr_param_t](#) that are not shown below can not be modified.

```

1006 typedef struct {
1007     ...
1008     it_path_t    path; /* IT_ADDR_PATH */
1009     ...
1010 } it_addr_param_t;
    
```

1012 The table below defines the meaning of each member of the [it_addr_param_t](#) structure.

it_addr_param_t member	Meaning
path	The new Path to be associated with this Address Handle. The Path will be associated with the Address Handle as a single unit. If the Consumer only wishes to modify a portion of the Path attributes, it can call it_address_handle_query to retrieve the current Path, modify the Path attributes as desired, and then call <i>it_address_handle_modify</i> with the resulting Path. See the it_address_handle_create man page for details about which

	portions of the Path are relevant for Address Handles.
--	--

1013

1014

1015

1016

The Consumer may not be allowed to access the Path that the requested modification to the Address Handle would imply. If that is the case, a permission violation error will be returned by this routine.

1017 **RETURN VALUE**

1018

1019

A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

1020

`IT_ERR_INVALID_AH`The Address Handle (*addr_handle*) was invalid.

1021

`IT_ERR_INVALID_MASK`The *mask* contained invalid flag values.

1022

1023

`IT_ERR_NO_PERMISSION`

The Consumer did not have the proper permissions to perform the requested operation.

1024

1025

`IT_ERR_INVALID_SGID`The Source Global ID (*ib.local_port_gid*) member of the supplied Path was invalid

1026

1027

`IT_ERR_INVALID_SLID`The Source Local ID (*ib.local_port_lid*) member of the supplied Path was invalid.

1028

`IT_ERR_INVALID_SPIGOT`

An invalid Spigot ID was specified.

1029

1030

1031

1032

`IT_ERR_IA_CATASTROPHE`The Interface Adapter has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See [it_ia_info_t](#) for a description of the disabled state.

1033

ERRORS

1034

None.

1035

SEE ALSO

1036

[it_address_handle_create\(\)](#), [it_address_handle_query\(\)](#), [it_address_handle_free\(\)](#)

it_address_handle_query()

1037

1038 **NAME**1039 `it_address_handle_query` – query an Address Handle1040 **SYNOPSIS**

```

1041 #include <it_api.h>
1042
1043 it_status_t it_address_handle_query(
1044     IN          it_addr_handle_t      addr_handle,
1045     IN          it_addr_param_mask_t  mask,
1046     OUT         it_addr_param_t       *params
1047 );
1048
1049 typedef enum {
1050     IT_ADDR_PARAM_ALL    = 0x0001,
1051     IT_ADDR_PARAM_IA    = 0x0002,
1052     IT_ADDR_PARAM_PZ    = 0x0004,
1053     IT_ADDR_PARAM_PATH  = 0x0008
1054 } it_addr_param_mask_t;
1055
1056 typedef struct {
1057     it_ia_handle_t      ia;          /* IT_ADDR_PARAM_IA */
1058     it_pz_handle_t      pz;          /* IT_ADDR_PARAM_PZ */
1059     it_path_t           path;        /* IT_ADDR_PARAM_PATH */
1060 } it_addr_param_t;

```

1061 **DESCRIPTION**1062 *addr_handle* Address Handle to query.1063 *mask* Logical OR of flags for desired parameters.1064 *params*: Structure whose members are written with the desired parameters.

1065 *it_address_handle_query* returns the desired attributes of the Address Handle *addr_handle* in
 1066 the structure pointed to by *params*. On return, each field of *params* is only valid if the
 1067 corresponding flag as shown above in the comment to the right of the field is set in the *mask*
 1068 argument. The *mask* value `IT_ADDR_PARAM_ALL` causes all fields to be returned.

1069 The table below defines the meaning of each member of the *it_addr_param_t* structure.

it_addr_param_t member	meaning
ia	The Handle for the IA that this Address Handle is associated with.
pz	The Handle for the Protection Zone that this Address Handle is associated with.
path	The Path that is associated with this Address Handle. Not all fields in the Path are relevant for an Address Handle; see the it_address_handle_create man page for details.

1070 **RETURN VALUE**

1071 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 1072 below.

1073 IT_ERR_INVALID_AH The Address Handle (*addr_handle*) was invalid.

1074 IT_ERR_INVALID_MASK The *mask* contained invalid flag values.

1075 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic error
 1076 and is in the disabled state. None of the output parameters
 1077 from this routine are valid. See [it_ia_info_t](#) for a
 1078 description of the disabled state.

1079 **ERRORS**

1080 None.

1081 **SEE ALSO**

1082 [it_address_handle_create\(\)](#), [it_address_handle_modify\(\)](#), [it_address_handle_free\(\)](#)

it_convert_net_addr()

1083

1084 **NAME**1085 `it_convert_net_addr` – convert a Network Address from one format to another1086 **SYNOPSIS**

```

1087 #include <it_api.h>
1088
1089 it_status_t it_convert_net_addr(
1090     IN      const      it_net_addr_t          *source_addr,
1091     IN      it_net_addr_type_t              addr_type,
1092     OUT     it_net_addr_t                    *destination_addr
1093 );

```

1094 **DESCRIPTION**1095 *source_addr* The input Network Address that is to be converted.1096 *addr_type* The new type of address to convert the *source_addr* address to.1097 *destination_addr* The returned Network Address.

1098 The *it_convert_net_addr* routine is used to convert one form of Network Address into another.
 1099 The type of Network Address desired is specified by *addr_type*, and upon successful return from
 1100 this routine *destination_addr* will contain an address of that type. If this routine does not return
 1101 success, the contents of *destination_addr* are undefined.

1102 The Implementation might not support the requested Network Address conversion. If it does
 1103 not, an error will be returned.

1104 The set of Network Addresses that are associated with a given Spigot is dynamic, and can
 1105 change over time. (For example, a link on a switch or router could become inoperative, thus
 1106 decreasing the set of Network Addresses by which a given Spigot can be reached.) There is
 1107 therefore no guarantee that given the same input parameters two different invocations of
 1108 *it_convert_net_addr* will return the same results. The Network Address returned by
 1109 *it_convert_net_addr* is chosen from amongst the Network Addresses that match the selection
 1110 criteria at the time of the call. In addition, since multiple Network Addresses of a given type can
 1111 be associated with the same Spigot, the Implementation may return a different Network Address
 1112 for two different invocations of *it_convert_net_addr* regardless of the state of the network.

1113 **RETURN VALUE**

1114
 1115 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described
 1116 below.

1117 `IT_ERR_INVALID_ADDRESS` The Network Address specified in *source_addr* was
 1118 invalid.

1119 `IT_ERR_INVALID_NETADDR` The format of the Network Address was not recognized.

1120 `IT_ERR_INVALID_CONVERSION` The requested Network Address conversion was not
 1121 supported by the Implementation.

1122 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic
1123 error and is in the disabled state. None of the output
1124 parameters from this routine are valid. See [it ia info t](#)
1125 for a description of the disabled state.

1126 **ERRORS**
1127 None.

1128 **APPLICATION USAGE**
1129 When a Consumer Receives an incoming Connection establishment attempt Event, the
1130 *source_addr_info* field in that Event will contain the Network Address of the initiator of the
1131 Connection establishment attempt. The type of Network Address contained within
1132 *source_addr_info* is Implementation-specific, and therefore may not be one that the Consumer
1133 wishes to deal with. The Consumer can use *it_convert_net_addr* to convert the Network
1134 Address supplied in *source_addr_info* into a type more to its liking.

1135 **SEE ALSO**
1136 [it_listen_create\(\)](#), [it_net_addr t](#)

it_ep_accept()

1137

1138 **NAME**1139 `it_ep_accept` - accept an incoming Connection Request or Connection Reply1140 **SYNOPSIS**

```

1141 #include <it_api.h>
1142
1143 it_status_t it_ep_accept(
1144     IN          it_ep_handle_t          ep_handle,
1145     IN          it_cn_est_identifier_t  cn_est_id,
1146     IN    const unsigned char          *private_data,
1147     IN          size_t                  private_data_length
1148 );
1149
1150 typedef uint64_t it_cn_est_identifier_t;

```

1151 **DESCRIPTION**

1152 *ep_handle* Local Endpoint to be bound to the Connection Request being
 1153 accepted.

1154 *cn_est_id* Connection Establishment Identifier associated with the Connection
 1155 Request being accepted. The *cn_est_id* is obtained from the data
 1156 delivered in the Connection Request Event
 1157 (IT_CM_REQ_CONN_REQUEST_EVENT). See the
 1158 ([it_cm_req_events](#)) manual page for details.

1159 *private_data* Opaque Private Data provided by the IT_CM_MSG_CONN_
 1160 PEER_REJECT_EVENT Event delivered to the Remote Consumer
 1161 that will be sent to the Remote Endpoint. If the Interface Adapter
 1162 does not support Private Data, *private_data_length* must be zero.
 1163 The delivery of Private Data to the Remote Endpoint is unreliable.
 1164

1165 *private_data_length* Length of *private_data*. This field must be 0 if the IA does not
 1166 support Private Data.

1167 *it_ep_accept* accepts an incoming Connection Request Event (IT_CM_REQ_CONN_
 1168 REQUEST_EVENT) or Connection accept arrival Event (IT_CM_MSG_CONN_ACCEPT_
 1169 ARRIVAL_EVENT). Calling *it_ep_accept* is the last Local Consumer step in establishing an
 1170 Endpoint-to-Endpoint Connection for a three-way Connection Establishment. The Consumer is
 1171 notified of an established Connection by an IT_CM_MSG_CONN_ESTABLISHED_EVENT
 1172 Event being delivered on the connect EVD of the Endpoint. The Event is generated on both the
 1173 active and passive side of the Connection establishment. See the Communication Management
 1174 Message Event ([it_cm_msg_events](#)) manual page for details.

1175 If the initial [it_ep_connect](#) specified two-way Connection Establishment then *it_ep_accept* is
 1176 called only on the Passive side of the Endpoint-to-Endpoint Connection.

1177

1178 If the initial [it_ep_connect](#) specified three-way Connection Establishment then *it_ep_accept* is
 1179 called on both the Active and the Passive sides of the Endpoint-to-Endpoint Connection. An
 1180 IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT Event will be delivered to the Active-
 1181 side Consumer after the Passive side calls *it_ep_accept*.

1182 On the Passive side, the Endpoint will transition into the IT_EP_STATE_
 1183 PASSIVE_CONNECTION_PENDING state when the Consumer calls *it_ep_accept*. The
 1184 Passive side successfully calling *it_ep_accept* will cause the Active Endpoint to eventually
 1185 transition into the IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state .

1186 *it_ep_accept* destroys the Connection Establishment Identifier, *cn_est_id*. After *it_ep_accept*
 1187 returns, *cn_est_id* is no longer valid and cannot be used.

1188 The Connection Establishment process can not be successfully completed unless the attributes of
 1189 the Local and Remote Endpoints are compatible; see [it_cm_msg_events](#) for details. The
 1190 Consumer can call [it_ep_modify](#) to make the Local Endpoint attributes compatible before calling
 1191 [it_ep_accept](#).

1192 RETURN VALUE

1193
 1194 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 1195 below.

1196	IT_ERR_PDATA_NOT_SUPPORTED	Private Data was supplied by the Consumer but this Interface Adapter does not support Private Data.
1197		
1198		
1199	IT_ERR_INVALID_PDATA_LENGTH	The Interface Adapter supports Private Data, but the length specified exceeded the Interface Adapter's capabilities.
1200		
1201		
1202	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
1203	IT_ERR_INVALID_EP_STATE	The Endpoint was not in the proper state for the attempted operation. See it_ep_state_t manual page.
1204		
1205		
1206	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
1207		
1208	IT_ERR_INVALID_CN_EST_ID	The Connection Establishment Identifier (<i>cn_est_id</i>) was invalid.
1209		
1210	IT_ERR_INVALID_EP_ATTR	The Local and Remote Endpoint attributes conflicted. Either the <i>max_message_size</i> , the number of <i>rdma_read_inflight_incoming</i> or the number of <i>rdma_read_inflight_outgoing</i> conflicted between the two Endpoints. This error will not be reported on the Passive-side accept of a three-way Connection Establishment.
1211		
1212		
1213		
1214		
1215		
1216		

1217	IT_ERR_EP_TIMEWAIT	The Endpoint provided to <i>it_ep_accept</i> was in the
1218		TimeWait condition, therefore the Connection
1219		could not be established. See it_ep_rc_create for
1220		details of the TimeWait condition.
1221	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is
1222		in the disabled state. None of the output
1223		parameters from this routine are valid. See
1224		it_ia_info_t for a description of the disabled state.
1225	ERRORS	
1226	None.	
1227	APPLICATION USAGE	
1228	1.	The Consumer should be aware that the delivery of Private Data to the Remote Endpoint
1229		may be unreliable and should be used accordingly. For some transports the Passive
1230		side's <i>it_ep_accept</i> Private Data is delivered reliably.
1231	2.	Calls to routines such as <i>it_ep_accept</i> , it_reject and it_ep_disconnect that pertain to the
1232		same Endpoint or Connection Establishment Identifier should be serialized by the
1233		Consumer. Failure to abide by this restriction may result in a segmentation violation or
1234		other error.
1235	3.	If <i>it_ep_accept</i> returns the IT_ERR_EP_TIMEWAIT error, the Consumer can recover
1236		either by retrying the Connection Establishment after the TimeWait interval has elapsed,
1237		or by retrying the Connection Establishment using a different Endpoint that is not under
1238		a TimeWait condition.
1239	4.	The Consumer should post at least one Receive buffer using the it_post_recv routine
1240		before calling <i>it_ep_accept</i> . Failure to do so can prevent a Connection from being
1241		established under certain circumstances on some transports.
1242	SEE ALSO	
1243		it_ep_connect() , it_reject() , it_ep_disconnect() , it_handoff() , it_ep_state_t , it_cm_req_events ,
1244		it_cm_msg_events
1245		

it_ep_connect()

1246

1247 **NAME**1248 `it_ep_connect` – initiate an Endpoint Connection establishment request1249 **SYNOPSIS**

```

1250 #include <it_api.h>
1251
1252 it_status_t it_ep_connect(
1253     IN          it_ep_handle_t          ep_handle,
1254     IN    const it_path_t*             path,
1255     IN    const it_conn_attributes_t*   conn_attr,
1256     IN    const it_conn_qual_t*       connect_qual,
1257     IN          it_cn_est_flags_t      cn_est_flags,
1258     IN    const unsigned char*        private_data,
1259     IN          size_t                 private_data_length
1260 );
1261
1262 /* Transport-specific connection attributes for InfiniBand */
1263 typedef struct {
1264
1265     /* Remote CM Response Timeout, as defined in the REQ
1266      message for the IB CM protocol */
1267     uint8_t          remote_cm_timeout : 5;
1268
1269     /* Local CM Response Timeout, as defined in the REQ
1270      message for the IB CM protocol */
1271     uint8_t          local_cm_timeout : 5;
1272
1273     /* Retry Count, as defined in the REQ message for the
1274      IB CM protocol */
1275     uint8_t          retry_count : 3;
1276
1277     /* RNR Retry Count, as defined in the REQ message for
1278      the IB CM protocol */
1279     uint8_t          rnr_retry_count : 3;
1280
1281     /* Max CM retries, as defined in the REQ message for
1282      the IB CM protocol */
1283     uint8_t          max_cm_retries : 4;
1284
1285     /* Local ACK Timeout, as defined in the REQ message
1286      for the IB CM protocol */
1287     uint8_t          local_ack_timeout : 5;
1288 } it_ib_conn_attributes_t;
1289
1290 /* Transport-specific connection attributes for VIA */
1291 typedef struct {
1292
1293     /* VIA currently has no transport-specific connection
1294      attributes */
1295
1296

```

```

1297     } it_via_conn_attributes_t;
1298
1299     /* Transport-specific connection attributes. This union is
1300     discriminated by the transport type being used to form the
1301     connection. This can be determined by examining
1302     the transport_type member in the it_ia_info_t that is
1303     associated with the IA that contains ep_handle. */
1304
1305     typedef union {
1306         it_ib_conn_attributes_t ib;
1307         it_via_conn_attributes_t via;
1308     } it_conn_attributes_t;
1309
1310     typedef enum {
1311         IT_CONNECT_FLAG_TWO_WAY      = 0x0001,
1312         IT_CONNECT_FLAG_THREE_WAY   = 0x0002
1313     } it_cn_est_flags_t;

```

1314 **DESCRIPTION**

- 1315 *ep_handle* Handle for an instance of the local Endpoint.
- 1316 *path* Path for Connection establishment request.
- 1317 *conn_attr* The transport-specific attributes for the Connection establishment
1318 attempt.
- 1319 *connect_qual:* The Connection Qualifier that the Consumer is initiating a
1320 Connection establishment request to. See [it_conn_qual_t](#) for more
1321 information on Connection Qualifiers.
- 1322 *cn_est_flags* Flags for the Connection establishment request.

1323

Features	Name	Bit value	Description
Two-way Connection establishment	IT_CONNECT_FLAG_TWO_WAY	0x0001	The Connection is established once the passive side of the Connection establishment calls it_ep_accept .
Three-way Connection establishment	IT_CONNECT_FLAG_THREE_WAY	0x0002	The Connection is established once the active side of the Connection establishment calls it_ep_accept .

1324

- 1325 *private_data* Opaque Private Data to be sent in the IT_CM_MSG_CONN_PEER_
- 1326 REJECT_EVENT Event delivered to the Remote Consumer. If the
- 1327 IA does not support Private Data, *private_data_length* must be zero.

1328 The delivery of Private Data to the Remote Endpoint is unreliable.
1329

1330 *private_data_length* Length of *private_data*. This field must be 0 if the IA does not
1331 support Private Data.

1332
1333 The *it_ep_connect* routine initiates a Connection establishment request for an existing local
1334 Endpoint using an *it_path_t*. The *path* can be found by using the *it_get_pathinfo* function. This
1335 request generates a connect establishment request (IT_CM_REQ_CONN_REQUEST_EVENT)
1336 Event on the passive side based on the Path provided. Once the Connection establishment
1337 request has been initiated the active side Endpoint transitions into the
1338 IT_EP_STATE_ACTIVE1_CONNECTION_PENDING state.

1339 Consumers that wish to write implementation-independent code should pass a NULL value for
1340 the *conn_attr* parameter. If the Consumer passes a NULL value for this parameter, the
1341 Implementation will choose implementation-dependent default values for the transport-specific
1342 Connection attributes that maximize the probability of the Connection being successfully
1343 established and maintained. If the Consumer wishes to have control over the transport-specific
1344 Connection attributes, they can pass a non-NULL value for the *conn_attr* parameter. If the
1345 Consumer passes a non-NULL value for this parameter and the Implementation determines that
1346 some portion of the transport-specific Connection attributes are invalid, it will return an error
1347 from this routine. What constitutes invalid transport-specific Connection attributes is
1348 implementation-dependent. The Implementation will not return an error indicating some portion
1349 of the transport-specific Connection attributes are invalid if the Consumer passes a NULL value
1350 for the *conn_attr* parameter.

1351 The passive side Consumer can either choose to either accept or reject the Connection Request.
1352 If the passive side chooses to reject the Connection by calling *it_reject* then an
1353 IT_CM_MSG_CONN_PEER_REJECT_EVENT Event is generated on the active side and the
1354 active side Endpoint transitions into the IT_EP_STATE_NONOPERATIONAL state. If it
1355 chooses to accept the Connection by calling *it_ep_accept* then the behavior is dependent on the
1356 type Connection setup specified by the *cn_est_flags*.

1357 For three-way Connection establishments, an IT_CM_MSG_CONN_ACCEPT_ARRIVAL_
1358 EVENT Event is generated on the active side if the passive side Consumer accepts the
1359 Connection establishment request by calling *it_ep_accept*. The active Consumer can choose to
1360 either accept or reject the Connection by calling *it_ep_accept* or *it_reject* respectively. For a
1361 two-way Connection establishment, an IT_CM_MSG_CONN_ESTABLISHED_EVENT Event
1362 is generated on the active side after the passive side Consumer accepts the Connection and the
1363 Endpoint transitions into the (IT_EP_STATE_CONNECTED) connected state. See the
1364 *it_ep_state_t* manual page for a complete description on the Endpoint state diagram for both the
1365 three-way and two-way Connection establishment.

1366 Whenever an Endpoint transitions to the connected (IT_EP_STATE_CONNECTED) state the
1367 Consumer will Receive an IT_CM_MSG_CONN_ESTABLISHED_EVENT Event on the
1368 simple Event Dispatcher to which the Communication Management Message Event Stream is
1369 routed after the Endpoint transitions into the IT_EP_STATE_CONNECTED state. This Event is
1370 generated on both the active and passive side of the Connection establishment after the Endpoint
1371 state transition takes place.

1372 For a complete definition of Endpoint state and a more complete description of the state
 1373 transitions see the [it_ep_state_t](#) manual page. If for any reason an Endpoint Connection fails to
 1374 be established the Endpoint will transition into the IT_EP_STATE_NONOPERATIONAL state
 1375 and any Receive DTO operations that were successfully posted to the Endpoint will be
 1376 completed with an IT_DTO_ERR_FLUSHED status.

1377 EXTENDED DESCRIPTION

1378 An Endpoint can only be connected to a different Endpoint. An Endpoint can not be connected
 1379 to itself.

1380 RETURN VALUE

1381
 1382 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 1383 below.

1384 IT_ERR_NO_PERMISSION The Consumer did not have the proper
 1385 permissions to perform the requested operation.

1386 IT_ERR_RESOURCES The operation failed due to resource limitations.

1387 IT_ERR_INVALID_CONN_QUAL The Connection Qualifier is invalid.

1388 IT_ERR_PDATA_NOT_SUPPORTED Private Data was supplied by the Consumer but
 1389 this IA does not support Private Data. See
 1390 [it_ia_query](#) for the IA's capabilities to support
 1391 Private Data.

1392 IT_ERR_INVALID_PDATA_LENGTH The IA supports Private Data, but the length
 1393 specified exceeds the IA's capabilities.

1394 IT_ERR_INVALID_SOURCE_PATH The Source Path specified in the [it_path_t](#) was
 1395 invalid.

1396 IT_ERR_INVALID_SPIGOT The Spigot specified in the [it_path_t](#) was invalid.

1397 IT_ERR_INVALID_EP The *ep_handle* was invalid.

1398 IT_ERR_INVALID_EP_STATE The Endpoint is not in the proper state to be
 1399 connected.

1400 IT_ERR_INVALID_EP_TYPE The Endpoint Service Type does not support this
 1401 operation.

1402 IT_ERR_INVALID_CN_EST_FLAGS The Connection establishment flags are invalid.

1403 IT_ERR_INVALID_RTIMEOUT The *conn_attr.ib.remote_cm_timeout* value was
 1404 invalid. The criteria for determining what
 1405 constitutes an invalid value are implementation-
 1406 dependent.

1407 IT_ERR_INVALID_LTIMEOUT The *conn_attr.ib.local_cm_timeout* value was
 1408 invalid. The criteria for determining what

1409		constitutes an invalid value are implementation-dependent.
1410		
1411	IT_ERR_INVALID_RETRY	The <i>conn_attr.ib.retry_count</i> value was invalid.
1412		The criteria for determining what constitutes an
1413		invalid value are implementation-dependent.
1414	IT_ERR_INVALID_RNR_RETRY	The <i>conn_attr.ib.rnr_retry_count</i> value was
1415		invalid. The criteria for determining what
1416		constitutes an invalid value are implementation-
1417		dependent.
1418	IT_ERR_INVALID_CM_RETRY	The <i>conn_attr.ib.max_cm_retries</i> value was
1419		invalid. The criteria for determining what
1420		constitutes an invalid value are implementation-
1421		dependent.
1422	IT_ERR_INVALID_ETIMEOUT	The <i>conn_attr.ib.local_ack_timeout</i> value was
1423		invalid. The criteria for determining what
1424		constitutes an invalid value are implementation-
1425		dependent.
1426	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is
1427		in the disabled state. None of the output
1428		parameters from this routine are valid. See
1429		it_ia_info_t for a description of the disabled state.
1430	ERRORS	
1431		None.
1432	APPLICATION USAGE	
1433		It is possible that between the time the Consumer calls it_get_pathinfo to retrieve a valid Path
1434		and the time the Consumer passes that Path as the <i>path</i> parameter to this routine, the set of
1435		available Paths through the network for forming the Connection may change, rendering some
1436		portion of the given <i>path</i> invalid. If IT_ERR_INVALID_SOURCE_PATH or IT_ERR_
1437		INVALID_SPIGOT is returned from this routine and the <i>path</i> the Consumer provided is one that
1438		was returned from it_get_pathinfo and was not subsequently modified by the Consumer, the
1439		Consumer can attempt to recover from the error by calling it_get_pathinfo again to retrieve an
1440		up-to-date Path to form the Connection and retrying the call to <i>it_ep_connect</i> with the new Path.
1441	SEE ALSO	
1442		it_ep_accept() , it_reject() , it_ep_disconnect() , it_handoff() , it_ep_state_t , it_ia_query() ,
1443		it_conn_qual_t

it_ep_disconnect()

1444

1445 **NAME**1446 `it_ep_disconnect` - disconnect an existing Endpoint-to-Endpoint Connection1447 **SYNOPSIS**

```

1448 #include <it_api.h>
1449
1450 it_status_t it_ep_disconnect (
1451     IN          it_ep_handle_t    ep_handle,
1452     IN          const unsigned char * private_data,
1453     IN          size_t            private_data_length
1454 );

```

1455 **DESCRIPTION**1456 `ep_handle` Endpoint to be disconnected.

1457 `private_data` Opaque Private Data to be delivered in the
 1458 IT_CM_MSG_CONN_DISCONNECT_EVENT Event at the
 1459 Remote Endpoint. If the IA does not support Private Data,
 1460 `private_length` must be zero.

1461 `private_data_length`: Length of `private_data`. This field must be 0 if the IA does not
 1462 support Private Data.

1463 `it_ep_disconnect` either breaks the existing Endpoint-to-Endpoint Connection or terminates
 1464 Endpoint-to-Endpoint Connection in the process of being established identified by the
 1465 `ep_handle`. An IT_CM_MSG_CONN_DISCONNECT_EVENT Event will be generated on both
 1466 the Local and Remote sides of the Connection. The generation of the Event on the remote Event
 1467 is not guaranteed. The Endpoints will transition into the IT_EP_STATE_NONOPERATIONAL
 1468 state. See the [it_ep_reset](#) manual page for how to restore an Endpoint back into the
 1469 IT_EP_STATE_UNCONNECTED state. `it_ep_disconnect` is ungraceful in the sense that the
 1470 remote Endpoints transitions directly into the IT_EP_STATE_NONOPERATIONAL state
 1471 without the Consumer's intervention.

1472 `it_ep_disconnect` may be successfully called in all states except IT_EP_STATE_
 1473 UNCONNECTED state.

1474 Once the Endpoint is in the IT_EP_STATE_NONOPERATIONAL state any pending Data
 1475 Transfer Operations or Bind or Unbind operations on the Endpoint will be flushed and will
 1476 generate Completion Events with a Status of IT_DTO_ERR_FLUSHED.

1477 See the [it_ep_state_t](#) manual page for a complete description of the Endpoint state behavior and
 1478 transitions.

1479 The delivery of Private Data to the Remote Endpoint is unreliable.

1480 **RETURN VALUE**

1481

1482 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 1483 below.

1484		
1485	IT_ERR_INVALID_EP_STATE	The Endpoint was not in the proper state for the attempted operation.
1486		
1487	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
1488	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
1489		
1490	IT_ERR_PDATA_NOT_SUPPORTED	Private Data was supplied by the Consumer but this Interface Adapter does not support Private Data. See it_ia_query for the IA's capabilities to support Private Data.
1491		
1492		
1493		
1494		
1495	IT_ERR_INVALID_PDATA_LENGTH	The Interface Adapter supports Private Data, but the length specified exceeded the Interface Adapter's capabilities.
1496		
1497		
1498	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
1499		
1500		
1501		
1502	ERRORS	
1503	See above.	
1504	APPLICATION USAGE	
1505	The Consumer should be aware that the delivery of Private Data to the Remote Endpoint is unreliable and should be used accordingly.	
1506		
1507	The Consumer is responsible for coordinating the use of functions that free a Connection Establishment Identifier (<i>cn_est_id</i>) such as it_ep_accept , it_reject , it_ep_disconnect and it_handoff . The behavior of functions that are passed as invalid Connection Establishment Identifier is indeterminate.	
1508		
1509		
1510		
1511	The Consumer should be aware that successfully returning from this routine does not guarantee that any interaction whatsoever will take place with the Remote Endpoint. If the Local Consumer wishes to ensure that the Remote Consumer takes some action, an explicit message should be sent to initiate that action before calling it_ep_disconnect .	
1512		
1513		
1514		
1515	With the three-way handshake Connection establishment method, there is also a potential race condition between the Implementation generating the <code>IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT</code> Event and the Consumer calling it_ep_free . The Consumer should not use the <i>cn_est_id</i> if the <code>IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT</code> Event arrives after it_ep_free was called, regardless of whether the call returned yet, and regardless of whether the Event was dequeued before or after the call was made. If the Consumer does use the <i>cn_est_id</i> then the Implementation generate an <code>IT_ERR_INVALID_CN_EST_ID</code> error, or it may generate a segmentation fault or other error.	
1516		
1517		
1518		
1519		
1520		
1521		
1522		

1523 **SEE ALSO**

1524 [*it_ep_accept\(\)*](#), [*it_reject\(\)*](#), [*it_ep_connect\(\)*](#), [*it_ep_state_t*](#), [*it_cm_msg_events*](#), [*it_ep_reset\(\)*](#),
1525 [*it_ia_query\(\)*](#)

1526

it_ep_free()

1527

1528 **NAME**1529 `it_ep_free` – destroy an RC or UD Endpoint1530 **SYNOPSIS**

```

1531 #include <it_api.h>
1532
1533 it_status_t it_ep_free(
1534     IN          it_ep_handle_t    ep_handle
1535 );

```

1536 **DESCRIPTION**1537 `ep_handle` Endpoint.1538 `it_ep_free` destroys an Endpoint.

1539 The Endpoint can be destroyed in any state. The freeing of an Endpoint also terminates the
 1540 generation of Events to any of the EVDs associated with the Endpoint.

1541 Use of the Handle `ep_handle` of the destroyed Endpoint in any subsequent operation fails.

1542 Freeing an Endpoint potentially means Events might be lost on the `recv_sevd_handle` or
 1543 `request_sevd_handle` SEVDs associated with the Endpoint. There is also potential to lose Events
 1544 on the `connect_sevd_handle` SEVD associated with the Endpoint. The Consumer should first
 1545 drain these EVDs before calling `it_ep_free`.

1546 Freeing an Endpoint in the `IT_EP_STATE_CONNECTED` state while DTOs are in progress
 1547 causes incoming DTOs to be ignored. The outstanding DTOs and RMRs may be flushed to the
 1548 `request_sevd_handle` and `recv_sevd_handle`. All entries on the Endpoint `request_sevd_handle`,
 1549 `recv_sevd_handle`, and `connect_sevd_handle` may or may not be on the EVDs after the Endpoint
 1550 is destroyed.

1551 Freeing an Endpoint in the `IT_EP_STATE_ACTIVE1_CONNECTION_PENDING`,
 1552 `IT_EP_STATE_ACTIVE2_CONNECTION_PENDING`, or `IT_EP_STATE_PASSIVE_`
 1553 `CONNECTION_PENDING` state may cause a Connection establishment timeout or non-peer
 1554 reject to be sent to the remote side.

1555 **RETURN VALUE**1556 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

1557 `IT_ERR_INVALID_EP` The Endpoint Handle (`ep_handle`) was invalid.

1558 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 1559 disabled state. None of the output parameters from this
 1560 routine are valid. See [it_ia_info.t](#) for a description of the
 1561 disabled state.

1562 **ERRORS**

1563 None.

1564 **APPLICATION USAGE**

1565 Since the Implementation may not immediately free underlying resources, the user must not rely
1566 upon being immediately able to reallocate an Endpoint that has been freed.

1567 If the Consumer wants to ensure that all Completion Events are dequeued prior to calling
1568 *it_ep_free*, the following method will work for both Request and Receive Queues:

1569 The Consumer should call *it_ep_disconnect* first. Then post a DTO or RMR operation set up as
1570 a "marker" that is flushed by the Implementation to *recv_evd_handle* or *request_evd_handle*.
1571 The DTO or RMR is made a "marker" operation by setting the IT_COMPLETION_FLAG on
1572 the operation. Now, when the Consumer dequeues the completion of the "marker" from the
1573 EVD, it is guaranteed that all previously posted DTO and RMR completions (including those
1574 posted with IT_COMPLETION_FLAG cleared) for the Endpoint were dequeued from that EVD
1575 for the Request or Receive Queue of the Endpoint to which the "marker" had been posted.

1576 For *connect_evd_handle*, Consumer should dequeue the disconnect or broken Connection Event
1577 for this Endpoint.

1578 After all of the previous steps, it is safe to destroy or reset the Endpoint without losing any
1579 completions or Connection Events.

1580 With the three-way handshake Connection establishment method, there is also a potential race
1581 condition between the Implementation generating the IT_CM_MSG_CONN_ACCEPT_
1582 ARRIVAL_EVENT Event and the Consumer calling *it_ep_disconnect*. The Consumer should
1583 not use the *cn_est_id* if the IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT Event arrives
1584 after *it_ep_disconnect* was called, regardless of whether the call returned yet, and regardless of
1585 whether the Event was dequeued before or after the call was made. If the Consumer does use the
1586 *cn_est_id* then the Implementation generate an IT_ERR_INVALID_CN_EST_ID error, or it
1587 may generate a segmentation fault or other error.

1588 **SEE ALSO**

1589 [*it_ep_rc_create\(\)*](#), [*it_ep_ud_create\(\)*](#), [*it_ep_modify\(\)*](#), [*it_ep_query\(\)*](#), [*it_ep_reset\(\)*](#),
1590 [*it_ep_disconnect\(\)*](#), [*it_ia_info_t*](#), [*it_ep_state_t*](#), [*it_dto_flags_t*](#), [*it_post_send\(\)*](#), [*it_post_sendto\(\)*](#),
1591 [*it_post_recv\(\)*](#), [*it_post_recvfrom\(\)*](#), [*it_post_rdma_read\(\)*](#), [*it_post_rdma_write\(\)*](#)

it_ep_modify()

1592

1593 **NAME**1594 `it_ep_modify` – modify attributes of an existing Endpoint1595 **SYNOPSIS**

```

1596 #include <it_api.h>
1597
1598 it_status_t it_ep_modify(
1599     IN          it_ep_handle_t          ep_handle,
1600     IN          it_ep_param_mask_t     mask,
1601     IN    const it_ep_attributes_t     *ep_attr
1602 );

```

1603 **DESCRIPTION**1604 `ep_handle` Endpoint.1605 `mask` Logical OR of flags for desired attributes to be modified.1606 `ep_attr` Pointer to Consumer-allocated structure that contains new
1607 Consumer-requested Endpoint attributes.1608 *it_ep_modify* changes selected attributes of the Endpoint *ep_handle*.1609 Attributes to be modified are specified by flags in *mask*. New values for the attributes are
1610 specified by the corresponding fields in the structure pointed to by *ep_attr*. See
1611 [it_ep_attributes_t](#) for the definition of the structure.1612 Flag values for the *mask* parameter are shown below. Note that attributes represented by fields
1613 of *ep_attr* for which no flag value is shown below can not be modified. The requested attribute
1614 changes only affect the local Endpoint and have no effect on attributes of any remote Endpoint.

1615 Flag values for attributes that may be potentially modified:

1616 `IT_EP_PARAM_MAX_PAYLOAD`1617 `IT_EP_PARAM_MAX_REQ_DTO`1618 `IT_EP_PARAM_MAX_RECV_DTO`1619 `IT_EP_PARAM_RDMA_RD_ENABLE`1620 `IT_EP_PARAM_RDMA_WR_ENABLE`1621 `IT_EP_PARAM_MAX_IRD`1622 `IT_EP_PARAM_MAX_ORD`1623 `IT_EP_PARAM_EP_KEY`1624 See [it_ep_attributes_t](#) for the definition of valid states in which each of the above attributes may
1625 be modified.1626 Values for *mask* must be created as the logical OR of the Endpoint attributes flag values (above)
1627 that Consumer desires to change. *it_ep_modify* must succeed in modifying all the requested

1628 attributes atomically; if the attempt to modify any of the requested attributes generates an error,
 1629 none of the other attributes supplied to the call will be applied.

1630 **RETURN VALUE**

1631
 1632 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below:
 1633

1634	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
1635	IT_ERR_INVALID_MASK	The mask contained invalid flag values.
1636	IT_ERR_INVALID_EP_STATE	The Endpoint was not in the proper state for the attempted operation. See it_ep_attributes_t .
1637		
1638	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
1639		
1640	IT_ERR_PAYLOAD_SIZE	The requested <i>max_dto_payload_size</i> exceeds the maximum payload size supported by the underlying transport.
1641		
1642		
1643	IT_ERR_RESOURCE_REQ_DTO	The underlying transport could not allocate the requested <i>max_req_dtos</i> resources at this time.
1644		
1645	IT_ERR_RESOURCE_RECV_DTO	The underlying transport could not allocate the requested <i>max_recv_dtos</i> resources at this time.
1646		
1647	IT_ERR_RESOURCE_IRD	The underlying transport could not allocate the requested <i>rdma_read_inflight_incoming</i> resources at this time.
1648		
1649		
1650	IT_ERR_RESOURCE_ORD	The underlying transport could not allocate the requested <i>rdma_read_inflight_outgoing</i> resources at this time.
1651		
1652		
1653	IT_ERR_INVALID_EP_KEY	Invalid Endpoint Key value. The Consumer doesn't have local permissions to use the specified Endpoint Key.
1654		
1655		
1656	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
1657		
1658		
1659		

1660 **ERRORS**

1661 None.

1662 **SEE ALSO**

1663 [it_ep_attributes_t](#), [it_ep_rc_create\(\)](#), [it_ep_ud_create\(\)](#), [it_ep_query\(\)](#), [it_ep_free\(\)](#), [it_ia_info_t](#)

it_ep_query()

1664

1665 **NAME**1666 `it_ep_query` – query an existing Endpoint1667 **SYNOPSIS**

```

1668 #include <it_api.h>
1669
1670 it_status_t it_ep_query(
1671     IN          it_ep_handle_t          ep_handle,
1672     IN          it_ep_param_mask_t     mask,
1673     OUT         ep_param_t             *params
1674 );
1675
1676 typedef struct {
1677     it_ia_handle_t    ia;                /* IT_EP_PARAM_IA */
1678     size_t           spigot_id;         /* IT_EP_PARAM_SPIGOT */
1679     it_ep_state_t    ep_state;         /* IT_EP_PARAM_STATE */
1680     it_transport_service_type_t
1681         service_type;                /* IT_EP_PARAM_SERV_TYPE */
1682     it_path_t        dst_path;         /* IT_EP_PARAM_PATH */
1683     it_pz_handle_t   pz;              /* IT_EP_PARAM_PZ */
1684     it_evd_handle_t  request_sevd;     /* IT_EP_PARAM_REQ_SEVD */
1685     it_evd_handle_t  recv_sevd;       /* IT_EP_PARAM_RECV_SEVD */
1686     it_evd_handle_t  connect_sevd;    /* IT_EP_PARAM_CONN_SEVD */
1687     it_ep_attributes_t attr;          /* see it_ep_attributes_t
1688                                     for mask flags for attr */
1689 } it_ep_param_t;

```

1690 **DESCRIPTION**

1691 `ep_handle` Endpoint.

1692 `mask` Logical OR of flags for desired parameters and attributes.

1693 `params` Pointer to Consumer-allocated structure whose members are
1694 written with the desired Endpoint parameters and attributes.

1695 `it_ep_query` returns the desired parameters and attributes of the Endpoint `ep_handle` in the
1696 structure pointed to by `params`. On return, each field of `params` is only valid if the corresponding
1697 flag as shown below each `it_ep_param_t` member is set in the `mask` argument. The `mask` value
1698 `IT_EP_PARAM_ALL` causes all fields to be returned. The `it_ep_param_mask_t` enum is defined
1699 in [it_ep_attributes_t](#).

1700 The definition of each field follows:

1701 `ia` Handle for the Interface Adapter specified to create the EP.

1702 `spigot_id` Spigot Identifier. For RC Endpoint valid only if not in
1703 `IT_EP_STATE_UNCONNECTED` state otherwise, value is
1704 undefined.

1705 `ep_state` State of the Endpoint.

1706	<i>service_type</i>	Endpoint Service Type.
1707	<i>dst_path</i>	For RC it is invalid and contents are undefined if the Endpoint is in the IT_EP_STATE_UNCONNECTED state. Otherwise, on the active side of a Connection, it is the Path that was specified at Connection Request time; on the passive side of a Connection, it is the Path used by the Implementation to reach the requesting remote Endpoint. Invalid for UD Endpoint.
1708		
1709		
1710		
1711		
1712		
1713	<i>pz</i>	Handle for the Protection Zone specified while creating the EP.
1714	<i>request_sevd</i>	Handle for the IT_DTO_EVENT_STREAM Simple Event Dispatcher for DTO request Completion Events of the created Endpoint.
1715		
1716		
1717	<i>recv_sevd</i>	Handle for the IT_DTO_EVENT_STREAM Simple Event Dispatcher for DTO Receive Completion Events of the created Endpoint.
1718		
1719		
1720	<i>connect_sevd</i>	Handle for the IT_CM_MSG_EVENT_STREAM Simple Event Dispatcher for Connection Events of the created Endpoint. Invalid for UD Endpoint.
1721		
1722		
1723	<i>attr</i>	Attributes of Endpoint – definitions and mask values found in it_ep_attributes.t . Consumer ORs the appropriate mask values for each attribute field desired into the <i>mask</i> parameter to <i>it_ep_query</i> .
1724		
1725		
1726		
1726	RETURN VALUE	
1727		A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.
1728	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
1729	IT_ERR_INVALID_MASK	The mask contained invalid flag values.
1730	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
1731		
1732	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info.t for a description of the disabled state.
1733		
1734		
1735		
1736	ERRORS	
1737		None.
1738	SEE ALSO	
1739		it_ep_attributes.t , it_ep_rc_create() , it_ep_ud_create() , it_ep_modify() , it_ep_free() ,
1740		it_ia_info.t

it_ep_rc_create()

1741

1742 **NAME**1743 `it_ep_rc_create` – create an Endpoint for Reliable Connection1744 **SYNOPSIS**

```

1745 #include <it_api.h>
1746
1747 it_status_t it_ep_rc_create (
1748     IN          it_pz_handle_t           pz_handle,
1749     IN          it_evd_handle_t         request_sevd_handle,
1750     IN          it_evd_handle_t         rcv_sevd_handle,
1751     IN          it_evd_handle_t         connect_sevd_handle,
1752     IN          it_ep_rc_creation_flags_t flags,
1753     IN    const it_ep_attributes_t      *ep_attr,
1754     OUT         it_ep_handle_t          *ep_handle
1755 );
1756
1757 typedef enum {
1758     IT_EP_NO_FLAG      = 0x00,
1759     IT_EP_REUSEADDR   = 0x01
1760 } it_ep_rc_creation_flags_t;

```

1761 **DESCRIPTION**

1762 *pz_handle* Handle for the Protection Zone of the created Endpoint. Implicitly
1763 identifies the Interface Adapter to be used.

1764 *request_sevd_handle* Handle for the IT_DTO_EVENT_STREAM Simple Event
1765 Dispatcher for DTO request Completion Events of the created
1766 Endpoint.

1767 *rcv_sevd_handle* Handle for the IT_DTO_EVENT_STREAM Simple Event
1768 Dispatcher for DTO Receive Completion Events of the created
1769 Endpoint.

1770 *connect_sevd_handle* Handle for the IT_CM_MSG_EVENT_STREAM Simple Event
1771 Dispatcher for Connection Events of the created Endpoint.

1772 *flags* Flags allowing Consumer optionally to control behavior of the
1773 Implementation on Endpoint creation. Default is IT_EP_NO_FLAG.

1774 *ep_attr* Pointer to a structure that contains Consumer-requested Endpoint
1775 Attributes.

1776 *ep_handle* Handle for the created Endpoint.

1777 *it_ep_rc_create* creates, on the Interface Adapter implicitly identified by *pz_handle*, a
1778 Connection Endpoint that is provided to the Consumer as *ep_handle*. The value of *ep_handle* is
1779 only defined if the return value of *it_ep_rc_create* is IT_SUCCESS.

1780 The Connection Endpoint is created in the IT_EP_STATE_UNCONNECTED state. See
1781 [it_ep_state_t](#) for details.

1782 The created Endpoint is not associated with an IA Spigot. An Endpoint is associated with a
1783 Spigot as part of Connection setup.

1784 The Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint
1785 can access for DTOs and what memory remote RDMA operations can access through the newly
1786 created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint
1787 Protection Zone can be accessed through the Endpoint.

1788 *recv_sevd_handle* and *request_sevd_handle* are Event Dispatcher instances where the Consumer
1789 collects completion Notifications of DTOs and RMR operations. Completions of Receive DTOs
1790 are reported in *recv_sevd_handle* Event Dispatcher, and completions of Send, RDMA Read,
1791 RDMA Write DTOs, RMR Bind and RMR Unbind are reported in *request_sevd_handle*. It is
1792 permissible for *recv_sevd_handle* and *request_sevd_handle* to reference the same EVD. DTO
1793 and RMR operation Completion Events are defined in [it_dto_events](#).

1794 The Consumer should not specify an SEVD in *recv_sevd_handle* or *request_sevd_handle* that is
1795 in overflowed state for use in the Endpoint creation call (see [it_evd_create](#) for more details on
1796 overflow). If Consumer attempts to do so the operation will fail with
1797 IT_ERR_INVALID_RECV_EVD_STATE or IT_ERR_INVALID_REQ_EVD_STATE.

1798 All Connection Events for the Endpoint are reported to the Consumer through the SEVD
1799 specified in *connect_sevd_handle*. For a complete list of Endpoint Connection Events, see
1800 [it_cm_msg_events](#).

1801 The *flags* parameter allows the Consumer to control the behavior of the Implementation on
1802 Endpoint creation. Use of the *flags* value IT_EP_REUSEADDR allows the Consumer to specify
1803 that they allow the Implementation to return an Endpoint on creation that is possibly in the
1804 TimeWait state. Normally, the Implementation will only return Endpoints that are not in the
1805 TimeWait state.

1806 The TimeWait state exists for the purpose of preventing packets that were transmitted over one
1807 Connection from being inadvertently received in another subsequently established Connection.
1808 The TimeWait state is not a state of the Endpoint per se, but rather a state associated with a
1809 Connection the Endpoint had previously established. A Connection enters the TimeWait state
1810 when a disconnect is performed, and exits the TimeWait state after a TimeWait interval has
1811 elapsed. The duration of the TimeWait interval is transport-dependent, and for some transports
1812 it is also dependent upon network configuration parameters as well. This interval can be on the
1813 order of a minute or two in length.

1814 An Endpoint that is “in the TimeWait state” still has at least one Connection that it had
1815 previously established for which the TimeWait interval has not elapsed. (It is possible for an
1816 Endpoint to be in the TimeWait state with respect to multiple Connections it had previously
1817 established.) If an Endpoint attempts to establish a Connection that will use the same pair of
1818 Spigots that were involved in a previous Connection involving the Endpoint, and if that previous
1819 Connection is currently in the TimeWait state, the Connection establishment attempt may fail
1820 with an IT_ERR_EP_TIMEWAIT error; see [it_ep_accept](#). This error will never be returned
1821 unless the Consumer specifies IT_EP_REUSEADDR in *flags* for the *it_ep_rc_create* routine.

1822 The *ep_attr* parameter specifies the Consumer-requested attributes of the created Endpoint. The
1823 Implementation is required to satisfy all requested attributes or fail the operation. Hence, the
1824 Implementation must allocate all necessary resources to satisfy Consumer-requested attributes.
1825 The Implementation is allowed to allocate more resources than Consumer requested in *ep_attr*.

1826 The Consumer can find the actual allocated resources by using [it_ep_query](#). For detailed
1827 Endpoint attributes see man page for [it_ep_attributes](#).

1828 RETURN VALUE

1829 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below:

1830	IT_ERR_INVALID_PZ	The Protection Zone Handle (<i>pz_handle</i>) was 1831 invalid.
1832	IT_ERR_INVALID_REQ_EVD	The Simple Event Dispatcher Handle for Data 1833 Transfer Operation request completions 1834 (<i>request_sevd_handle</i>) was invalid.
1835	IT_ERR_INVALID_RECV_EVD	The Simple Event Dispatcher Handle for Data 1836 Transfer Operation Receive completions 1837 (<i>recv_sevd_handle</i>) was invalid.
1838	IT_ERR_INVALID_CONN_EVD	The Connection Simple Event Dispatcher 1839 Handle was invalid.
1840	IT_ERR_INVALID_EVD_TYPE	The Event Stream Type for the Event 1841 Dispatcher was invalid.
1842	IT_ERR_INVALID_REQ_EVD_STATE	The Simple Event Dispatcher for Data 1843 Transfer Operation request completions was 1844 in an unusable state.
1845	IT_ERR_INVALID_RECV_EVD_STATE	The Simple Event Dispatcher for Data 1846 Transfer Operation Receive completions was 1847 in an unusable state.
1848	IT_ERR_INVALID_FLAGS	The flags value was invalid.
1849	IT_ERR_RESOURCES	The requested operation failed due to 1850 insufficient resources.
1851	IT_ERR_PAYLOAD_SIZE	The requested <i>max_dto_payload_size</i> exceeds 1852 the maximum payload size supported by the 1853 underlying transport.
1854	IT_ERR_RESOURCE_REQ_DTO	The underlying transport could not allocate 1855 the requested <i>max_req_dtos</i> resources at this 1856 time.
1857	IT_ERR_RESOURCE_RECV_DTO	The underlying transport could not allocate 1858 the requested <i>max_recv_dtos</i> resources at this 1859 time.
1860	IT_ERR_RESOURCE_SSEG	The underlying transport could not allocate 1861 the requested <i>max_send_segments</i> resources 1862 at this time.

1863	IT_ERR_RESOURCE_RSEG	The underlying transport could not allocate the requested <i>max_rcv_segments</i> resources at this time.
1864		
1865		
1866	IT_ERR_RESOURCE_RRSEG	The underlying transport could not allocate the requested <i>max_rdma_read_segments</i> resources at this time.
1867		
1868		
1869	IT_ERR_RESOURCE_RWSEG	The underlying transport could not allocate the requested <i>max_rdma_write_segments</i> resources at this time.
1870		
1871		
1872	IT_ERR_RESOURCE_IRD	The underlying transport could not allocate the requested <i>rdma_read_inflight_incoming</i> resources at this time.
1873		
1874		
1875	IT_ERR_RESOURCE_ORD	The underlying transport could not allocate the requested <i>rdma_read_inflight_outgoing</i> resources at this time.
1876		
1877		
1878	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it ia info for a description of the disabled state.
1879		
1880		
1881		
1882		

1883 **ERRORS**

1884 None.

1885 **APPLICATION USAGE**

1886 Use of IT_EP_REUSEADDR requires the Consumer to handle a potential
 1887 IT_ERR_EP_TIMEWAIT error from [it ep accept](#) if the Endpoint and an incoming Connection
 1888 Request are in the TimeWait state with respect to each other.

1889 Sometimes the required attribute values for an Endpoint depend on parameters in an incoming
 1890 Connection Request and are not known at Endpoint creation time. The Consumer should specify
 1891 these attributes at a later time using [it ep modify](#), for example, before accepting an incoming
 1892 Connection Request.

1893 Specifying an overflowed SEVD in *connect_sevd_handle* is recoverable but may result in
 1894 connect Events being lost.

1895 **SEE ALSO**

1896 [it ep attributes t](#), [it ep ud create\(\)](#), [it ep query\(\)](#), [it ep modify\(\)](#), [it ep free\(\)](#), [it ep accept\(\)](#),
 1897 [it cm msg events\(\)](#), [it dto events\(\)](#), [it ia info t](#)

it_ep_reset()

1898

1899 **NAME**1900 `it_ep_reset` – reset a Reliable Connected Endpoint to the initial state1901 **SYNOPSIS**

```

1902 #include <it_api.h>
1903
1904 it_status_t it_ep_reset(
1905     IN          it_ep_handle_t          ep_handle
1906 );

```

1907 **DESCRIPTION**1908 `ep_handle` Reliable Connected Endpoint.

1909 `it_ep_reset` resets a Reliable Connected Endpoint into the `IT_EP_STATE_UNCONNECTED`
 1910 state it had at original creation while maintaining the other attributes of the Endpoint in their
 1911 current settings. `it_ep_reset` may only be applied to Reliable Connected Endpoints in the
 1912 `IT_EP_STATE_NONOPERATIONAL` state. An Endpoint in the `IT_EP_STATE_`
 1913 `NONOPERATIONAL` due to overflow of a DTO completion EVD can not be reset.

1914 Upon return of this operation any Completions for the Endpoint not yet harvested by Consumer
 1915 may be dropped or not delivered to the EVD(s) associated with the Request or Receive Queue
 1916 for the Endpoint. This operation is only needed if Consumers would like to reuse the Endpoint.
 1917 Otherwise they can just free the Endpoint using [it_ep_free](#).

1918 **RETURN VALUE**1919 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below:

1920	<code>IT_ERR_INVALID_EP</code>	The Endpoint Handle (<code>ep_handle</code>) was invalid
1921	<code>IT_ERR_INVALID_EP_STATE</code>	The Endpoint was not in the proper state for the attempted operation.
1922		
1923	<code>IT_ERR_INVALID_EP_TYPE</code>	The attempted operation was invalid for the Service Type of the Endpoint.
1924		
1925	<code>IT_ERR_CANNOT_RESET</code>	The Endpoint could not be reset due to an overflow of one of its Data Transfer Operation Event Stream Event Dispatchers.
1926		
1927		
1928	<code>IT_ERR_IA_CATASTROPHE</code>	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
1929		
1930		
1931		

1932 **ERRORS**

1933 None.

1934 **SEE ALSO**1935 [it_ep_rc_create\(\)](#), [it_ep_disconnect\(\)](#), [it_ep_free\(\)](#), [it_ia_info_t](#)

it_ep_ud_create()

1936

1937 **NAME**

1938

ep_ud_create – create an Endpoint for Unreliable Datagram

1939 **SYNOPSIS**

1940

```
#include <it_api.h>
```

1941

1942

```
it_status_t it_ep_ud_create (
```

1943

IN

it_pz_handle_t

pz_handle,

1944

IN

it_evd_handle_t

request_sevd_handle,

1945

IN

it_evd_handle_t

recv_sevd_handle,

1946

IN

const

it_ep_attributes_t

*ep_attr,

1947

IN

size_t

spigot_id,

1948

OUT

it_ep_handle_t

*ep_handle

1949

1950

```
);
```

1951 **DESCRIPTION**

1952

pz_handle

1953

Handle for the Protection Zone of the created Endpoint. Implicitly identifies the Interface Adapter.

1954

request_sevd_handle:

1955

1956

Handle for the IT_DTO_EVENT_STREAM Simple Event Dispatcher for DTO request Completion Events of the created Endpoint.

1957

recv_sevd_handle:

1958

1959

Handle for the IT_DTO_EVENT_STREAM Simple Event Dispatcher for DTO Receive Completion Events of the created Endpoint.

1960

ep_attr

1961

Pointer to a structure that contains Consumer-requested Endpoint Attributes.

1962

spigot_id

Interface Adapter Spigot identifier to use when creating Endpoint.

1963

ep_handle

Handle for the created Endpoint.

1964

1965

it_ep_ud_create creates, on the requested *spigot_id* of the Interface Adapter implicitly identified by *pz_handle*, an Unreliable Datagram Endpoint that is provided to the Consumer as *ep_handle*. The value of *ep_handle* is only defined if the return value is IT_SUCCESS.

1968

1969

The Unreliable Datagram Endpoint is created in the IT_EP_STATE_OPERATIONAL state. See [it_ep_state_t](#) for details.

1970

1971

1972

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs. Only memory referred to by LMRs that match the Endpoint Protection Zone can be accessed by the Endpoint.

1973

1974

1975

recv_sevd_handle and *request_sevd_handle* are Event Dispatcher instances where the Consumer collects completion Notifications of DTOs. Completions of Receive DTOs are reported in the *recv_sevd_handle* Event Dispatcher, and completions of Send DTOs are reported in

1976		<i>request_sevd_handle</i> . It is permissible for <i>recv_sevd_handle</i> and <i>request_sevd_handle</i> to
1977		reference the same EVD. DTO Completion Events are defined in it dto events .
1978		The Consumer should not specify an SEVD in <i>recv_sevd_handle</i> or <i>request_sevd_handle</i> that is
1979		in overflowed state for use in the Endpoint creation call (see it evd create for more details on
1980		overflow). If Consumer attempts to do so the operation will fail with
1981		IT_ERR_INVALID_RECV_EVD_STATE or IT_ERR_INVALID_REQ_EVD_STATE.
1982		The <i>ep_attr</i> parameter specifies the Consumer-requested attributes of the created Endpoint. The
1983		Implementation is required to satisfy all requested attributes or fail the operation. Hence, the
1984		Implementation must allocate all necessary resources to satisfy Consumer-requested attributes.
1985		The Implementation is allowed to allocate more resources than Consumer requested in <i>ep_attr</i> .
1986		Consumer can find the actual allocated resources by using it ep query . For detailed Endpoint
1987		attributes see man page for it ep attributes t .
1988	RETURN VALUE	
1989		A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below:
1990	IT_ERR_INVALID_PZ	The Protection Zone Handle (<i>pz_handle</i>) was
1991		invalid.
1992	IT_ERR_INVALID_REQ_EVD	The Simple Event Dispatcher Handle for Data
1993		Transfer Operation request completions
1994		(<i>request_sevd_handle</i>) was invalid.
1995	IT_ERR_INVALID_RECV_EVD	The Simple Event Dispatcher Handle for Data
1996		Transfer Operation Receive completions
1997		(<i>recv_sevd_handle</i>) was invalid.
1998	IT_ERR_INVALID_EVD_TYPE -	The Event Stream Type for the Event
1999		Dispatcher was invalid.
2000	IT_ERR_INVALID_REQ_EVD_STATE	The Simple Event Dispatcher for Data Transfer
2001		Operation request completions was in an
2002		unusable state.
2003	IT_ERR_INVALID_RECV_EVD_STATE	The Simple Event Dispatcher for Data Transfer
2004		Operation Receive completions was in an
2005		unusable state.
2006	IT_ERR_INVALID_SPIGOT	An invalid Spigot ID was specified.
2007	IT_ERR_RESOURCES	The requested operation failed due to
2008		insufficient resources.
2009	IT_ERR_PAYLOAD_SIZE	The requested <i>max_dto_payload_size</i> exceeds
2010		the maximum payload size supported by the
2011		underlying transport.
2012	IT_ERR_RESOURCE_REQ_DTO	The underlying transport could not allocate the
2013		requested <i>max_req_dtos</i> resources at this time.

2014	IT_ERR_RESOURCE_RECV_DTO	The underlying transport could not allocate the requested <i>max_rcv_dtos</i> resources at this time.
2015		
2016	IT_ERR_RESOURCE_SSEG	The underlying transport could not allocate the requested <i>max_send_segments</i> resources at this time.
2017		
2018		
2019	IT_ERR_RESOURCE_RSEG	The underlying transport could not allocate the requested <i>max_rcv_segments</i> resources at this time.
2020		
2021		
2022	IT_ERR_INVALID_EP_KEY	Invalid Endpoint Key value. The Consumer doesn't have local permissions to use the specified Endpoint Key.
2023		
2024		
2025	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it ia info t for a description of the disabled state.
2026		
2027		
2028		
2029		
2030	ERRORS	
2031	None.	
2032	SEE ALSO	
2033		it ep attributes t , it ep rc create() , it ep query() , it ep modify() , it ep free() , it ep state t ,
2034		it dto events , it ia info t

it_evd_create()

2035

2036 **NAME**

2037

it_evd_create – create Simple or Aggregate Event Dispatcher

2038 **SYNOPSIS**

2039

```
#include <it_api.h>
```

2040

2041

```

it_status_t    evd_create (
    IN          it_ia_handle_t          ia_handle,
    IN          it_event_type_t        event_number,
    IN          it_evd_flags_t         evd_flag,
    IN          size_t                  sevd_queue_size,
    IN          size_t                  sevd_threshold,
    IN          it_evd_handle_t        aevd_handle,
    OUT         it_evd_handle_t        *evd_handle,
    OUT         int                     *fd
);

```

2050

2051

2052

2053

```
#define IT_THRESHOLD_DISABLE 0
```

2054

2055

```

typedef enum {
    IT_EVD_DEQUEUE_NOTIFICATIONS = 0x01,
    IT_EVD_CREATE_FD             = 0x02,
    IT_EVD_OVERFLOW_DEFAULT      = 0x04,
    IT_EVD_OVERFLOW_NOTIFY       = 0x08,
    IT_EVD_OVERFLOW_AUTO_RESET   = 0x10
} it_evd_flags_t;

```

2056

2057

2058

2059

2060

2061

2062 **DESCRIPTION**

2063

ia_handle

Handle for the Interface Adapter to which created Event Dispatcher belongs.

2064

2065

event_number

Identifier for Event Stream type that can be enqueued to the EVD.

2066

evd_flag

Logical OR of flag values for creation operation.

2067

sevd_queue_size

Minimum size of the Simple EVD Event queue. This parameter is ignored for Aggregate EVD.

2068

2069

*sevd_threshold*Number of Events on the Simple EVD queue required for Notification of the associated AEVD or *fd* and for SEVD waiters unblocking. This parameter is ignored for Aggregate EVD.

2070

2071

2072

*aevd_handle*Optional Handle to associate an Aggregate EVD with the Simple EVD. This parameter must be IT_NULL_HANDLE when IT_EVD_CREATE_FD *evd_flag* is set. This parameter must also be IT_NULL_HANDLE when using *it_evd_create* to create an Aggregate EVD.

2073

2074

2075

2076

2077 *evd_handle* Handle for the created Event Dispatcher.

2078 *fd* Pointer to optional *file descriptor* corresponding to Event
2079 Dispatcher. Only valid if return value is IT_SUCCESS and
2080 IT_EVD_CREATE_FD *evd_flag* was set.

2081 *it_evd_create* creates an instance of an Event Dispatcher (EVD) that is provided to the
2082 Consumer as *evd_handle*. Two different types of EVDs are supported by the Implementation:
2083 Simple EVDs (SEVD) and Aggregate EVDs (AEVD). An SEVD is an EVD for a single Event
2084 Stream. An AEVD is an aggregation of SEVDs and thus can potentially return Events for more
2085 than one Event Stream type. *it_evd_create* can also optionally return a file descriptor (*fd*)
2086 associated with an EVD.

2087 The values of *evd_handle* and *fd* are only defined if the return value is IT_SUCCESS.

2088 The scope of an EVD is a single Interface Adapter identified by *ia_handle*.

2089 *event_number* identifies the type of Event Stream that the created EVD will handle. Multiple
2090 Event Streams of the same Event Stream type (such as DTO Completion Event Streams) can
2091 feed the EVD. Event Stream types are defined in [it_event t](#).

2092 To create an Aggregate EVD, the *event_number* must be set to IT_AEVD_
2093 NOTIFICATION_EVENT_STREAM; a Simple EVD (SEVD) is created otherwise. To create a
2094 Simple EVD the *event_number* can be any one of IT_DTO_EVENT_STREAM,
2095 IT_CM_REQ_EVENT_STREAM, IT_CM_MSG_EVENT_STREAM, IT_ASYNC_AFF_
2096 EVENT_STREAM, IT_ASYNC_UNAFF_EVENT_STREAM, or IT_SOFTWARE_EVENT_
2097 STREAM.

2098 A Simple EVD may feed only one Aggregate EVD. An Aggregate EVD may be fed by many
2099 Simple EVDs. The Consumer may create multiple AEVDs and SEVDs with the following two
2100 exceptions:

2101 Only one IT_ASYNC_AFF_EVENT_STREAM Simple EVD may be created per Interface
2102 Adapter instance. Subsequent calls to *it_evd_create* for the IT_ASYNC_AFF_EVENT_
2103 STREAM Event Stream, without intervening calls to [it_evd_free](#) the EVD, will fail with the
2104 error return IT_ERR_ASYNC_AFF_EVD_EXISTS.

2105 Only one IT_ASYNC_UNAFF_EVENT_STREAM Simple EVD may be created per Interface
2106 Adapter instance. Subsequent calls to *it_evd_create* for the IT_ASYNC_UNAFF_
2107 EVENT_STREAM Event Stream, without intervening calls to [it_evd_free](#) the EVD, will fail
2108 with the error return IT_ERR_ASYNC_UNAFF_EVD_EXISTS.

2109 For all Event Stream types except IT_SOFTWARE_EVENT_STREAM, IT_ASYNC_
2110 AFF_EVENT_STREAM, and IT_ASYNC_UNAFF_EVENT_STREAM, upon creation there is
2111 no Event Stream of *event_number* feeding Events to the created EVD. For an Aggregate EVD
2112 this means that there are no Simple EVDs associated with *evd_handle*. No Events are fed to
2113 *evd_handle* until *evd_handle* is associated with an object that feeds Events to it. For Aggregate
2114 EVD this means that no Events are fed to *evd_handle* until *evd_handle* is associated with a
2115 Simple EVD. For Simple EVD this means that no Events are fed to *evd_handle* until *evd_handle*
2116 is associated with an Endpoint, Listen Handle, or UD Service Request Handle depending on the
2117 stream type.

2118 For IT_ASYNC_AFF_EVENT_STREAM Event Stream type, the Simple EVD receives the
 2119 Async Affiliated Events for the *ia_handle*. For IT_ASYNC_UNAFF_EVENT_STREAM Event
 2120 Stream type, the Simple EVD receives the Async Unaffiliated Events for the *ia_handle*.

2121 Multiple Event Streams of the same Event Stream type can be associated with the same EVD,
 2122 with exception of IT_ASYNC_AFF_EVENT_STREAM, IT_ASYNC_UNAFF_EVENT_
 2123 STREAM and IT_SOFTWARE_EVENT_STREAM Event Stream types. For IT_AEVD_
 2124 NOTIFICATION_EVENT_STREAM Event Stream type multiple SEVDs can be associated
 2125 with the same AEVD. For IT_DTO_EVENT_STREAM multiple EPs can be associated with the
 2126 same SEVD. For IT_CM_REQ_EVENT_STREAM multiple Listens can be associated with the
 2127 same SEVD. For IT_CM_MSG_EVENT_STREAM multiple RC EPs and/or UD Service
 2128 Requests can be associated with the same SEVD. For IT_ASYNC_AFF_EVENT_STREAM,
 2129 IT_ASYNC_UNAFF_EVENT_STREAM, and IT_SOFTWARE_EVENT_STREAM Event
 2130 Stream types, only a single Event Stream feeds each EVD, respectively, and the corresponding
 2131 Event Stream is created upon EVD creation. For IT_SOFTWARE_EVENT_STREAM, the
 2132 Events are generated explicitly by the Consumer calling [it_evd_post_se](#).

2133 When the Implementation attempts to enqueue more Events on an SEVD than the queue size of
 2134 the SEVD will permit, the SEVD is said to overflow. An AEVD can not overflow.

2135 Once a SEVD overflows, subsequent Events from the Event Stream will be dropped. For all
 2136 Event Streams, with the exception of the IT_DTO_EVENT_STREAM, Events will no longer be
 2137 dropped once the Consumer makes more space available in the SEVD's Event queue. The
 2138 Consumer can make room in a SEVD either by dequeuing an Event, or by using [it_evd_modify](#)
 2139 to increase the queue size of the SEVD. For the IT_DTO_EVENT_STREAM, however, Events
 2140 will continue to be dropped; the overflow can not be corrected.

2141 The behavior of a SEVD after an overflow depends upon the Event Stream associated with the
 2142 SEVD, and upon whether the default overflow behavior has been configured for the SEVD. The
 2143 man page associated with each Event Stream type provides details of the default overflow
 2144 behavior. The Consumer specifies they desire default overflow behavior by setting the
 2145 IT_EVD_OVERFLOW_DEFAULT *evd_flag* value.

2146 If default overflow behavior is not configured (IT_EVD_OVERFLOW_DEFAULT is cleared in
 2147 *evd_flag*), then the Consumer can control two possible parameters: Whether the overflow
 2148 occurrence causes generation (IT_EVD_OVERFLOW_NOTIFY flag value) of an overflow
 2149 Event on the Affiliated or Unaffiliated SEVD, and, if configured, how the generation of the
 2150 overflow Event is controlled (IT_EVD_OVERFLOW_AUTO_RESET flag value). Each
 2151 subsequent SEVD Event that arrives after overflow of the SEVD initially occurs can potentially
 2152 generate an overflow Event.

2153 The Consumer can request that an overflow Event be generated when an overflow occurs by
 2154 setting IT_EVD_OVERFLOW_NOTIFY in *evd_flag*. For SEVDs associated with any Event
 2155 Streams other than the IT_ASYNC_AFF_EVENT_STREAM or the IT_ASYNC_UNAFF_
 2156 EVENT_STREAM, an IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE Event is enqueued on
 2157 the affiliated asynchronous error Event Stream of *ia_handle*. For a SEVD associated with the
 2158 IT_ASYNC_AFF_EVENT_STREAM, an IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE
 2159 Event is enqueued on the unaffiliated asynchronous error Event Stream of *ia_handle*. The Event
 2160 identifies the SEVD that overflowed. Overflow of the IT_ASYNC_UNAFF_EVENT_
 2161 STREAM is never detected and no indication of such overflow is ever generated; however, no
 2162 adverse consequences occur other than the dropping of some Unaffiliated Events.

2163 If a SEVD overflow has occurred, the *evd_overflowed* member of the [it_evd_param_t](#) structure
 2164 (as returned by the [it_evd_query](#) routine) will have an IT_TRUE value until the condition is
 2165 corrected or manually changed. When the Consumer creates a SEVD to hold Events of an Event
 2166 Stream and has enabled generation of overflow Events on the SEVD
 2167 (IT_EVD_OVERFLOW_NOTIFY flag value), the Consumer must chose one of two modes for
 2168 generation of overflow Events using the IT_EVD_OVERFLOW_AUTO_RESET flag:
 2169 automatic, or Consumer-controlled. In automatic mode, overflow Events may again be
 2170 enqueued on the Affiliated or Unaffiliated SEVD as soon as the Consumer makes more space
 2171 available in the EVD's Event queue. In Consumer-controlled generation, overflow Events are
 2172 only again generated after the Consumer calls [it_evd_modify](#) to clear the *evd_overflowed* field.
 2173 See [it_evd_modify](#) for more details.

2174 Note that even if overflow generation is disabled, the Consumer may still clear *evd_overflowed*
 2175 using [it_evd_modify](#) if they so choose. A subsequent overflow will again set the *evd_overflowed*
 2176 member of the [it_evd_param_t](#) structure.

2177 For a newly created SEVD, the *evd_overflowed* member of the [it_evd_param_t](#) structure is not
 2178 set.

2179 The *evd_flag* value of IT_EVD_DEQUEUE_NOTIFICATIONS applies only to AEVDs.

2180 When the IT_EVD_DEQUEUE_NOTIFICATIONS bit is set in *evd_flag*, then wait and dequeue
 2181 operations on the AEVD will dequeue IT_AEVD_NOTIFICATION_EVENT_STREAM
 2182 Events; such Events provide the SEVD Handle of the underlying SEVD that caused the
 2183 Notification. To retrieve the underlying Event, the Consumer must call [it_evd_dequeue](#) on the
 2184 SEVD Handle provided in the IT_AEVD_NOTIFICATION_EVENT_STREAM Event from the
 2185 AEVD.

2186 When the IT_EVD_DEQUEUE_NOTIFICATIONS bit is cleared in *evd_flag*, then calling
 2187 [it_evd_wait](#) on the AEVD directly returns the first Event from a notifying underlying SEVD
 2188 (such as IT_DTO_EVENT_STREAM Events, etc.). The dequeue operation on the AEVD
 2189 directly returns the first Event from an underlying SEVD. These Events will be of whatever
 2190 Event Stream types that feed each of these associated SEVDs. The associated SEVD can be
 2191 determined from the *evd_handle* found in every Event.

2192 If an underlying SEVD of an AEVD has been disabled then the SEVD will no longer generate
 2193 Notification Events for the AEVD until the SEVD is enabled (see [it_evd_modify](#)). Previously
 2194 generated SEVD Notifications for the AEVD are unaffected by the enabling and disabling of
 2195 SEVD.

2196 For a Simple EVD that does not have an associated AEVD, the Consumer can wait on and
 2197 dequeue from the SEVD.

2198 If the SEVD has an associated AEVD with the IT_EVD_DEQUEUE_NOTIFICATIONS
 2199 *evd_flag* cleared, then it is an error for the Consumer to wait on or dequeue from the SEVD.
 2200 Attempting to wait on or dequeue from the SEVD will return IT_ERR_INVALID_EVD_
 2201 STATE.

2202 If the SEVD has an associated AEVD with the IT_EVD_DEQUEUE_NOTIFICATIONS
 2203 *evd_flag* set, then the Consumer can always dequeue from the SEVD, and the Consumer can
 2204 wait on the SEVD but only if they disable the SEVD first (see [it_evd_modify](#)). Attempting to
 2205 wait on the SEVD when disallowed will return IT_ERR_INVALID_EVD_STATE.

2206 The *evd_flag* bit value of IT_EVD_CREATE_FD set indicates that the Consumer requests
 2207 creation of a File Descriptor associated with the EVD (either SEVD or AEVD). If the EVD has
 2208 an associated *fd*, then the Consumer can wait on the EVD if they disable the EVD first (see
 2209 [it_evd_modify](#)). Attempting to wait on the EVD when disallowed will return
 2210 IT_ERR_INVALID_EVD_STATE. If the EVD has an associated *fd*, then Consumer can
 2211 dequeue from the feeding EVD.

2212 Values for *evd_flag* are constructed by a bitwise-inclusive OR of flags from the following list,
 2213 defined in <it_api.h>.

Flag Value	Description
IT_EVD_DEQUEUE_NOTIFICATIONS	Only applicable to AEVD. When set, wait and dequeue on the AEVD shall dequeue IT_AEVD_NOTIFICATION_EVENT_STREAM Events from the created AEVD. Otherwise, wait and dequeue on the AEVD will dequeue the underlying Events (of potentially various Event Stream types) from the SEVDs that feed the AEVD.
IT_EVD_CREATE_FD	Implementation will allocate and return a file descriptor usable as a Notification object for this EVD. It is an error to set this flag as well as specify an AEVD for SEVD.
IT_EVD_OVERFLOW_DEFAULT	Only applicable to an SEVD. When set, the overflow behavior for the SEVD will be the default behavior for the <i>event_number</i> as specified in the man page for the Event Stream. When clear, the behavior is determined by how the IT_EVD_OVERFLOW_NOTIFY and IT_EVD_OVERFLOW_AUTO_RESET flags are set. It is an error to set this flag as well as IT_EVD_OVERFLOW_NOTIFY and/or IT_EVD_OVERFLOW_AUTO_RESET.
IT_EVD_OVERFLOW_NOTIFY	Only applicable to an SEVD. When clear, EVD overflow is ignored. When set, causes an IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE Event to be generated if the EVD overflows if <i>event_number</i> is anything other than IT_ASYNC_AFF_EVENT_STREAM or IT_ASYNC_UNAFF_EVENT_STREAM. Causes an IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE Event to be generated if the EVD overflows and <i>event_number</i> is IT_ASYNC_AFF_EVENT_STREAM. It is invalid to set this flag if <i>event_number</i> is IT_ASYNC_UNAFF_EVENT_STREAM. It is invalid to set both this flag and IT_EVD_OVERFLOW_DEFAULT.

IT_EVD_OVERFLOW_AUTO_RESET	<p>Only applicable to an SEVD. When set, this flag specifies that the Implementation will automatically reset overflow Event generation (i.e. when Consumer make space available, further Events that again overflow EVD will cause another overflow Event to attempt to be queued to the Affiliated or Unaffiliated SEVD); when clear, this flag specifies that Consumer must manually reset the <i>evd_overflowed</i> state of the EVD (see it_evd_modify) and Implementation shall not reset EVD overflow Event generation on its own. It is invalid to set both this flag and IT_EVD_OVERFLOW_DEFAULT. If IT_EVD_OVERFLOW_NOTIFY is not set, it is an error to set this flag. Since a DTO overflow can not be corrected, it is an error to set this flag for the DTO Event Stream.</p>
----------------------------	--

2214

2215

2216

2217

2218

2219

sevd_queue_size is only applicable for a Simple EVD. It defines the size of the Event queue that the Consumer requested. The Implementation is required to provide a queue size of at least *sevd_queue_size*, but is free to provide a larger queue size. The Consumer can determine the actual queue size by querying the created Simple Event Dispatcher. This parameter is ignored for Aggregate EVD.

2220

2221

2222

2223

2224

2225

2226

2227

2228

2229

2230

2231

The *sevd_threshold* is only applicable to an SEVD and allows the Consumer to request an accumulation of up to *sevd_threshold* number of enqueued “non-Notification Events” for the Simple EVD queue prior to waking up the Consumer or notifying *fd* or *aevd_handle*. A “non-Notification Event” is one of the following: An Event with *dto_status* of IT_DTO_SUCCESS corresponding to a non-Recv DTO that was posted with the IT_NOTIFY_FLAG bit cleared. An Event with *dto_status* of IT_DTO_SUCCESS corresponding to a Recv DTO that was posted with the IT_NOTIFY_FLAG bit cleared and with the IT_SOLICITED_WAIT bit cleared in the corresponding remote Send. See [it_dto_flags_t](#) for more details. Only DTO Event Streams support non-notification Events; on all other Event Streams, every Event is a Notification Event (thus thresholds have no function on non-DTO Event Streams). Arrival of a “Notification Event” before *sevd_threshold* number of non-notification Events have arrived will cause wakeup or Notification.

2232

A “Notification Event” is one of the following:

2233

An Event corresponding to a DTO that was posted with the IT_NOTIFY_FLAG bit set.

2234

An Event with a *dto_status* that is not IT_DTO_SUCCESS.

2235

2236

An Event corresponding to a Recv DTO with the IT_SOLICITED_WAIT bit set in the corresponding remote Send.

2237

Any Event of Event Stream other than IT_DTO_EVENT_STREAM.

2238

2239

2240

An SEVD is in the “notification criteria” when one of the following is true: There is a Notification Event queued on the SEVD. The number of Events on SEVD is larger or equal to the *sevd_threshold*.

2241 For SEVD the *sevd_threshold* must be set to either the value `IT_THRESHOLD_DISABLE` or to
 2242 a value greater than or equal to one. Setting *sevd_threshold* to `IT_THRESHOLD_DISABLE`
 2243 will cause [it evd wait](#) to return only for Notification Events (specifically not for a threshold
 2244 number of Events). For AEVD the *sevd_threshold* is ignored.

2245 An *aevd_handle* specified on creation of Simple EVD allows a Consumer to consolidate
 2246 Notifications from multiple Simple Event Dispatchers (from the same Interface Adapter) to a
 2247 single higher-level Aggregate Event Dispatcher. For SEVD the *aevd_handle* value of
 2248 `IT_NULL_HANDLE` means that no AEVD is associated with nor fed by the created SEVD. For
 2249 Aggregate EVD creation this parameter must be `IT_NULL_HANDLE`; otherwise *it_evd_create*
 2250 will return `IT_ERR_AEVD_NOT_ALLOWED`.

2251 Alternatively, if the `IT_EVD_CREATE_FD evd_flag` bit value is set, then the Implementation
 2252 will return a new unique file descriptor associated with the EVD. The file descriptor is placed
 2253 into the contents of the *fd* pointer. The *fd* may be used in *select()* or *poll()* system calls and will
 2254 be identified as ready to read when a Notification occurs on the underlying EVD. It is up to a
 2255 Consumer then to go and dequeue Events from the EVD which is one-to-one associated with the
 2256 particular *fd*. It is the Consumer's responsibility to keep track of the one-to-one association of *fd*
 2257 and *EVD*.

2258 For Simple EVD the use of a value other than `IT_NULL_HANDLE` for *aevd_handle* is mutually
 2259 exclusive with use of the `IT_EVD_CREATE_FD evd_flag`; specifying both an *aevd_handle* not
 2260 equal to `IT_NULL_HANDLE` and the `IT_EVD_CREATE_FD evd_flag` in a call to
 2261 [it_evd_create](#) will fail and return value of `IT_ERR_MISMATCH_FD`.

2262 IT-API supports the following configurations: Simple EVD, Simple EVD with associated *fd*,
 2263 Simple EVD feeding Aggregate EVD, and Simple EVD feeding Aggregate EVD that is
 2264 associated with *fd*.

2265 The Aggregate EVD specified by *aevd_handle* or the *fd* will be notified by the Implementation
 2266 when a Notification Event arrives or *sevd_threshold* value is reached when EVD is enabled.

2267 When a SEVD feeds an AEVD or *fd*, control of the capability of the feeding EVD to notify the
 2268 fed AEVD or *fd* is done by enabling or disabling the feeding SEVD (see [it_evd_modify](#)).

2269 By default the created EVD is enabled. An enabled SEVD will cause *aevd_handle* or *fd* (if
 2270 applicable) to be notified when an Event arrival causes Notification criteria to be reached on that
 2271 SEVD. An enabled AEVD will cause *fd* (if applicable) to be notified when an Event arrival
 2272 causes Notification criteria to be reached. Notification is done on *aevd_handle* by generating an
 2273 `IT_AEVD_NOTIFICATION_EVENT` for the AEVD if the `IT_EVD_DEQUEUE_`
 2274 `NOTIFICATIONS evd_flag` bit set on AEVD creation. When Notification is necessary for an
 2275 AEVD with the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit cleared, no `IT_AEVD_`
 2276 `NOTIFICATION_EVENT` will be enqueued; rather, the AEVD Consumer will be unblocked
 2277 with the underlying SEVD Event delivered to it. Notification is done on the *fd* by marking it as
 2278 ready to read.

2279 Consumers can not wait on an enabled SEVD that feeds an AEVD or *fd*.

2280 A disabled feeding EVD will not generate Notification to the fed AEVD or *fd*. Consumers can
 2281 wait on a disabled SEVD, unless it is associated with an AEVD with the `IT_EVD_`
 2282 `DEQUEUE_NOTIFICATIONS evd_flag` bit cleared.

2283 An SEVD preserves the order of Events within each individual Event Stream as provided by the
 2284 underlying Transport. No order is defined between Events of different Event Streams, even

2285 when they are of the same Event Stream type. For IT_DTO_EVENT_STREAM Event Stream
 2286 type, the order of the Event completions is defined for each DTO and RMR post operation on the
 2287 Endpoint. No order is defined between Events of Event Streams coming from different SEVDs
 2288 for an AEVD with IT_EVD_DEQUEUE_NOTIFICATIONS *evd_flag* bit cleared. The order of
 2289 Events of IT_AEVD_NOTIFICATION_EVENT Event Stream is Implementation dependent.

2290 If the IT_EVD_DEQUEUE_NOTIFICATIONS bit is cleared in *evd_flag* on AEVD creation, the
 2291 Consumer, when blocked in *it_evd_wait* and an SEVD Notification occurs, is unblocked and
 2292 dequeues an lower-level Event from the same SEVD that caused Notification.

2293 Multiple SEVDs can feed the same AEVD. An SEVD generates a Notification for AEVD when
 2294 an SEVD arriving Event causes SEVD to reach Notification criteria if SEVD is enabled.

2295 SEVD and AEVD can support multiple waiters. For SEVD the *sevd_threshold* value must be 1
 2296 for multiple waiters to be supported.

2297 An SEVD waiter will block when SEVD queue is empty. An AEVD waiter will block when all
 2298 associated SEVDs are empty. An SEVD waiter may block when SEVD is not in the Notification
 2299 criteria. An AEVD waiter may block when all associated SEVDs are not in the Notification
 2300 criteria. An SEVD waiter will return if there is a Notification Event on the queue or if the
 2301 number of Events on the SEVD is equal or larger then *threshold*. An AEVD waiter will return if
 2302 there is a Notification Event on any of the associated SEVDs or any of the associated SEVDs
 2303 has number of Events larger or equal to its *sevd_threshold*.

2304 If arriving Event causes SEVD to reach Notification criteria then SEVD waiter will be
 2305 unblocked if one exists and the SEVD is disabled and not associated with AEVD with
 2306 IT_EVD_DEQUEUE_NOTIFICATIONS *evd_flag* bit cleared. As many waiters as there are
 2307 Events available on SEVD may be unblocked. If an arriving Notification Event causes SEVD to
 2308 reach Notification criteria and the SEVD is enabled then Notification will be generated for the
 2309 associated AEVD or *fd*. As many Notifications can be generated as there are Events available on
 2310 all SEVDs associated with the AEVD.

2311 [*it_evd_dequeue*](#) from SEVD will return an Event, if one exists, from SEVD queue regardless if
 2312 there are waiters except when the SEVD is associated with AEVD with IT_EVD_
 2313 DEQUEUE_NOTIFICATIONS *evd_flag* bit cleared. In the latter case dequeue from the SEVD
 2314 is not allowed. For AEVD with IT_EVD_DEQUEUE_NOTIFICATIONS *evd_flag* bit cleared,
 2315 dequeue from the AEVD will return an Event, if one exists, from any of its associated SEVDs.
 2316 For AEVD with IT_EVD_DEQUEUE_NOTIFICATIONS *evd_flag* bit set, dequeue from the
 2317 AEVD will return an IT_AEVD_NOTIFICATION_EVENT if any of the associated enabled
 2318 SEVDs is in Notification criteria or may return an IT_AEVD_NOTIFICATION_
 2319 EVENT if any of the associated enabled SEVD simply has an Event.

2320 RETURN VALUE

2321 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

2322	IT_ERR_INVALID_IA	The Interface Adapter Handle (<i>ia_handle</i>) was
2323		invalid.
2324	IT_ERR_INVALID_EVD_TYPE	The Event Stream Type for the Event Dispatcher
2325		was invalid.
2326	IT_ERR_INVALID_FLAGS	The flags value was invalid.

2327 2328	IT_ERR_RESOURCE_QUEUE_SIZE	The underlying transport could not allocate the requested <code>sevd_queue_size</code> resources at this time.
2329 2330	IT_ERR_INVALID_THRESHOLD	An invalid value for the Simple Event Dispatcher threshold was specified.
2331 2332	IT_ERR_INVALID_AEVD	The Aggregation Event Dispatcher Handle (<i>aevd_handle</i>) was invalid.
2333 2334	IT_ERR_RESOURCES	The requested operation failed due to insufficient resources.
2335 2336	IT_ERR_MISMATCH_FD	An illegal request was made for both the File Descriptor and the Aggregation Event Dispatcher.
2337 2338 2339 2340	IT_ERR_AEVD_NOT_ALLOWED	The <i>aevd_handle</i> was non-NULL and the <i>event_number</i> was IT_AEVD_NOTIFICATION_EVENT_STREAM .
2341 2342	IT_ERR_ASYNC_AFF_EVD_EXISTS	The Asynchronous Affiliated Event Dispatcher already exists.
2343 2344	IT_ERR_ASYNC_UNAFF_EVD_EXISTS	The Asynchronous Unaffiliated Event Dispatcher already exists.
2345 2346 2347 2348	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
2349	ERRORS	
2350	None.	
2351	APPLICATION USAGE	
2352 2353 2354		Consumers may use SEVDs with a pure polling model. Consumers create SEVDs and dequeue from them directly. The Consumer threads never wait on the SEVDs and just dequeue Events when they are ready to process them.
2355 2356		Alternatively, Consumers may create SEVDs and wait on and dequeue from them directly. This also potentially requires many waiting threads, one per SEVD.
2357 2358 2359 2360		For the “non-thread-safe” Implementation the Consumer cannot have multiple threads calling on the same EVD Handle simultaneously. When multiple threads retrieve Events concurrently from the same SEVD, each Event will be retrieved exactly once but it is unpredictable which thread will retrieve any particular Event.
2361		
2362 2363		The use of an AEVD can reduce the number of distinct waiting threads required for an application. EVDs must be enabled to generate Notifications for the AEVD.

2364 Consumers can wait on an AEVD that had been created with the
 2365 IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag bit set and all feeding SEVDs enabled.
 2366 When wait returns, a returned IT_AEVD_NOTIFICATION_EVENT_STREAM Event identifies
 2367 the SEVD that caused the unblocking. Consumer can then dequeue Events directly from that
 2368 SEVD or any other SEVD that feeds the AEVD. Thus, the Consumer can choose to service the
 2369 SEVDs feeding the AEVD in any order they wish.

2370 Consumers can wait on an AEVD that had been created with the
 2371 IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag bit cleared and all feeding SEVDs enabled.
 2372 When wait returns, it provides the first Event from an SEVD that is in Notification status.
 2373 Consumer can then dequeue Events only from the AEVD. This dequeuing will provide Events
 2374 from all SEVDs that feed the AEVD.

2375 The order of returned Events from the AEVD is implementation-dependent. If Events are
 2376 retrieved from a given AEVD strictly by a single thread, the order of each Event from its
 2377 underlying SEVDs is maintained, but the order in which SEVDs are selected by the AEVD is
 2378 implementation-dependent. If Events are retrieved from a given AEVD by more than one thread,
 2379 no order guarantees are made.

2380 The use of a file descriptor can also reduce the number of distinct waiting threads. File
 2381 descriptors also can be used to wait for Notification Events across multiple Interface Adapters or
 2382 Events not generated by this API. EVDs must be enabled to generate Notifications for the file
 2383 descriptor.

2384 Consumers can *select* or *poll* on multiple *fds* that are associated with EVDs. The return for the
 2385 *select* or *poll* call identifies the notifying *fd*. It is the Consumer's responsibility to keep track of
 2386 which EVD is associated with each *fd*. Consumer can dequeue Events from the EVD one-to-one
 2387 associated with that *fd* using [it evd dequeue](#).

2388 Typically, if the Consumer chooses to use an AEVD, they are then prohibited from waiting on
 2389 the underlying SEVDs (see DESCRIPTION section above for exceptions) and also may be
 2390 prohibited from dequeuing from the underlying SEVDs (again see DESCRIPTION section
 2391 above for details). If the Consumer chooses to use an *fd*, then they are prohibited from waiting
 2392 on the underlying AEVD(s) or SEVD(s).

2393 Overflow may occur and may not be reported to Consumer via Events if there is no Simple EVD
 2394 for IT_ASYNC_AFF_EVENT_STREAM or for IT_ASYNC_UNAFF_EVENT_STREAM.
 2395 Additionally, an IA can enter catastrophic state and not notify Consumer about it if there is no
 2396 Simple EVD for IT_ASYNC_UNAFF_EVENT_STREAM or if it has overflowed. For the effect
 2397 of catastrophic error see [it unaffiliated event t](#) and [it ia create](#).

2398 When an IA supports Spigot *online* or *offline* Events the number of Events that can be generated
 2399 for the Unaffiliated Asynchronous Event Stream is potentially unbounded, but the queuing
 2400 capacity of an EVD is finite. This can potentially lead to Events that are generated for the
 2401 Unaffiliated Asynchronous Event Stream being silently discarded by the
 2402 Implementation. Events that are generated for the Affiliated (or Unaffiliated) Asynchronous
 2403 Event Stream will be silently discarded by the Implementation until such time as an EVD is
 2404 created to hold the Affiliated (or Unaffiliated) asynchronous Event Stream. If the Consumer
 2405 needs to know with certainty the state of an entity that can generate an Unaffiliated
 2406 Asynchronous Event (e.g. a Spigot), it should query for that state itself rather than relying upon
 2407 getting a state change Notification via the Unaffiliated Asynchronous Event Stream.

2408 IT_ASYNC_AFF_EVENT_STREAM and IT_ASYNC_UNAFF_EVENT_STREAM SEVDs
 2409 store Events that notify users of errors and other conditions that affect IA operation. These
 2410 Events are usually unpredictable, which can make determining an appropriate size for these
 2411 queues a challenge. Users should consider the size and type of the fabric, their resource usage
 2412 and their message patterns when setting the *sevd_queue_size* parameter for these EVDs.

2413 **FUTURE DIRECTIONS**

2414 IT-API support for a callback routine being invoked when an Event is enqueued on an SEVD
 2415 may be added in the future.

2416 Aggregate EVD support for multiple IAs may be added in the future.

2417 **SEE ALSO**

2418 [*it_evd_post_se\(\)*](#), [*it_ep_rc_create\(\)*](#), [*it_ep_ud_create\(\)*](#), [*it_listen_create\(\)*](#),
 2419 [*it_ud_service_request_handle_create\(\)*](#), [*it_evd_query\(\)*](#), [*it_evd_modify\(\)*](#), [*it_evd_wait\(\)*](#),
 2420 [*it_evd_dequeue\(\)*](#), [*it_evd_free\(\)*](#), [*it_event_t*](#), [*it_dto_flags_t*](#), [*it_unaffiliated_event_t*](#),
 2421 [*it_ia_create\(\)*](#), [*it_ia_info_t*](#)

it_evd_dequeue()

2422

2423 **NAME**

2424

`it_evd_dequeue` – dequeue for Events from Event Dispatcher2425 **SYNOPSIS**

2426

`#include <it_api.h>`

2427

2428

`it_status_t it_evd_dequeue(`

2429

`IN it_evd_handle_t evd_handle,`

2430

`OUT it_event_t *event`

2431

`);`2432 **DESCRIPTION**

2433

evd_handle: Handle for simple or aggregate Event Dispatcher.

2434

event: Pointer to the Consumer-allocated structure that the Implementation

2435

fills with the Event information.

2436

it_evd_dequeue removes the first Event from the Event Dispatcher Event queue and fills the Consumer-allocated *event* structure with Event information. For the Event information and *event* structure see [it_event_t](#). The Consumer should allocate an Event structure big enough to hold any Event that the Event Dispatcher can deliver.

2437

2438

2439

2440

it_evd_dequeue returns the first Event from an EVD, if one exists, regardless of whether EVD has waiters.

2441

2442

The return value for *event* is defined only if *it_evd_dequeue* returns `IT_SUCCESS`.

2443

For AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit clear, the operation dequeues the first Event from one of its associated SEVDs.

2444

2445

For AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit set, the operation returns a Notification Event of the `IT_AEVD_NOTIFICATION_EVENT` Event Stream which identifies an *evd_handle* from one of its associated SEVDs. The order in which the associated SEVD's AEVD Notification Events are delivered is implementation-dependent.

2446

2447

2448

2449

For a Simple EVD that does not have an associated AEVD, the Consumer can dequeue from the SEVD.

2450

2451

2452

If the SEVD has an associated AEVD with the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` cleared, then it is an error for the Consumer to dequeue from the SEVD.

2453

2454

If the SEVD has an associated AEVD with the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` set, then the Consumer may dequeue from the SEVD at will.

2455

2456

Attempting to dequeue from the SEVD when disallowed will return `IT_ERR_INVALID_EVD_STATE`.

2457

2458

The Consumer can always dequeue from AEVD regardless of the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` value or associated *fd*. If the EVD is empty, then *it_evd_dequeue* will return `IT_ERR_QUEUE_EMPTY`.

2459

2460

2461 For IT_DTO_EVENT_STREAM Events when a Completion Event is returned for a given Send,
 2462 RDMA Read, RDMA Write, RMR Bind or RMR Unbind operation that was posted to an
 2463 Endpoint, the Implementation guarantees that all Send, RDMA Read, RDMA Write, RMR Bind
 2464 and RMR Unbind operations that were posted to the Endpoint prior to the one whose
 2465 Completion Event was returned have also completed regardless of their *dto_flag* value for
 2466 *IT_COMPLETION_FLAG*.

2467 The SEVD *sevd_threshold* value has no effect on this operation.

2468 RETURN VALUE

2469 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

2470 IT_ERR_QUEUE_EMPTY There were no entries on the Event Dispatcher
 2471 queue.

2472 IT_ERR_INVALID_EVD The Event Dispatcher Handle (*evd_handle*) was
 2473 invalid.

2474 IT_ERR_INVALID_EVD_STATE The attempted operation was invalid for the current
 2475 state of the Event Dispatcher.

2476 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in
 2477 the disabled state. None of the output parameters
 2478 from this routine are valid. See [it_ia_info_t](#) for a
 2479 description of the disabled state.

2480 ERRORS

2481 None.

2482 APPLICATION USAGE

2483 For an AEVD with IT_EVD_DEQUEUE_NOTIFICATIONS set, receipt of an
 2484 IT_AEVD_NOTIFICATION_EVENT Event indicates that the SEVD (identified by *evd_handle*
 2485 in the Event) reached Notification status or has Events available. By the time the Consumer calls
 2486 *it_evd_dequeue* on the returned SEVD it may be empty or may not be in the Notification Criteria
 2487 any longer if there are multiple dequeuers from the SEVD.

2488 For the “non-thread-safe” Implementation Consumer cannot have multiple threads
 2489 calling *dequeue* on the same EVD Handle simultaneously.

2490 When multiple threads retrieve Events concurrently from the same SEVD, each Event will be
 2491 retrieved exactly once but it is unpredictable which thread will retrieve any particular Event.

2492 SEE ALSO

2493 [it_evd_create\(\)](#), [it_evd_wait\(\)](#), [it_event_t](#).

it_evd_free()

2494

2495 NAME2496 `it_evd_free` – destroy an Event Dispatcher**2497 SYNOPSIS**

```

2498 #include <it_api.h>
2499
2500 it_status_t it_evd_free(
2501     IN          it_evd_handle_t    evd_handle
2502 );

```

2503 DESCRIPTION2504 `evd_handle` Handle to Simple or Aggregate Event Dispatcher.2505 `it_evd_free` Destroys an Event Dispatcher.

2506 On successful completion, all Events on the queue of the specified Event Dispatcher are lost.

2507 `it_evd_free` will return `IT_ERR_EVD_BUSY` if the EVD is still associated with an active Event
 2508 Stream feeding it for all Event Streams except `IT_ASYNC_AFF_EVENT_STREAM`, `IT_`
 2509 `ASYNC_UNAFF_EVENT_STREAM`, and `IT_SOFTWARE_EVENT_STREAM`. `it_evd_free`
 2510 may be called at any time for `IT_ASYNC_AFF_EVENT_STREAM`, `IT_ASYNC_`
 2511 `UNAFF_EVENT_STREAM`, and `IT_SOFTWARE_EVENT_STREAM` Event Streams but
 2512 Events may be lost.

2513 An AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS` set may be dissociated from its
 2514 SEVDs through use of `it_evd_modify` on each SEVD or through use of `it_evd_free` on each
 2515 SEVD. An AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit clear may be
 2516 dissociated from SEVDs through use of `it_evd_free` on each SEVD. DTO SEVDs may be
 2517 disassociated from their DTO Event Streams through use of [it_ep_free](#) on each associated
 2518 Endpoint. Communication Management Request SEVDs may be disassociated from their Event
 2519 Streams through use of [it_listen_free](#) on each associated listen Handle. Communication
 2520 Management Message SEVDs may be disassociated from their Event Streams through use of
 2521 [it_ep_free](#) on each associated Endpoint.

2522 Use of the Handle `evd_handle` in any subsequent operation fails.

2523 This operation is applicable to both AEVD and SEVD Handles.

2524 RETURN VALUE2525 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

2526

2527 `IT_ERR_INVALID_EVD` The Event Dispatcher Handle (`evd_handle`) was invalid.

2528 `IT_ERR_EVD_BUSY` The Event Dispatcher was still associated with active
 2529 Event Streams.

2530 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 2531 disabled state. None of the output parameters from this

2532 routine are valid. See [it ia info t](#) for a description of the
2533 disabled state.

2534 **ERRORS**

2535 None.

2536 **SEE ALSO**

2537 [it evd create\(\)](#), [it evd modify\(\)](#), [it evd query\(\)](#), [it ep free\(\)](#), [it listen free\(\)](#).

it_evd_modify()

2538

2539 **NAME**2540 `it_evd_modify` – modify an existing Event Dispatcher2541 **SYNOPSIS**

```

2542 #include <it_api.h>
2543
2544 it_status_t it_evd_modify(
2545     IN          it_evd_handle_t          evd_handle,
2546     IN          it_evd_param_mask_t     mask,
2547     IN    const it_evd_param_t          *params
2548 );

```

2549 **DESCRIPTION**

2550 *evd_handle* Simple or Aggregate Event Dispatcher.

2551 *mask* Logical OR of flags for requested EVD parameters.

2552 *params* Pointer to Consumer-allocated structure that contains new
2553 Consumer-requested Event Dispatcher parameters.

2554 *it_evd_modify* changes the desired parameters of the Simple or Aggregate Event Dispatcher
2555 *evd_handle*. Parameters to be modified are specified by flags in *mask*. New values for the
2556 parameters are specified by the corresponding fields in the structure pointed to by *params*. Fields
2557 and their flag values are shown below. Note that parameters represented by fields of
2558 *it_evd_param_t* that are not shown below can not be modified. See [it_evd_query](#) for definition
2559 of *it_evd_param_t* and *it_evd_param_mask_t*.

```

2560 typedef struct {
2561     ...
2562     size_t          sevd_queue_size; /* IT_EVD_PARAM_QUEUE_SIZE */
2563     size_t          sevd_threshold; /* IT_EVD_PARAM_THRESHOLD */
2564     it_evd_handle_t aevd;          /* IT_EVD_PARAM_AEVD_HANDLE */
2565     ...
2566     it_boolean_t    evd_enabled;    /* IT_EVD_PARAM_ENABLED */
2567     it_boolean_t    evd_overflowed; /* IT_EVD_PARAM_OVERFLOWED */
2568 } it_evd_param_t;

```

2569

2570 The definition of each field follows:

2571 *sevd_queue_size* Minimum size of the Simple EVD Event queue. Attempting to modify
2572 this field for an AEVD will return an IT_ERR_INVALID_MASK
2573 error code.

2574 *sevd_threshold* For Simple EVD only. Number of Events on a single Event Dispatcher
2575 queue required for Notification of the associated AEVD or FD and for
2576 SEVD waiters unblocking. Attempting to modify this field for an
2577 AEVD will return an IT_ERR_INVALID_MASK error code.

2578 *aevd* For Simple EVD only. The Handle for the new associated Aggregate
2579 EVD. Attempting to modify this field for an AEVD will return an

2580		IT_ERR_INVALID_MASK error code if the above criteria are not
2581		met.
2582	<i>evd_enabled</i>	Consumer may set this it_boolean_t to the value IT_TRUE to indicate
2583		that an EVD should notify an associated AEVD or <i>fd</i> when
2584		Notification criteria are reached. Clearing <i>evd_enabled</i> (making it
2585		equal to IT_FALSE) will disable this capability. May be done at any
2586		time.
2587	<i>evd_overflowed</i>	Consumer may clear this it_boolean_t (make it equal IT_FALSE) to
2588		reset an overflow condition on the EVD. See it_evd_create for more
2589		details.
2590		AEVD can only be changed for the SEVD that is disabled, and IT_EVD_
2591		DEQUEUE_NOTIFICATIONS is set in the <i>evd_flag</i> for the current <i>aevd</i> (i.e. for the AEVD),
2592		and there is no <i>fd</i> associated with the SEVD (i.e., for the SEVD). Otherwise,
2593		IT_ERR_INVALID_EVD_STATE is returned.
2594		The new AEVD may have IT_EVD_DEQUEUE_NOTIFICATIONS set or cleared.
2595		If the new AEVD has IT_EVD_DEQUEUE_NOTIFICATIONS cleared, the Consumer can not
2596		subsequently disassociate the SEVD from the new AEVD.
2597		The Consumer may disassociate an SEVD from an AEVD by specifying the value of
2598		IT_NULL_HANDLE for <i>aevd</i> only if the SEVD is disabled and the AEVD has IT_EVD_
2599		DEQUEUE_NOTIFICATIONS set. Otherwise, IT_ERR_INVALID_EVD_STATE is returned.
2600		Consumer can not use <i>it_evd_modify</i> to request <i>fd</i> to be associated with SEVD; instead, the
2601		Consumer can only do so at it_evd_create time.
2602		To disassociate the <i>fd</i> , the Consumer can simply <i>close</i> the <i>fd</i> . This clears the bit for IT_EVD_
2603		CREAT_FD in <i>evd_flag</i> for the SEVD or AEVD.
2604		If <i>sevd_queue_size</i> is requested to be changed for SEVD then the Implementation is required to
2605		provide a queue size of at least <i>sevd_queue_size</i> , but is free to provide a larger queue size (or
2606		provide dynamic queue enlargement when needed). The Consumer can determine the actual
2607		queue size by querying the modified Simple Event Dispatcher.
2608		Attempting to modify <i>sevd_queue_size</i> to be less than <i>sevd_threshold</i> returns
2609		IT_ERR_INVALID_QUEUE_SIZE. Attempting to modify <i>sevd_threshold</i> to be greater than
2610		<i>sevd_queue_size</i> returns IT_ERR_INVALID_THRESHOLD. In both error cases, the operation
2611		will not change the respective parameter from its current value.
2612		If the number of entries on the Event queue is greater than the requested <i>sevd_queue_size</i> , the
2613		operation will return IT_ERR_INVALID_QUEUE_SIZE and not change the Event queue size.
2614		The Consumer can enable the SEVD (set <i>evd_enabled</i>) so that the SEVD will generate
2615		Notifications for the current or future associated AEVD or <i>fd</i> , if either of them exists. Enabling
2616		an SEVD prohibits the Consumer from waiting on the SEVD if it has an associated AEVD or <i>fd</i> .
2617		If SEVD is in Notification criteria then SEVD generates the Notification for an associated
2618		existing AEVD or <i>fd</i> .
2619		The Consumer can enable AEVD so that the AEVD will generate Notification for an associated
2620		<i>fd</i> . Enabling the AEVD disallows the Consumer from waiting on the AEVD if it has an

2621 associated *fd*. For an AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` cleared,
 2622 Consumer can still dequeue Events from the AEVD.

2623 Enabling the enabled EVD has no effect.

2624 The Consumer can disable the SEVD (clear *evd_enabled*) so that the SEVD will not generate
 2625 Notifications for an associated AEVD or *fd*, if either of them exists. If an associated AEVD has
 2626 the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` cleared the AEVD can dequeue Events
 2627 from the SEVD. The Consumer can not wait on or dequeue from the SEVD which is associated
 2628 with the AEVD has the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` cleared even when
 2629 SEVD is disabled. The Consumer can wait on or dequeue from the SEVD which is associated
 2630 with the AEVD has the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` set when SEVD is
 2631 disabled.

2632 The Consumer can disable AEVD so that the AEVD will not generate Notification for an
 2633 associated *fd*. Disabling the AEVD allows the Consumer to wait on the AEVD.

2634 Disabling the disabled EVD has no effect.

2635

2636 RETURN VALUE

2637 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

2638 2639	<code>IT_ERR_INVALID_EVD</code>	The Event Dispatcher Handle (<i>evd_handle</i>) was invalid.
2640	<code>IT_ERR_INVALID_MASK</code>	The mask contained invalid flag values.
2641 2642	<code>IT_ERR_INVALID_EVD_STATE</code>	The attempted operation was invalid for the current state of the Event Dispatcher.
2643 2644	<code>IT_ERR_RESOURCE_QUEUE_SIZE</code>	The underlying transport could not allocate the requested <i>sevd_queue_size</i> resources at this time.
2645 2646 2647	<code>IT_ERR_INVALID_QUEUE_SIZE</code>	The requested Simple Event Dispatcher queue size (<i>sevd_queue_size</i>) was less than the outstanding Events on the Event queue.
2648 2649	<code>IT_ERR_INVALID_THRESHOLD</code>	An invalid value for the Simple Event Dispatcher threshold was specified.
2650 2651	<code>IT_ERR_INVALID_AEVD</code>	The Aggregation Event Dispatcher Handle (<i>aevd</i>) was invalid.
2652 2653	<code>IT_ERR_RESOURCES</code>	The requested operation failed due to insufficient resources.
2654 2655 2656 2657	<code>IT_ERR_IA_CATASTROPHE</code>	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.

2658 **ERRORS**

2659 None.

2660 **SEE ALSO**2661 [*it_evd_create\(\)*](#), [*it_evd_query\(\)*](#), [*it_evd_free\(\)*](#)

it_evd_post_se()

2662

2663 **NAME**2664 `it_evd_post_se` – post software Event on Simple Event Dispatcher2665 **SYNOPSIS**

```

2666 #include <it_api.h>
2667
2668     it_status_t it_evd_post_se(
2669         IN          it_evd_handle_t    evd_handle,
2670         IN          const void        *event
2671     );

```

2672 **DESCRIPTION**

2673 `evd_handle` Simple Event Dispatcher of IT_SOFTWARE_EVENT_STREAM
 2674 Event Stream type.

2675 `event` Pointer to the Consumer-created Software Event.

2676 `it_evd_post_se` posts a software Event to the IT_SOFTWARE_EVENT_STREAM simple Event
 2677 Dispatcher Event queue. This causes an Event to arrive on the Event Dispatcher Software Event
 2678 Stream. The `event` pointer is opaque to the Implementation and release of the memory referenced
 2679 by the `event` pointer in a software Event is the Consumer's responsibility.

2680 If the Event queue is full, the operation is completed unsuccessfully and returns
 2681 IT_ERR_EVD_QUEUE_FULL. The `event` is not queued. Since the Event queue for software
 2682 Events can never overflow, the Affiliated Asynchronous Event Dispatcher is not affected.

2683 `it_evd_post_se` can only be used to post software Events within the same process since
 2684 `evd_handle` has the scope of a single IA.

2685 **RETURN VALUE**

2686 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

2687 IT_ERR_EVD_QUEUE_FULL The Simple Event Dispatcher queue was full.

2688 IT_ERR_INVALID_EVD The Event Dispatcher Handle (`evd_handle`) was
 2689 invalid.

2690 IT_ERR_INVALID_SOFT_EVD The Simple Event Dispatcher Handle (`evd_handle`)
 2691 was not an IT_SOFTWARE_EVENT_STREAM
 2692 Event Dispatcher.

2693 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in
 2694 the disabled state. None of the output parameters
 2695 from this routine are valid. See [it_ia_info_t](#) for a
 2696 description of the disabled state.

2697 **ERRORS**

2698 None.

2699 APPLICATION USAGE

2700 Consumer can use this operation to unblock an AEVD waiter as well as passing specific
2701 instruction for the unblocked waiter. The SEVD for the Software Event should be associated
2702 with the AEVD. A software Event is Notification Event and will unblock the waiter.

2703 SEE ALSO

2704 [*it_evd_create\(\)*](#), [*it_software_event_t*](#), [*it_evd_wait\(\)*](#)

it_evd_query()

2705

2706 **NAME**

2707

it_evd_query – query an existing Simple or Aggregate Event Dispatcher2708 **SYNOPSIS**

2709

```
#include <it_api.h>
```

2710

2711

```
it_status_t it_evd_query(
```

2712

```
    IN          it_evd_handle_t          evd_handle,
```

2713

```
    IN          it_evd_param_mask_t      mask,
```

2714

```
    OUT         it_evd_param_t           *params
```

2715

```
);
```

2716

2717

```
typedef enum {
```

2718

```
    IT_EVD_PARAM_ALL           = 0x0000001,
```

2719

```
    IT_EVD_PARAM_IA           = 0x0000002,
```

2720

```
    IT_EVD_PARAM_EVENT_NUMBER = 0x0000004,
```

2721

```
    IT_EVD_PARAM_FLAG         = 0x0000008,
```

2722

```
    IT_EVD_PARAM_QUEUE_SIZE   = 0x0000010,
```

2723

```
    IT_EVD_PARAM_THRESHOLD    = 0x0000020,
```

2724

```
    IT_EVD_PARAM_AEVD_HANDLE  = 0x0000040,
```

2725

```
    IT_EVD_PARAM_FD           = 0x0000080,
```

2726

```
    IT_EVD_PARAM_BOUND        = 0x0000100,
```

2727

```
    IT_EVD_PARAM_ENABLED      = 0x0000200,
```

2728

```
    IT_EVD_PARAM_OVERFLOWED   = 0x0000400
```

2729

```
} it_evd_param_mask_t;
```

2730

2731

```
typedef struct {
```

2732

```
    it_ia_handle_t      ia;                /* IT_EVD_PARAM_IA */
```

2733

```
    it_event_type_t    event_number;       /* IT_EVD_PARAM_EVENT_NUMBER*/
```

2734

```
    it_evd_flags_t     evd_flag;          /* IT_EVD_PARAM_FLAG */
```

2735

```
    size_t              sevq_queue_size;   /* IT_EVD_PARAM_QUEUE_SIZE */
```

2736

```
    size_t              sevq_threshold;    /* IT_EVD_PARAM_THRESHOLD */
```

2737

```
    it_evd_handle_t    aevid;             /* IT_EVD_PARAM_AEVD_HANDLE*/
```

2738

```
    int                 fd;               /* IT_EVD_PARAM_FD */
```

2739

```
    it_boolean_t        evd_bound;        /* IT_EVD_PARAM_BOUND */
```

2740

```
    it_boolean_t        evd_enabled;      /* IT_EVD_PARAM_ENABLED */
```

2741

```
    it_boolean_t        evd_overflowed;   /* IT_EVD_PARAM_OVERFLOWED */
```

2742

```
} it_evd_param_t;
```

2743 **DESCRIPTION**

2744

evd_handle Event Dispatcher.

2745

mask Logical OR of flags for requested EVD parameters.

2746

params Pointer to Consumer-allocated structure that the Implementation fills with Consumer-requested Event Dispatcher parameters.

2747

2748

it_evd_query returns the desired parameters of the Simple or Aggregate Event Dispatcher *evd_handle* in the structure pointed to by *params*. On return, each field of *params* is only valid if

2749

2750		the corresponding flag as shown below each field is set in the <i>mask</i> argument. The <i>mask</i> value
2751		IT_EVD_PARAM_ALL causes all fields to be returned.
2752		The definition of each field follows:
2753	<i>ia</i>	Handle for the Interface Adapter.
2754	<i>event_number</i>	Identifier for Event Stream type that can be enqueued to the EVD.
2755	<i>evd_flag</i>	Flags for Event Dispatcher. See it_evd_create for definitions and use
2756		of <i>evd_flag</i> .
2757	<i>sevd_queue_size</i>	Minimum size of the SEVD Event queue or zero for an AEVD.
2758	<i>sevd_threshold</i>	The number of non-notification Events on the Simple Event
2759		Dispatcher queue for Notification, unblocking.
2760	<i>aevd</i>	Handle for Aggregate EVD associated with SEVD or
2761		IT_NULL_HANDLE if none.
2762	<i>fd</i>	<i>File descriptor</i> corresponding to Event Dispatcher or ‘-1’ if none.
2763	<i>evd_bound</i>	When it has the value IT_TRUE indicates that the EVD is tied to an
2764		Event Stream so Events can be queued on EVD. For an AEVD,
2765		indicates that SEVDs are tied to the AEVD.
2766	<i>evd_enabled</i>	When it has the value IT_TRUE indicates: for an SEVD that it has
2767		been configured to notify an associated AEVD or <i>fd</i> when
2768		Notification criteria is reached; for an AEVD that it has been
2769		configured to notify an associated <i>fd</i> when it is notified by one of its
2770		associated SEVDs. See it_evd_modify .
2771	<i>evd_overflowed</i>	When it has the value IT_TRUE indicates that the EVD has
2772		overflowed. See it_evd_create for more details.
2773	RETURN VALUE	
2774		A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.
2775		
2776	IT_ERR_INVALID_EVD	The Event Dispatcher Handle (<i>evd_handle</i>) was
2777		invalid.
2778	IT_ERR_INVALID_MASK	The mask contained invalid flag values.
2779	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in
2780		the disabled state. None of the output parameters
2781		from this routine are valid. See it_ia_info_t for a
2782		description of the disabled state.
2783	ERRORS	
2784		None.

2785 **SEE ALSO**
2786 [*it_evd_create\(\)*](#), [*it_evd_modify\(\)*](#), [*it_evd_free\(\)*](#), [*it_ia_info_t*](#)

it_evd_wait()

2787

2788 **NAME**2789 `it_evd_wait` – wait for Events on Event Dispatcher2790 **SYNOPSIS**

```

2791 #include <it_api.h>
2792
2793     it_status_t it_evd_wait(
2794         IN         it_evd_handle_t     evd_handle,
2795         IN         uint64_t            timeout,
2796         OUT        it_event_t          *event,
2797         OUT        size_t              *nmore
2798     );

```

2799 **DESCRIPTION**2800 *evd_handle* Handle for Simple or Aggregate Event Dispatcher.2801 *timeout* The duration of time, in microseconds, that Consumer is willing to
2802 wait for an Event.2803 *event* Pointer to the Consumer-allocated structure that the Implementation
2804 fills with the Event information.2805 *nmore* The snapshot of the number of Events queued on the EVD at the
2806 time of `it_evd_wait` return. Only applicable for SEVD.

2807 `it_evd_wait` removes the first Event from the Event Dispatcher Event queue and fills the
2808 Consumer-allocated *event* structure with Event information. For the Event information and *event*
2809 structure see [it_event_t](#). The Consumer should allocate an Event structure big enough to hold
2810 any Event that the Event Dispatcher can deliver.

2811 The return value for *event* is defined only if `it_evd_wait` returns IT_SUCCESS.2812 The Consumer can wait on an EVD that is not associated with any higher level object (AEVD or
2813 fd).

2814 Consumer should not wait on an SEVD that has an associated AEVD with the
2815 `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit clear. An attempt by Consumer to wait on
2816 *evd_handle* for that type of SEVD will result in routine failure with the return value of
2817 `IT_ERR_INVALID_EVD_STATE`.

2818 Consumer should not wait on an EVD that is associated with and enabled for Notification to
2819 higher level objects. An attempt by Consumer to wait on *evd_handle* that is associated with and
2820 enabled for Notification to a higher level object will result in routine failure with the return value
2821 of `IT_ERR_INVALID_EVD_STATE`. However, the Consumer can wait on the EVD associated
2822 with the higher level object if the EVD is disabled for Notification (except if the object is an
2823 AEVD with the `IT_EVD_DEQUEUE_NOTIFICATIONS evd_flag` bit clear, as stated above).

2824 An Implementation can support one or more simultaneous waiters on the same EVD (for thread
2825 safety models see Section 3.2) if `sevd_threshold` value of *evd_handle* (see [it_evd_create](#)) is
2826 greater than one then only a single waiter is supported. An attempt for more than one waiter to

2827 wait on the EVD will result in an immediate error with IT_ERR_WAITER_LIMIT return value.
 2828 If *sevd_threshold* value of *evd_handle* is 1, then one or more simultaneous waiters can supported
 2829 for the SEVD.

2830 A waiter can be blocked. An SEVD waiter will block when SEVD queue is empty. An AEVD
 2831 waiter will block when all associated SEVDs are empty. An SEVD waiter may block when the
 2832 SEVD has not reached the Notification criteria (see [it_evd_create](#) for the definition of the
 2833 Notification criteria). An AEVD waiter may block when all associated SEVDs have not reached
 2834 their Notification criteria.

2835 An SEVD waiter will return immediately if there is a Notification Event (see [it_evd_create](#) for
 2836 the definition of the Notification Event) on the queue or if the number of Events on the SEVD is
 2837 equal or larger than *sevd_threshold*. An AEVD waiter with the IT_EVD_DEQUEUE_
 2838 NOTIFICATIONS *evd_flag* bit cleared will return immediately if there is a Notification Event
 2839 on any of the associated SEVDs or any of the associated SEVDs has number of Events larger
 2840 than or equal to its *sevd_threshold*. An AEVD waiter with the IT_EVD_DEQUEUE_
 2841 NOTIFICATIONS *evd_flag* bit set will return immediately if there is an IT_AEVD_
 2842 NOTIFICATION_EVENT available.

2843 If arriving Event causes SEVD to reach Notification criteria then SEVD waiter will be
 2844 unblocked if one exists and if the SEVD is disabled and not associated with AEVD with the
 2845 IT_EVD_DEQUEUE_NOTIFICATIONS *evd_flag* bit cleared. As many waiters as there are
 2846 Events available on the SEVD can be unblocked. If arriving Event causes the SEVD to reach
 2847 Notification criteria and the SEVD is enabled for Notification to higher level objects then
 2848 Notification will be generated for the associated AEVD or *fd*. If the associated AEVD has a
 2849 waiter then the waiter will be unblocked. As many Notifications can be generated as there are
 2850 Events available on all SEVDs of the AEVD. As many waiters as there are Notifications can be
 2851 unblocked. Which waiters will be woken and in what order they will be woken is
 2852 implementation-dependent.

2853 The *timeout* allows the Consumer to restrict the amount of time it will be blocked waiting for an
 2854 Event arrival. The value of IT_TIMEOUT_INFINITE indicates that Consumer will wait
 2855 indefinitely for an Event arrival. Consumers should use caution in using this value because wait
 2856 may never return if Notification is not generated. Consumers can use *signal* to unblock the
 2857 waiter in this case.

2858 For IT_DTO_EVENT_STREAM Events, when a Completion Event is returned for a given
 2859 Send, RDMA Read, RDMA Write, RMR Bind or RMR Unbind operation that was posted to an
 2860 Endpoint, the Implementation guarantees that all Send, RDMA Read, RDMA Write, RMR Bind
 2861 and RMR Unbind operations that were posted to the Endpoint prior to the one whose
 2862 Completion Event was returned have also completed regardless of their *dto_flag* value for
 2863 *IT_COMPLETION_FLAG*.

2864 For an SEVD, if the return value is neither IT_SUCCESS nor IT_ERR_TIMEOUT_EXPIRED,
 2865 then the returned values of *nmore* and *Event* are undefined. If the return value is
 2866 IT_ERR_TIMEOUT_EXPIRED, then the return value of *event* is undefined, but the return value
 2867 of *nmore* is defined. If the return value is IT_SUCCESS, then the return values of both *nmore*
 2868 and *event* are defined.

2869 For an AEVD *nmore* is undefined for all returns. If the return value is not IT_SUCCESS, then
 2870 returned value *event* is undefined.

2871 The routine returns with return value `IT_ERR_INTERRUPT` when the waiter is unblocked by an
 2872 OS signal.

2873 RETURN VALUE

2874 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

2875 `IT_ERR_WAITER_LIMIT` No more waiters are permitted for the Event
 2876 Dispatcher.

2877 `IT_ERR_INVALID_EVD` The Event Dispatcher Handle (*evd_handle*) was
 2878 invalid.

2879 `IT_ERR_INVALID_EVD_STATE` The attempted operation was invalid for the current
 2880 state of the Event Dispatcher.

2881 `IT_ERR_ABORT` The Event Dispatcher has been destroyed.

2882 `IT_ERR_INTERRUPT` The Event Dispatcher waiter was unblocked by a
 2883 signal.

2884 `IT_ERR_TIMEOUT_EXPIRED` The operation timed out.

2885 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in
 2886 the disabled state. None of the output parameters
 2887 from this routine are valid. See [it_ia_info_t](#) for a
 2888 description of the disabled state.

2889 ERRORS

2890 None.

2891 APPLICATION USAGE

2892 The Consumer should allocate an Event structure big enough to hold any Event that the Event
 2893 Dispatcher can deliver. The Implementation is not able to check that the *event* that Consumer
 2894 provides is sufficient to hold a returned Event. As a result a segmentation fault or memory
 2895 corruption may occur if the Implementation over-runs the user-specified memory.

2896 For an AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS` set, any `IT_AEVD_`
 2897 `NOTIFICATION_EVENT` Event only indicates that the SEVD (identified by *evd_handle* in the
 2898 Event) reached Notification criteria. The order in which the associated SEVD's AEVD
 2899 Notification Events are delivered is implementation-dependent. No restriction is imposed by the
 2900 Implementation on dequeuing Events from the underlying SEVD. If other Consumer threads
 2901 are independently dequeuing Events from the SEVD, the thread receiving the
 2902 `IT_AEVD_NOTIFICATION_EVENT` may find the SEVD to be empty when it dequeues from
 2903 the SEVD.

2904 For an AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS` set, receipt of an `IT_AEVD_`
 2905 `NOTIFICATION_EVENT` Event indicates that the SEVD (identified by *evd_handle* in the
 2906 Event) reached Notification status. If the Consumer fails to dequeue Events from the SEVD
 2907 sufficient to remove it from Notification status, then an additional
 2908 `IT_AEVD_NOTIFICATION_EVENT` Event for the SEVD will appear at the AEVD when the
 2909 Consumer next calls *it_evd_wait* or [it_evd_dequeue](#).

2910 For an AEVD with IT_EVD_DEQUEUE_NOTIFICATIONS set, receipt of an IT_AEVD_
 2911 NOTIFICATION_EVENT Event indicates that the SEVD (identified by *evd_handle* in the
 2912 Event) reached Notification status. By the time the Consumer calls [it_evd_dequeue](#) on the
 2913 returned SEVD it may be empty or may not be in the Notification Criteria any longer if there are
 2914 multiple dequeuers from the SEVD.

2915 The Consumer must be prepared to handle return from *it_evd_wait* with fewer than the expected
 2916 number of Events or without any Notification Events on an EVD. This can occur for the
 2917 following reasons:

2918 The underlying Implementation does not support thresholding.

2919 The underlying Implementation does not support IT_NOTIFY_FLAG.

2920 For *sevd_threshold* value of one, if an Event is on the SEVD then *it_evd_wait* will return
 2921 immediately with IT_SUCCESS for the SEVD or the AEVD fed by the SEVD.

2922 For the “non-thread-safe” Implementation Consumer should not have multiple threads calling on
 2923 the same EVD Handle simultaneously. Consumer should choose an Implementation that
 2924 supports multithreaded applications if they want to have multiple waiters. Consumer should set
 2925 the *sevd_threshold* to one for an SEVD if they want to use multiple waiters on the SEVD.

2926 When multiple threads retrieve Events concurrently from the same SEVD, each Event will be
 2927 retrieved exactly once but it is unpredictable which thread will retrieve any particular Event.

2928 The Consumer is advised not to destroy an EVD that it is currently waiting on. If the Consumer
 2929 does so, the *it_evd_wait* routine may return IT_ERR_ABORT, or a segmentation violation may
 2930 take place. Which behavior occurs is implementation-dependent.

2931 **SEE ALSO**

2932 [it_evd_create\(\)](#), [it_event_t](#), [it_post_send\(\)](#), [it_post_sendto\(\)](#), [it_post_rdma_read\(\)](#),
 2933 [it_post_rdma_write\(\)](#), [it_rmr_bind\(\)](#), [it_rmr_unbind\(\)](#), [it_dto_events](#), [it_dto_flags_t](#)

it_get_consumer_context()

2934

2935 **NAME**2936 `it_get_consumer_context` – return the Consumer Context associated with an IT Object Handle2937 **SYNOPSIS**

```

2938 #include <it_api.h>
2939
2940 it_status_t it_get_consumer_context(
2941     IN          it_handle_t      handle,
2942     OUT         it_context_t     *context
2943 );

```

2944 **DESCRIPTION**

2945 *handle* Handle of the IT-API object associated with the Consumer Context
 2946 to be retrieved.

2947 *context* The address of the location where the retrieved Consumer Context is
 2948 returned.

2949 *it_get_consumer_context* retrieves the Consumer Context associated with the specified *handle*.
 2950 If the Consumer Context was never set (by a call to [it_set_consumer_context](#)), then the value of
 2951 the returned Consumer Context is 0.

2952 The *handle* must be one of the IT-API Handle types, cast as an [it_handle_t](#). See [it_handle_t](#) for
 2953 a description of the valid Handle types.

2954 **RETURN VALUE**

2955 A successful call returns SUCCESS. Otherwise, an error code is returned as described below.

2956 IT_ERR_INVALID_HANDLE The *handle* was invalid.

2957 IT_ERR_NO_CONTEXT The *handle* does not have an associated Context.

2958 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
 2959 disabled state. None of the output parameters from this
 2960 routine are valid. See [it_ia_info_t](#) for a description of the
 2961 disabled state.

2962 **ERRORS**

2963 None.

2964 **EXAMPLES**

2965 The following code example demonstrates the use of a cast in the call to
 2966 *it_get_consumer_context*. The *lmr* object is cast to the generic [it_handle_t](#) type for the call.

```

2967 it_lmr_handle_t lmr;
2968 it_context_t cxt;
2969 it_get_consumer_context( (it_handle_t) lmr, &cxt);
2970

```

2971 **SEE ALSO**
2972 [*it_set_consumer_context\(\), it_context_t, it_handle_t*](#)

it_get_handle_type()

2973

2974 **NAME**2975 `it_get_handle_type` – return the Handle type value associated with an IT Object Handle2976 **SYNOPSIS**

```

2977 #include <it_api.h>
2978
2979 it_status_t it_get_handle_type(
2980     IN          it_handle_t          handle,
2981     OUT         it_handle_type_enum_t *type_of_handle
2982 );

```

2983 **DESCRIPTION**2984 *handle* Handle of an IT-API object.2985 *type_of_handle* Type of the Handle of *handle*.

2986 The *it_get_handle_type* interface allows the Consumer to retrieve the type of an IT Object using
 2987 its Handle. See [it_handle_t](#) for a description of the Handle types and associated enumeration
 2988 values returned.

2989 **RETURN VALUE**

2990 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 2991 below.

2992 IT_ERR_INVALID_HANDLE The *handle* was invalid.

2993 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
 2994 disabled state. None of the output parameters from this
 2995 routine are valid. See [it_ia_info_t](#) for a description of the
 2996 disabled state.

2997 **ERRORS**

2998 None.

2999 **SEE ALSO**3000 [it_handle_t](#)

it_get_pathinfo()

3001

3002 **NAME**

3003 `it_get_pathinfo` – retrieve a set of Paths that can be used to communicate with a given remote
 3004 Network Address

3005 **SYNOPSIS**

```

3006 #include <it_api.h>
3007
3008 it_status_t it_get_pathinfo(
3009     IN          it_ia_handle_t    ia_handle,
3010     IN          size_t            spigot_id,
3011     IN          const it_net_addr_t *net_addr,
3012     IN OUT     size_t            *num_paths,
3013     OUT        size_t            *total_paths,
3014     OUT        it_path_t         *paths
3015 );
  
```

3016 **DESCRIPTION**

3017 *ia_handle* The Handle for the IA that the caller wishes to use for
 3018 communicating with the remote Network Address .

3019 *spigot_id* The Spigot on the IA that the caller wishes to use for communicating
 3020 with the remote Network Address.

3021 *net_addr* The remote Network Address to communicate with.

3022 *num_paths* On input, points to the count of the maximum number of Paths that
 3023 the Consumer wishes to have returned. On output, points to the
 3024 count of the total number of Paths that were actually returned, which
 3025 is guaranteed to be less than or equal to the number that the
 3026 Consumer requested. This is only valid on output if the call returns
 3027 IT_SUCCESS.

3028 *total_paths* The total number of Paths that were available to access the remote
 3029 Network Address. This may be greater than the number of Paths
 3030 returned via *num_paths* if there were more Paths available than the
 3031 maximum the Consumer wished to have returned. This is only valid
 3032 if the call returns IT_SUCCESS.

3033 *paths* An array allocated by the Consumer that holds the returned Path(s).
 3034 This only contains valid information if the call returns
 3035 IT_SUCCESS.

3036 *it_get_pathinfo* is used to retrieve a set of Paths that can be used to reach the specified remote
 3037 Network Address. The local component of the Path is given by the combination of *ia_handle*
 3038 (which identifies the local IA to use), and *spigot_id* (which identifies the Spigot to be used on
 3039 that IA). The set of Paths that can be used is returned in *paths*.

3040 How the Consumer chooses which IA and Spigot to use for the local component of the Path is
 3041 outside the scope of the API. The API does, however, provide the [it_interface_list](#) routine to

3042 enumerate all Interfaces that could be used to find a possible Path. The Consumer can use the
 3043 names returned as input to the [it_ia_create](#) routine, which will return an IA Handle that can
 3044 subsequently be fed to [it_ia_query](#) to determine what the valid Spigot identifiers are for that IA.

3045 Several different Network Address formats are supported: see the man page for [it_net_addr_t](#)
 3046 for details. The mechanism by which the Consumer determines the remote Network Address to
 3047 target is outside the scope of the API. If the underlying transport is one that supports IP
 3048 Network Addresses, existing APIs (such as *gethostbyname*) for translating host names into IP
 3049 addresses can be used to convert a hostname into an IP address.

3050 The Consumer is responsible for allocating the storage necessary to hold the returned set of
 3051 *paths*. Since the Consumer may not know how many Paths are available, it passes the number of
 3052 Paths for which it has allocated storage in the *num_paths* parameter on input. This routine will
 3053 return no more than that number of Paths to the Consumer. If more Paths are available than the
 3054 Consumer has allocated space for, an arbitrary subset of the available Paths will be provided to
 3055 the Consumer. A Consumer that does not wish to deal with Path selection can therefore avoid
 3056 doing so by always specifying a value of one for the total number of Paths it wishes to have
 3057 returned.

3058 The set of Paths that are available to reach a given remote Network Address is dynamic, and can
 3059 change over time. (For example, a link on a switch or router could become inoperative, thus
 3060 decreasing the set of available Paths.) There is therefore no guarantee that given the same input
 3061 parameters two different invocations of *it_get_pathinfo* will return the same results. The
 3062 information returned by *it_get_pathinfo* is a snapshot of the Paths available at the time of the
 3063 call. In addition, if the Consumer asks for fewer Paths than are available, the API may return a
 3064 different set of Paths for two different invocations of *it_get_pathinfo* regardless of the state of
 3065 the network.

3066 It is possible that no Paths are available to reach the given remote Network Address. In that
 3067 case, *it_get_pathinfo* will return IT_SUCCESS, but the total number of Paths available pointed
 3068 to by *num_paths* will be zero.

3069 Once the Consumer has chosen one of the set of Paths returned, it can furnish that Path as input
 3070 to the [it_ep_connect](#) routine. Consumers that wish to construct their own Path can also do so by
 3071 populating the [it_path_t](#) data structure themselves, although this is inherently a transport-
 3072 dependent programming practice. See the man page for [it_path_t](#) for details on the internal
 3073 structure of a Path.

3074 RETURN VALUE

3075 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 3076 below.

3077 IT_ERR_INVALID_IA The Interface Adapter Handle (*ia_handle*) was
 3078 invalid.

3079 IT_ERR_INVALID_SPIGOT An invalid Spigot ID was specified.

3080 IT_ERR_INVALID_NETADDR The format of the Network Address was not
 3081 recognized.

3082 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic
 3083 error and is in the disabled state. None of the output

3084 parameters from this routine are valid. See
3085 [it ia info t](#) for a description of the disabled state.

3086 **ERRORS**
3087 None.

3088 **SEE ALSO**
3089 [it interface list\(\)](#), [it ia create\(\)](#), [it ia query\(\)](#), [it ep connect\(\)](#), [it listen create\(\)](#)

it_handoff()

3090

3091 **NAME**

3092 `it_handoff` - forward an incoming Connection Request to another Spigot and Connection
 3093 Qualifier

3094 **SYNOPSIS**

```
3095 #include <it_api.h>
3096
3097 it_status_t it_handoff(
3098     IN      const      it_conn_qual_t          * conn_qual,
3099     IN      size_t      spigot_id,
3100     IN      it_cn_est_identifier_t  cn_est_id
3101 );
3102
3103 typedef uint64_t it_cn_est_identifier_t;
```

3104 **DESCRIPTION**

3105 *conn_qual* The Connection Qualifier to which the Connection Request should
 3106 be forwarded.

3107 *spigot_id* Interface Adapter Spigot to which the Connection Request should be
 3108 forwarded.

3109 *cn_est_id* Connection Establishment Identifier associated with the Connection
 3110 Request to be forwarded.

3111 *it_handoff* forwards a Connection Request to the specified Spigot and Connection Qualifier of
 3112 the IA on which the Connection Request originally arrived. The forwarded Connection Request
 3113 generates an IT_CM_REQ_CONN_REQUEST_EVENT Event at the Listen Point to which the
 3114 request was forwarded. Forwarded Connection Request Events look identical to the original
 3115 Events, therefore the Consumer can not distinguish them. The Connection Establishment
 3116 Identifier, *cn_est_id*, is destroyed by this function.

3117 **RETURN VALUE**

3118 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below:

3119 ERR_INVALID_CN_EST_ID The Connection Establishment Identifier (*cn_est_id*)
 3120 was invalid.

3121 IT_ERR_INVALID_CONN_QUAL The Connection Qualifier (*conn_qual*) was invalid.

3122 IT_ERR_INVALID_SPIGOT An invalid Spigot ID was specified.

3123 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in
 3124 the disabled state. None of the output parameters
 3125 from this routine are valid. See [it_ia_info_t](#) for a
 3126 description of the disabled state.

3127 **ERRORS**

3128 None.

3129 **APPLICATION USAGE**

3130 Calls to [it_reject](#), [it_ep_accept](#), and *it_handoff* that pertain to the same Endpoint should be
3131 serialized by the Consumer. Failure to abide by this restriction may result in a segmentation
3132 violation or other error.

3133 **SEE ALSO**

3134 [it_ep_connect\(\)](#), [it_reject\(\)](#), [it_ep_accept\(\)](#), [it_cm_req_events](#)

it_hton64()

3135
 3136 **NAME**
 3137 `it_hton64, it_ntoh64` – convert 64-bit integers between host and network byte order

3138 **SYNOPSIS**
 3139 `#include <it_api.h>`
 3140
 3141 `uint64_t it_hton64(`
 3142 `uint64_t hostint`
 3143 `);`
 3144
 3145 `uint64_t it_ntoh64(`
 3146 `uint64_t netint`
 3147 `);`

3148 **DESCRIPTION**

3149 `hostint` 64-bit integer stored in host byte order.

3150 `netint` 64-bit integer stored in network byte order.

3151 The `it_hton64` routine converts its input argument `hostint` from host byte order to network byte
 3152 order and returns the result.

3153 `it_ntoh64` converts its input argument `netint` from network byte order to host byte order and
 3154 returns the result.

3155 On some platforms, host byte order and network byte order are identical and these functions
 3156 simply return their input argument.

3157 **RETURN VALUE**

3158 Both functions always succeed and return their converted input argument.

3159 **ERRORS**

3160 None.

3161 **APPLICATION USAGE**

3162 The individual bytes of integer variables are stored in memory in an order that is platform
 3163 dependent, which is known as host byte order. To facilitate the exchange of integer variables
 3164 between platforms having different host byte orders, a platform independent byte order known as
 3165 network byte order has been defined. To portably send an integer to a network peer, the
 3166 Consumer should convert it from host to network byte order and send the network byte order
 3167 value. The receiving peer then converts from network byte order to its own host byte order.

3168 Note that an integer must be stored in host byte order to be used correctly in normal arithmetic
 3169 operations.

3170 **SEE ALSO**

3171 `htonl(), ntohl()`

it_ia_create()

3172

3173 **NAME**3174 `it_ia_create` – create an Interface Adapter3175 **SYNOPSIS**

```

3176 #include <it_api.h>
3177
3178 it_status_t it_ia_create(
3179     IN      const      char          *name,
3180     IN      uint32_t    major_version,
3181     IN      uint32_t    minor_version,
3182     OUT     it_ia_handle_t *ia_handle
3183 );

```

3184 **DESCRIPTION**

3185 *name* The name of the Interface for which to create an Interface Adapter.

3186 *major_version* The IT-API major version that the Consumer will use in subsequent
3187 calls to the IA.

3188 *minor_version* The IT-API minor version that the Consumer will use in subsequent
3189 calls to the IA.

3190 *ia_handle* Upon successful return, points to an Interface Adapter Handle for
3191 the created Interface Adapter.

3192 *it_ia_create* is used to create an Interface Adapter. The Consumer identifies the Interface
3193 Adapter to be created by its Interface name, major and minor version numbers for the most
3194 recent version of the IT-API supported. The Consumer may select these parameters from the list
3195 returned by the [it_interface_list](#) call.

3196 The major version number associated with the first release of the IT-API is 1, and the minor
3197 version number associated with the first release of the IT-API is 0. When a new version of the
3198 IT-API is released, a unique combination of major and minor version numbers is associated with
3199 it. If the new release is source code compatible with the previous release the major version
3200 number of the new release will be the same as that of the previous release, and the minor version
3201 number will be incremented by one. If the new release is not source code compatible with the
3202 previous release the major version number of the new release will be incremented by one, and
3203 the minor number will be zero.

3204 The latest version of the IT-API that an Implementation supports is returned from the
3205 [it_interface_list](#) call. For the *major_version* returned from that call, the Implementation shall
3206 support all minor versions less than or equal to the *minor_version* returned from that call. The
3207 Implementation is not required to support major versions of the IT-API previous to the one
3208 returned from [it_interface_list](#). If the Implementation does not support conversing with the IA
3209 using the requested previous major version of the IT-API, an error will be returned from
3210 *it_ia_create*.

3211 If successful, this routine returns an Interface Adapter Handle. The returned Interface Adapter
 3212 Handle may be passed to other IT-API routines that create and manage Interface Adapter objects
 3213 such as Event Dispatchers and Local Memory Regions.

3214 **RETURN VALUE**

3215 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 3216 below.

3217 IT_ERR_RESOURCES The requested operation failed due to
 3218 insufficient resources.

3219 IT_ERR_INVALID_NAME The specified *name* was invalid.

3220 IT_ERR_INVALID_MAJOR_VERSION The requested IT-API major version number
 3221 was not supported for this Interface Adapter.

3222 IT_ERR_INVALID_MINOR_VERSION The requested IT-API minor version number
 3223 was not supported for this Interface Adapter.

3224 **ERRORS**

3225 None.

3226 **SEE ALSO**

3227 [*it_interface_list\(\), it_ia_query\(\), it_ia_free\(\)*](#)

it_ia_free()

3228

3229 **NAME**3230 `it_ia_free` – free Interface Adapter Handle3231 **SYNOPSIS**

```

3232 #include <it_api.h>
3233
3234 it_status_t it_ia_free(
3235     IN          it_ia_handle_t    ia_handle
3236 );

```

3237 **DESCRIPTION**3238 `ia_handle` Identifies the Interface Adapter Handle to be freed.3239 `it_ia_free` is used to free an Interface Adapter Handle.

3240 All IT Objects associated with the specified Interface Adapter Handle are freed before this
 3241 routine returns. The documented semantics associated with freeing the various IT Objects are
 3242 observed when these objects are freed by the call to `it_ia_free`. Further use by the Consumer of
 3243 Handles for those freed IT Objects after this routine returns successfully may have unpredictable
 3244 effects. All [it_ia_info_t](#) structures that were returned to the Consumer by [it_ia_query](#) that have
 3245 not already been freed by the Consumer (via [it_ia_info_free](#)) are freed. Examining an
 3246 [it_ia_info_t](#) that was associated with `ia_handle` after this routine returns may have unpredictable
 3247 effects.

3248 All pending operations associated with the specified `ia_handle` will be terminated before this
 3249 routine returns. Posted Data Transfer Operations that are currently in progress will be
 3250 terminated before this routine returns. The completion status of such DTOs is indeterminate; if
 3251 the Consumer wishes to know the completion status of the DTOs they have issued they should
 3252 dequeue the relevant Completion Events before freeing the IA. All callers blocked in
 3253 [it_evd_wait](#) calls associated with the specified `ia_handle` will be unblocked.

3254 All Connections and pending Connection Requests associated with the specified `ia_handle` are
 3255 terminated before this routine returns.

3256 **RETURN VALUE**3257 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
3258 below.3259 IT_ERR_INVALID_IA The Interface Adapter Handle (`ia_handle`) was invalid.3260 **ERRORS**

3261 None.

3262 **SEE ALSO**3263 [it_ia_create\(\)](#), [it_ia_query\(\)](#)

it_ia_info_free()

3264

3265 **NAME**3266 `it_ia_info_free` – free an [it_ia_info_t](#) structure that was returned by [it_ia_query](#)3267 **SYNOPSIS**

```

3268 #include <it_api.h>
3269
3270 void it_ia_info_free(
3271     IN          it_ia_info_t      *ia_info
3272 );

```

3273 **DESCRIPTION**

3274 `ia_info` Points to an [it_ia_info_t](#) data structure that was previously returned
 3275 from a call to [it_ia_query](#).

3276 `it_ia_info_free` is used to free the memory for the data structure allocated and returned by the
 3277 [it_ia_query](#) routine. The Consumer should use this routine rather than the `free` routine to
 3278 deallocate the data structure pointed to by `ia_info`; unpredictable behavior can result if `free` is
 3279 used. Since this routine deallocates the input data structure, the Consumer should not attempt to
 3280 access it after successfully returning from this routine.

3281 This routine does not free any of the resources that are associated with the [it_ia_info_t](#) data
 3282 structure; it only frees the data structure itself. In particular, calling this routine does not cause
 3283 the EVD Handle associated with the EVD that contains the Affiliated Asynchronous Event
 3284 Stream (if present) or the EVD Handle associated with the EVD that contains the Unaffiliated
 3285 Asynchronous Event Stream (if present) to be freed.

3286 When an IA is freed (by calling [it_ia_free](#)), any [it_ia_info_t](#) structures that were returned by
 3287 [it_ia_query](#) for that IA will also be freed. The Consumer can call `it_ia_info_free` to free an
 3288 [it_ia_info_t](#) structure before the IA is freed. After the IA has been freed, calling `it_ia_info_free`
 3289 to free an [it_ia_info_t](#) associated with that IA will have undefined results, and may result in
 3290 memory corruption.

3291 **RETURN VALUE**

3292

3293 None.

3294 **ERRORS**

3295

None.

3296 **SEE ALSO**

3297

[it_ia_info_t\(\)](#), [it_ia_query\(\)](#)

it_ia_query()

3298

3299 **NAME**3300 `it_ia_query` – retrieve attributes of given Interface Adapter and its Spigots3301 **SYNOPSIS**

```

3302 #include <it_api.h>
3303
3304 it_status_t it_ia_query(
3305     IN          it_ia_handle_t    ia_handle,
3306     OUT         it_ia_info_t      ** ia_info
3307 );

```

3308 **DESCRIPTION**3309 `ia_handle` Identifies the Interface Adapter to be queried.

3310 `ia_info` Points to a pointer to an [it_ia_info_t](#) structure upon successful
 3311 return. The [it_ia_info_t](#) structure contains the attributes of the
 3312 Interface Adapter and the identity of its Spigots.

3313 `it_ia_query` is used to retrieve the attributes of an Interface Adapter and its associated Spigots.
 3314 See the man page [it_ia_info_t](#) for details of the attributes structure.

3315 This routine allocates the storage necessary to hold the returned [it_ia_info_t](#) structure. The
 3316 Consumer should free the allocated storage using the [it_ia_info_free](#) routine; if the Consumer
 3317 fails to do so, the Implementation will free the storage when [it_ia_free](#) is called for `ia_handle`.

3318 If the query is unsuccessful, the return value will indicate failure, no [it_ia_info_t](#) structure will
 3319 be allocated, and `ia_info` will point to a NULL pointer.

3320 **RETURN VALUE**

3321 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 3322 below.

3323 IT_ERR_INVALID_IA The Interface Adapter Handle (`ia_handle`) was
 3324 invalid.

3325 IT_ERR_RESOURCES The requested operation failed due to insufficient
 3326 resources.

3327 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic
 3328 error and is in the disabled state. None of the output
 3329 parameters from this routine are valid. See
 3330 [it_ia_info_t](#) for a description of the disabled state.

3331 **ERRORS**

3332 None.

3333 **SEE ALSO**3334 [it_ia_create\(\)](#), [it_ia_free\(\)](#), [it_ia_info_t](#), [it_ia_info_free\(\)](#)

it_interface_list()

3335

3336 **NAME**

3337

it_interface_list – retrieve information about the available Interfaces

3338 **SYNOPSIS**

3339

```
#include <it_api.h>
```

3340

3341

```
void it_interface_list(
```

3342

```
    OUT          it_interface_t      *interfaces,
```

3343

```
    IN OUT      size_t              *num_interfaces,
```

3344

```
    IN OUT      size_t              *total_interfaces
```

3345

```
);
```

3346

```
typedef struct {
```

3347

3348

```
    /* Most recent major version number of the IT-API supported by the  
    Interface */
```

3349

3350

```
    uint32_t    major_version;
```

3351

3352

```
    /* Most recent minor version number of the IT-API supported by the  
    Interface */
```

3353

3354

```
    uint32_t    minor_version;
```

3355

3356

```
    /* The transport that the Interface uses, as defined in
```

3357

3358

```
    it_ia_info_t. */  
    it_transport_type_t    transport_type;
```

3359

3360

```
    /* The name of the Interface, suitable for input to it_ia_create.  
    The name is a string of maximum length IT_INTERFACE_NAME_SIZE,  
    including the terminating NULL character. */  
    char    name[IT_INTERFACE_NAME_SIZE];
```

3361

3362

3363

3364

3365

```
    } it_interface_t;
```

3366

3367 **DESCRIPTION**

3368

interfaces

3369

An array allocated by the Consumer that contains the information returned for the Interface(s).

3370

num_interface

3371

3372

3373

3374

3375

On input, points to the count of the maximum number of Interfaces for which the Consumer wishes to have information returned. On output, points to the count of the number of Interfaces for which information was actually returned, which is guaranteed to be less than or equal to the number that the Consumer requested.

3376

total_interfaces

3377

3378

3379

Upon return, points to the number of Interfaces potentially available for Consumer use. A Consumer may specify NULL for this parameter if it does not wish to know how many Interfaces are potentially available.

3380 *it_interface_list* is used to retrieve information about the set of available Interfaces. The
 3381 Consumer may select an Interface from the returned set, and furnish the name and version
 3382 number for that Interface as input to the [it_ia_create](#) call.

3383 The Consumer is responsible for allocating the storage necessary to hold the information for the
 3384 returned set of Interfaces. Since the local Consumer may not know how many Interfaces are
 3385 available, it passes the number of Interfaces for which it has allocated storage in the
 3386 *num_interfaces* parameter on input. This routine will return information for no more than that
 3387 number of Interfaces to the Consumer. If more Interfaces are available than the Consumer has
 3388 allocated space for, information will be provided to the Consumer for only *num_interfaces* such
 3389 Interfaces; which Interfaces information will be returned for is arbitrary in this case. Upon
 3390 return, the value pointed to by *total_interfaces* is the total number of available Interfaces.

3391 The set of Interfaces available to the Consumer is dynamic, and can change over time. (For
 3392 example, an Interface can become inoperative, thus decreasing the set of available Interfaces.)
 3393 There is therefore no guarantee that given the same input parameters two different invocations of
 3394 *it_interface_list* will return the same results. The information returned by *it_interface_list* is a
 3395 snapshot of the Interfaces available at the time of the call. In addition, if the Consumer asks for
 3396 fewer Interfaces than are available, the API may return information for a different set of
 3397 Interfaces for two different invocations of *it_interface_list* regardless of the state of the
 3398 Interfaces.

3399 It is possible that no Interfaces are available. In that case the total number of Interfaces available
 3400 pointed to by *num_interfaces* will be zero.

3401 **RETURN VALUE**

3402 None.

3403 **ERRORS**

3404 None.

3405 **EXAMPLES**

3406 The following example illustrates how the Consumer can check after it has created the IA to
 3407 ensure that the information it retrieved from the *it_interface_list* call is still valid.

```

3408 it_interface_t interface;
3409 size_t          num_interfaces;
3410 it_ia_handle_t  ia;
3411 it_ia_info_t    *infop;
3412
3413 num_interfaces = 1;
3414 it_interface_list(&interface, &num_interfaces, NULL);
3415 if (num_interfaces != 1) {
3416
3417     /* Failed to find any IAs; */
3418
3419 }
3420
3421 if (it_ia_create( interface.name, interface.major_version,
3422 interface.minor_version, &ia) != IT_SUCCESS) {
3423
3424     /* The IA wasn't found. Assuming sufficient resources were available,
3425        this can happen if the Interface that was retrieved by the
  
```

```

3426         it_interface_list call isn't available anymore. */
3427     }
3428 }
3429
3430     if (it_ia_query(ia, &infop) != IT_SUCCESS) {
3431
3432         /* This can happen if the Implementation didn't have sufficient
3433            resources to return the information for the IA */
3434     }
3435 }
3436
3437     if ((infop->transport_type != interface.transport_type) {
3438
3439         /* This can happen if the Interface that was retrieved by the
3440            it_interface_list call isn't available anymore. (A different
3441            Interface with the same name as was retrieved by it_interface_list
3442            is available as it turns out. The most likely reason this happened
3443            is that a new Interface was added to the system between the time
3444            it_interface_list was called and the time that it_ia_create was
3445            called.) */
3446     }
3447 }
3448
3449     /* Validation of the information returned by it_interface_list is
3450        complete. */

```

3451 **APPLICATION USAGE**

3452 The Interface Adapter associated with a given *name* can change between the time that the
3453 *it_interface_list* routine is called and the time that [it_ia_create](#) is called to actually create the IA.
3454 For that reason, the Consumer should check after it has created the IA to ensure that the
3455 information it retrieved from the *it_interface_list* call is still valid.

3456 **SEE ALSO**

3457 [it_ia_create\(\)](#), [it_ia_info t](#)

3458

it_listen_create()

3459

3460 NAME

3461 `it_listen_create` - create a Listen Point for incoming Connection Requests to a
 3462 Connection Qualifier

3463 SYNOPSIS

```

3464 #include <it_api.h>
3465
3466 it_status_t it_listen_create(
3467     IN          it_ia_handle_t          ia_handle,
3468     IN          size_t                  spigot_id,
3469     IN          it_evd_handle_t         connect_evd,
3470     IN          it_listen_flags_t       flags,
3471     IN OUT     it_conn_qual_t          *conn_qual,
3472     OUT        it_listen_handle_t      *listen_handle
3473 );
3474
3475 typedef enum {
3476     IT_LISTEN_NO_FLAG           = 0x0000,
3477     IT_LISTEN_CONN_QUAL_INPUT  = 0x0001
3478 } it_listen_flags_t;

```

3479 DESCRIPTION

3480 *ia_handle*: Interface Adapter Handle.

3481 *spigot_id*: Interface Adapter Spigot identifier.

3482 *connect_evd*: The Handle of the Simple Event Dispatcher where Connection
 3483 Request Events for this Listen Point will be posted. The Event
 3484 Stream Type of the Simple Event Dispatcher must be
 3485 IT_CM_REQ_EVENT_STREAM.

3486 *flags*: Specifies whether the Connection Qualifier is an input or output
 3487 parameter.

3488 *conn_qual*: The Connection Qualifier for which the Consumer wants to listen for
 3489 Connection Requests.

3490 *listen_handle*: Upon successful return points to a Handle to the created Listen
 3491 Point.

3492 *it_listen_create* establishes a Listen Point for incoming Connection Requests for a particular
 3493 Connection Qualifier on the Spigot identified. Incoming Connection Request Events will be
 3494 posted to the Simple Event Dispatcher specified until the Listen Point is destroyed. The
 3495 *listen_handle* returned can be passed to [it_listen_free](#) when the Listen Point is no longer needed.

3496 When the IT_LISTEN_CONN_QUAL_INPUT bit is set in *flags*, *conn_qual* is an input
 3497 parameter. When this bit is clear, *conn_qual* is an output parameter and an available Connection
 3498 Qualifier is returned through that parameter.

3499 A backlog for the incoming Connection Request Events is provided by the size of the Simple
 3500 Event Dispatcher that the Events are directed to. If a Connection Request arrives while the
 3501 Simple Event Dispatcher is full it is discarded and the Active side of the Connection
 3502 establishment attempt will receive an IT_CM_MSG_CONN_NONPEER_REJECT_EVENT
 3503 Event, with IT_CN_REJ_TIMEOUT as the reject reason code.

3504 **RETURN VALUE**

3505 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below:

3506	IT_ERR_INVALID_CONN_QUAL	The Connection Qualifier (<i>conn_qual</i>) was invalid.
3507	IT_ERR_CONN_QUAL_BUSY	The Connection Qualifier was already in use.
3508	IT_ERR_NO_PERMISSION	The Consumer did not have the proper permissions to perform the requested operation.
3509		
3510	IT_ERR_RESOURCES	The requested operation failed due to insufficient resources.
3511		
3512	IT_ERR_INVALID_CONN_EVD	The Connection Simple Event Dispatcher Handle was invalid.
3513		
3514	IT_ERR_INVALID_IA	The Interface Adapter Handle (<i>ia_handle</i>) was invalid.
3515		
3516	IT_ERR_INVALID_SPIGOT	An invalid Spigot ID was specified.
3517	IT_ERR_INVALID_FLAGS	The <i>flags</i> value was invalid.
3518	IT_ERR_INVALID_EVD_TYPE	The Event Stream Type for the Event Dispatcher was invalid.
3519		
3520	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
3521		
3522		
3523		

3524 **ERRORS**

3525 None.

3526 **SEE ALSO**

3527 [it_listen_free\(\)](#), [it_listen_query\(\)](#), [it_cm_msg_events](#)

it_listen_free()

3528

3529 **NAME**3530 `it_listen_free` - free a Listen Point.3531 **SYNOPSIS**

```

3532 #include <it_api.h>
3533
3534 it_status_t it_listen_free(
3535     IN it_listen_handle_t listen_handle
3536 );

```

3537 **DESCRIPTION**3538 `listen_handle` Identifies the Listen Point to be destroyed.

3539 Frees a Listen Point associated with a Connection Qualifier. Upon return no more Connection
 3540 Requests will be posted for the associated Connection Qualifier. Previously posted un-reaped
 3541 Connection Requests, if any, will remain valid on the `connect_evd` and therefore can be used as
 3542 input to either [it_ep_accept](#) or [it_reject](#).

3543 **RETURN VALUE**3544 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.3545 `IT_ERR_INVALID_LISTEN` The Listen Handle (`listen_handle`) was invalid.

3546 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 3547 disabled state. None of the output parameters from this
 3548 routine are valid. See [it_ia_info_t](#) for a description of
 3549 the disabled state.

3550 **ERRORS**

3551 None.

3552 **SEE ALSO**3553 [it_listen_create\(\)](#), [it_listen_query\(\)](#)

it_listen_query()

3554

3555 **NAME**

3556

it_listen_query - query parameters associated with a Listen Point

3557 **SYNOPSIS**

3558

3559

3560

3561

3562

3563

3564

3565

3566

3567

3568

3569

3570

3571

3572

3573

3574

3575

3576

3577

3578

3579

```

#include <it_api.h>

it_status_t it_listen_query(
    IN          it_listen_handle_t    listen_handle,
    IN          it_listen_param_mask_t mask,
    OUT         it_listen_param_t     *params
);

typedef enum {
    IT_LISTEN_PARAM_ALL           = 0x0001,
    IT_LISTEN_PARAM_IA_HANDLE    = 0x0002,
    IT_LISTEN_PARAM_SPIGOT_ID    = 0x0004,
    IT_LISTEN_PARAM_CONNECT_EVD  = 0x0008,
    IT_LISTEN_PARAM_CONN_QUAL    = 0x0010
} it_listen_param_mask_t;

typedef struct {
    it_ia_handle_t    ia_handle;    /* IT_LISTEN_PARAM_IA_HANDLE */
    size_t            spigot_id;    /* IT_LISTEN_PARAM_SPIGOT_ID */
    it_evd_handle_t  connect_evd;  /* IT_LISTEN_PARAM_CONNECT_EVD */
    it_conn_qual_t   connect_qual; /* IT_LISTEN_PARAM_CONN_QUAL */
} it_listen_param_t;

```

3580 **DESCRIPTION**

3581

listen_handle Handle associated with the Listen Point being queried.

3582

mask Bitwise OR of flags for desired parameters.

3583

3584

params Pointer to Consumer-allocated structure whose members are written with the desired Listen Point parameters and attributes.

3585

3586

3587

it_listen_query queries the parameters associated with a Listen Point. On return, each field of *params* is only valid if the corresponding flag as shown in the Synopsis is set in the *mask* argument. The *mask* value IT_LISTEN_PARAM_ALL causes all fields to be returned.

3588

The definition of each field of *params* follows

3589

ia_handle: Interface Adapter Handle.

3590

spigot_id Interface Adapter Spigot identifier.

3591

3592

connect_evd The Handle of the Simple Event Dispatcher where incoming Connection Request Events for this Listen Point are posted.

3593

connect_qual The Connection Qualifier associated with the Listen Point.

3594 **RETURN VALUE**

3595 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

3596 IT_ERR_INVALID_LISTEN The Listen Handle (*listen_handle*) was invalid.

3597 IT_ERR_INVALID_MASK The mask contained invalid flag values.

3598 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
3599 disabled state. None of the output parameters from this
3600 routine are valid. See [it_ia_info_t](#) for a description of
3601 the disabled state.3602 **ERRORS**

3603 None.

3604 **SEE ALSO**3605 [it_listen_free\(\)](#), [it_listen_create\(\)](#)

it_lmr_create()

3606

3607 **NAME**3608 `it_lmr_create` – create a Local Memory Region (LMR) and register with an Interface Adapter3609 **SYNOPSIS**

```

3610 #include <it_api.h>
3611
3612 it_status_t it_lmr_create(
3613     IN          it_pz_handle_t    pz_handle,
3614     IN          void              *addr,
3615     IN          it_length_t       length,
3616     IN          it_mem_priv_t     privs,
3617     IN          it_lmr_flag_t     flags,
3618     IN          uint32_t          shared_id,
3619     OUT         it_lmr_handle_t   *lmr_handle,
3620     IN OUT     it_rmr_context_t   *rmr_context
3621 );
3622
3623 typedef uint32_t it_rmr_context_t;
3624
3625 #ifdef IT_32BIT
3626     typedef uint32_t it_length_t; /* a 32-bit platform */
3627 #else
3628     typedef uint64_t it_length_t; /* a 64-bit platform */
3629 #endif
3630
3631 typedef enum {
3632     IT_PRIV_NONE           = 0x0001,
3633     IT_PRIV_READ_ONLY     = 0x0002,
3634     IT_PRIV_REMOTE_READ   = 0x0004,
3635     IT_PRIV_REMOTE_WRITE  = 0x0008,
3636     IT_PRIV_REMOTE        = 0x0010,
3637     IT_PRIV_ALL           = 0x0020,
3638     IT_PRIV_DEFAULT       = 0x0040
3639 } it_mem_priv_t;
3640
3641 typedef enum {
3642     IT_LMR_FLAG_NONE       = 0x0001,
3643     IT_LMR_FLAG_SHARED    = 0x0002,
3644     IT_LMR_FLAG_NONCOHERENT = 0x0004
3645 } it_lmr_flag_t;

```

3646 **DESCRIPTION**

3647	<i>pz_handle</i>	Protection Zone in which to create memory region.
3648	<i>addr</i>	Virtual address for start of memory region.
3649	<i>length</i>	Length of memory region in bytes.
3650	<i>privs</i>	Logical OR of access privilege flags for region.

3651 *flags* Logical OR of modifier flags.

3652 *shared_id* Optional identifier for sharing Interface Adapter translation
3653 resources.

3654 *lmr_handle* Returned Handle for created memory region.

3655 *rmr_context* Optionally returned Context allowing remote access to this memory.

3656 The *it_lmr_create* routine allows an Interface Adapter to access a contiguous Local Memory
3657 Region in a process' virtual address space; memory that is to be the Source or Destination of a
3658 DTO must first be registered using this call. The region starts at virtual address *addr* and
3659 extends for *length* bytes, and this address range must already be valid in the process's virtual
3660 address space. The Interface Adapter is implicitly identified by the *pz_handle* argument.
3661 Registering a memory range that does not correspond to physically backed memory, such as the
3662 non-cacheable I/O address space, may work on some Implementations but not others.
3663 Applications that rely on this behavior will not be portable. The range can refer to memory that
3664 is exclusive to the calling process, or is being shared with other processes. Some Interface
3665 Adapters may require a memory region to be locked in physical memory. Such locking, if
3666 required, will be performed by the *it_lmr_create* implementation and is not the Consumer's
3667 responsibility.

3668 The type of access granted is specified by the *privs* argument as the logical OR of one or more of
3669 the following flags. Unless otherwise noted, all combinations are allowed.

IT_PRIV_READ_ONLY	Specifies that only read accesses will be allowed to the region. If this flag is omitted, read and write access will be allowed. Note that this flag does not by itself enable remote access; one or more of the remote access flags must also be specified for that purpose.
IT_PRIV_REMOTE_READ	Grants remote read access to the IA, enabling the LMR to be used as an RMR, as a Source buffer for remote RDMA Read DTOs.
IT_PRIV_REMOTE_WRITE	Grants remote write access to the IA, enabling the LMR to be used as an RMR, as a Destination buffer for remote RDMA Write DTOs.
IT_PRIV_REMOTE	Grant remote read and write access to the IA.
IT_PRIV_ALL	Grant all types of access to the IA.

3670

3671 The special value IT_PRIV_DEFAULT or 0 may be used to grant default access, which includes
3672 local read and write access. The value IT_PRIV_NONE is invalid here. It is invalid to request
3673 remote write access in combination with IT_PRIV_READ_ONLY. It is not possible to grant
3674 access privileges to which a process is not already entitled. If the calling process does not have
3675 read or write access privileges to the memory region, then any attempt to grant those privileges
3676 to the Interface Adapter will cause *it_lmr_create* to fail.

3677 The *flags* argument is a logical OR of zero or more of the following options. The value
3678 IT_LMR_FLAG_NONE or 0 may be used to specify no options.

3679 IT_LMR_FLAG_SHARED
3680 This flag may be used to conserve finite Interface Adapter translation resources by sharing
3681 resources between multiple LMRs. The LMRs may have been created in the caller's process or in
3682 a different process. If set, then the Implementation will re-use resources from a matching LMR,
3683 which is defined as an existing LMR that was created using the same value of the *shared_id*
3684 argument, refers to the same physical memory pages as the new LMR, and has the same
3685 coherency mode. If any of these conditions are not met, the LMRs do not match. If a matching
3686 LMR is not found, then a new LMR is created and *shared_id* is associated with it. The value of
3687 *shared_id* is only an efficiency aid for the matching process and need not be unique. For
3688 example, if unrelated callers supply the same value for *shared_id*, matches for an LMR will still
3689 be found if they exist, with no false matches, but the search may take longer on some
3690 Implementations. If IT_LMR_FLAG_SHARED is not specified, then *shared_id* is ignored.

3691 IT_LMR_FLAG_NONCOHERENT
3692 Controls whether an LMR is created in coherent or non-coherent mode. Coherent mode is the
3693 default and is supported by all Implementations. Non-coherent mode is not supported by all
3694 Implementations, and IT_LMR_FLAG_NONCOHERENT is silently ignored on such
3695 Implementations.

3696
3697 Set IT_LMR_FLAG_NONCOHERENT to create an LMR in non-coherent mode. Non-coherent
3698 mode may yield higher throughput for large DTOs, but may also increase latency for small
3699 DTOs. The downside of requesting non-coherent mode is that the Consumer must synchronize
3700 between local and remote access to the memory region using the [it_lmr_sync_rdma_write](#) and
3701 [it_lmr_sync_rdma_read](#) calls.

3702 The coherency mode of an LMR is inherited by any RMR that is bound to it.

3703 An RMR Context allowing remote access to the memory region will be created if the *privs*
3704 argument includes either IT_PRIV_REMOTE_READ or IT_PRIV_REMOTE_WRITE. The
3705 Context will be returned in the location pointed to by *rmr_context* if the input value of
3706 *rmr_context* is not NULL. The returned *rmr_context* is only valid if *privs* includes remote
3707 privileges and the call returns successfully; otherwise it is undefined. The RMR Context may
3708 also be retrieved using [it_lmr_query](#). The *rmr_context* is returned in network byte order, and
3709 may be passed by value to any remote process that wishes to use the Context in DTOs that target
3710 the corresponding LMR. Modifying or destroying the LMR may revoke remote access using this
3711 RMR Context.

3712 The newly created LMR Handle is returned in the *lmr_handle* argument. A process may create
3713 multiple LMRs, and the address ranges of different LMRs may overlap. The Implementation
3714 may round the requested *addr* down and/or round the requested *length* up, and thus allow the
3715 Interface Adapter to access memory slightly outside the specified boundaries, but never beyond
3716 the IA pages that include the requested starting and ending addresses. The actual boundaries
3717 may be queried using [it_lmr_query](#). Note that if the *privs* argument enables remote access, then
3718 remote Consumers may also access memory slightly outside the requested boundaries. If this is
3719 undesirable, the Consumer should not enable remote access in this routine, but should instead
3720 create an RMR using [it_rmr_bind](#), which guarantees byte level registration granularity.

3721 After a memory range has been registered with the IA using [it_lmr_create](#), the Consumer should
3722 not call routines outside of the IT-API that would invalidate any part of the memory referred to
3723 by the LMR or revoke access privileges that were granted to IA at registration time. Disallowed
3724 operations include but are not limited to unmapping part of the range using [munmap](#), revoking

3725 privileges using *mprotect*, unlocking memory using *munlock*, truncating a file for file-backed
 3726 regions, etc. Violation of this rule may result in DTO failures, data corruption in the Consumer's
 3727 LMR, and/or program termination. These effects may extend to other Consumer processes if the
 3728 LMR is in shared memory. However, the Implementation must prevent any adverse effect on
 3729 unrelated processes that do not use this memory object.

3730 RETURN VALUE

3731 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 3732 below.

3733 IT_ERR_INVALID_PZ The Protection Zone Handle (*pz_handle*) was
 3734 invalid.

3735 IT_ERR_INVALID_PRIVS The requested memory privileges (*privs*) contained
 3736 an invalid flag.

3737 IT_ERR_INVALID_FLAGS The *flags* value was invalid.

3738 IT_ERR_FAULT Part or all of the supplied address range was invalid.

3739 IT_ERR_ACCESS The Consumer was not allowed to have the
 3740 requested memory privileges.

3741 IT_ERR_RESOURCES The requested operation failed due to insufficient
 3742 resources.

3743 IT_ERR_RESOURCE_LMR_LENGTH The underlying transport could not allocate an LMR
 3744 of the requested *length* at this time.

3745 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in
 3746 the disabled state. None of the output parameters
 3747 from this routine are valid. See [it_ia_info_t](#) for a
 3748 description of the disabled state.

3749 ERRORS

3750 None.

3751 APPLICATION USAGE

3752 Memory that is to be the Source or Destination of a DTO must first be registered using
 3753 *it_lmr_create*. The Consumer typically copies the returned *lmr_handle* to an [it_lmr_triplet_t](#)
 3754 structure that is used in DTO calls such as [it_post_send](#).

3755 If the LMR is created with flags that enable remote access, then the Consumer typically passes
 3756 the returned RMR Context to a remote peer using a DTO. The remote peer uses the RMR
 3757 Context in RDMA calls such as [it_post_rdma_write](#) that access memory within the range of the
 3758 LMR.

3759 Consumers can enable remote access more selectively over any portion of an LMR by creating
 3760 an RMR and binding the RMR to the desired region of the LMR using [it_rmr_bind](#). This
 3761 operation returns an RMR Context.

3762 **SEE ALSO**

3763 [*it_lmr_free\(\)*](#), [*it_lmr_query\(\)*](#), [*it_lmr_modify\(\)*](#), [*it_lmr_sync_rdma_read\(\)*](#),
3764 [*it_lmr_sync_rdma_write\(\)*](#)

it_lmr_free()

3765

3766 **NAME**3767 `it_lmr_free` – destroy a Local Memory Region3768 **SYNOPSIS**

```

3769 #include <it_api.h>
3770
3771 it_status_t it_lmr_free(
3772     IN          it_lmr_handle_t  lmr_handle
3773 );

```

3774 **DESCRIPTION**3775 `lmr_handle` Handle of Local Memory Region to be destroyed.

3776 The `it_lmr_free` routine destroys the Local Memory Region `lmr_handle`. On return, the Handle
 3777 `lmr_handle` may no longer be used. A Local Memory Region may not be destroyed if it has an
 3778 RMR bound to it; an attempt to do so will fail and the LMR will not be affected. `it_lmr_free`
 3779 does not invalidate the memory range represented by `lmr_handle`, and the caller may continue to
 3780 reference memory in this range for non-transport operations. If the memory range was locked in
 3781 physical memory as a side effect of the corresponding [it_lmr_create](#) call, then it will be
 3782 unlocked immediately if no portion of the range overlaps with the range of other non-freed
 3783 LMRs. Otherwise, the unlock operation may be deferred until the overlapping LMRs are
 3784 themselves freed. Note that these may include LMRs created by other Consumers if the range is
 3785 in shared memory.

3786 LMRs with memory ranges that overlap the range of `lmr_handle` are not affected by its
 3787 destruction. Outstanding DTO's, Bind, and Unbind operations that use an LMR that has been
 3788 destroyed may or may not complete successfully.

3789 **RETURN VALUE**

3790 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described
 3791 below.

3792 `IT_ERR_INVALID_LMR` The Local Memory Region Handle (`lmr_handle`)
 3793 was invalid.

3794 `IT_ERR_LMR_BUSY` The Local Memory Region was still referenced by a
 3795 Remote Memory Region.

3796 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in
 3797 the disabled state. None of the output parameters
 3798 from this routine are valid. See [it_ia_info_t](#) for a
 3799 description of the disabled state.

3800 **ERRORS**

3801 None.

3802 **SEE ALSO**3803 [it_lmr_create\(\)](#), [it_lmr_query\(\)](#), [it_lmr_modify\(\)](#)

it_lmr_modify()

3804

3805 **NAME**3806 `it_lmr_modify` – modify selected attributes of a Local Memory Region3807 **SYNOPSIS**

```

3808 #include <it_api.h>
3809
3810 it_status_t it_lmr_modify(
3811     IN          it_lmr_handle_t          lmr_handle,
3812     IN          it_lmr_param_mask_t     mask,
3813     IN          const it_lmr_param_t     *params
3814 );

```

3815 **DESCRIPTION**3816 *lmr_handle* Local Memory Region.3817 *mask* Logical OR of flags for specified parameters.3818 *params* Structure whose members contain the new parameter values.

3819 The *it_lmr_modify* routine changes selected attributes of the Local Memory Region *lmr_handle*.
 3820 Attributes to be modified are specified by flags in *mask*. New values for the attributes are
 3821 specified by the corresponding fields in the structure pointed to by *params*. Fields and their
 3822 corresponding flag values are shown in [it_lmr_param_t](#). Note that attributes represented by
 3823 fields of [it_lmr_param_t](#) that are not shown below can not be modified. The definition of each
 3824 field follows:

3825 *pz* The new Protection Zone Handle for the LMR.

3826 *privs* The new memory access privileges for the LMR. See [it_lmr_create](#)
 3827 for flag definitions and restrictions.

3828 On successful return, the previous RMR Context (if any) is invalidated. If remote access
 3829 privileges are specified in *privs*, then a new RMR Context is created and associated with the
 3830 LMR. The new RMR Context may be retrieved using [it_lmr_query](#).

3831 A Local Memory Region may not be modified if it is still referenced by bound Remote Memory
 3832 Regions; an attempt to do so will fail with an error return, and the LMR will not be modified or
 3833 affected.

3834 The Consumer should not modify an LMR whose LMR Handle or RMR Context is used in
 3835 outstanding DTO's, Bind, or Unbind operations. The Consumer must dequeue the Completion
 3836 Events for all such operations prior to modifying the LMR. If this rule is not followed, the
 3837 Outstanding Operations may fail and complete with an error status.

3838 If the modify operation fails because the caller was not allowed to have the requested memory
 3839 privileges, or fails due to insufficient resources, then the old RMR Context (if any) is
 3840 invalidated. The *lmr_handle* is also invalidated and may no longer be used in any calls. Any
 3841 resources that were associated with the LMR are freed in this case.

3842 **RETURN VALUE**

3843 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
3844 below.

3845 IT_ERR_INVALID_LMR The Local Memory Region Handle (*lmr_handle*) was
3846 invalid.

3847 IT_ERR_INVALID_PZ The Protection Zone Handle (*pz_handle*) was invalid.

3848 IT_ERR_INVALID_MASK The *mask* contained invalid flag values.

3849 IT_ERR_INVALID_PRIVS The requested memory privileges (*privs*) contained an
3850 invalid flag.

3851 IT_ERR_ACCESS The Consumer was not allowed to have the requested
3852 memory privileges.

3853 IT_ERR_LMR_BUSY The Local Memory Region was still referenced by a Remote
3854 Memory Region.

3855 IT_ERR_RESOURCES The requested operation failed due to insufficient resources.

3856 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
3857 disabled state. None of the output parameters from this
3858 routine are valid. See [it_ia_info_t](#) for a description of the
3859 disabled state.

3860 **ERRORS**

3861 None.

3862 **APPLICATION USAGE**

3863 Although `it_lmr_modify` can be used to change the remote access privileges for an LMR, this is
3864 a much more expensive operation than binding an RMR to an LMR using [it_rmr_bind](#).

3865 **SEE ALSO**

3866 [it_lmr_create\(\)](#), [it_lmr_free\(\)](#), [it_lmr_query\(\)](#), [it_lmr_param_t](#), [it_lmr_param_mask_t](#)

it_lmr_query()

3867

3868 **NAME**

3869

it_lmr_query – get attributes of a Local Memory Region3870 **SYNOPSIS**

3871

3872

3873

3874

3875

3876

3877

3878

3879

3880

3881

3882

3883

3884

3885

3886

3887

3888

3889

3890

3891

3892

3893

3894

3895

3896

3897

3898

3899

3900

3901

3902

3903

3904

```

#include <it_api.h>

it_status_t it_lmr_query(
    IN          it_lmr_handle_t      lmr_handle,
    IN          it_lmr_param_mask_t  mask,
    OUT         it_lmr_param_t       *params
);

typedef enum {
    IT_LMR_PARAM_ALL           = 0x000001,
    IT_LMR_PARAM_IA           = 0x000002,
    IT_LMR_PARAM_PZ           = 0x000004,
    IT_LMR_PARAM_ADDR         = 0x000008,
    IT_LMR_PARAM_LENGTH       = 0x000010,
    IT_LMR_PARAM_MEM_PRIV     = 0x000020,
    IT_LMR_PARAM_FLAG         = 0x000040,
    IT_LMR_PARAM_SHARED_ID    = 0x000080,
    IT_LMR_PARAM_RMR_CONTEXT  = 0x000100,
    IT_LMR_PARAM_ACTUAL_ADDR  = 0x000200,
    IT_LMR_PARAM_ACTUAL_LENGTH = 0x000400
} it_lmr_param_mask_t;

typedef struct {
    it_ia_handle_t      ia;           /* IT_LMR_PARAM_IA */
    it_pz_handle_t     pz;           /* IT_LMR_PARAM_PZ */
    void                *addr;       /* IT_LMR_PARAM_ADDR */
    it_length_t        length;       /* IT_LMR_PARAM_LENGTH */
    it_mem_priv_t      privs;        /* IT_LMR_PARAM_MEM_PRIV */
    it_lmr_flag_t      flags;        /* IT_LMR_PARAM_FLAG */
    uint32_t           shared_id;     /* IT_LMR_PARAM_SHARED_ID */
    it_rmr_context_t   rmr_context;   /* IT_LMR_PARAM_RMR_CONTEXT */
    void                *actual_addr; /* IT_LMR_PARAM_ACTUAL_ADDR */
    it_length_t        actual_length; /* IT_LMR_PARAM_ACTUAL_LENGTH */
} it_lmr_param_t;

```

3905 **DESCRIPTION**

3906

lmr_handle Local Memory Region.

3907

mask Logical OR of flags for desired parameters.

3908

params Structure whose members are written with the desired parameters.

3909

3910

3911

3912

The *it_lmr_query* routine returns the desired attributes of the Local Memory Region *lmr_handle* in the structure pointed to by *params*. On return, each field of *params* is only valid if the corresponding flag as shown in the Synopsis is set in the *mask* argument. The *mask* value `IT_LMR_PARAM_ALL` causes all fields to be returned.

3913 The definition of each field of *params* follows:

3914	<i>ia</i>	The Interface Adapter Handle specified to create the LMR.
3915	<i>pz</i>	The Protection Zone Handle specified to create the LMR.
3916	<i>addr</i>	The requested starting address of the LMR.
3917	<i>length</i>	The requested length in bytes of the LMR.
3918	<i>privs</i>	The memory access privileges specified to create the LMR.
3919	<i>flags</i>	The flags specified to create the LMR.
3920	<i>shared_id</i>	The <i>shared_id</i> specified to create the LMR, if <i>flags</i> included
3921		IT_LMR_FLAG_SHARED. Otherwise, undefined.
3922	<i>rmr_context</i>	The RMR Context associated with the LMR, or undefined if <i>privs</i>
3923		does not include remote access. Returned in network byte order.
3924	<i>actual_addr</i>	The actual starting address for which bounds checking is done for
3925		data transfers.
3926	<i>actual_length</i>	The actual length for which bounds checking is done for data
3927		transfers.

3928 **RETURN VALUE**

3929 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
3930 below.

3931	IT_ERR_INVALID_LMR	The Local Memory Region Handle (<i>lmr_handle</i>) was
3932		invalid.
3933	IT_ERR_INVALID_MASK	The <i>mask</i> contained invalid flag values.
3934	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the
3935		disabled state. None of the output parameters from this
3936		routine are valid. See it_ia_info_t for a description of the
3937		disabled state.

3938 **ERRORS**

3939 None.

3940 **SEE ALSO**

3941 [it_lmr_create\(\)](#), [it_lmr_modify\(\)](#), [it_lmr_free\(\)](#)

it_lmr_sync_rdma_read()

3942

3943 **NAME**

3944

it_lmr_sync_rdma_read – make memory changes visible to an incoming RDMA Read operation

3945 **SYNOPSIS**

3946

```
#include <it_api.h>
```

3947

3948

```
it_status_t it_lmr_sync_rdma_read(
```

3949

```
    IN      const      it_lmr_triplet_t  *local_segments,
```

3950

```
    IN      size_t      num_segments
```

3951

```
);
```

3952 **DESCRIPTION**

3953

local_segments Array of buffer segments.

3954

num_segments Number of segments in the array.

3955

The *it_lmr_sync_rdma_read* routine is needed if and only if an LMR was created in non-coherent mode using IT_LMR_FLAG_NONCOHERENT.

3956

3957

If a Local Memory Region is created in non-coherent mode, then the Consumer must call *it_lmr_sync_rdma_read* after modifying data in a memory range in this region that will be the target of an *incoming* RDMA Read operation. *it_lmr_sync_rdma_read* must be called after the Consumer has modified the memory range but before the RDMA Read operation starts, and the memory range that will be accessed by the RDMA Read must be supplied by the caller in the *local_segments* array. After this call returns, the RDMA Read operation may safely see the modified contents of the memory range. It is permissible to batch synchronizations for multiple RDMA Read operations in a single call, by passing a *local_segments* array that includes all modified memory ranges. The *local_segments* entries need not contain the same LMR, and need not be in the same Protection Zone.

3958

3959

3960

3961

3962

3963

3964

3965

3966

3967

If an RDMA Read operation on an LMR created in non-coherent mode attempts to read from a memory range that is not properly synchronized using *it_lmr_sync_rdma_read*, the returned contents are undefined.

3968

3969

3970 **RETURN VALUE**

3971

This call is a no-op and always returns successfully if the Implementation does not support non-coherent mode, or if none of the LMRs in *local_segments* were created using the IT_LMR_FLAG_NONCOHERENT flag.

3972

3973

3974

A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

3975

3976

IT_ERR_RANGE

The address range for a local segment fell outside the boundaries of the corresponding Local Memory Region and the Local Memory Region was created in non-coherent mode.

3977

3978

3979

3980

IT_ERR_IA_CATASTROPHE

The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this

3981

3982 routine are valid. See [it_ia_info_t](#) for a description of the
3983 disabled state.

3984

3985 **ERRORS**

3986 None.

3987 **APPLICATION USAGE**

3988 Determining when an RDMA Read will start and what memory range it will read is the
3989 Consumer's responsibility. One possibility is to have the Consumer that is modifying memory to
3990 call [it_lmr_sync_rdma_read](#) and then post a Send DTO message that identifies the range in the
3991 body of the Send. The Consumer wishing to do the RDMA Read can receive this message and
3992 thus know when it is safe to initiate the RDMA Read operation.

3993 **SEE ALSO**

3994 [it_lmr_create\(\)](#), [it_lmr_sync_rdma_write\(\)](#), [it_lmr_triplet_t](#)

it_lmr_sync_rdma_write()

3995

3996 **NAME**

3997 `it_lmr_sync_rdma_write` – make effects of an incoming RDMA Write operation visible to
 3998 Consumer

3999 **SYNOPSIS**

```
4000 #include <it_api.h>
4001
4002 it_status_t it_lmr_sync_rdma_write(
4003     IN      const      it_lmr_triplet_t  *local_segments,
4004     IN      size_t      num_segments
4005 );
```

4006 **DESCRIPTION**

4007 *local_segments* Array of buffer segments.

4008 *num_segments* Number of segments in the array.

4009 The *it_lmr_sync_rdma_write* routine is needed if and only if an LMR was created in non-
 4010 coherent mode using `IT_LMR_FLAG_NONCOHERENT`.

4011 If a Local Memory Region is created in non-coherent mode, then the Consumer must call
 4012 *it_lmr_sync_rdma_write* before reading data from a memory range in this region that was the
 4013 target of an *incoming* RDMA Write operation. *it_lmr_sync_rdma_write* must be called after the
 4014 RDMA Write operation completes, and the memory range that was modified by the RDMA
 4015 Write must be supplied by the caller in the *local_segments* array. After this call returns, the
 4016 Consumer may safely see the modified contents of the memory range. It is permissible to batch
 4017 synchronizations of multiple RDMA Write operations in a single call, by passing a
 4018 *local_segments* array that includes all modified memory ranges. The *local_segments* entries
 4019 need not contain the same LMR, and need not be in the same Protection Zone.

4020 The Consumer must also use *it_lmr_sync_rdma_write* when performing local writes to a
 4021 memory range that was or will be the target of incoming RDMA Writes. After performing the
 4022 local write, the Consumer must call *it_lmr_sync_rdma_write* before the RDMA Write is
 4023 initiated. Conversely, after an RDMA Write completes, the Consumer must call
 4024 *it_lmr_sync_rdma_write* before performing a local write to the same range.

4025 If the Consumer attempts to read from a memory range in an LMR that was created in non-
 4026 coherent mode, without properly synchronizing using *it_lmr_sync_rdma_write*, the returned
 4027 contents are undefined. If the Consumer attempts to write to a memory range without properly
 4028 synchronizing, the contents of the memory range become undefined.

4029 **RETURN VALUE**

4030 This call is a no-op and always returns successfully if the Implementation does not support non-
 4031 coherent mode, or if none of the LMRs in *local_segments* were created using the
 4032 `IT_LMR_FLAG_NONCOHERENT` flag.

4033 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described
 4034 below.

4035	IT_ERR_RANGE	The address range for a local segment fell outside the
4036		boundaries of the corresponding Local Memory Region and
4037		the Local Memory Region was created in non-coherent
4038		mode.
4039	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the
4040		disabled state. None of the output parameters from this
4041		routine are valid. See it_ia_info_t for a description of the
4042		disabled state.
4043	ERRORS	
4044	None.	
4045	APPLICATION USAGE	
4046		Determining when an RDMA Write completes and determining which memory range was
4047		modified is the Consumer's responsibility. One possibility is for the RDMA Write initiator to
4048		post a Send DTO message after each RDMA Write that identifies the range in the body of the
4049		Send. The Consumer at the target of the RDMA Write can receive the message and thus know
4050		when and how to call <i>it_lmr_sync_rdma_write</i> .
4051	SEE ALSO	
4052		it_lmr_create() , it_lmr_sync_rdma_read() , it_lmr_triplet_t

it_make_rdma_addr()

4053

4054 **NAME**4055 `it_make_rdma_addr` – make a platform independent RDMA address4056 **SYNOPSIS**

```

4057 #include <it_api.h>
4058
4059 typedef uint64_t it_rdma_addr_t;
4060
4061 it_rdma_addr_t it_make_rdma_addr(
4062                 void *addr
4063 );

```

4064 **DESCRIPTION**4065 `addr` Local address.

4066 The `it_make_rdma_addr` routine takes a local address `addr` that may be the target of a remote
 4067 operation, and returns a 64-bit platform independent representation of that address in network
 4068 byte order called an RDMA address. A network peer may use this RDMA address in RDMA
 4069 Read and Write operations. `it_make_rdma_addr` performs no validity checking on `addr`, so `addr`
 4070 is not required to lie within a currently registered LMR when `it_make_rdma_addr` is called.

4071 **RETURN VALUE**

4072 This function always succeeds and returns a 64-bit RDMA address.

4073 **ERRORS**

4074 None.

4075 **APPLICATION USAGE**

4076 The returned RDMA address must be communicated to a network peer in order to be used in
 4077 RDMA operations. The Consumer is responsible for performing this communication.

4078 Because the RDMA address is in network byte order, a Consumer wishing to perform address
 4079 arithmetic must first convert it to host byte order, which may be done using the [it_ntoh64](#)
 4080 function. Derived addresses must be converted back to network byte order using [it_hton64](#)
 4081 before being used in RDMA operations.

4082 **SEE ALSO**4083 [it_post_rdma_write\(\)](#), [it_ntoh64](#), [it_hton64](#)

it_post_rdma_read()

4084
 4085 **NAME**
 4086 `it_post_rdma_read` – post an RDMA Read DTO to a Reliable Connected Endpoint

SYNOPSIS

```
4087 #include <it_api.h>
4088
4089
4090 it_status_t it_post_rdma_read (
4091     IN          it_ep_handle_t          ep_handle,
4092     IN          const it_lmr_triplet_t  *local_segments,
4093     IN          size_t                  num_segments,
4094     IN          it_dto_cookie_t        cookie,
4095     IN          it_dto_flags_t         dto_flags,
4096     IN          it_rdma_addr_t         rdma_addr,
4097     IN          it_rmr_context_t       rmr_context
4098 );
```

DESCRIPTION

4100 *ep_handle* Handle for the Endpoint – the local side of the Connection.

4101 *local_segments* Vector of [it_lmr_triplet_t](#) data structures that specifies the local
 4102 buffer where data should be deposited. Can be NULL for a zero-
 4103 sized message.

4104 *num_segments* Number of [it_lmr_triplet_t](#) data structures in *local_segments*. Can be
 4105 zero for a zero-sized message.

4106 *cookie* Consumer-provided cookie that is returned to the Consumer in the
 4107 Completion Event corresponding to the RDMA Read.

4108 *dto_flags* Flags for posted RDMA Read.

4109 *rdma_addr* The starting address of the remote buffer to read from.

4110 *rmr_context* The RMR Context for the remote buffer to read from.

4111 *it_post_rdma_read* requests a transfer of the data from a remote buffer into the local buffer
 4112 specified by *num_segments* and *local_segments* over the reliable Connection of the *ep_handle*
 4113 Endpoint. The size of the data transferred is specified by the sum of sizes of the *local_segments*.
 4114 A zero-sized message may be transferred over the Connection. The *it_post_rdma_read* is only
 4115 applicable to reliable Connections.

4116 *num_segments* specifies the number of segments in the *local_segments* vector.

4117 The Implementation allows the buffer segments described by the *local_segments* to overlap but
 4118 the resulting content of the local buffer is undefined.

4119 Once a successful Completion Event has been generated for the RDMA Read, the order of the
 4120 bytes in the local buffer specified by *num_segments* and *local_segments* corresponds to the order
 4121 defined by the remote buffer unless there is local overlap. If there is local overlap, the byte order

4122 in the local buffer is undefined. Prior to the Completion Event being generated, the content of
4123 the local buffer is implementation-dependent.

4124 A Consumer shall not modify the local buffer specified by *num_segments* and *local_segments*
4125 until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the
4126 Implementation and the underlying transport is not defined. A Consumer does get back
4127 ownership of the *num_segments* and *local_segments* arguments (but not the local buffer
4128 identified by them) when *it_post_rdma_read* returns and is free to use the *num_segments* and
4129 *local_segments* arguments for other calls, or to modify them, or to destroy them.

4130 The completion of the posted RDMA Read is reported asynchronously to the Consumer
4131 according to the rules defined in [it_dto_flags_t](#). Any generated DTO Completion Event
4132 manifests on the EVD associated with the Endpoint. See [it_ep_rc_create](#), [it_dto_status_t](#) and
4133 [it_dto_events](#). A completion status other than IT_DTO_SUCCESS will break the Connection. If
4134 the reported status of the completion DTO Event corresponding to the posted RDMA Read is not
4135 IT_DTO_SUCCESS, the content of the local buffer is not defined.

4136 The *dto_flags* value is used as specified in [it_dto_flags_t](#).

4137 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4138 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4139 to the Consumer in the Completion Event for the posted RDMA Read. See [it_dto_cookie_t](#).

4140 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4141 Operations.

4142 Data corruption (at local and/or at remote) and data loss will be reported to the Consumer in the
4143 DTO Completion Event. These conditions will cause the Connection to be broken. Once the
4144 Connection is broken, all outstanding and in-progress operations on the Connection will
4145 complete with a failure reported in their corresponding DTO Completion Events.

4146 The Implementation ensures that the RDMA Read in no way corresponds to any Send or Recv
4147 Data Transfer Operations over the same Connection.

4148 The Implementation ensures that all RDMA Read operations start and complete in the order
4149 posted.

4150 Send and RDMA DTOs following an RDMA Read DTO may start during execution of the
4151 RDMA Read DTO and complete before the RDMA Read completes. To ensure
4152 deterministically that subsequent Sends and RDMA DTOs following an RDMA Read DTO
4153 do start after the RDMA Read completes, specify the IT_BARRIER_FENCE_FLAG on the
4154 DTOs following the RDMA Read.

4155 The Implementation ensures that all data for a given RDMA Read operation is transferred from
4156 the remote buffers and into the local buffers before a RDMA Read completion is generated with
4157 the status of IT_DTO_SUCCESS.

4158 Posting to an Endpoint that is not in the IT_EP_STATE_CONNECTED or IT_EP_STATE_
4159 NONOPERATIONAL state will return the IT_ERR_INVALID_EP_STATE error.

4160 Any posting to an Endpoint that is in the IT_EP_STATE_NONOPERATIONAL state will be
4161 flushed with completion status ([it_dto_status_t](#)) set to IT_DTO_ERR_FLUSHED.

4162 IT_SUCCESS returned from the *it_post_rdma_read* call means that the RDMA Read operation
4163 was successfully posted to the transport layer in use.

4164 **RETURN VALUE**

4165 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4166 IT_ERR_TOO_MANY_POSTS The operation failed due to an overflow of a
4167 work queue.

4168 IT_ERR_INVALID_EP The Endpoint Handle (*ep_handle*) was
4169 invalid.

4170 IT_ERR_INVALID_EP_STATE The Endpoint was not in the proper state for
4171 the attempted operation.

4172 IT_ERR_INVALID_EP_TYPE The attempted operation was invalid for the
4173 Service Type of the Endpoint.

4174 IT_ERR_INVALID_DTO_FLAGS The Data Transfer Operation flags
4175 (*dto_flags*) value was invalid.

4176 IT_ERR_INVALID_NUM_SEGMENTS The requested number of segments
4177 (*num_segments*) was larger than the
4178 Endpoint supports.

4179 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error
4180 and is in the disabled state. None of the
4181 output parameters from this routine are
4182 valid. See [it ia info t](#) for a description of
4183 the disabled state

4184 **ERRORS**

4185 None.

4186 **APPLICATION USAGE**

4187 This function is used after a Connection has been established to transfer data from a Consumer-
4188 specified remote buffer to a Consumer-specified local buffer.

4189 The Consumer should use unique identifiers for *cookie* if they desire to identify each DTO. If
4190 the Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.

4191 For best RDMA Read operation performance, the Consumer should align each buffer segment of
4192 *local_segments* to the *dto_alignment_hint* in the IA attributes obtained via [it ia query](#).

4193 **SEE ALSO**

4194 [it_post_send\(\)](#), [it_post_sendto\(\)](#), [it_post_rcv\(\)](#), [it_post_rcvfrom\(\)](#), [it_post_rdma_write\(\)](#),
4195 [it_rmr_bind\(\)](#), [it_rmr_unbind\(\)](#), [it_dto_status_t](#), [it_dto_events](#), [it_dto_flags_t](#), [it_ep_rc_create\(\)](#),
4196 [it_lmr_triplet_t](#), [it_ia_query\(\)](#), [it_dto_cookie_t](#), [it_ia_info_t](#)

it_post_rdma_write()

4197

4198 **NAME**4199 `it_post_rdma_write` – post an RDMA Write DTO to a connected Endpoint4200 **SYNOPSIS**

```

4201 #include <it_api.h>
4202
4203 it_status_t it_post_rdma_write (
4204     IN          it_ep_handle_t          ep_handle,
4205     IN          const it_lmr_triplet_t   *local_segments,
4206     IN          size_t                  num_segments,
4207     IN          it_dto_cookie_t         cookie,
4208     IN          it_dto_flags_t          dto_flags,
4209     IN          it_rdma_addr_t          rdma_addr,
4210     IN          it_rmr_context_t        rmr_context
4211 );

```

4212 **DESCRIPTION**

4213 *ep_handle* Handle for the Endpoint – the local side of the Connection.

4214 *local_segments* Vector of [it_lmr_triplet_t](#) data structures that specifies the local
4215 buffer that contains data to be transferred. Can be NULL for a zero-
4216 sized message.

4217 *num_segment* Number of [it_lmr_triplet_t](#) data structures in *local_segments*. Can be
4218 zero for a zero-sized message.

4219 *cookie* Consumer-provided cookie that is returned to the Consumer in the
4220 Completion Event corresponding to the RDMA Write.

4221 *dto_flags* Flags for posted RDMA Write.

4222 *rdma_addr* The starting address of the remote buffer to write to.

4223 *rmr_context* The RMR Context for the remote buffer to write to.

4224 *it_post_rdma_write* requests a transfer of all the data from *local_segments* into a remote buffer
4225 on the other side of the Connection. The Connection is implemented on a reliable transport. A
4226 zero-sized message may be transferred over the Connection.

4227 *num_segments* specifies the number of segments in the *local_segments* vector. The Completion
4228 Event for the *it_post_rdma_write* call indicates to the Consumer that the local buffer is under
4229 Consumer control again. It does not guarantee that the contents of the local buffer have been
4230 successfully delivered into the memory of the remote Consumer. However, once the contents of
4231 the local buffer reach the remote Consumer memory, the order of the bytes in the remote
4232 memory corresponds to the order defined by the *local_segments*. Prior to the Completion Event
4233 being generated, the content of the remote buffer is implementation-dependent.

4234 A Consumer should not modify the local buffer specified by *num_segments* and *local_segments*
4235 until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the
4236 Implementation and the underlying transport is not defined. A Consumer does get back

4237 ownership of the *num_segments* and *local_segments* arguments (but not the local buffer
4238 identified by them) when *it_post_rdma_write* returns and is free to use the *num_segments* and
4239 *local_segments* arguments for other calls, to modify them, or to destroy them.

4240 The completion of the posted RDMA Write is reported asynchronously to the Consumer
4241 according to the rules defined in [it_dto_flags_t](#). Any generated DTO Completion Event
4242 manifests on the EVD associated with the Endpoint. See [it_ep_rc_create](#), [it_dto_status_t](#) and
4243 [it_dto_events](#). A completion status other than IT_DTO_SUCCESS will break the Connection. If
4244 the reported status of the completion DTO Event corresponding to the posted RDMA Write
4245 DTO is not IT_DTO_SUCCESS, then the contents of the remote buffer are not defined.

4246 The *dto_flags* value is used as specified in [it_dto_flags_t](#).

4247 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4248 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4249 to the Consumer in the Completion Event for the posted RDMA Write.

4250 The buffer segments described by *local_segments* can overlap.

4251 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4252 Operations.

4253 Data corruption (at local and/or at remote) and data loss will be reported to the Consumer in the
4254 DTO Completion Event. These conditions will cause the Connection to be broken. Once the
4255 Connection is broken, all outstanding and in-progress operations on the Connection will
4256 complete with an error status.

4257 The Implementation ensures that the RDMA Write in no way corresponds to any Receive Data
4258 Transfer Operations over the same Connection.

4259 The Implementation ensures that all RDMA Write operations start and complete in the order
4260 posted.

4261 If the RDMA Write operation exceeds the bounds of the remote buffer, the completion status
4262 will be IT_DTO_ERR_REMOTE_ACCESS.

4263 The Implementation ensures that each RDMA Write Data Transfer Operation posted on a
4264 Connection prior to a Send Data Transfer Operation posted to the same Connection has its
4265 complete data payload delivered to the remote memory prior to the completion of the Receive
4266 Data Transfer Operation at the remote side matching that Send.

4267 Send and RDMA DTOs following an RDMA Read DTO may start during execution of the
4268 RDMA Read DTO and complete before the RDMA Read completes. To ensure that Sends and
4269 RDMA DTOs following an RDMA Read DTO do start after the RDMA Read completes,
4270 specify the IT_BARRIER_FENCE_FLAG on the DTOs following the RDMA Read.

4271 Posting to an Endpoint that is not in the IT_EP_STATE_CONNECTED or IT_EP_STATE_
4272 NONOPERATIONAL state will return the IT_ERR_INVALID_EP_STATE error.

4273 Any posting to an Endpoint that is in the IT_EP_STATE_NONOPERATIONAL state will be
4274 flushed with completion status ([it_dto_status_t](#)) set to IT_DTO_ERR_FLUSHED.

4275 IT_SUCCESS returned from the *it_post_rdma_write* call means that the RDMA Write operation
4276 was successfully posted to the transport layer in use.

4277 **EXTENDED DESCRIPTION**

4278 See [it_lmr_sync_rdma_write](#) for a discussion of ramifications of RDMA Write use on non-
4279 coherent systems.

4280 **RETURN VALUE**

4281 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4282 IT_ERR_TOO_MANY_POSTS The operation failed due to an overflow of a
4283 work queue.

4284 IT_ERR_INVALID_EP The Endpoint Handle (*ep_handle*) was
4285 invalid.

4286 IT_ERR_INVALID_EP_STATE The Endpoint was not in the proper state for
4287 the attempted operation.

4288 IT_ERR_INVALID_EP_TYPE The attempted operation was invalid for the
4289 Service Type of the Endpoint.

4290 IT_ERR_INVALID_DTO_FLAGS The Data Transfer Operation flags
4291 (*dto_flags*) value was invalid.

4292 IT_ERR_INVALID_NUM_SEGMENTS The requested number of segments
4293 (*num_segments*) was larger than the
4294 Endpoint supports.

4295 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error
4296 and is in the disabled state. None of the
4297 output parameters from this routine are
4298 valid. See [it_ia_info_t](#) for a description of
4299 the disabled state.

4300 **ERRORS**

4301 None.

4302 **APPLICATION USAGE**

4303 This function is used after Connection has been established to transfer data from Consumer
4304 specified local buffer to a remote buffer on the other side of the Connection.

4305 The Consumer should use unique identifiers for *cookie* if they desire to identify each DTO. If
4306 the Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.

4307 For best RDMA Write operation performance, the Consumer should align each buffer segment
4308 of *local_segments* to the *dto_alignment_hint* in the IA attributes obtained via [it_ia_query](#).

4309 There are a variety of ways to guarantee the delivery of a local buffer via RDMA Write into the
4310 memory of the remote Consumer. One way would be for the Consumer to send a message over
4311 the Connection after the RDMA Write had completed, and then wait for the remote peer to reply
4312 to that message. By requiring the remote Consumer to reap the Receive Completion for the Send
4313 from the local Consumer, the payload of the RDMA Write is delivered into the remote memory.

4314 **SEE ALSO**

4315 [*it_post_send\(\)*](#), [*it_post_sendto\(\)*](#), [*it_post_rcv\(\)*](#), [*it_post_rcvfrom\(\)*](#), [*it_post_rdma_read\(\)*](#),
4316 [*it_rmr_bind\(\)*](#), [*it_rmr_unbind\(\)*](#), [*it_dto_status t*](#), [*it_dto_events*](#), [*it_dto_flags t*](#), [*it_ep_rc_create\(\)*](#),
4317 [*it_lmr_triplet t*](#), [*it_ia_query\(\)*](#), [*it_dto_cookie t*](#), [*it_ia_info t*](#)

it_post_recv()

4318

4319 **NAME**

4320

it_post_recv – post a Receive DTO to a connected Endpoint

4321 **SYNOPSIS**

4322

```
#include <it_api.h>
```

4323

4324

```
it_status_t it_post_recv(
```

4325

```
    IN          it_ep_handle_t    ep_handle,
```

4326

```
    IN          const it_lmr_triplet_t *local_segments,
```

4327

```
    IN          size_t           num_segments,
```

4328

```
    IN          it_dto_cookie_t  cookie,
```

4329

```
    IN          it_dto_flags_t   dto_flags
```

4330

```
);
```

4331 **DESCRIPTION**

4332

ep_handle

Handle for the Endpoint – the local side of the Connection.

4333

*local_segments*Vector of *it_lmr_triplet_t* data structures that specifies the local buffer to contain the data to be received. Can be NULL for a zero-sized message.

4334

4335

4336

*num_segments*Number of *it_lmr_triplet_t* data structures in *local_segments*. Can be zero for a zero-sized message.

4337

4338

cookie

Consumer-provided cookie that is returned to the Consumer in the Completion Event corresponding to the Receive.

4339

4340

dto_flags

Flags for posted Receive.

4341

it_post_recv supplies the local Receive buffer specified by *local_segments* and *num_segments* to the *ep_handle* Endpoint. A single incoming message from a single corresponding Send over the Connection is deposited into the local Receive buffer. Zero-sized messages are supported and will consume a posted Receive.

4342

4343

4344

4345

num_segments specifies the number of segments in the *local_segments* vector.

4346

The Implementation allows the buffer segments described by the *local_segments* vector to overlap but the resulting Receive behavior is undefined.

4347

4348

Once a successful Completion Event has been generated for the Receive, the order of the bytes in the local buffer specified by *local_segments* and *num_segments* corresponds to the order defined by the *local_segments* of the corresponding Send operation unless there is overlap among the segments of the local Receive buffer. If there is such an overlap, the content of the local Receive buffer after the Completion Event has been generated is undefined. Prior to the Completion Event being generated, the content of the local buffer is implementation-dependent.

4349

4350

4351

4352

4353

4354

A Consumer shall not modify the local buffer specified by *num_segments* and *local_segments* until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Implementation and the underlying transport is not defined. A Consumer does get back ownership of the *num_segments* and *local_segments* arguments (but not the local buffer

4355

4356

4357

4358 identified by them) when *it_post_recv* returns and is free to use the *num_segments* and
4359 *local_segments* arguments for other calls, to modify them, or to destroy them.

4360 The completion of the posted Receive is reported asynchronously to the Consumer according to
4361 the rules defined in [it_dto_flags_t](#). Exactly one Receive DTO Completion Event is always
4362 generated and manifests on the EVD associated with the Endpoint. See [it_ep_rc_create](#),
4363 [it_dto_status_t](#) and [it_dto_events](#). A completion status other than IT_DTO_SUCCESS will break
4364 the Connection. If the reported *dto_status* of the Completion DTO Event corresponding to the
4365 posted Receive DTO is not IT_DTO_SUCCESS, the content of the local buffer is not defined.

4366 The *dto_flags* value is used as specified in [it_dto_flags_t](#).

4367 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4368 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4369 to the Consumer in the Completion Event for the posted Receive.

4370 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4371 Operations.

4372 Data corruption (at local and/or at remote) and data loss will be reported to the Consumer in the
4373 DTO Completion Event. These conditions will cause the Connection to be broken. Once the
4374 Connection is broken, all outstanding and in-progress operations on the Connection will
4375 complete with an error status.

4376 The Implementation ensures that each Receive corresponds to one and only one remote Send and
4377 in no way corresponds to any RDMA Read or RDMA Write Data Transfer Operations over the
4378 same Connection.

4379 The Implementation ensures that a RDMA Write DTO from the remote connected Endpoint
4380 preceding a Send from the remote connected Endpoint has fully delivered its payload prior to the
4381 completion of the Receive corresponding to the Send.

4382 The Implementation ensures that all Receives start and complete in the order posted.

4383 Receive Data Transfer Operations on a Connection are completed in the order of posting of their
4384 corresponding Sends at the remote Endpoint. Since Sends start and complete in order, the Recvs
4385 complete in order.

4386 There is no order relationship between completions of Receive Data Transfer Operations and all
4387 other Data Transfer Operations (including RMR operations) on the same Connection.

4388 The Implementation ensures that all data from a given Send operation is transferred from the
4389 remote buffers and into the local buffers before a Receive completion is generated with
4390 IT_DTO_SUCCESS.

4391 Any posting to an Endpoint that is in the IT_EP_STATE_NONOPERATIONAL state will be
4392 flushed with completion status ([it_dto_status_t](#)) set to IT_DTO_ERR_FLUSHED.

4393 IT_SUCCESS returned from the *it_post_recv* call means that the Receive operation was
4394 successfully posted to the transport layer for use.

4395 **RETURN VALUE**

4396 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4397	IT_ERR_TOO_MANY_POSTS	The operation failed due to an overflow of a work
4398		queue.

4399	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
4400 4401	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
4402 4403	IT_ERR_INVALID_DTO_FLAGS	The Data Transfer Operation flags (<i>dto_flags</i>) value was invalid.
4404 4405 4406	IT_ERR_INVALID_NUM_SEGMENTS	The requested number of segments (<i>num_segments</i>) was larger than the Endpoint supports.
4407 4408 4409 4410	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
4411	ERRORS	
4412	None.	
4413	APPLICATION USAGE	
4414 4415 4416		This function is used after a Connection has been established to transfer data into a Consumer-specified local buffer from a buffer specified by the corresponding Send operation on the other side of the Connection.
4417 4418		The Consumer should use unique identifiers for <i>cookie</i> if they desire to identify each DTO. If the Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.
4419 4420		For best Receive operation performance, the Consumer should align each buffer segment of <i>local_segments</i> to the <i>dto_alignment_hint</i> in the IA attributes obtained via it_ia_query .
4421	SEE ALSO	
4422 4423 4424		it_post_send() , it_post_sendto() , it_post_rcvfrom() , it_post_rdma_read() , it_post_rdma_write() , it_rmr_bind() , it_rmr_unbind() , it_dto_status_t , it_dto_events , it_dto_flags_t , it_ep_rc_create() , it_lmr_triplet_t , it_ia_query() , it_dto_cookie_t , it_ia_info_t

it_post_recvfrom()

4425

4426 **NAME**

4427

it_post_recvfrom – post a Receive DTO to a datagram Endpoint

4428 **SYNOPSIS**

4429

```
#include <it_api.h>
```

4430

4431

```
it_status_t it_post_recvfrom(
```

4432

```
    IN
```

```
    it_ep_handle_t
```

```
    ep_handle,
```

4433

```
    IN
```

```
    const
```

```
    it_lmr_triplet_t
```

```
    *local_segments,
```

4434

```
    IN
```

```
    size_t
```

```
    num_segments,
```

4435

```
    IN
```

```
    it_dto_cookie_t
```

```
    cookie,
```

4436

```
    IN
```

```
    it_dto_flags_t
```

```
    dto_flags
```

4437

```
);
```

4438 **DESCRIPTION**

4439

ep_handle

Handle for the local datagram Endpoint.

4440

local_segments

Vector of [it_lmr_triplet_t](#) data structures that specifies the local buffer that will contain the received data. Local buffer must be at least 40 bytes.

4441

4442

4443

num_segments

Number of [it_lmr_triplet_t](#) data structures in *local_segments*. Must be at least one.

4444

4445

cookie

Consumer-provided cookie that is returned to the Consumer in the Completion Event corresponding to the Receive.

4446

4447

dto_flags

Flags for posted Receive.

4448

it_post_recvfrom supplies the local Receive buffer specified by *local_segments* and *num_segments* to the *ep_handle* datagram Endpoint. A single incoming message from a single corresponding Send from a remote datagram Endpoint is deposited into the local Receive buffer.

4449

4450

4451

The first 40 bytes of the Consumer's local buffer are reserved for Implementation use. For a zero-sized message, the minimum size for the local buffer in *local_segments* is 40 bytes. To accommodate a larger message, the Consumer should provide a local buffer in *local_segments* at least 40 bytes bigger than their expected incoming message size. See [it_dto_events](#) for more details.

4452

4453

4454

4455

4456

num_segments specifies the number of segments in the *local_segments* vector.

4457

The Implementation allows the buffer segments described by the *local_segments* vector to overlap but the resulting Receive behavior is undefined.

4458

4459

Once a successful Completion Event has been generated for the Receive, the order of the bytes in the local buffer specified by *local_segments* and *num_segments* corresponds to the order defined by the *local_segments* of the corresponding Send operation unless there is overlap among the segments of the local Receive buffer. If there is such an overlap, the content of the local buffer after the Completion Event has been generated is undefined. Prior to the Completion

4460

4461

4462

4463

4464 Event being generated, the content of the local buffer is implementation-dependent. A successful
4465 Completion Event indicates that the data has been delivered uncorrupted into the local buffer.

4466 A Consumer should not modify the local buffer specified by *num_segments* and *local_segments*
4467 until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the
4468 Implementation and the underlying transport is not defined. A Consumer does get back
4469 ownership of the *num_segments* and *local_segments* arguments (but not the local buffer
4470 identified by them) when *it_post_rcvfrom* returns and is free to use the *num_segments* and
4471 *local_segments* arguments for other calls, to modify them, or to destroy them.

4472 The completion of the posted Receive is reported asynchronously to the Consumer according to
4473 the rules defined in [it_dto_flags t](#). Exactly one Receive DTO Completion Event is always
4474 generated and manifests on the EVD associated with the Endpoint Receive Queue. See
4475 [it_ep_ud_create](#), [it_dto_status t](#) and [it_dto_events](#). If the reported *dto_status* of the Completion
4476 DTO Event corresponding to the posted Receive DTO is not IT_DTO_SUCCESS, the content of
4477 the local buffer is not defined.

4478 The *dto_flags* value is used as specified in [it_dto_flags t](#).

4479 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4480 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4481 to the Consumer in the Completion Event for the posted Receive.

4482 If the size of an incoming message is larger than the size of the local buffer or larger than the
4483 MTU of the local Spigot, the reported status of the posted Receive DTO in the corresponding
4484 Completion DTO Event is IT_DTO_ERR_LOCAL_LENGTH.

4485 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4486 Operations.

4487 The Implementation ensures that each Receive corresponds to one and only one remote Send.

4488 There is no relationship guaranteed on the order of Receive completions and the order of the
4489 posting of the corresponding Sends at remote datagram Endpoints.

4490 The Implementation ensures that all Receives start and complete in the order posted.

4491 The Implementation ensures that all data from a given Send operation is transferred from the
4492 remote buffer and into the local buffer before a Receive completion is generated with
4493 IT_DTO_SUCCESS.

4494 Any posting to an Endpoint that is in the IT_EP_STATE_NONOPERATIONAL state will be
4495 flushed with completion status ([it_dto_status t](#)) set to IT_DTO_ERR_FLUSHED.

4496 IT_SUCCESS returned from the *it_post_rcvfrom* call means that the Receive operation was
4497 successfully posted to the transport layer for use.

4498 **RETURN VALUE**

4499 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4500	IT_ERR_TOO_MANY_POSTS	The operation failed due to an overflow of a
4501		work queue.
4502	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was
4503		invalid.

4504	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service Type of the Endpoint.
4506	IT_ERR_INVALID_DTO_FLAGS	The Data Transfer Operation flags (<i>dto_flags</i>) value was invalid.
4508	IT_ERR_INVALID_NUM_SEGMENTS	The requested number of segments (<i>num_segments</i>) was larger than the Endpoint supports.
4511	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it ia info t for a description of the disabled state.

4516 **ERRORS**

4517 None.

4518 **APPLICATION USAGE**

4519 This function is used to transfer data into a Consumer-specified local buffer from a buffer
4520 specified by the corresponding Send operation at the remote Endpoint.

4521 The Consumer should use unique identifiers for *cookie* if they desire to identify each DTO. If the
4522 Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.

4523 For best Receive operation performance, the Consumer should align each buffer segment of
4524 *local_segments* to the *dto_alignment_hint* in the IA attributes obtained via [it ia query](#).

4525 **SEE ALSO**

4526 [it post send\(\)](#), [it post sendto\(\)](#), [it post rcv\(\)](#), [it post rdma read\(\)](#), [it post rdma write\(\)](#),
4527 [it rmr bind\(\)](#), [it rmr unbind\(\)](#), [it dto status t](#), [it dto events](#), [it dto flags t](#), [it ep ud create\(\)](#),
4528 [it lmr triplet t](#), [it ia query\(\)](#), [it dto cookie t](#), [it ia info t](#)

it_post_send()

4529

4530 **NAME**4531 `it_post_send` – post a Send DTO to a connected Endpoint4532 **SYNOPSIS**

```

4533 #include <it_api.h>
4534
4535 it_status_t it_post_send(
4536     IN          it_ep_handle_t          ep_handle,
4537     IN          const it_lmr_triplet_t  *local_segments,
4538     IN          size_t                  num_segments,
4539     IN          it_dto_cookie_t        cookie,
4540     IN          it_dto_flags_t         dto_flags
4541 );

```

4542 **DESCRIPTION**

4543 *ep_handle* Handle for the Endpoint – the local side of the Connection.

4544 *local_segments* Vector of [it_lmr_triplet_t](#) data structures that specifies the local
4545 buffer that contains data to be transferred. Can be NULL for a zero-
4546 sized message.

4547 *num_segments* Number of [it_lmr_triplet_t](#) data structures in *local_segments*. Can be
4548 zero for a zero-sized message.

4549 *cookie* Consumer-provided cookie that is returned to the Consumer in the
4550 Completion Event corresponding to the Send.

4551 *dto_flags* Flags for posted Send.

4552 *it_post_send* requests a transfer of all the data from *local_segments* into a remote buffer
4553 specified by a single corresponding Receive on the other side of the Connection. The Connection
4554 is implemented on a reliable transport. A zero-sized message may be transferred over the
4555 Connection and will consume a buffer specified by the corresponding Receive.

4556 *num_segments* specifies the number of segments in the *local_segments* vector.

4557 The Completion Event for the *it_post_send* call indicates to the Consumer that the local buffer is
4558 under Consumer control again. It does not guarantee that the contents of the local buffer have
4559 been successfully received by the remote Consumer. The contents of the local buffer are only
4560 guaranteed to have reached the remote Consumer's memory when the remote Consumer reaps a
4561 successful completion for the Receive operation that matches the Send initiated by the
4562 *it_post_send* call.

4563 Once the local buffer has reached the remote Consumer memory, the order of the bytes in the
4564 remote buffer specified by the Receive operation at the remote Endpoint corresponds to the order
4565 defined by the Send side *local_segments* subject to overlap constraints. See [it_post_recv](#)
4566 for details on overlap constraints. Prior to the Completion Event being generated, the contents of the
4567 receiver's *local_segments* are implementation-dependent.

4568 A Consumer should not modify the local buffer specified by *num_segments* and *local_segments*
4569 until the DTO is completed. When a Consumer does not adhere to this rule, the content of the
4570 local buffer at the receiving side is undefined after the matching Receive operation completes. A
4571 Consumer does get back the ownership of the *num_segments* and *local_segments* arguments (but
4572 not the local buffer identified by them) when *it_post_send* returns and is free to use the
4573 *num_segments* and *local_segments* arguments for other calls, to modify them, or to destroy
4574 them.

4575 The completion of the posted Send is reported asynchronously to the Consumer according to the
4576 rules defined in [it_dto_flags_t](#). Any generated DTO Completion Event manifests on the EVD
4577 associated with the Endpoint. See [it_ep_rc_create](#), [it_dto_status_t](#) and [it_dto_events](#). A
4578 completion status other than IT_DTO_SUCCESS will break the Connection.

4579 The *dto_flags* value is used as specified in [it_dto_flags_t](#).

4580 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4581 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4582 to the Consumer in the Completion Event for the posted Send.

4583 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4584 Operations.

4585 The Implementation allows the buffer segments described by the *local_segments* to overlap.

4586 Data corruption (at local and/or at remote) and data loss will be reported to the Consumer in the
4587 DTO Completion Event. These conditions will cause the Connection to be broken. Once the
4588 Connection is broken, all outstanding and in-progress operations on the Connection will
4589 complete with an error status.

4590 The Implementation ensures that each Send corresponds to one and only one remote Receive and
4591 in no way corresponds to any locally or remotely posted RDMA Read or RDMA Write Data
4592 Transfer Operations over the same Connection. If no Receive resources are ever posted at the
4593 remote end, then a Send will eventually abort with a completion error and the Connection will be
4594 broken. In order to avoid this scenario the remote Consumer should post Receive resources prior
4595 to the local Consumer posting the Send.

4596 The Implementation ensures that a RDMA Write DTO preceding the Send has fully delivered its
4597 payload prior to the completion of the remote Receive corresponding to the Send.

4598 The Implementation ensures that all Sends start and complete in the order posted.

4599 The Implementation ensures that all data from the given Send operation is transferred from the
4600 local buffer and to the remote buffer before a Send completion is generated with the status of
4601 IT_DTO_SUCCESS. If the corresponding remote Receive buffer is not sufficient in size for the
4602 Send data buffer then the operation will complete with an error status. In order to avoid this
4603 scenario the remote Consumer should post a buffer large enough for the incoming Send data.

4604 Send and RDMA DTOs following an RDMA Read DTO may start during execution of the
4605 RDMA Read DTO and complete before the RDMA Read completes. To ensure
4606 deterministically that subsequent Sends and RDMA DTOs following an RDMA Read DTO
4607 do start after the RDMA Read completes, specify the IT_BARRIER_FENCE_FLAG on the
4608 DTOs following the RDMA Read.

4609 Posting to an Endpoint that is not in the IT_EP_STATE_CONNECTED or IT_EP_STATE_
4610 NONOPERATIONAL state will return the IT_ERR_INVALID_EP_STATE error.

4611 Any posting to an Endpoint that is in the `IT_EP_STATE_NONOPERATIONAL` state will be
4612 flushed with completion status ([it_dto_status_t](#)) set to `IT_DTO_ERR_FLUSHED`.

4613 `IT_SUCCESS` returned from the `it_post_send` call means that the Send operation was
4614 successfully posted to the transport layer.

4615 **RETURN VALUE**

4616 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described below.

4617 `IT_ERR_TOO_MANY_POSTS` The operation failed due to an overflow of a
4618 work queue.

4619 `IT_ERR_INVALID_EP` The Endpoint Handle (`ep_handle`) was
4620 invalid.

4621 `IT_ERR_INVALID_EP_STATE` The Endpoint was not in the proper state for
4622 the attempted operation.

4623 `IT_ERR_INVALID_EP_TYPE` The attempted operation was invalid for the
4624 Service Type of the Endpoint.

4625 `IT_ERR_INVALID_DTO_FLAGS` The Data Transfer Operation flags (`dto_flags`)
4626 value was invalid.

4627 `IT_ERR_INVALID_NUM_SEGMENTS` The requested number of segments
4628 (`num_segments`) was larger than the Endpoint
4629 supports.

4630 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error
4631 and is in the disabled state. None of the
4632 output parameters from this routine are valid.
4633 See [it_ia_info_t](#) for a description of the
4634 disabled state.

4635 **ERRORS**

4636 None.

4637 **APPLICATION USAGE**

4638 This function is used after a Connection has been established to transfer data from a Consumer-
4639 specified local buffer to a buffer specified by the corresponding Receive operation on the other
4640 side of the Connection.

4641 The Consumer should use unique identifiers for `cookie` if they desire to identify each DTO. If the
4642 Consumer does not require a unique DTO identifier, the value of zero or `NULL` can be used.

4643 For best Send operation performance, the Consumer should align each buffer segment of
4644 `local_segments` to the `dto_alignment_hint` in the IA attributes obtained via [it_ia_query](#).

4645 **SEE ALSO**

4646 [it_post_sendto\(\)](#), [it_post_rcv\(\)](#), [it_post_rcvfrom\(\)](#), [it_post_rdma_read\(\)](#), [it_post_rdma_write\(\)](#),
4647 [it_rmr_bind\(\)](#), [it_rmr_unbind\(\)](#), [it_dto_status_t](#), [it_dto_events](#), [it_dto_flags_t](#), [it_ep_rc_create\(\)](#),
4648 [it_lmr_triplet_t](#), [it_ia_query\(\)](#), [it_dto_cookie_t](#), [it_ia_info_t](#)

it_post_sendto()

4649

4650 **NAME**4651 `it_post_sendto` – post a Send DTO to a datagram Endpoint4652 **SYNOPSIS**

```

4653 #include <it_api.h>
4654
4655 it_status_t it_post_sendto(
4656     IN          it_ep_handle_t          ep_handle,
4657     IN          const it_lmr_triplet_t  *local_segments,
4658     IN          size_t                  num_segments,
4659     IN          it_dto_cookie_t         cookie,
4660     IN          it_dto_flags_t          dto_flags,
4661     IN          const it_dg_remote_ep_addr_t *remote_ep_addr
4662 );

```

4663 **DESCRIPTION**

4664 *ep_handle* Handle for the local datagram Endpoint.

4665 *local_segments* Vector of [it_lmr_triplet_t](#) that specifies the local buffer that contains
4666 data to be transferred. Can be NULL for a zero-sized message.

4667 *num_segments* Number of [it_lmr_triplet_t](#) data structures in *local_segments*. Can be
4668 zero for a zero-sized message.

4669 *cookie* Consumer-provided cookie that is returned to the Consumer in the
4670 Completion Event corresponding to the Send.

4671 *dto_flags* Flags for posted Send.

4672 *remote_ep_addr* Remote datagram Endpoint address.

4673 *it_post_sendto* requests a transfer of all the data from *local_segments* via the local datagram
4674 *ep_handle* into a remote buffer specified by a single corresponding Receive at the remote
4675 datagram Endpoint as specified by *remote_ep_addr*. No guarantee of delivery is provided.

4676 *num_segments* specifies the number of segments in the *local_segments* vector.

4677 Once a successful Completion Event has been generated at the receiver, the order of the bytes in
4678 the remote buffer specified by the Receive operation at the remote Endpoint corresponds to the
4679 order defined by the Send side *local_segments* subject to overlap constraints. See
4680 [it_post_recvfrom](#) for details on overlap constraints. Prior to the Completion Event being
4681 generated, the contents of the receiver's *local_segments* are implementation-dependent.

4682 A Consumer should not modify the local buffer specified by *num_segments* and *local_segments*
4683 until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the
4684 Implementation and the underlying transport is not defined. A Consumer does get back the
4685 ownership of the *num_segments* and *local_segments* arguments (but not the local buffer
4686 identified by them) when *it_post_sendto* returns and is free to use the *num_segments* and
4687 *local_segments* arguments for other calls, to modify them, or to destroy them.

4688 The completion of the posted Send is reported asynchronously to the Consumer according to the
4689 rules defined in [it_dto_flags_t](#). Any generated DTO Completion Event manifests on the EVD
4690 associated with the Endpoint. See [it_ep_ud_create](#), [it_dto_status_t](#), and [it_dto_events](#).

4691 The *dto_flags* value is used as specified in [it_dto_flags_t](#).

4692 The *cookie* allows the Consumer to associate an identifier with each DTO. This identifier is
4693 completely under Consumer control and is opaque to the Implementation. The *cookie* is returned
4694 to the Consumer in the Completion Event for the posted Send.

4695 *remote_ep_addr* specifies the Destination for the *it_post_sendto* operation. See
4696 [it_dg_remote_ep_addr_t](#) for details on the format of this data structure.

4697 The Implementation ensures that an LMR Triplet supports byte alignment for Data Transfer
4698 Operations.

4699 The Implementation allows the buffer segments described by *local_segments* to overlap.

4700 The Implementation ensures that all Sends start and complete in the order posted.

4701 The Implementation makes no delivery order guarantees for Unreliable Datagrams.

4702 There is no delivery order or completion order between Receive Data Transfer Operations on
4703 different Destinations that correspond to the Sends posted in order to the same Unreliable
4704 Datagram Endpoint.

4705 Any posting to an Endpoint that is in the IT_EP_STATE_NONOPERATIONAL state will be
4706 flushed with completion status ([it_dto_status_t](#)) set to IT_DTO_ERR_FLUSHED.

4707 IT_SUCCESS returned from the *it_post_sendto* call means that the Send operation was
4708 successfully posted to the transport layer.

4709 When *it_post_sendto* completes with IT_SUCCESS or IT_DTO_ERR_LOCAL_EP there is no
4710 guarantee that the DTO has reached the remote Endpoint.

4711 **RETURN VALUE**

4712 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4713	IT_ERR_TOO_MANY_POSTS	The operation failed due to an overflow of a
4714		work queue.
4715	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
4716	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the
4717		Service Type of the Endpoint.
4718	IT_ERR_INVALID.DTO_FLAGS	The Data Transfer Operation flags (<i>dto_flags</i>)
4719		value was invalid.
4720	IT_ERR_INVALID_NUM_SEGMENTS	The requested number of segments
4721		(<i>num_segments</i>) was larger than the Endpoint
4722		supports.
4723	IT_ERR_INVALID_AH	The Address Handle within <i>remote_ep_addr</i>
4724		was invalid or the does not match the <i>spigot_id</i>
4725		of the Endpoint.

4726 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and
 4727 is in the disabled state. None of the output
 4728 parameters from this routine are valid. See
 4729 [it ia info t](#) for a description of the disabled
 4730 state.

4731 **ERRORS**
 4732 None.

4733 **APPLICATION USAGE**

4734 This function is used to transfer data from a Consumer-specified local buffer to a buffer
 4735 specified by the corresponding Receive operation at a remote datagram Endpoint.

4736 The Consumer should use unique identifiers for *cookie* if they desire to identify each DTO. If the
 4737 Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.

4738 For best Send operation performance, the Consumer should align each buffer segment of
 4739 *local_segments* to the *dto_alignment_hint* in the IA attributes obtained via [it ia query](#).

4740 An Address Handle corresponds to a specific Spigot on an IA. Attempting to *it_post_sendto* on
 4741 an Endpoint using an Address Handle that does not correspond to the Spigot associated with the
 4742 Endpoint must be avoided by the Consumer. If the Consumer persists in this practice, they must
 4743 write error handling code to deal with three possible error cases: One - the *it_post_sendto* call
 4744 will return the IT_ERR_INVALID_AH error immediately. Or two - the DTO will complete in
 4745 error with the *it_dto_status* set to IT_DTO_ERR_LOCAL_EP. Or three - there will be no
 4746 indication of error. The three possible cases represent the allowable implementations of the
 4747 underlying technology.

4748 **SEE ALSO**

4749 [it_post_send\(\)](#), [it_post_recv\(\)](#), [it_post_recvfrom\(\)](#), [it_post_rdma_read\(\)](#), [it_post_rdma_write\(\)](#),
 4750 [it_rmr_bind\(\)](#), [it_rmr_unbind\(\)](#), [it_dto_status_t](#), [it_dto_events](#), [it_dg_remote_ep_addr_t](#),
 4751 [it_dto_flags_t](#), [it_ep_ud_create\(\)](#), [it_lmr_triplet_t](#), [it_ia_query\(\)](#), [it_dto_cookie_t](#), [it_ia_info_t](#)

it_pz_create()

4752

NAME4753 **NAME**
4754 `it_pz_create` – create a new Protection Zone**SYNOPSIS**

```

4755 #include <it_api.h>
4756
4757
4758 it_status_t it_pz_create(
4759     IN          it_ia_handle_t    ia_handle,
4760     OUT         it_pz_handle_t    *pz_handle
4761 );

```

DESCRIPTION4762 `ia_handle` Interface Adapter on which the Protection Zone will be created.4763 `pz_handle` Handle of new Protection Zone

4765 The `it_pz_create` routine creates a new Protection Zone that may be used to create Local
 4766 Memory Regions, Remote Memory Regions, transport Endpoints, or Address Handles on the
 4767 Interface Adapter identified by `ia_handle`. The Protection Zone is returned in `pz_handle`.

RETURN VALUE4768 A successful call returns `IT_SUCCESS`. Otherwise, returns an error code as described below.4770 `IT_ERR_RESOURCES` The requested operation failed due to insufficient resources.4771 `IT_ERR_INVALID_IA` The Interface Adapter Handle (`ia_handle`) was invalid.

4772 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 4773 disabled state. None of the output parameters from this
 4774 routine are valid. See [it_ia_info_t](#) for a description of the
 4775 disabled state.

ERRORS4776 None.
4777**APPLICATION USAGE**

4778 An LMR, RMR, Endpoint or Address Handle can not be created without supplying a Protection
 4779 Zone. An LMR, RMR, or Address Handle may only be used in concert with an Endpoint having
 4780 the same Protection Zone. In DTO, Bind, and Unbind operations, the Protection Zone of the
 4781 local Endpoint and the LMR must match, or the operation will fail. In RDMA operations, the
 4782 Protection Zone of the RMR associated with the RMR Context must match that of the remote
 4783 Endpoint. In datagram DTO operations, the Protection Zone of the local Address Handle
 4784 identifying the Destination must match that of the local Endpoint. In Bind and Unbind
 4785 operations, the Protection Zone of the LMR and RMR must match.
 4786

SEE ALSO4787 [it_pz_free\(\)](#), [it_pz_query\(\)](#)
4788

it_pz_free()

4789

4790 **NAME**4791 `it_pz_free` – destroy a Protection Zone4792 **SYNOPSIS**

```

4793 #include <it_api.h>
4794
4795 it_status_t it_pz_free(
4796     IN          it_pz_handle_t      pz_handle
4797 );

```

4798 **DESCRIPTION**4799 `pz_handle` Handle of Protection Zone to be destroyed.

4800 The `it_pz_free` routine destroys the Protection Zone `pz_handle`. On successful return, the
 4801 `pz_handle` may no longer be used. An attempt to free a Protection Zone that is still referenced by
 4802 undestroyed Endpoints, Local Memory Regions, Remote Memory Regions, or Address Handles
 4803 will fail with `IT_ERR_PZ_BUSY`, and the Protection Zone will be unaffected.

4804 **RETURN VALUE**4805 A successful call returns `IT_SUCCESS`. Otherwise, returns an error code as described below.4806 `IT_ERR_INVALID_PZ` The Protection Zone Handle (`pz_handle`) was invalid.4807 `IT_ERR_PZ_BUSY` The Protection Zone was still in use.

4808 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 4809 disabled state. None of the output parameters from this
 4810 routine are valid. See [it_ia_info_t](#) for a description of the
 4811 disabled state.

4812 **ERRORS**

4813 None.

4814 **SEE ALSO**4815 [it_pz_create\(\)](#), [it_pz_query\(\)](#)

it_pz_query()

4816

4817 **NAME**4818 `it_pz_query` – get attributes of a Protection Zone4819 **SYNOPSIS**

```

4820 #include <it_api.h>
4821
4822 it_status_t it_pz_query(
4823     IN          it_pz_handle_t      pz_handle,
4824     IN          it_pz_param_mask_t  mask,
4825     OUT         it_pz_param_t      *params
4826 );
4827
4828 typedef enum {
4829     IT_PZ_PARAM_ALL   = 0x01,
4830     IT_PZ_PARAM_IA   = 0x02
4831 } it_pz_param_mask_t;
4832
4833 typedef struct {
4834     it_ia_handle_t ia; /* IT_PZ_PARAM_IA */
4835 } it_pz_param_t;

```

4836 **DESCRIPTION**4837 *pz_handle* Protection Zone.4838 *mask* Logical OR of flags for desired parameters.4839 *params* Structure whose members are written with the desired parameters.

4840 The *it_pz_query* routine returns the desired parameters of the Protection Zone *pz_handle* in the
 4841 structure pointed to by *params*. On return, each field of *params* is only valid if the
 4842 corresponding flag as shown in the Synopsis is set in the mask argument. The mask value
 4843 IT_PZ_PARAM_ALL causes all fields to be returned.

4844 The definition of each field of *params* follows:

4845 *ia* The Interface Adapter Handle specified to create the Protection
 4846 Zone.

4847 **RETURN VALUE**

4848 A successful call returns IT_SUCCESS. Otherwise, returns an error code as described below.

4849 IT_ERR_INVALID_PZ The Protection Zone Handle (*pz_handle*) was invalid.4850 IT_ERR_INVALID_MASK The *mask* contained invalid flag values.

4851 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
 4852 disabled state. None of the output parameters from this
 4853 routine are valid. See [it_ia_info_t](#) for a description of the
 4854 disabled state.

4855 **ERRORS**

4856 None.

4857 **SEE ALSO**4858 [*it_pz_create\(\)*](#), [*it_pz_free\(\)*](#)

it_reject()

4859

4860 **NAME**4861 `it_reject` - reject an incoming Connection Request or Connection Reply4862 **SYNOPSIS**

```

4863 #include <it_api.h>
4864
4865 it_status_t it_reject(
4866     IN          it_cn_est_identifier_t  cn_est_id,
4867     IN          const unsigned char    *private_data,
4868     IN          size_t                  private_data_length
4869 );
4870
4871 typedef uint64_t it_cn_est_identifier_t;

```

4872 **DESCRIPTION**

4873 *cn_est_id* Connection Establishment Identifier associated with the Connection
 4874 Request to be rejected. Calling *it_reject* destroys the identifier. See
 4875 [it_ep_accept](#) for a definition of this data type.

4876 *private_data* Opaque Private Data to be sent in the IT_CM_MSG_CONN_PEER_
 4877 REJECT_EVENT Event delivered to the Remote Consumer. If the
 4878 IA does not support Private Data, *private_data_length* must be zero.
 4879 The delivery of Private Data to the Remote Endpoint is unreliable.

4880 *private_data_length* Length of *private_data*. This field must be 0 if the IA does not
 4881 support Private Data.

4882 *it_reject* rejects an incoming Connection Request or Connection Reply. The Remote Endpoint
 4883 will receive an IT_CM_MSG_CONN_PEER_REJECT_EVENT Event on its IT_CM_MSG_
 4884 EVENT_STREAM Simple Event Dispatcher, and that Endpoint will transition into the
 4885 IT_EP_STATE_NONOPERATIONAL state.

4886 For two-way Connection establishment, *it_reject* can only be called on the Passive side in
 4887 response to the IT_CM_REQ_CONN_REQUEST_EVENT Event.

4888 For three-way Connection establishment, *it_reject* can be called on the Passive side in response
 4889 to the IT_CM_REQ_CONN_REQUEST_EVENT, or on the Active side in response to the
 4890 IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT. If *it_reject* is called on the active side,
 4891 the local Endpoint associated with the Connection establishment transitions to the IT_EP_
 4892 STATE_NONOPERATIONAL state. See the [it_ep_state_t](#) manual page for a description of
 4893 this Endpoint state.

4894 Once the Endpoint is in the IT_EP_STATE_NONOPERATIONAL state any pending Data
 4895 Transfer Operations or Bind or Unbind operations on the Endpoint will be flushed and will
 4896 generate Completion Events with a Status of IT_DTO_ERR_FLUSHED.

4897 The Connection Establishment Identifier, *cn_est_id*, is freed by *it_reject*.

4898 **RETURN VALUE**

4899 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

4900	IT_ERR_INVALID_CN_EST_ID	The Connection Establishment Identifier
4901		(<i>cn_est_id</i>) was invalid.
4902	IT_ERR_PDATA_NOT_SUPPORTED	Private Data was supplied by the Consumer but
4903		this Interface Adapter does not support Private
4904		Data. See it_ia_query for the IAs capabilities to
4905		support Private Data.
4906	IT_ERR_INVALID_PDATA_LENGTH	The Interface Adapter supports Private Data, but
4907		the length specified exceeded the Interface
4908		Adapter's capabilities.
4909	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is
4910		in the disabled state. None of the output
4911		parameters from this routine are valid. See
4912		it_ia_info_t for a description of the disabled state.
4913	ERRORS	
4914	None.	
4915	APPLICATION USAGE	
4916	1.	The Consumer is responsible for coordinating the use of functions that free a Connection
4917		Establishment Identifier (<i>cn_est_id</i>) such as it_ep_accept , it_reject , it_ep_disconnect and
4918		it_handoff . The behavior of functions that are passed in an invalid Connection
4919		Establishment Identifier is indeterminate.
4920	2.	The Consumer should be aware that the delivery of Private Data to the Remote Endpoint
4921		is unreliable.
4922	SEE ALSO	
4923		it_ep_accept() , it_ep_connect() , it_cm_req_events , it_cm_msg_events , it_ep_state_t ,
4924		it_handoff() , it_ia_query()

it_rmr_bind()

4925
 4926 **NAME**
 4927 `it_rmr_bind` – post operation to Bind a Remote Memory Region to a memory range

SYNOPSIS

```

4928 #include <it_api.h>
4929
4930
4931 it_status_t it_rmr_bind(
4932     IN          it_rmr_handle_t    rmr_handle,
4933     IN          it_lmr_handle_t    lmr_handle,
4934     IN          void                *addr,
4935     IN          it_length_t        length,
4936     IN          it_mem_priv_t      privs,
4937     IN          it_ep_handle_t      ep_handle,
4938     IN          it_dto_cookie_t     cookie,
4939     IN          it_dto_flags_t     dto_flags,
4940     OUT         it_rmr_context_t    *rmr_context
4941 );

```

DESCRIPTION

4943 *rmr_handle* Handle of RMR that will be bound.

4944 *lmr_handle* LMR to which RMR will be bound.

4945 *addr* Starting address of region to be bound.

4946 *length* Length in bytes of region to be bound. Must not be 0.

4947 *privs* Logical OR of requested remote access privilege flags for bound
 4948 region.

4949 *ep_handle* Endpoint on which to post the Bind operation.

4950 *cookie* Consumer-provided cookie that is returned to the Consumer in the
 4951 Completion Event corresponding to the RMR Bind operation.

4952 *dto_flags* Logical OR of options for operation handling.

4953 *rmr_context* Returned Context allowing remote access to the bound region.

4954 The `it_rmr_bind` routine posts to Endpoint `ep_handle` an operation to Bind the Remote Memory
 4955 Region `rmr_handle` to the segment of an LMR specified by the `lmr_handle`, `addr`, and `length`
 4956 arguments. It returns a new `rmr_context` value in network byte order that can be transferred by
 4957 the local Consumer to a remote Consumer to be used for an RDMA operation. The `ep_handle`
 4958 should be a Reliable Connected Endpoint; if it is not, an immediate error will be returned. The
 4959 Protection Zones of the `lmr_handle`, `rmr_handle`, and `ep_handle` must match; if they do not, a
 4960 completion error will be generated with completion status ([it_dto_status_t](#)) set to
 4961 IT_DTO_ERR_LOCAL_PROTECTION. Like DTOs, the Bind operation completes
 4962 asynchronously, and its completion is reported to the Consumer through a Completion Event
 4963 based on the specified `dto_flags` value. The Consumer defined `cookie` argument is opaque to the

4964 Implementation and is returned in the Completion Event. A Bind operation will only complete
 4965 successfully if it is posted to an Endpoint in the `IT_EP_STATE_CONNECTED` state. Any
 4966 posting to an Endpoint that is in the `IT_EP_STATE_NONOPERATIONAL` state will be flushed
 4967 with completion status set to `IT_DTO_ERR_FLUSHED`. A Bind operation may be submitted for
 4968 an RMR that has never been bound, is currently bound, or has been unbound using
 4969 [it_rmr_unbind](#).

4970 The starting virtual address and length of the region to be bound is specified by *addr* and *length*,
 4971 respectively. Remote access to the region is enforced with byte level granularity, unlike an
 4972 LMR. The specified address range must fall within the LMR given by *lmr_handle*.

4973 The type of remote access to be allowed is specified by the *privs* argument as a logical OR of
 4974 zero or more of the following values:

4975	<code>IT_PRIV_REMOTE_READ</code>	Enable access for RDMA Read operations.
4976	<code>IT_PRIV_REMOTE_WRITE</code>	Enable access for RDMA Write operations.
4977	<code>IT_PRIV_REMOTE</code>	Enable access for both remote RDMA Read and RDMA
4978		Write.
4979	<code>IT_PRIV_ALL</code>	Equivalent to <code>IT_PRIV_REMOTE</code> .

4980 Pass 0 or the value `IT_PRIV_NONE` to disallow remote access to the RMR. The flags
 4981 `IT_PRIV_READ_ONLY` and `IT_PRIV_DEFAULT` are invalid in this Context. It is invalid to
 4982 request remote write access if the memory access flags for *lmr_handle* include
 4983 `IT_PRIV_READ_ONLY`.

4984 Request handling is specified by the *dto_flags* argument and is the logical OR of zero or more of
 4985 the following flags:

4986	<code>IT_COMPLETION_FLAG</code>
4987	<code>IT_NOTIFY_FLAG</code>
4988	<code>IT_BARRIER_FENCE_FLAG</code>

4989 For the definition of these flags, see [it_dto_flags t](#). In addition, *it_rmr_bind* automatically fences
 4990 all DTO, Bind, and Unbind operations subsequently submitted on the Endpoint *ep_handle* such
 4991 that none of these operations starts until the currently posted Bind operation completes.

4992 The value of *rmr_context* is immediately available when *it_rmr_bind* returns, but it may not be
 4993 used by a remote host for an RDMA operation until the Bind Completion Event occurs.
 4994 Violation of this rule may result in an error and a broken Connection for the reliable Connection
 4995 Endpoint on which the RDMA operation is posted. See Application Usage for more details.

4996 After a successful Bind Completion Event, any previous binding for the RMR is invalidated.
 4997 Any RDMA operation that uses the previous RMR Context will fail with a protection violation;
 4998 beware that this may include operations that are outstanding when *it_rmr_bind* is called.
 4999 Completions for such operations should be dequeued prior to calling *it_rmr_bind*.

5000 The new binding remains valid until the next Bind or Unbind operation completes successfully,
 5001 or until the RMR is destroyed. A Bind operation will never be partially successful over a subset
 5002 of the requested memory range; it either succeeds completely or fails without invalidating any
 5003 portion of the previous binding.

5004		If <i>it_rmr_bind</i> returns successfully, but the Bind Completion Event status indicates failure, then
5005		the previous binding and RMR Context remains valid. If <i>ep_handle</i> is part of a Reliable
5006		Connection, then the Connection is broken, the Endpoint transitions into the
5007		IT_EP_STATE_NONOPERATIONAL state, and an IT_CM_MSG_CONN_BROKEN_EVENT
5008		Event is delivered to the Connect EVD of <i>ep_handle</i> .
5009		The Bind Completion Event is defined by it_dto_cmpl_event_t . The Event Stream type is
5010		IT_RMR_BIND_CMPL_EVENT.
5011	RETURN VALUE	
5012		A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
5013		below, and the previous binding for the RMR remains valid. It is possible for <i>it_rmr_bind</i> to
5014		return success but for the Completion Event to indicate failure.
5015	IT_ERR_INVALID_PRIVS	The requested memory privileges (<i>privs</i>) contained an
5016		invalid flag.
5017	IT_ERR_INVALID_DTO_FLAGS	The Data Transfer Operation flags (<i>dto_flags</i>) value
5018		was invalid.
5019	IT_ERR_RESOURCES	The requested operation failed due to insufficient
5020		resources.
5021	IT_ERR_ADDRESS	The address (<i>addr</i>) fell outside the boundaries
5022		specified by the Local Memory Region.
5023	IT_ERR_INVALID_LENGTH	The value of <i>length</i> fell outside the boundaries of the
5024		Local Memory Region or the value of <i>length</i> was 0.
5025	IT_ERR_INVALID_LMR	The Local Memory Region Handle (<i>lmr_handle</i>) was
5026		invalid.
5027	IT_ERR_INVALID_RMR	The Remote Memory Region Handle (<i>rmr_handle</i>)
5028		was invalid.
5029	IT_ERR_TOO_MANY_POSTS	The operation failed due to an overflow of a work
5030		queue.
5031	IT_ERR_INVALID_EP_STATE	The Endpoint was not in the proper state for the
5032		attempted operation.
5033	IT_ERR_INVALID_EP_TYPE	The attempted operation was invalid for the Service
5034		Type of the Endpoint.
5035	IT_ERR_INVALID_EP	The Endpoint Handle (<i>ep_handle</i>) was invalid.
5036	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in
5037		the disabled state. None of the output parameters from
5038		this routine are valid. See it_ia_info_t for a
5039		description of the disabled state.

5040 **ERRORS**

5041 None.

5042 **APPLICATION USAGE**

5043 The *it_rmr_bind* operation is lightweight compared to creating an RMR or an LMR. An
 5044 application concerned with efficiency would typically create one or more RMRs at initialization
 5045 time that could be bound multiple times to enable remote access to different peers as needed.

5046 The Consumer should use unique identifiers for *cookie* if they desire to identify each DTO. If the
 5047 Consumer does not require a unique DTO identifier, the value of zero or NULL can be used.

5048 The local Consumer has several options for ensuring that the remote Consumer does not use
 5049 *rmr_context* before the Bind Completion Event occurs. One is to wait for the Completion Event
 5050 on the Send EVD of the specified Endpoint *ep_handle* before sending the *rmr_context* to a peer.
 5051 Another option is to send the *rmr_context* to a peer by posting a DTO to the same Endpoint
 5052 *ep_handle* that was used to Bind the RMR. The barrier-fencing behavior of *it_rmr_bind* ensures
 5053 that the DTO does not start until the Bind Completion Event has occurred. If the Bind fails with
 5054 a completion error, the Connection will be broken and the DTO flushed, so the *rmr_context* will
 5055 not be sent.

5056 For reasons already described, the Bind Completion Event marks an important change in the
 5057 status of an RMR that some Consumers may need to monitor. It is inadvisable for such
 5058 Consumers to suppress this Completion Event by omitting `IT_COMPLETION_FLAG`, although
 5059 the completion status of the Bind operation may be inferred by other means. For example,
 5060 successful completion of a subsequently posted operation of any type indicates that the Bind
 5061 operation has completed successfully. If the Bind operation fails, a Bind Completion Event is
 5062 generated regardless.

5063 **FUTURE DIRECTIONS**

5064 Currently the Consumer is allowed to call *it_rmr_bind* on an RMR that is already in the bound
 5065 state. A future version of the IT-API may require the Consumer on some transports to first
 5066 Unbind a bound RMR using [*it_rmr_unbind*](#) before performing a Bind operation.

5067 **SEE ALSO**5068 [*it_lmr_create\(\)*](#), [*it_rmr_unbind\(\)*](#), [*it_rmr_create\(\)*](#), [*it_dto_flags_t*](#), [*it_dto_events*](#)

it_rmr_create()

5069

5070 NAME

5071

it_rmr_create – create a Remote Memory Region (RMR)

5072 SYNOPSIS

5073

#include <it_api.h>

5074

5075

it_status_t it_rmr_create(

5076

IN it_pz_handle_t pz_handle,

5077

OUT it_rmr_handle_t *rmr_handle

5078

);

5079 DESCRIPTION

5080

pz_handle

5081

Protection Zone in which the Remote Memory Region will be created.

5082

rmr_handle

Handle of new Remote Memory Region.

5083

5084

5085

5086

The *it_rmr_create* routine creates a Remote Memory Region that may be used as the target for Data Transfer Operations over the Interface Adapter that is implicitly identified by the *pz_handle* argument. The returned RMR must be bound to a Local Memory Region using [it_rmr_bind](#) before it can be used as a target, however.

5087 RETURN VALUE

5088

A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described below.

5089

5090

IT_ERR_INVALID_PZ

The Protection Zone Handle (*pz_handle*) was invalid.

5091

IT_ERR_RESOURCES

The requested operation failed due to insufficient resources.

5092

5093

IT_ERR_IA_CATASTROPHE

The IA has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See [it_ia_info_t](#) for a description of the disabled state.

5094

5095

5096

5097 ERRORS

5098

None.

5099 APPLICATION USAGE

5100

5101

5102

5103

5104

Creating an RMR is a relatively expensive operation. Once created, however, an RMR may be bound repeatedly to different LMR address ranges using the more efficient [it_rmr_bind](#) call, as long as the Protection Zone of the RMR matches that of the LMR. Binding an RMR is much more efficient than granting and changing remote access privileges using [it_lmr_create](#) and [it_lmr_modify](#).

5105 SEE ALSO

5106

[it_rmr_bind\(\)](#), [it_rmr_free\(\)](#), [it_rmr_query\(\)](#)

it_rmr_free()

5107

5108 **NAME**5109 `it_rmr_free` – destroy a Remote Memory Region5110 **SYNOPSIS**

```

5111 #include <it_api.h>
5112
5113 it_status_t it_rmr_free(
5114     IN          it_rmr_handle_t  rmr_handle
5115 );

```

5116 **DESCRIPTION**5117 `rmr_handle` Handle of Remote Memory Region to be destroyed.

5118 The `it_rmr_free` routine destroys the Remote Memory Region `rmr_handle`. If the RMR is
 5119 currently bound to an LMR, then the RMR binding is also destroyed. On return, the `rmr_handle`
 5120 may no longer be used, and the associated RMR Context may no longer be used. RMRs with
 5121 memory ranges that overlap the range of `rmr_handle` are not affected by its destruction.

5122 Outstanding remote DTOs that use the RMR Context of this RMR may either complete
 5123 successfully or fail with an access violation error. Note also that the number of possible RMR
 5124 Context values is finite, and the Implementation will eventually reuse previously freed values in
 5125 a new binding. If a DTO using an RMR Context is posted after that Context is freed, it is
 5126 theoretically possible for the Context to be reused before the DTO completes, and for the DTO
 5127 to complete under the new binding for the Context, resulting in data corruption. To avoid this,
 5128 the Consumer should not free an RMR which may be the target of outstanding DTOs. This may
 5129 require coordination between local and remote Consumers, and such coordination is the
 5130 Consumer's responsibility.

5131 **RETURN VALUE**

5132 A successful call returns `IT_SUCCESS`. Otherwise, an error code is returned as described
 5133 below.

5134 `IT_ERR_INVALID_RMR` The Remote Memory Region Handle (`rmr_handle`) was
 5135 invalid.

5136 `IT_ERR_IA_CATASTROPHE` The IA has experienced a catastrophic error and is in the
 5137 disabled state. None of the output parameters from this
 5138 routine are valid. See [it_ia_info_t](#) for a description of the
 5139 disabled state.

5140 **ERRORS**

5141 None.

5142 **SEE ALSO**5143 [it_rmr_create\(\)](#), [it_rmr_query\(\)](#)

it_rmr_query()

5144

5145 **NAME**

5146

it_rmr_query – get attributes of a Remote Memory Region

5147 **SYNOPSIS**

5148

5149

5150

5151

5152

5153

5154

5155

5156

5157

5158

5159

5160

5161

5162

5163

5164

5165

5166

5167

5168

5169

5170

5171

5172

5173

5174

5175

5176

5177

5178

```

#include <it_api.h>

it_status_t it_rmr_query(
    IN  it_rmr_handle_t    rmr_handle,
    IN  it_rmr_param_mask_t mask,
    OUT it_rmr_param_t     *params
);

typedef enum {
    IT_RMR_PARAM_ALL      = 0x000001,
    IT_RMR_PARAM_IA      = 0x000002,
    IT_RMR_PARAM_PZ      = 0x000004,
    IT_RMR_PARAM_BOUND   = 0x000008,
    IT_RMR_PARAM_LMR     = 0x000010,
    IT_RMR_PARAM_ADDR    = 0x000020,
    IT_RMR_PARAM_LENGTH  = 0x000040,
    IT_RMR_PARAM_MEM_PRIV = 0x000080,
    IT_RMR_PARAM_RMR_CONTEXT = 0x000100
} it_rmr_param_mask_t;

typedef struct {
    it_ia_handle_t    ia;      /* IT_RMR_PARAM_IA */
    it_pz_handle_t    pz;      /* IT_RMR_PARAM_PZ */
    it_boolean_t      bound;   /* IT_RMR_PARAM_BOUND */
    it_lmr_handle_t   lmr;     /* IT_RMR_PARAM_LMR */
    void *            addr;    /* IT_RMR_PARAM_ADDR */
    it_length_t       length;  /* IT_RMR_PARAM_LENGTH */
    it_mem_priv_t     privs;   /* IT_RMR_PARAM_MEM_PRIV */
    it_rmr_context_t  rmr_context;
                        /* IT_RMR_PARAM_RMR_CONTEXT */
} it_rmr_param_t;

```

5179 **DESCRIPTION**

5180

rmr_handle Remote Memory Region

5181

mask logical OR of flags for desired parameters

5182

params structure whose members are written with the desired parameters

5183

5184

5185

5186

5187

The *it_rmr_query* routine returns the desired attributes of the Remote Memory Region *rmr_handle* in the structure pointed to by *params*. The *mask* argument specifies which fields of *params* are returned, and the values returned in other fields are undefined. See the Synopsis for the correspondence between *mask* values and fields. The *mask* value *IT_RMR_PARAM_ALL* causes all fields to be returned.

5188 The definition of each field of *params* follows, some of which depend on whether the RMR is
 5189 currently bound to an LMR as a result of using [it_rmr_bind](#):

5190	<i>ia</i>	The Interface Adapter Handle specified to create the RMR.
5191	<i>pz</i>	The Protection Zone Handle specified to create the RMR.
5192	<i>bound</i>	IT_TRUE if the RMR is currently bound, IT_FALSE otherwise.
5193	<i>lmr</i>	The Local Memory Region to which the RMR is currently bound, or
5194		undefined if RMR is not bound.
5195	<i>addr</i>	The currently bound starting address of the RMR, or undefined if not
5196		bound.
5197	<i>length</i>	The currently bound length in bytes of the RMR, or undefined if not
5198		bound.
5199	<i>privs</i>	The currently bound memory access privileges of the RMR, or
5200		undefined if not bound.
5201	<i>rmr_context</i>	The currently bound RMR Context associated with the RMR, or
5202		undefined if not bound. Returned in network byte order.

5203 If the Consumer calls *it_rmr_query* after posting a Bind or Unbind operation, and before
 5204 dequeuing the Completion Event of such an operation, then the returned *bound*, *lmr*, *addr*,
 5205 *length*, *privs*, and *rmr_context* fields may represent the RMR state as it was prior to posting, or a
 5206 new RMR state. The Consumer should not rely on the value of these fields during this time.

5207 RETURN VALUE

5208 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 5209 below.

5210	IT_ERR_INVALID_RMR	The Remote Memory Region Handle (<i>rmr_handle</i>) was
5211		invalid.
5212	IT_ERR_INVALID_MASK	The <i>mask</i> contained invalid flag values.
5213	IT_ERR_IA_CATASTROPHE	The IA has experienced a catastrophic error and is in the
5214		disabled state. None of the output parameters from this
5215		routine are valid. See it_ia_info_t for a description of the
5216		disabled state.

5217 ERRORS

5218 None.

5219 SEE ALSO

5220 [it_rmr_create\(\)](#), [it_rmr_free\(\)](#), [it_rmr_bind\(\)](#), [it_rmr_context_t](#)

it_rmr_unbind()

5221

5222 **NAME**5223 `it_rmr_unbind` – post operation to Unbind a Remote Memory Region from its memory range5224 **SYNOPSIS**

```

5225 #include <it_api.h>
5226
5227 it_status_t it_rmr_unbind(
5228     IN          it_rmr_handle_t  rmr_handle,
5229     IN          it_ep_handle_t   ep_handle,
5230     IN          it_dto_cookie_t  cookie,
5231     IN          it_dto_flags_t   dto_flags
5232 );

```

5233 **DESCRIPTION**5234 `rmr_handle` Handle of RMR that will be unbound.5235 `ep_handle` Endpoint on which to post the operation.5236 `cookie` Consumer-provided cookie that is returned to the Consumer in the
5237 Completion Event corresponding to the operation.5238 `dto_flags` Logical OR of options for operation handling.

5239 The `it_rmr_unbind` routine posts to Endpoint `ep_handle` an Unbind operation to Unbind the
5240 Remote Memory Region `rmr_handle`. The `ep_handle` should be a Reliable Connected Endpoint;
5241 if it is not, an immediate error will be returned. The Protection Zones of the `rmr_handle` and
5242 `ep_handle` must match; if they do not, a completion error will be generated with completion
5243 status (`it_dto_status_t`) set to `IT_DTO_ERR_LOCAL_PROTECTION`. The operation
5244 completes asynchronously, and its completion is reported to the Consumer through a Completion
5245 Event based on the specified `dto_flags` value. The Consumer defined `cookie` argument is opaque
5246 to the Implementation and is returned in the Completion Event. An Unbind operation will only
5247 complete successfully if it is posted to an Endpoint in the `IT_EP_STATE_CONNECTED` state.
5248 Any posting to an Endpoint that is in the `IT_EP_STATE_NONOPERATIONAL` state will be
5249 flushed with completion status set to `IT_DTO_ERR_FLUSHED`.

5250 Request handling is specified by the `dto_flags` argument as the logical OR of zero or more of the
5251 following flags:

```

5252     IT_COMPLETION_FLAG
5253     IT_NOTIFY_FLAG
5254     IT_BARRIER_FENCE_FLAG

```

5255 For the definition of these flags, see [it_dto_flags_t](#). In addition, `it_rmr_unbind` automatically
5256 fences all DTO, Bind, and Unbind operations subsequently submitted on the Endpoint `ep_handle`
5257 such that none of these operations starts until the currently posted Unbind operation completes.

5258 After a successful Unbind Completion Event, any previous binding for the RMR is invalidated,
5259 and the RMR Context for the RMR is no longer defined. Any RDMA operation that uses the
5260 previous RMR Context will fail with a protection violation; beware that this may include
5261 operations that are outstanding when `it_rmr_unbind` is called. The Consumer must ensure that

5262 such operations have completed prior to calling *it_rmr_unbind* if successful completions are
 5263 desired. An Unbind operation will never be partially successful over a subset of the requested
 5264 memory range; it either succeeds completely or fails without invalidating any portion of the
 5265 previous binding.

5266 If *it_rmr_unbind* returns successfully but the Completion Event status indicates failure, then the
 5267 previous binding and RMR Context remains valid. If *ep_handle* is part of a Reliable
 5268 Connection, then the Connection is broken, the Endpoint transitions into the
 5269 IT_EP_STATE_NONOPERATIONAL state, and an IT_CM_MSG_CONN_BROKEN_EVENT
 5270 Event is delivered to the Connect EVD of *ep_handle*.

5271 The Unbind operation generates an [it_dto_compl_event_t](#) Completion Event. The Event Stream
 5272 type is IT_RMR_BIND_CMPL_EVENT.

5273 RETURN VALUE

5274 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 5275 below, and the previous binding for the RMR remains valid. It is possible for *it_rmr_unbind* to
 5276 return success but for the Completion Event to indicate failure.

5277 IT_ERR_INVALID_DTO_FLAGS The Data Transfer Operation flags (*dto_flags*) value was
 5278 invalid.

5279 IT_ERR_INVALID_RMR The Remote Memory Region Handle (*rmr_handle*) was
 5280 invalid.

5281 IT_ERR_TOO_MANY_POSTS The operation failed due to an overflow of a work
 5282 queue.

5283 IT_ERR_INVALID_EP_STATE The Endpoint was not in the proper state for the
 5284 attempted operation.

5285 IT_ERR_INVALID_EP_TYPE The attempted operation was invalid for the Service
 5286 Type of the Endpoint.

5287 IT_ERR_INVALID_EP The Endpoint Handle (*ep_handle*) was invalid.

5288 IT_ERR_IA_CATASTROPHE The IA has experienced a catastrophic error and is in the
 5289 disabled state. None of the output parameters from this
 5290 routine are valid. See [it_ia_info_t](#) for a description of
 5291 the disabled state.

5292 ERRORS

5293 None.

5294 APPLICATION USAGE

5295 *it_rmr_unbind* may be used to revoke remote Consumer access to an RMR that was previously
 5296 granted. In addition, the Consumer must Unbind all RMRs that refer to an LMR in order to
 5297 destroy or modify the LMR. Note that the RMR is not considered unbound until a successful
 5298 Completion Event is generated; thus, the Consumer should dequeue the Completion Event
 5299 before calling *it_lmr_free*. A difficulty can arise if the Endpoint that the Consumer was using to
 5300 Bind the RMR has become disconnected, because an Unbind operation can only be posted to a
 5301 connected Endpoint. One solution is for the Consumer to create a special pair of Endpoints to be

5302 used in this situation that are connected in loopback mode to each other, created using the same
5303 Protection Zone as the RMR. Another solution is to destroy the RMR by calling *it_rmr_free*.

5304 For reasons already described, the Unbind Completion Event marks an important change in the
5305 status of an RMR that some Consumers may need to monitor. It is inadvisable for such
5306 Consumers to suppress this Completion Event by omitting IT_COMPLETION_FLAG, although
5307 the completion status of the Unbind operation may be inferred by other means. For example,
5308 completion of a subsequently posted operation of any type indicates that the Unbind operation
5309 has completed successfully. If the Unbind operation fails, a Completion Event is generated
5310 regardless.

5311 **SEE ALSO**

5312 [*it_rmr_create\(\)*](#), [*it_rmr_bind\(\)*](#), [*it_dto_flags_t*](#)

5313 **it_set_consumer_context()**

5314 **NAME**

5315 `it_set_consumer_context` – associate a Consumer Context with an IT Object Handle

5316 **SYNOPSIS**

```
5317 #include <it_api.h>
5318
5319 it_status_t it_set_consumer_context(
5320     IN          it_handle_t      handle,
5321     IN          it_context_t     context
5322 );
```

5323 **DESCRIPTION**

5324 *handle* Handle for the IT-API object to be associated with the Consumer
5325 Context.

5326 *context* The Consumer Context to be associated with the object Handle.

5327 *it_set_consumer_context* associates a Consumer Context with the specified *handle*. See
5328 [it_handle_t](#) for a description of the valid Handle types.

5329 Only a single Consumer Context is provided for any IT Object Handle. If there is a previous
5330 Consumer Context associated with the specified Handle, the new Context replaces the old one.
5331 The value of Context is opaque to the Implementation. The Consumer can disassociate the
5332 existing Context by providing a NULL value for the Context. The Implementation makes no
5333 attempt to synchronize access to the Context.

5334 **RETURN VALUE**

5335 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
5336 below.

5337 IT_ERR_INVALID_HANDLE The *handle* was invalid

5338 IT_ERR_IA_CATASTROPHE The IA has experiences a catastrophic error and is in the
5339 disabled state. None of the output parameters from this
5340 routine are valid. See [it_ia_info_t](#) for a description of the
5341 disabled state.

5342 **ERRORS**

5343 None.

5344 **EXAMPLES**

5345 The following code example demonstrates the use of a cast in the call to
5346 *it_set_consumer_context*. The lmr object is cast to the generic [it_handle_t](#) type for the call.

```
5347 it_lmr_handle_t lmr;
5348 it_context_t cxt = 1234;
5349 it_set_consumer_context( (it_handle_t) lmr, cxt);
```

5350 **SEE ALSO**

5351 [it_get_consumer_context\(\)](#), [it_context_t](#), [it_handle_t](#)

it_ud_service_reply()

5352

5353 **NAME**

5354 `it_ud_service_reply` – return the information necessary to communicate via Unreliable
 5355 Datagram (UD) messages with the entity specified by the Connection
 5356 Qualifier in the UD Service Request Event

5357 **SYNOPSIS**

```
5358 #include <it_api.h>
5359
5360 it_status_t it_ud_service_reply (
5361     IN          it_ud_svc_req_identifiaer_t    ud_svc_req_id,
5362     IN          it_ud_svc_req_status_t        status,
5363     IN          it_remote_ep_info_t          ep_info,
5364     IN          const unsigned char          *private_data,
5365     IN          size_t                        private_data_length
5366 );
5367
5368 typedef uint64_t it_ud_svc_req_identifiaer_t;
```

5369 **DESCRIPTION**

5370 *ud_svc_req_id* Unique identifier from the
 5371 IT_CM_REQ_UD_SERVICE_REQUEST_EVENT Event generated
 5372 from the UD Service Request that is being responded to with this
 5373 invocation of *it_ud_service_reply*.

5374 *status* Status to return in the
 5375 IT_CM_MSG_UD_SERVICE_REPLY_EVENT data indicating the
 5376 outcome of the UD Service Request.

5377 *ep_info* End-point information to be used by the UD Service Requester to
 5378 communicate with this UD Service.

5379 *private_data* Opaque Private Data provided by the Consumer which will be sent
 5380 as part of the *it_ud_service_reply*. If the IA does not support Private
 5381 Data, *private_data_length* must be 0. The delivery of Private Data to
 5382 the Remote Endpoint is unreliable.

5383 *private_data_length* Length of the *private_data* provided by the Consumer. If the IA does
 5384 not support Private Data, this field must be 0.

5385 The *it_ud_service_reply* routine will be called by the Consumer to respond to an
 5386 IT_CM_REQ_UD_SERVICE_REQUEST_EVENT Event. The IT_CM_REQ_UD_SERVICE_
 5387 REQUEST_EVENT Event data ([it_ud_svc_request_event_t](#)) contains a unique Service Request
 5388 Handle, the Connection Qualifier of interest, Source address information and optional Private
 5389 Data. The recipient of the IT_CM_REQ_UD_SERVICE_REQUEST_EVENT Event needs to
 5390 respond to the request by calling *it_ud_service_reply*.

5391 The *ud_svc_req_id* is a unique identifier allowing this response to be correlated to the request
 5392 being responded to. The *ud_svc_req_id* should be copied from the IT_CM_REQ_UD_
 5393 SERVICE_REQUEST_EVENT Event data, *ud_svc_req_id* field. Once *it_ud_service_reply* has

5394 been successfully invoked, the supplied *ud_svc_req_id* is no longer valid. The resources
5395 associated with the *ud_svc_req_id* are released and the *ud_svc_req_id* can not be reused.

5396 Valid status codes for the *status* field are defined in [it ud svc req status t](#). A valid status code
5397 must be provided. IT_UD_REQ_REDIRECTED can not be supplied as input for this parameter,
5398 even though it may appear in the Event given to the requester. The Implementation is
5399 responsible for redirection, not the Consumer.

5400 The *ep_info* is only used by this routine if the *status* field is set to IT_UD_SVC_EP_
5401 INFO_VALID. See [it ud svc req status t](#) for details.

5402 The IA can be queried via [it ia query](#) to determine if it supports the transfer of Private Data.
5403 This is indicated by the *private_data_support* field of the [it ia info t](#) structure. If Private Data is
5404 not supported, *private_data_length* must be 0. The maximum length of *private_data* can be
5405 determined by examining the *ud_rep_private_data_len* member of the [it ia info t](#) structure.

5406 EXTENDED DESCRIPTION

5407 *it_ud_service_reply* is called by the Consumer in response to receiving an
5408 IT_CM_REQ_UD_SERVICE_REQUEST_EVENT Event. The Consumer chooses how to
5409 respond to the Service Request and makes that choice known via the value of *status* passed into
5410 the *it_ud_service_reply* call. The value of *status* determines whether the Implementation uses the
5411 *ep_info* input parameter. The table below describes the meaning of each *status* value, and
5412 whether the Implementation uses the *ep_info* input parameter when that *status* value is present.

<i>status</i> value	Implication of the status value
IT_UD_SVC_EP_INFO_VALID	The supplied <i>ep_info</i> (it remote ep info t) is valid and can be used by the recipient of the <i>it_ud_service_reply</i> to communicate with this service via UD messages. The Consumer must supply an it remote ep info t structure containing a valid <i>ud_ep_id</i> and <i>ud_ep_key</i> .
IT_UD_SVC_ID_NOT_SUPPORTED	The service described by the <i>conn_qual</i> (it conn qual t) in the it ud svc request event t is not supported by this service. The Implementation does not use the <i>ep_info</i> parameter.
IT_UD_SVC_REQ_REJECTED	Rejects the request for UD Service information. The Implementation does not use the <i>ep_info</i> parameter.
IT_UD_NO_EP_AVAILABLE	The Consumer responding via <i>it_ud_service_reply</i> does not have any Endpoints available for UD communication. The Implementation does not use the <i>ep_info</i> parameter.
IT_UD_REQ_REDIRECTED	The Consumer can not set this status. This status can only be set by the Implementation.

5413 In order for the Implementation to be able to correctly correlate this *it_ud_service_reply* call
5414 with the request Event being responded to, the Consumer must supply the *ud_svc_req_id* from
5415 the [it ud svc request event t](#) as the *ud_svc_req_id* passed into the *it_ud_service_reply* call.

5416 It is possible to receive duplicate UD Service Requests as a result of the active side retrying an
 5417 [it_ud_service_request](#) operation. It is the Consumer's responsibility to detect and handle
 5418 duplicate requests. Requests are uniquely identified by a combination of the *ud_svc_req_id* and
 5419 the *source_addr* from the [it_ud_svc_request_event_t](#) data. This combination can be used to
 5420 detect duplicate UD Service Requests.

5421 **RETURN VALUE**

5422 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 5423 below.

5424 IT_ERR_PDATA_NOT_SUPPORTED Private Data was supplied by the Consumer, but this
 5425 Interface Adapter does not support Private Data.

5426 IT_ERR_INVALID_PDATA_LENGTH The Interface Adapter supports Private Data, but the
 5427 length specified exceeded the Interface Adapter's
 5428 capabilities.

5429 IT_ERR_INVALID_UD_SVC_REQ_ID The Unreliable Datagram Service Request ID
 5430 (*ud_svc_req_id*) was invalid.

5431 IT_ERR_INVALID_UD_STATUS The Unreliable Datagram Service Request *status*
 5432 was invalid.

5433 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a
 5434 catastrophic error and is in the disabled state. None
 5435 of the output parameters from this routine are valid.
 5436 See [it_ia_info_t](#) for a description of the disabled
 5437 state.

5438 **ERRORS**

5439 None.

5440 **SEE ALSO**

5441 [it_ia_query\(\)](#), [it_ud_service_request\(\)](#), [it_ep_attributes_t](#), [it_cm_msg_events](#),
 5442 [it_cm_req_events](#)

5443

it_ud_service_request()5444 **NAME**

5445 `it_ud_service_request` – request that the recipient of this message return the information
 5446 necessary to communicate via Unreliable Datagram (UD) messages to
 5447 the entity specified by the UD Service Handle

5448 **SYNOPSIS**

```
5449 #include <it_api.h>
5450
5451 it_status_t it_ud_service_request (
5452     IN          it_ud_svc_req_handle_t  ud_svc_handle
5453 );
```

5454 **DESCRIPTION**

5455 `ud_svc_handle` UD Service Request Handle created by a call to
 5456 [it_ud_service_request_handle_create](#). This Handle uniquely
 5457 identifies this UD Service Request operation. The UD Service
 5458 Request Handle is associated with a specific UD Service described
 5459 during the creation of the UD Service Request Handle.

5460 The `it_ud_service_request` routine is called by a Consumer to request a remote entity specified
 5461 by the UD Service Handle to return information necessary to communicate via Unreliable
 5462 Datagram messages.

5463 The `ud_svc_handle` provides the Consumer with a means of correlating this
 5464 `it_ud_service_request` with the `IT_CM_MSG_UD_SERVICE_REPLY_EVENT` Event that the
 5465 Consumer will receive when the remote Endpoint responds to this UD Service Request. See
 5466 [it_cm_msg_events](#).

5467 Due to the nature of Unreliable Datagrams, even though an invocation of `it_ud_service_request`
 5468 returns success, the target of the UD Service Request may not receive it. Therefore, the
 5469 Consumer may have to call `it_ud_service_request` multiple times with the same `ud_svc_handle`
 5470 before the recipient actually receives the request and is able to reply to it. In addition, if the
 5471 Consumer issues multiple requests with the same `ud_svc_handle`, the Consumer may receive
 5472 multiple replies. It is up to the Consumer to detect and handle duplicate replies.

5473 The `ud_svc_req_id` ([it_ud_svc_req_identifier_t](#)) associated with a given `ud_svc_handle` does not
 5474 change. Therefore, all retries using a given `ud_svc_handle` will result in the same `ud_svc_req_id`
 5475 being presented to the recipient of the `IT_CM_REQ_UD_SERVICE_REQUEST_EVENT` Event
 5476 in the Event data ([it_ud_svc_request_event_t](#)).

5477 Upon a successful invocation and transmission of the `it_ud_service_request`, once the recipient
 5478 of the request replies via [it_ud_service_reply](#), the Consumer will receive an
 5479 `IT_CM_MSG_UD_SERVICE_REPLY_EVENT` Event. The `IT_CM_MSG_UD_SERVICE_REPLY_EVENT`
 5480 Event data ([it_ud_svc_reply_event_t](#)) contains the results of the Service
 5481 Request query. The `status` field of the [it_ud_svc_reply_event_t](#) structure in the
 5482 `IT_CM_MSG_UD_SERVICE_REPLY_EVENT` indicates the state of the information in the
 5483 [it_ud_svc_reply_event_t](#) structure. See [it_cm_msg_events](#).

5484 **EXTENDED DESCRIPTION**

5485 The *ud_service_request* call requests information from the remote UD Service. Once that remote
 5486 UD Service responds, an IT_CM_MSG_UD_SERVICE_REPLY Event will be generated. The
 5487 data associated with the Event, [it_ud_svc_reply_event t](#), contains information the Consumer
 5488 needs in order to perform Data Transfer Operations with the remote UD Service. The *status*
 5489 ([it_ud_svc_req_status t](#)) field of the [it_ud_svc_reply_event t](#) indicates the validity of other
 5490 fields in the structure. The *status* field should be checked by the Consumer prior to making any
 5491 assumptions about the data in the rest of the structure. The table below summarizes the *status*
 5492 values and the implications on the data in the [it_ud_svc_reply_event t](#) structure:

<i>status</i> value	implication for it_ud_svc_reply_event t data
IT_UD_SVC_EP_INFO_VALID	The <i>ep_info</i> (it_remote_ep_info t) is valid. The <i>ud_ep_id</i> and <i>ud_ep_key</i> from the it_remote_ep_info t structure, combined with the it_path t from the <i>ud_svc_handle</i> provides the Consumer with the necessary information to perform Data Transfer Operations with the remote UD Service. All fields except <i>destination_path</i> contain valid data.
IT_UD_SVC_ID_NOT_SUPPORTED	The Service described by the <i>connection_qualifier</i> (it_conn_qual t) in the <i>ud_svc_handle</i> is not supported on the Spigot to which the <i>it_ud_service_request</i> was sent. All fields except <i>ep_info</i> and <i>destination_path</i> contain valid data.
IT_UD_SVC_REQ_REJECTED	The recipient of the <i>it_ud_service_request</i> rejected the UD Service Request operation. All fields except <i>ep_info</i> and <i>destination_path</i> contain valid data.
IT_UD_NO_EP_AVAILABLE	The recipient of the <i>it_ud_service_request</i> does support the UD Service requested, but is out of Endpoint resources. That is, the remote node does not have any Endpoints that can be used to perform Data Transfer Operations with the UD Consumer. All fields except <i>ep_info</i> and <i>destination_path</i> contain valid data.
IT_UD_REQ_REDIRECTED	The Implementation on the receiving side of the <i>it_ud_service_request</i> has requested that the Consumer redirect the Service Request operation to a new Destination. The <i>destination_path</i> (it_path t) contains valid data. The <i>destination_path</i> should be used to create a new <i>ud_svc_handle</i> to be used in another call to <i>it_ud_service_request</i> . All fields except <i>ep_info</i> , <i>private_data</i> , and <i>private_data_length</i> contain valid data.

5493 **RETURN VALUE**

5494 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 5495 below.

5496 IT_ERR_INVALID_UD_SVC The Unreliable Datagram Service Handle (*ud_svc_handle*)
 5497 was invalid.

5498 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic error
 5499 and is in the disabled state. None of the output parameters
 5500 from this routine are valid. See [it_ia_info_t](#) for a description
 5501 of the disabled state.

5502 **ERRORS**

5503 None.

5504 **SEE ALSO**

5505 [it_ud_service_request_handle_create\(\)](#), [it_ud_service_reply\(\)](#), [it_cm_msg_events](#), [it_path_t](#),
 5506 [it_ep_attributes_t](#)

5507

it_ud_service_request_handle_create()

5508

5509 **NAME**

5510 `it_ud_service_request_handle_create` – create an Unreliable Datagram (UD) Service Request
 5511 Handle

5512 **SYNOPSIS**

```
5513 #include <it_api.h>
5514
5515 it_status_t it_ud_service_request_handle_create (
5516     IN      const      it_conn_qual_t      *conn_qual,
5517     IN      it_ev_d_handle_t      reply_ev_d,
5518     IN      const      it_path_t      *destination_path,
5519     IN      const      unsigned char      *private_data,
5520     IN      size_t      private_data_length,
5521     OUT     it_ud_svc_req_handle_t      *ud_svc_handle
5522 );
```

5523 **DESCRIPTION**

5524 *conn_qual* The Connection Qualifier describing the UD Service for which the
 5525 Consumer is requesting information.

5526 *reply_ev_d* The Simple EVD on which the
 5527 IT_CM_MSG_UD_SERVICE_REPLY_EVENT Event will be
 5528 received. *reply_ev_d* must be of the
 5529 IT_CM_MSG_EVENT_STREAM Event Stream Type. See
 5530 [it_cm_msg_events](#).

5531 *destination_path* *destination_path* specifies a Path to the Destination of the
 5532 [it_ud_service_request](#) operation.

5533 *private_data* Opaque Private Data provided by the Consumer which will be sent
 5534 as part of the [it_ud_service_request](#). If the IA does not support
 5535 Private Data, *private_data_length* must be 0. The delivery of Private
 5536 Data to the Remote Endpoint is unreliable.

5537 *private_data_length*: Length of the *private_data* provided by the Consumer. If the IA does
 5538 not support Private Data, this field must be 0.

5539 *ud_svc_handle* UD Service Request Handle created by this call. This Handle will be
 5540 used in a call to [it_ud_service_request](#).

5541 The *it_ud_service_request_handle_create* routine is called by the Consumer to create an
 5542 Unreliable Datagram Service Request Handle to be used in a call to [it_ud_service_request](#).

5543 The *destination_path* can be obtained by calling [it_get_pathinfo](#). The *spigot_id* in the [it_path_t](#)
 5544 will be the Spigot Identifier used for this UD Service Request.

5545 The IA can be queried via [it_ia_query](#) to determine if it supports the transfer of Private Data.
 5546 This is indicated by the *private_data_support* field of the [it_ia_info_t](#) structure. If Private Data is

5547 not supported, *private_data_length* must be 0. The maximum length of *private_data* can be
5548 determined by examining the *ud_req_private_data_len* member of the [it_ia_info_t](#) structure.

5549 The returned *ud_svc_handle* is used to identify the UD Service Request. It provides the
5550 Consumer with a means of correlating this [it_ud_service_request](#) with the
5551 IT_CM_MSG_UD_SERVICE_REPLY_EVENT Event that the Consumer will receive when the
5552 remote Endpoint responds to this UD Service Request.

5553 The *ud_svc_req_id* ([it_ud_svc_req_idenfier_t](#)) associated with a given *ud_svc_handle* does not
5554 change. Therefore, all retries using a given *ud_svc_handle* will result in the same *ud_svc_req_id*
5555 being presented to the recipient of the IT_CM_REQ_UD_SERVICE_REQUEST_EVENT Event
5556 in the Event data ([it_ud_svc_request_event_t](#)).

5557 EXTENDED DESCRIPTION

5558 The members of the [it_path_t](#) structure that are pertinent for creating a UD Service Request
5559 Handle are listed in the table below.

it_path_t member	Description
spigot_id	Spigot Identifier
ib.partition_key	Partition Key
ib.local_port_lid	Source LID
ib.remote_port_lid	Destination LID
ib.sl	Service level

5560 RETURN VALUE

5561 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
5562 below.

5563 IT_ERR_INVALID_CONN_EVD The Connection Simple Event Dispatcher Handle
5564 was invalid.

5565 IT_ERR_INVALID_EVD_TYPE The Event Stream Type for the Event Dispatcher
5566 was invalid.

5567 IT_ERR_PDATA_NOT_SUPPORTED Private Data was supplied by the Consumer, but this
5568 Interface Adapter does not support Private Data.

5569 IT_ERR_INVALID_PDATA_LENGTH The Interface Adapter supports Private Data, but the
5570 length specified exceeded the Interface Adapter's
5571 capabilities.

5572 IT_ERR_INVALID_CONN_QUAL The Connection Qualifier (*conn_qual*) was invalid.

5573 IT_ERR_INVALID_SOURCE_PATH One of the components of the Source portion of the
5574 supplied Path was invalid.

5575 IT_ERR_INVALID_SPIGOT An invalid Spigot ID was specified (*spigot_id*
5576 member of the *destination_path*).

5577	IT_ERR_RESOURCES	The requested operation failed due to insufficient resources.
5578		
5579	IT_ERR_IA_CATASTROPHE	The Interface Adapter has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it ia info t for a description of the disabled state.
5580		
5581		
5582		
5583		
5584	ERRORS	
5585	None.	
5586	APPLICATION USAGE	
5587		The resulting <i>ud_svc_handle</i> (it ud service request handle t) produced by this call will be used in calls to it ud service request to obtain information describing how to communicate with the remote UD Service described by the <i>conn_qual</i> (it conn qual t).
5588		
5589		
5590		The it ud service request call requests information from the remote UD Service. Once that remote UD Service responds, an IT_CM_MSG_UD_SERVICE_REPLY Event will be generated. The data associated with the Event, it ud svc reply event t , contains an it remote ep info t structure and other information. The <i>ud_ep_id</i> and <i>ud_ep_key</i> from the it remote ep info t , combined with the information from the <i>destination_path</i> (it path t) provides the Consumer the necessary information to perform Data Transfer Operations with the remote UD Service.
5591		
5592		
5593		
5594		
5595		
5596		
5597		Note that the <i>spigot_id</i> of the Endpoint that will be used for Data Transfer Operations with the UD Service being requested must match the <i>spigot_id</i> in the <i>destination_path</i> .
5598		
5599		See it ud service request and it ud service reply for more information.
5600	SEE ALSO	
5601		it ud service request handle free() , it ud request handle query() , it ia query() ,
5602		it ud service request() , it get pathinfo() , it path t , it cm msg events , it ep attributes t

5603 **it_ud_service_request_handle_free()**

5604 **NAME**

5605 `it_ud_service_request_handle_free` – free a previously created `it_ud_svc_req_handle_t`

5606 **SYNOPSIS**

```
5607 #include <it_api.h>
5608
5609 it_status_t it_ud_service_request_handle_free (
5610     IN          it_ud_svc_req_handle_t  ud_svc_handle
5611 );
```

5612 **DESCRIPTION**

5613 `ud_svc_handle` Unreliable Datagram (UD) Service Request Handle previously
5614 created by a call to [it_ud_service_request_handle_create](#).

5615 `it_ud_service_request_handle_free` removes an existing UD Service Request Handle and frees
5616 all associated underlying resources. Once `it_ud_service_request_handle_free` returns,
5617 `ud_svc_handle` can no longer be used in UD Service Request operations. In addition, once
5618 `it_ud_service_request_handle_free` returns, any replies to outstanding UD Service Request
5619 operations associated with this `ud_svc_handle` will be silently dropped.

5620
5621 Any IT_CM_MSG_UD_SERVICE_REPLY_EVENT Events associated with this request that
5622 have been enqueued on the Event Dispatcher (EVD) will not be removed. It is the Consumer's
5623 responsibility to dequeue and dispose of them.

5624 **RETURN VALUE**

5625 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
5626 below.

5627 IT_ERR_INVALID_UD_SVC The Unreliable Datagram Service Handle (`ud_svc_handle`)
5628 was invalid.

5629 IT_ERR_IA_CATASTROPHE The Interface Adapter has experienced a catastrophic error
5630 and is in the disabled state. None of the output parameters
5631 from this routine are valid. See [it_ia_info_t](#) for a
5632 description of the disabled state.

5633 **ERRORS**

5634 None.

5635 **SEE ALSO**

5636 [it_ud_service_request_handle_create\(\)](#), [it_ud_service_request_handle_query\(\)](#),
5637 [it_cm_msg_events](#)

it_ud_service_request_handle_query()5639 **NAME**

5640 `it_ud_service_request_handle_query` – return information about a specified
 5641 [*it_ud_svc_req_handle_t*](#)

5642 **SYNOPSIS**

```

5643 #include <it_api.h>
5644
5645 it_status_t it_ud_service_request_handle_query (
5646     IN          it_ud_svc_req_handle_t      ud_svc_handle,
5647     IN          it_ud_svc_req_param_mask_t  mask,
5648     OUT         it_ud_svc_req_param_t      *ud_svc_handle_info
5649 );
5650
5651 typedef enum {
5652     IT_UD_PARAM_ALL                = 0x00000001,
5653     IT_UD_PARAM_IA_HANDLE          = 0x00000002,
5654     IT_UD_PARAM_REQ_ID             = 0x00000004,
5655     IT_UD_PARAM_REPLY_EVD         = 0x00000008,
5656     IT_UD_PARAM_CONN_QUAL         = 0x00000010,
5657     IT_UD_PARAM_DEST_PATH         = 0x00000020,
5658     IT_UD_PARAM_PRIV_DATA         = 0x00000040,
5659     IT_UD_PARAM_PRIV_DATA_LENGTH = 0x00000080
5660 } it_ud_svc_req_param_mask_t;
5661
5662 /*
5663  * The it_ud_svc_req_param_mask_t value in the comment above
5664  * each attribute in the it_ud_svc_req_param_t structure below
5665  * is the mask value used to select that attribute in a call
5666  * to it_ud_service_request_handle_query.
5667  */
5668 typedef struct {
5669     it_ia_handle_t    ia;          /* IT_UD_PARAM_IA_HANDLE */
5670     uint32_t          request_id; /* IT_UD_PARAM_REQ_ID */
5671     it_evd_handle_t   reply_evd;  /* IT_UD_PARAM_REPLY_EVD */
5672     it_conn_qual_t    conn_qual;  /* IT_UD_PARAM_CONN_QUAL */
5673     it_path_t         destination_path; /* IT_UD_PARAM_DEST_PATH */
5674     unsigned char     private_data[IT_MAX_PRIV_DATA];
5675                                     /* IT_UD_PARAM_PRIV_DATA */
5676     size_t            private_data_length; /* IT_UD_PARAM_PRIV_DATA_LEN */
5677 } it_ud_svc_req_param_t;

```

5678 **DESCRIPTION**

5679	<code>ud_svc_handle</code>	Unreliable Datagram (UD) Service Request Handle previously created by a call to <u><i>it_ud_service_request_handle_create</i></u> .
5681	<code>mask</code>	Logical OR of flags for the requested UD Service Request Handle parameters.
5683	<code>ud_svc_handle_info</code>	Data structure containing information about the UD Service Request Handle, <code>ud_svc_handle</code> .

5685 *it_ud_service_request_handle_query* collects the desired information about the *ud_svc_handle*
 5686 passed in and returns that information in the *it_ud_svc_req_param_t* structure provided in
 5687 *ud_svc_handle_info*. On return, each field of *ud_svc_handle_info* is only valid if the
 5688 corresponding flag is set in the *mask* argument. The flag values for the *mask* appear in the
 5689 comments above each of the fields in the *it_ud_svc_req_param_t* structure. The mask value
 5690 IT_UD_PARAM_ALL causes all fields to be returned.

5691
 5692 The definition of each field in the *it_ud_svc_req_param_t* structure is as follows:

5693	<i>ia</i>	Handle for the Interface Adapter associated with this UD Service Request.
5694		
5695	<i>request_id</i>	Unique identifier associated with the it_ud_svc_req_handle_t .
5696	<i>reply_evd</i>	The Simple EVD for reply Events associated with the it_ud_svc_req_handle_t .
5697		
5698	<i>conn_qual</i>	Connection Qualifier describing the UD Service associated with the it_ud_svc_req_handle_t .
5699		
5700	<i>destination_path</i>	Path to the Destination of the it_ud_service_request operation associated with the it_ud_svc_req_handle_t .
5701		
5702	<i>private_data</i>	Opaque Private Data provided by the Consumer if the IA supports Private Data.
5703		
5704	<i>private_data_length</i>	Length of the Private Data supplied by the Consumer.

5705 **RETURN VALUE**

5706 A successful call returns IT_SUCCESS. Otherwise, an error code is returned as described
 5707 below.

5708	IT_ERR_INVALID_UD_SVC	The Unreliable Datagram Service Handle (<i>ud_svc_handle</i>) was invalid.
5709		

5710		
5711	IT_ERR_INVALID_MASK	The <i>mask</i> contained invalid flag values.

5712		
5713	IT_ERR_IA_CATASTROPHE	The Interface Adapter has experienced a catastrophic error and is in the disabled state. None of the output parameters from this routine are valid. See it_ia_info_t for a description of the disabled state.
5714		
5715		
5716		

5717 **ERRORS**

5718 None.

5719 **SEE ALSO**

5720 [*it ud service request handle create\(\)*](#), [*it ud service request handle free\(\)*](#),

5721 [*it ud service request\(\)*](#)

5722

5723

5724 **5 Data Type Manual Pages**

- 5725 [*it_aevd_notification_event_t*](#) – Aggregate Event Dispatcher Notification Event type
- 5726 [*it_affiliated_event_t*](#) – Affiliated Asynchronous Event type
- 5727 [*it_boolean_t*](#) – the Boolean type used by the IT-API
- 5728 [*it_cm_msg_events*](#) – Communication Management Message Events
- 5729 [*it_cm_req_events*](#) – Communication Management Request Events
- 5730 [*it_conn_qual_t*](#) – encapsulates all supported Connection Qualifier types
- 5731 [*it_context_t*](#) – structure describing a Consumer Context
- 5732 [*it_dg_remote_ep_addr_t*](#) – DatagramTransport Endpoint address
- 5733 [*it_dto_cookie_t*](#) – DTO Cookie type
- 5734 [*it_dto_events*](#) – Completion Event types
- 5735 [*it_dto_flags_t*](#) – flags for Send, Receive, RDMA Read & Write, RMR Bind & Unbind
- 5736 [*it_dto_status_t*](#) – definition of DTO and RMR completion asynchronous status
- 5737 [*it_ep_attributes_t*](#) – Endpoint attributes
- 5738 [*it_ep_state_t*](#) – RC and UD Endpoint state type definition.
- 5739 [*it_event_t*](#) – definition of Event data structures
- 5740 [*it_handle_t*](#) – enumeration and type definitions for IT Handles
- 5741 [*it_ia_info_t*](#) – encapsulates all Interface Adapter attributes and Spigot information
- 5742 [*it_lmr_triplet_t*](#) – structure describing a DTO buffer in a Local Memory Region
- 5743 [*it_net_addr_t*](#) – encapsulates all supported Network Address types
- 5744 [*it_path_t*](#) – describes the Path between a pair of Spigots
- 5745 [*it_software_event_t*](#) – Software Event type
- 5746 [*it_unaffiliated_event_t*](#) – Unaffiliated Asynchronous Event type

it_aevd_notification_event_t

5747

5748 **NAME**5749 `it_aevd_notification_event_t` – Aggregate Event Dispatcher Notification Event type5750 **SYNOPSIS**

```

5751 #include <it_api.h>
5752
5753 typedef struct {
5754     it_event_type_t    event_number;
5755     it_evd_handle_t    aevd;
5756     it_evd_handle_t    sevd;
5757 } it_aevd_notification_event_t;

```

5758 **DESCRIPTION**

5759 *event_number* Identifier of the Event type. Valid values:
 5760 `IT_AEVD_NOTIFICATION_EVENT`

5761 *aevd* Handle for the Aggregate Event Dispatcher (AEVD) where the
 5762 Event was queued.

5763 *sevd* Handle to the Simple Event Dispatcher (SEVD) that experienced a
 5764 Notification Event.

5765 An `IT_AEVD_NOTIFICATION_EVENT_STREAM` Event is generated when a Notification
 5766 has occurred on an SEVD associated with an AEVD with the
 5767 `IT_EVD_DEQUEUE_NOTIFICATIONS` *evd_flag* set (see [it_evd_create](#)).

5768 The AEVD Notification Event passes the Handle for the associated SEVD on which a
 5769 Notification Event has occurred.

5770 The AEVD Notification Event only applies to AEVDs. AEVDs do not overflow.

5771 **RETURN VALUE**

5772
 5773 None.

5774 **ERRORS**

5775 None.

5776 **SEE ALSO**

5777 [it_event_t](#), [it_evd_create\(\)](#), [it_evd_wait\(\)](#)

it_affiliated_event_t

5778

5779 **NAME**5780 `it_affiliated_event_t` – Affiliated Asynchronous Event type5781 **SYNOPSIS**

```

5782 #include <it_api.h>
5783
5784 typedef struct {
5785     it_event_type_t    event_number;
5786     it_evd_handle_t    evd;
5787
5788     union {
5789         it_evd_handle_t    sevd;
5790         it_ep_handle_t     ep;
5791     } u;
5792 } it_affiliated_event_t;

```

5793 **DESCRIPTION**

5794 *event_number* Identifier of the Event type. Valid values:
5795 IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE,
5796 IT_ASYNC_AFF_EP_FAILURE,
5797 IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE,
5798 IT_ASYNC_AFF_EP_REQ_DROPPED,
5799 IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION,
5800 IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA,
5801 IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION

5802 *evd* Handle for the Event Dispatcher where the Event was queued.

5803 *sevd* The Handle for the SEVD that the Implementation failed to enqueue an
5804 Event for. Valid only for the
5805 IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE Event type.

5806 *ep* The Handle for the Endpoint that experienced the Event. Valid for all
5807 asynchronous errors affiliated with Endpoint other than
5808 IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE.

5809 IT_ASYNC_AFF_EVENT_STREAM Events are generated when an Affiliated Asynchronous
5810 Event occurs. There are several types of Affiliated Asynchronous Events, and each type is
5811 identified by *event_number*.

5812 The Consumer asks for Affiliated Asynchronous Events to be delivered when it used
5813 [it_evd_create](#) to create an EVD associated with the Affiliated Asynchronous Event Stream.

5814 The following table maps the Affiliated Asynchronous Error values in the [it_event_type_t](#)
5815 enumeration to a transport independent description.

5816

it_event_type_t value	Generic Event Description
IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE	The Implementation was unable to enqueue an entry into the SEVD. Applies to all SEVD Event Streams except for IT_ASYNC_AFF_EVENT_STREAM and IT_ASYNC_UNAFF_EVENT_STREAM.
IT_ASYNC_AFF_EP_FAILURE	The local Endpoint experienced a failure when attempting to enqueue on an EVD in the <i>it_evd_overflowed</i> state or on an EVD in an error state.
IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE	The local Endpoint detected an invalid transport opcode in an incoming request it was processing.
IT_ASYNC_AFF_EP_LOCAL_ACCESS_VIOLATION	The local Endpoint detected an access violation while processing an incoming request. Note that not all incoming requests that cause an access violation will cause an Affiliated Asynchronous Event to be generated.
IT_ASYNC_AFF_EP_REQ_DROPPED	The local Endpoint could not process an incoming Send operation because the Receive Queue was empty.
IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION	The remote Endpoint connected to the local Endpoint that is furnished via this Event detected an access violation while processing an RDMA Write operation.
IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA	The remote Endpoint connected to the local Endpoint that is furnished via this Event detected corruption in the incoming data.
IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION	The remote Endpoint connected to the local Endpoint that is furnished via this Event detected an access violation while processing an RDMA Read operation.

5817

5818

5819

All Events on an IT_ASYNC_AFF_EVENT_STREAM SEVD cause Notification. See [it_evd_create](#) for details of Notification.

5820 Default overflow behavior of an IT_ASYNC_AFF_EVENT_STREAM SEVD is overflow
 5821 Notification enabled with automatic rearming. This default behavior of the SEVD is equivalent
 5822 to IT_EVD_OVERFLOW_DEFAULT cleared and IT_EVD_OVERFLOW_NOTIFY set and
 5823 IT_EVD_OVERFLOW_AUTO_RESET set. See [it_evd_create](#) for details of overflow detection.
 5824 Note that overflow of an IT_ASYNC_AFF_EVENT_STREAM SEVD generates an
 5825 IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE Event on the Unaffiliated Asynchronous
 5826 Event SEVD of the IA.

5827 EXTENDED DESCRIPTION

5828 For the Infiniband transport, the following table maps the Affiliated Asynchronous Error values
 5829 in the [it_event_type_t](#) enumeration to their corresponding “Affiliated Asynchronous Errors” as
 5830 specified in Volume 1, Chapter 11 of the Infiniband specification.

5831

it_event_type_t value	IB “Affiliated Asynchronous Error” name
IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE	CQ Error
IT_ASYNC_AFF_EP_FAILURE	Local Work Queue Catastrophic Error
IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE	Invalid Request Local Work Queue Error
IT_ASYNC_AFF_EP_LOCAL_ACCESS_VIOLATION	Local Access Violation Work Queue Error
IT_ASYNC_AFF_EP_REQ_DROPPED	(Not applicable to the IB transport.)
IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION	(Not applicable to the IB transport.)
IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA	(Not applicable to the IB transport.)
IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION	(Not applicable to the IB transport.)

5832

5833 For the VIA transport, the following table maps the Affiliated Asynchronous Error values in the
 5834 [it_event_type_t](#) enumeration to their corresponding descriptions in the “VipErrorCallback” man
 5835 page in the Appendix of the VIA specification.

5836

it_event_type_t value	VIA “VipErrorCallback” name
IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE	(Not applicable to the VIA transport.)
IT_ASYNC_AFF_EP_FAILURE	Completion Protection Error
IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE	RDMA Write Packet Abort

IT_ASYNC_AFF_EP_LOCAL_ACCESS_VIOLATION	(Not applicable to the VIA transport.)
IT_ASYNC_AFF_EP_REQ_DROPPED	Receive Queue Empty
IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION	RDMA Write Protection Error
IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA	RDMA Write Data Error
IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION	RDMA Read Protection Error

5837

5838 **RETURN VALUE**

5839 None.

5840 **ERRORS**

5841 None.

5842 **SEE ALSO**5843 [*it_event_t*, *it_evd_wait\(\)*, *it_evd_create\(\)*](#)

it_boolean_t

5844

5845 **NAME**5846 `it_boolean_t` – the Boolean type used by the API5847 **SYNOPSIS**

```
5848 #include <it_api.h>
5849
5850 typedef enum {
5851     IT_FALSE = 0,
5852     IT_TRUE  = 1
5853 } it_boolean_t;
```

5854 **DESCRIPTION**

5855 The `it_boolean_t` type is used in several data structures in the API to describe a value that can
 5856 exist in one of two different states: `true` (`IT_TRUE`), or `false` (`IT_FALSE`).

5857 **RETURN VALUE**

5858 None.

5859 **ERRORS**

5860 None.

5861 **SEE ALSO**

5862 [it_cm_msg events](#), [it_cm_req events](#), [it_ep attributes t](#), [it_evd create\(\)](#), [it_evd modify\(\)](#),
 5863 [it_evd query\(\)](#), [it_ia_info t](#), [it_rmr query\(\)](#)

it_cm_msg_events

5864

5865 **NAME**

5866 Communication Management Message Events – definitions for communication management
 5867 Events other than Connection Requests and definition of Unreliable Datagram service resolution
 5868 reply Event

5869 **SYNOPSIS**

```

5870 #include <it_api.h>
5871
5872 #define IT_MAX_PRIV_DATA 256
5873
5874 typedef enum {
5875     IT_CN_REJ_OTHER                = 0,
5876 IT_CN_REJ_TIMEOUT                = 1,
5877     IT_CN_REJ_BAD_PATH             = 2,
5878     IT_CN_REJ_STALE_CONN          = 3,
5879     IT_CN_REJ_BAD_ORD             = 4,
5880     IT_CN_REJ_RESOURCES           = 5
5881 } it_conn_reject_code_t;
5882
5883 typedef struct {
5884     it_event_type_t                event_number;
5885     it_evd_handle_t                evd;
5886     it_cn_est_identifier_t         cn_est_id;
5887     it_ep_handle_t                ep;
5888     uint32_t                       rdma_read_inflight_incoming;
5889     uint32_t                       rdma_read_inflight_outgoing;
5890     it_path_t                       dst_path;
5891     it_conn_reject_code_t          reject_reason_code;
5892     unsigned char                  private_data[IT_MAX_PRIV_DATA];
5893     it_boolean_t                   private_data_present;
5894 } it_connection_event_t;
5895
5896 typedef enum {
5897     IT_UD_SVC_EP_INFO_VALID        = 0,
5898     IT_UD_SVC_ID_NOT_SUPPORTED     = 1,
5899     IT_UD_SVC_REQ_REJECTED        = 2,
5900     IT_UD_NO_EP_AVAILABLE        = 3,
5901     IT_UD_REQ_REDIRECTED          = 4
5902 } it_ud_svc_req_status_t;
5903
5904 typedef struct {
5905     it_event_type_t                event_number;
5906     it_evd_handle_t                evd;
5907     it_ud_svc_req_handle_t         ud_svc;
5908     it_ud_svc_req_status_t        status;
5909     it_remote_ep_info_t           ep_info;
5910     it_path_t                       dst_path;
5911     unsigned char                  private_data[IT_MAX_PRIV_DATA];
5912     it_boolean_t                   private_data_present;
5913 } it_ud_svc_reply_event_t;

```

5914 **DESCRIPTION**

5915 The Communication Management Message Event Stream, `IT_CM_MSG_EVENT_STREAM`,
 5916 generates Events for all of the possible state transitions following a Connection Request as well
 5917 as for Unreliable Datagram Service Resolution replies. These Events are all the Communication
 5918 Management Events except those invoked by incoming requests (see [it_cm_req_events](#) for
 5919 those).

5920 Only one Event will be generated when a Connection is destroyed for any reason, either the
 5921 `IT_CM_MSG_CONN_DISCONNECT_EVENT` or the `IT_CM_MSG_CONN_BROKEN_`
 5922 `EVENT`, but not both. Consumer should be ready to handle either of these Events being
 5923 generated even when the local or remote Consumer called [it_ep_disconnect](#).

5924 The Connection Events are represented by the `it_connection_event_t` structure and the
 5925 Unreliable Datagram Service Resolution replies are represented by the `it_ud_svc_reply_event_t`
 5926 structure.

5927 The `it_connection_event_t` structure has the following members:

5928	<i>event_number</i>	Identifier of the Event type. Valid values: 5929 <code>IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT</code> , 5930 <code>IT_CM_MSG_CONN_ESTABLISHED_EVENT</code> , 5931 <code>IT_CM_MSG_CONN_PEER_REJECT_EVENT</code> , 5932 <code>IT_CM_MSG_CONN_NONPEER_REJECT_EVENT</code> , 5933 <code>IT_CM_MSG_CONN_DISCONNECT_EVENT</code> , 5934 <code>IT_CM_MSG_CONN_BROKEN_EVENT</code>
5935	<i>evd</i>	Handle for the Event Dispatcher where the Event was 5936 queued.
5937	<i>cn_est_id</i>	Identifier for the Connection establishment Event.
5938	<i>ep</i>	Endpoint Handle associated with Connection in progress.
5939	<i>rdma_read_inflight_incoming</i>	Maximum number of incoming simultaneous RDMA Read 5940 operations supported. Only valid if the 5941 it_ia_info.ird_support value is <code>IT_TRUE</code> and as described 5942 under Application Usage below.
5943	<i>rdma_read_inflight_outgoing</i>	Maximum number of outgoing simultaneous RDMA Read 5944 operations supported. Only valid if the it_ia_info.ord_ 5945 support value is <code>IT_TRUE</code> and as described under 5946 Application Usage below.
5947	<i>dst_path</i>	Path to Destination node supporting Service. Valid only if 5948 remote has rejected the proposed Path (<i>reject_reason_code</i> 5949 is <code>IT_CN_REJ_BAD_PATH</code>). Consumer should use 5950 <i>dst_path</i> if they wish to retry Connection attempt.
5951	<i>reject_reason_code</i>	Reason for rejection of Connection attempt.
5952	<i>private_data</i>	Private Data buffer.

5953	<i>private_data_present</i>	When it has the value IT_TRUE then Private Data is present
5954		in the <i>private_data</i> buffer above.
5955		The <i>it_ud_svc_reply_event_t</i> structure has the following members:
5956	<i>event_number</i>	Identifier of the Event type. Valid values:
5957		IT_CM_MSG_UD_SERVICE_REPLY_EVENT
5958	<i>evd</i>	Handle for the Event Dispatcher where the Event was
5959		queued.
5960	<i>ud_svc</i>	Handle for the corresponding Service Request.
5961	<i>status</i>	Completion status for Service Request.
5962	<i>ep_info</i>	Resolution of Connection Qualifier for the UD service to a
5963		specific remote Endpoint. Only valid if <i>status</i> is
5964		IT_UD_SVC_EP_INFO_VALID. See it_ep_attributes_t
5965		for the definition of the <i>it_remote_ep_info_t</i> structure.
5966	<i>dst_path</i>	Path to Destination node supporting Service. Valid only if
5967		remote has redirected (<i>status</i> is
5968		IT_UD_REQ_REDIRECTED). Path returned is complete.
5969	<i>private_data</i>	Private Data buffer.
5970	<i>private_data_present</i>	When it has the value IT_TRUE then Private Data is present
5971		in the <i>private_data</i> buffer above.

5972 EXTENDED DESCRIPTION

5973 Connection Events are described in **Error! Reference source not found.:**

Event type	Description	Notes
IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT	The passive side of a three-way Connection establishment has issued an it_ep_accept for the specified Connection establishment request identifier.	Only applies to three-way Connection establishment. Second phase of three. The Endpoint is in IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state.
IT_CM_MSG_CONN_ESTABLISHED_EVENT	The Connection identified has been established and Data Transfer Operations can be performed. This Event is generated on both the passive and active sides of a Connection.	Applies to both two-way and three-way Connection establishment. Second phase on two-way, third phase on three-way. The Endpoint is in CONNECTED state.

Event type	Description	Notes
IT_CM_MSG_CONN_DISCONNECT_EVENT	The Connection identified has been disconnected, either by the local or remote side, through a call to it_ep_disconnect . No more Data Transfer Operations posted on the Endpoint will complete successfully.	Applies to both two-way and three-way Connection establishment. The Endpoint is in IT_EP_STATE_NONOPERATIONAL state. All posted DTOs and RMRs are flushed.
IT_CM_MSG_CONN_PEER_REJECT_EVENT	The remote side of a Connection establishment request has issued it_reject for the specified Connection establishment request.	Applies to both two-way and three-way Connection establishment. The Endpoint is in IT_EP_STATE_NONOPERATIONAL state. All preposted DTOs and RMRs are flushed.
IT_CM_MSG_CONN_NONPEER_REJECT_EVENT	This Event includes all other reasons for the remote side not establishing a Connection that are not related to the remote Consumer issuing it_reject . Such reasons include overflow of the remote EVD for Connection Events, timeouts of the Connection attempt, and the passive side rejecting the proposed Path for the Connection attempt.	Applies to both two-way and three-way Connection establishment. The Endpoint is in IT_EP_STATE_NONOPERATIONAL state. All preposted DTOs and RMRs are flushed.
IT_CM_MSG_CONN_BROKEN_EVENT	The Connection identified has been disconnected by the Implementation. Causes include transport errors.	Applies to both two-way and three-way Connection establishment. The Endpoint is in IT_EP_STATE_NONOPERATIONAL state. All posted DTOs and RMRs are flushed.

5974
5975

Table 3: Connection Management Event Definitions

5976
5977

Error! Reference source not found. identifies which fields are valid in each of the Connection management message Events. For any Event, *event_number* and *evd* are always valid.

Event type	Valid fields
IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT	<i>cn_est_id, ep, private_data, private_data_present.cn_est_id</i> may not be valid if Consumer called it_ep_disconnect of the associated Endpoint at any time before the <i>cn_est_id</i> is used. The following are valid only on active side when three-way Connection establishment is used: <i>rdma_read_inflight_incoming, rdma_read_inflight_outgoing</i> .
IT_CM_MSG_CONN_ESTABLISHED_EVENT	<i>ep, private_data, private_data_present</i>
IT_CM_MSG_CONN_DISCONNECT_EVENT	<i>ep, private_data, private_data_present</i>
IT_CM_MSG_CONN_PEER_REJECT_EVENT	<i>ep, private_data, private_data_present</i>
IT_CM_MSG_CONN_NONPEER_REJECT_EVENT	<i>ep, reject_reason_code</i> If the <i>reject_reason_code</i> is IT_CN_REJ_BAD_PATH, then <i>dst_path</i> is also valid.
IT_CM_MSG_CONN_BROKEN_EVENT	<i>ep</i>

5978
5979

Table 4: Event Management Event Fields

5980
5981

Error! Reference source not found. describes the meaning of the various *reject_reason_code* values that can be present in an IT_CM_MSG_CONN_NONPEER_REJECT_EVENT.

reject_reason_code value	Description
IT_CN_REJ_OTHER	The Connection establishment attempt was rejected for some reason other than those listed below.
IT_CN_REJ_TIMEOUT	The Connection could not be established within the timeout period defined by the <i>timeout</i> member of the it_path_t that was input to it_ep_connect . This <i>reject_reason_code</i> is only returned when the local timeout period has elapsed; a timeout that occurs at the remote peer does not cause this status to be returned.
IT_CN_REJ_BAD_PATH	The passive side replied to the request to establish a Connection by rejecting the proposed Path. If the Consumer wishes to retry the Connection establishment attempt, the <i>dst_path</i> member of the Event structure contains the suggested Path to use.

reject_reason_code value	Description
IT_CN_REJ_STALE_CONN	The remote peer detected a stale Connection using the local Endpoint that the Consumer furnished as part of the Connection establishment attempt, and has initiated the cleanup process for that stale Connection. If the Consumer wishes to retry the Connection establishment attempt with the remote peer, they should either use a different Endpoint when they retry, or wait for the stale Connection cleanup process to complete before doing the retry. (The duration of the stale Connection cleanup process is implementation-dependent.)
IT_CN_REJ_BAD_ORD	When this Event is received on the passive side of a Connection establishment attempt, it means that the active side was unwilling to accept the <i>rdma_read_inflight_incoming</i> limit in the passive-side Endpoint.
IT_CN_REJ_RESOURCES	The remote peer was unable to allocate resources necessary to establish the Connection.

5982
5983

Table 5: reject_reason_code Descriptions

5984

The UD Service Resolution Reply Event is described in **Error! Reference source not found.:**

Event type	Description
IT_CM_MSG_UD_SERVICE_REPLY_EVENT	The passive side of a UD service has responded to the request for Connection Qualifier resolution.

5985
5986

Table 6: UD Service Resolution Reply Event Definitions

5987
5988

UD service resolution replies return *status* in the Event data structure as described in **Error! Reference source not found.:**

Status	Description
IT_UD_SVC_EP_INFO_VALID	Reply is valid. <i>ep_info</i> resolves the remote Endpoint associated with Connection Qualifier
IT_UD_SVC_ID_NOT_SUPPORTED	Service is not supported by remote
IT_UD_SVC_REQ_REJECTED	Request is rejected by remote
IT_UD_NO_EP_AVAILABLE	Remote is out of resources
IT_UD_REQ_REDIRECTED	Remote redirected the request

5989
5990

Table 7: Service Resolution Reply Status

5991 All Events on an IT_CM_MSG_EVENT_STREAM SEVD cause Notification. See
 5992 [it_evd_create](#) for details of Notification.

5993 Default overflow behavior of an IT_CM_MSG_EVENT_STREAM SEVD is automatic
 5994 rearming. This default behavior of the SEVD is equivalent to IT_EVD_OVERFLOW_
 5995 DEFAULT cleared and IT_EVD_OVERFLOW_NOTIFY set and IT_EVD_OVERFLOW_
 5996 AUTO_RESET set. See [it_evd_create](#) for details of overflow detection.

5997 **EXTENDED DESCRIPTION**

5998 For the Infiniband transport, **Error! Reference source not found.** maps the values in the
 5999 *reject_reason_code* field to their corresponding “Rejection Reason Code” for the REJ message
 6000 as specified in Volume 1, Chapter 12 of the Infiniband specification. Rejection Reason Codes
 6001 that are not listed in this table should never be received by a Consumer that is using this API.

reject_reason_code value	Infiniband Rejection Reason Code Number
IT_CN_REJ_OTHER	4-9, 29-31
IT_CN_REJ_TIMEOUT	None. This code is generated based upon failure to establish the Connection within a given amount of time, not upon receiving a REJ message.
IT_CN_REJ_BAD_PATH	12-17, 24-26, 32
IT_CN_REJ_STALE_CONN	10
IT_CN_REJ_BAD_ORD	27
IT_CN_REJ_RESOURCES	1, 3

6002
 6003 **Table 8: InfiniBand reject_reason_code Mapping**

6004 For the VIA transport, **Error! Reference source not found.** maps the values in the
 6005 *reject_reason_code* field to their corresponding return values from the man pages for
 6006 “VipConnectRequest” and “VipConnectAccept” in the Appendix of the VIA specification.
 6007 Return values that are not listed in the table below are manifest to Consumers of the IT-API
 6008 through a mechanism other than a IT_CM_MSG_CONN_NONPEER_REJECT_EVENT.

reject_reason_code value	VIA Return Code
IT_CN_REJ_OTHER	VipConnectAccept – VIP_INVALID_RELIABILITY_LEVEL, VIP_INVALID_QOS, VIP_TIMEOUT, VIP_ERROR_RESOURCE VipConnectRequest – VIP_NO_MATCH
IT_CN_REJ_TIMEOUT	VipConnectAccept – VIP_NOT_REACHABLE VipConnectRequest – VIP_TIMEOUT, VIP_NOT_REACHABLE

reject_reason_code value	VIA Return Code
IT_CN_REJ_BAD_PATH	None. This code is not applicable to the VIA transport.
IT_CN_REJ_STALE_CONN	None. This code is not applicable to the VIA transport.
IT_CN_REJ_BAD_ORD	None. This code is not applicable to the VIA transport.
IT_CN_REJ_RESOURCES	VipConnectRequest – VIP_ERROR_RESOURCE

6009
6010 **Table 9: VIA reject_reason_code Mapping**

6011 For the Infiniband transport, **Error! Reference source not found.** maps the *ep_info* field
6012 elements in the UD Service Resolution Event data type to InfiniBand concepts as specified in
6013 Volume 1 of the Infiniband specification.

ep_info element	IB concept
it_ud_ep_id_t	Queue Pair Number (QPN)
it_ud_ep_key_t	Queue Key

6014
6015 **Table 10: ep_info Element Mapping**

6016 **RETURN VALUE**
6017 None.

6018 **ERRORS**
6019 None.

6020 **APPLICATION USAGE**
6021 The Consumer should use the [it_event_t](#) structure if it is desired to wait for both communication
6022 management Events ([it_connection_event_t](#)) and Unreliable Datagram service resolution reply
6023 Events ([it_ud_svc_reply_event_t](#)) via the same EVD. The [it_event_t](#) structure is of sufficient size
6024 to hold either Event type.

6025 When using three-way Connection establishment, the Consumer may receive an
6026 IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT containing [rdma_read_inflight_incoming](#)
6027 and [rdma_read_inflight_outgoing](#) values that differ from those of the Endpoint in use. The
6028 Consumer should use [it_ep_modify](#) to adjust the values associated with the Endpoint to agree
6029 with those from this Event before issuing the [it_ep_accept](#) call to complete the Connection
6030 establishment.

6031 With the three-way handshake Connection establishment method, there is also a potential race
6032 condition between the Implementation generating the IT_CM_MSG_CONN_ACCEPT_
6033 ARRIVAL_EVENT Event and the Consumer calling [it_ep_disconnect](#) or [it_ep_free](#). The
6034 Consumer should not use the [cn_est_id](#) if the IT_CM_MSG_CONN_ACCEPT_
6035 ARRIVAL_EVENT Event arrives after [it_ep_disconnect](#) or [it_ep_free](#) was called, regardless of
6036 whether the call returned yet, and regardless of the Event was dequeued before or after the call

6037 was made. If the Consumer does use the *cn_est_id* then the Implementation generate an
6038 IT_ERR_INVALID_CN_EST_ID error, or it may generate a segmentation fault, or other error.

6039 Neither the Active nor the Passive side Consumer should rely upon the
6040 IT_CM_MSG_CONN_ESTABLISHED_EVENT Event containing any Private Data, even if
6041 Private Data is input to the final [it_ep_accept](#) call that causes the Connection to be established.
6042 The side that makes the final call to [it_ep_accept](#) will never see any Private Data in the
6043 IT_CM_MSG_CONN_ESTABLISHED_EVENT Event, and because of races and unreliability
6044 inherent to the Connection establishment process of many of the transports that the IT-API
6045 supports Private Data can sometimes be dropped on the other side as well.

6046 See the [it_ep_disconnect](#) man page for details on the guarantee of delivery of Private Data when
6047 disconnecting Connections.

6048 The Consumer is advised to structure their ULP so that the Active side sends the first message
6049 after a Connection has been established. This is good practice because under some
6050 circumstances the completion of the first Receive operation is what causes the
6051 IT_CM_MSG_CONN_ESTABLISHED_EVENT Event to be generated on the Passive side.
6052 Depending upon the Passive side to send the first message after a Connection has been
6053 established can potentially result in the Connection establishment process timing out rather than
6054 completing successfully.

6055 Consult the Application Usage section of [it_cm_req_events](#) for the discussion of use of Private
6056 Data.

6057 When the Consumer receives an IT_CM_MSG_UD_SERVICE_REPLY_EVENT where the
6058 *status* is IT_UD_REQ_REDIRECTED, the Consumer can retry the attempt to retrieve UD
6059 service information. In order to do so the Consumer should free up its old UD Service Request
6060 Handle (by calling [it_ud_service_request_handle_free](#)), and create a new UD Service Request
6061 Handle by passing the *dst_path* returned in the IT_CM_MSG_UD_SERVICE_REPLY_EVENT
6062 to [it_ud_service_request_handle_create](#) to create a new Handle.

6063 FUTURE DIRECTIONS

6064 When an Endpoint gets connected, the Endpoint moves to the IT_EP_STATE_CONNECTED
6065 state and an IT_CM_MSG_CONN_ESTABLISHED_EVENT Event is generated on the
6066 Connection Event Stream for that Endpoint. An Active-side Endpoint can get connected without
6067 any DTO needing to be processed by either of the Endpoints in the Connection. A Passive-side
6068 Endpoint can also usually (but not always) get connected without any DTO needing to be
6069 processed. (A Passive-side Endpoint can always get connected if a Receive DTO posted to the
6070 Passive-side Endpoint completes.) A future version of the API may not allow a Passive-side
6071 Endpoint to get connected unless a Receive DTO posted to the Passive-side Endpoint first
6072 completes.

6073 Currently when [it_reject](#) is called by the remote side during a Connection establishment attempt
6074 an IT_CM_MSG_CONN_PEER_REJECT_EVENT Event is generated on the local side to let
6075 the local Consumer know that the attempt was rejected. A future version of the API may on
6076 some transports generate an IT_CM_MSG_CONN_NONPEER_REJECT_EVENT Event
6077 instead of an IT_CM_MSG_CONN_PEER_REJECT_EVENT in this circumstance.

6078 **SEE ALSO**

6079 [*it_event t, it_evd_create\(\), it_evd_wait\(\), it_ep_modify\(\), it_listen_create\(\), it_ep_accept\(\),*](#)
6080 [*it_ep_disconnect\(\), it_reject\(\), it_address_handle_create\(\),*](#)
6081 [*it_ud_service_request_handle_free\(\), it_path t, it_ia_info t, it_ep_attributes t*](#)

it_cm_req_events

6082

6083 **NAME**

6084 Communication Management Request Events – definitions for Connection Request and
 6085 Unreliable Datagram service resolution request communication management Events

6086 **SYNOPSIS**

```

6087 #include <it_api.h>
6088
6089 typedef struct {
6090     it_event_type_t           event_number;
6091     it_evd_handle_t          evd;
6092     it_cn_est_identifier_t    cn_est_id;
6093     it_conn_qual_t           conn_qual;
6094     it_net_addr_t            source_addr;
6095     size_t                    spigot_id;
6096     uint32_t                  max_message_size;
6097     uint32_t                  rdma_read_inflight_incoming;
6098     uint32_t                  rdma_read_inflight_outgoing;
6099     unsigned char             private_data[IT_MAX_PRIV_DATA];
6100     it_boolean_t              private_data_present;
6101 } it_conn_request_event_t;
6102
6103 typedef struct {
6104     it_event_type_t           event_number;
6105     it_evd_handle_t          evd;
6106     it_ud_svc_req_identifier_t ud_svc_req_id;
6107     it_conn_qual_t           conn_qual;
6108     it_net_addr_t            source_addr;
6109     size_t                    spigot_id;
6110     unsigned char             private_data[IT_MAX_PRIV_DATA];
6111     it_boolean_t              private_data_present;
6112 } it_ud_svc_request_event_t;
  
```

6113 **DESCRIPTION**

6114 The Communication Management Request Event Stream, *IT_CM_REQ_EVENT_STREAM*,
 6115 generates Events when an incoming Connection Request Event or incoming Unreliable
 6116 Datagram service resolution request occurs.

6117 Incoming Connection Request Events are represented by the *it_conn_request_event_t* structure
 6118 and incoming Unreliable Datagram Service Resolution requests are represented by the
 6119 *it_ud_svc_request_event_t* structure.

6120 The *it_conn_req_event_t* structure has the following members:

6121	<i>event_number</i>	Identifier of the Event type. Valid values: IT_CM_REQ_CONN_REQUEST_EVENT
6122		
6123	<i>evd</i>	Handle for the Event Dispatcher where the Event was queued.
6124		
6125	<i>cn_est_id</i>	Identifier for the Connection establishment Event.

6126	<i>conn_qual</i>	Connection Qualifier on which request was received.
6127	<i>source_addr</i>	Source address of requestor.
6128	<i>spigot_id</i>	Local Spigot on which request was received.
6129	<i>max_message_size</i>	Largest message supported on Connection by the requesting remote EP. Only valid if it_ia_info.max_message_size_support is IT_TRUE.
6130		
6131		
6132	<i>rdma_read_inflight_incoming</i>	Maximum number of incoming simultaneous RDMA Read operations of requesting remote EP supported. Only valid if it_ia_info.ird_support is IT_TRUE.
6133		
6134		
6135	<i>rdma_read_inflight_outgoing</i>	Maximum number of outgoing simultaneous RDMA Read operations of requesting remote EP supported. Only valid if it_ia_info.ord_support is IT_TRUE.
6136		
6137		
6138	<i>private_data</i>	Private Data buffer.
6139	<i>private_data_present</i>	When it has the value IT_TRUE then Private Data is present in the <i>private_data</i> buffer above.
6140		
6141		The <i>it_ud_svc_request_event_t</i> structure has the following members:
6142	<i>event_number</i>	Identifier of the Event type. Valid values: IT_CM_REQ_UD_SERVICE_REQUEST_EVENT
6143		
6144	<i>evd</i>	Handle for the Event Dispatcher where the Event was queued.
6145		
6146	<i>ud_svc_req_id</i>	Identifier for the Service Request. Must be passed into the it_ud_service_reply call used to respond.
6147		
6148	<i>conn_qual:</i>	Connection Qualifier on which request was received.
6149	<i>source_addr</i>	Source address of requestor.
6150	<i>spigot_id</i>	Local Spigot on which request was received.
6151	<i>private_data</i>	Private Data buffer.
6152	<i>private_data_present:</i>	When it has the value IT_TRUE then Private Data is present in the <i>private_data</i> buffer above.
6153		
6154		

Event type	Description
IT_CM_REQ_CONN_REQUEST_EVENT	An incoming request for Connection establishment. This Connection Request is identified by the Connection establishment request identifier (<i>cn_est_id</i>) in the Event.

Event type	Description
IT_CM_REQ_UD_SERVICE_REQUEST_EVENT	An incoming request for Unreliable Datagram service resolution. This request is identified by the <i>ud_svc_req_id</i> in the Event.

Table 11: Communication Management Request Event Definitions

6155
6156

All Events on an IT_CM_REQ_EVENT_STREAM SEVD cause Notification. See [it_evd_create](#) for details of Notification.

6157
6158

Default overflow behavior of an IT_CM_REQ_EVENT_STREAM SEVD is overflow Notification disabled. This default behavior of the SEVD is equivalent to IT_EVD_OVERFLOW_DEFAULT cleared and IT_EVD_OVERFLOW_NOTIFY cleared. See [it_evd_create](#) for details of overflow detection.

6159
6160
6161
6162

6163 RETURN VALUE

6164 None.

6165 ERRORS

6166 None.

6167 APPLICATION USAGE

6168 The Consumer should use the [it_event_t](#) structure if it is desired to wait for both Connection
6169 Request Events (*it_conn_request_event_t*) and Unreliable Datagram service resolution request
6170 Events (*it_ud_svc_request_event_t*) via the same EVD. The [it_event_t](#) structure is of sufficient
6171 size to hold either Event type.

6172 The Consumer must [it_evd_create](#) an IT_CM_REQ_EVENT_STREAM Simple EVD and pass
6173 the new EVD and a Connection Qualifier to [it_listen_create](#) in order to receive
6174 IT_CM_REQ_EVENT_STREAM Events via the [it_evd_wait](#) or [it_evd_dequeue](#) calls.

6175 The *private_data_present* field indicates whether Private Data is present in the *private_data*
6176 buffer. It is the Consumer's responsibility to convey the size of the data contained in the Private
6177 Data buffer using their own ULP. Each communication management Event type may have a
6178 different maximum Private Data buffer size. The Consumer can determine the maximum
6179 possible sizes for the Private Data buffers corresponding to each of the Event types from the
6180 *it_ia_info_t* structure (for instance, *connect_private_data_len*).

6181 SEE ALSO

6182 [it_event_t](#), [it_evd_create\(\)](#), [it_evd_wait\(\)](#), [it_evd_dequeue\(\)](#), [it_ep_modify\(\)](#), [it_listen_create\(\)](#),
6183 [it_ep_accept\(\)](#), [it_ep_disconnect\(\)](#), [it_reject\(\)](#), [it_ud_service_reply\(\)](#), [it_ia_info_t](#)

it_conn_qual_t

6184

6185 **NAME**6186 `it_conn_qual_t` – encapsulates all supported Connection Qualifier types6187 **SYNOPSIS**

```

6188 #include <it_api.h>
6189
6190 /* Enumerates all the possible Connection Qualifier types supported by
6191    the API. */
6192 typedef enum {
6193
6194     /* IANA (TCP/UDP) Port Number */
6195     IT_IANA_PORT = 0x1,
6196
6197     /* InfiniBand Service ID, as described in section 12.7.3 of
6198        Volume 1 of the InfiniBand specification. */
6199     IT_IB_SERVICEID = 0x2,
6200
6201     /* VIA Connection Discriminator */
6202     IT_VIA_DISCRIMINATOR = 0x4
6203
6204 } it_conn_qual_type_t;
6205
6206 /* Defines the Connection Qualifier format for a VIA "connection
6207    discriminator". The API imposes a fixed upper bound on the
6208    discriminator size. */
6209 #define IT_MAX_VIA_DISC_LEN    64
6210
6211 typedef struct {
6212
6213     /* The total number of bytes in the array below */
6214     /* that are significant */
6215     uint16_t          len;
6216
6217     /* VIA connection discriminator, which is an array of bytes */
6218     unsigned char     discriminator[IT_MAX_VIA_DISC_LEN];
6219
6220 } it_via_discriminator_t;
6221
6222 /* This defines the Connection Qualifier for InfiniBand, which is the
6223    64-bit Service ID */
6224 typedef uint64_t      it_ib_serviceid_t;
6225
6226 /* This describes a Connection Qualifier suitable for input to
6227    several routines in the API. */
6228 typedef struct {
6229
6230     /* The discriminator for the union below. */
6231     it_conn_qual_type_t    type;
6232
6233     union {
6234

```

```

6235
6236         /* IANA Port Number, in network byte order */
6237         uint16_t         port;
6238
6239         /* InfiniBand Service ID, in network byte order */
6240         it_ib_serviceid_t serviceid;
6241
6242         /* VIA connection discriminator. */
6243         it_via_discriminator_t discriminator;
6244
6245     } conn_qual;
6246
6247 } it_conn_qual_t;

```

6248 DESCRIPTION

6249 The *it_conn_qual_t* type is used by several routines in the API to encapsulate a Connection
6250 Qualifier. A Connection Qualifier is used by a Consumer on the Active side of the Connection
6251 establishment process in the [it_ep_connect](#) routine to target the remote Consumer that should be
6252 responding to the Connection establishment attempt. It is used on the Passive side of the
6253 Connection establishment process in the [it_listen_create](#) routine to steer incoming Connection
6254 Requests to an appropriate EVD for further processing.

6255 Each Spigot on an IA can support one or more types of Connection Qualifier. All Spigots will
6256 support the IANA Port Number type of Connection Qualifier, regardless of which transport the
6257 IA that houses the Spigot is using. Which types of Connection Qualifier a Spigot supports can
6258 be determined using the [it_ia_query](#) routine.

6259 In order to aid Consumers in writing portable applications that span platforms with different
6260 native byte orders, all Connection Qualifiers that are supported by the API with the exception of
6261 the VIA “connection discriminator” are required to be input to the API in network byte order,
6262 and will be output from the API in network byte order. (The VIA “connection discriminator” is
6263 defined to be an array of bytes, and hence is not affected by which native byte order a platform
6264 uses.)

6265 RETURN VALUE

6266 None.

6267 ERRORS

6268 None.

6269 SEE ALSO

6270 [it_ep_connect\(\)](#), [it_listen_create\(\)](#), [it_ia_query\(\)](#)

6271

it_context_t

6272

6273 NAME6274 `it_context_t` – structure describing a Consumer Context**6275 SYNOPSIS**6276 `#include <it_api.h>`

6277

6278 `typedef union {`6279 `void * ptr;`6280 `uint64_t index;`6281 `} it_context_t;`**6282 DESCRIPTION**6283 The `it_context_t` union describes storage definitions for the Consumer Context associated with
6284 an IT Object Handle.6285 `ptr` storage space for an address pointer.6286 `index` storage space for an unsigned 64-bit integer.**6287 RETURN VALUE**

6288 None.

6289 ERRORS

6290 None.

6291 SEE ALSO6292 [`it_get_consumer_context\(\)`, `it_set_consumer_context\(\)`](#)

it_dg_remote_ep_addr_t

6293

6294 **NAME**6295 `it_dg_remote_ep_addr_t` - Datagram Transport Endpoint address6296 **SYNOPSIS**

```

6297 #include <it_api.h>
6298
6299 typedef struct
6300 {
6301     it_addr_handle_t      addr;
6302     it_remote_ep_info_t   ep_info;
6303 } it_ib_ud_addr_t;
6304
6305 typedef enum
6306 {
6307     IT_DG_TYPE_IB_UD
6308 } it_dg_type_t;
6309
6310 typedef struct
6311 {
6312     it_dg_type_t          type; /* IT_DG_TYPE_IB_UD */
6313     union {
6314         it_ib_ud_addr_t   ud;
6315     } addr;
6316 } it_dg_remote_ep_addr_t;

```

6317 **DESCRIPTION**

6318 For datagram transports, the Endpoint address is specified in DTO operations using the
 6319 *it_dg_remote_ep_addr_t* data structure. The structure is intended to allow support of more than
 6320 one datagram transport type.

6321 The datagram transport type is specified as *type* in the *it_dg_remote_ep_addr_t* structure. In this
 6322 revision of the API, only the InfiniBand Unreliable Datagram transport, `IT_DG_TYPE_IB_UD`,
 6323 is supported.

6324 For InfiniBand Unreliable Datagram, the transport specific Endpoint address is contained in the
 6325 *it_ib_ud_addr_t* sub-structure of the *it_dg_remote_ep_addr_t*. The components of the
 6326 InfiniBand Unreliable Datagram Endpoint address are:

6327 `addr` An Address Handle created by the Consumer using
 6328 [it_address_handle_create](#).

6329 `ep_info` An Endpoint Info structure (see [it_ep_attributes_t](#)) containing the Endpoint
 6330 ID, `ud_ep_id`, and the Endpoint Key, `ud_ep_key`. The Consumer may make
 6331 use of [it_ud_service_request](#) to obtain `ud_ep_id` and `ud_ep_key` or may
 6332 obtain them by their own means.

6333 **EXTENDED DESCRIPTION**

6334 For InfiniBand Unreliable Datagram, the Endpoint ID is equivalent to an InfiniBand QP number
 6335 and the Endpoint Key is equivalent to an InfiniBand Q_key.

6336 RETURN VALUE

6337 None.

6338 ERRORS

6339 None.

6340 FUTURE DIRECTIONS

6341 Support for Reliable Datagram Service Type may be provided in a future revision of this API.

6342

6343 SEE ALSO

6344 [*it_post_sendto\(\)*](#), [*it_post_rcvfrom\(\)*](#), [*it_address_handle_create\(\)*](#), [*it_ud_service_request\(\)*](#),

6345 [*it_ep_attributes_t*](#)

6346

it_dto_cookie_t

6347

6348 **NAME**6349 `it_dto_cookie_t` – definition of implementation-opaque Consumer cookie6350 **SYNOPSIS**

```
6351 #include <it_api.h>
6352
6353 typedef uint64_t it_dto_cookie_t;
```

6354 **DESCRIPTION**

6355 `it_dto_cookie_t` is an object that can be provided by the Consumer on every DTO or RMR
 6356 operation and is returned to the Consumer in the corresponding DTO Completion Event (see
 6357 [it_dto_events](#)) if a DTO Completion Event is generated (see [it_dto_flags_t](#)). The `it_dto_cookie_t`
 6358 object is opaque to the Implementation and is returned unchanged to the Consumer in the DTO
 6359 Completion Event corresponding to the posted DTO or RMR.

6360 **RETURN VALUE**

6361 None.

6362 **ERRORS**

6363 None.

6364 **SEE ALSO**

6365 [it_dto_events](#), [it_dto_flags_t](#), [it_post_send\(\)](#), [it_post_sendto\(\)](#), [it_post_rcv\(\)](#),
 6366 [it_post_rcvfrom\(\)](#), [it_post_rdma_read\(\)](#), [it_post_rdma_write\(\)](#), [it_rmr_bind\(\)](#), [it_rmr_unbind\(\)](#)

it_dto_events

6367

6368 **NAME**

6369

DTO and RMR Bind/Unbind Completion Event types

6370 **SYNOPSIS**

6371

```
#include <it_api.h>
```

6372

```
typedef enum {
```

6373

```
    IB_UD_IB_GRH_PRESENT = 0x01
```

6374

```
} it_dto_ud_flags_t;
```

6375

```
typedef struct {
```

6376

```
    it_event_type_t          event_number;
```

6377

```
    it_evd_handle_t         evd;
```

6378

```
    it_ep_handle_t          ep;
```

6379

```
        it_dto_cookie_t      cookie;
```

6380

```
        it_dto_status_t      dto_status;
```

6381

```
        uint32_t             transferred_length;
```

6382

```
} it_dto_cmpl_event_t;
```

6383

```
typedef struct {
```

6384

```
    it_event_type_t          event_number;
```

6385

```
    it_evd_handle_t         evd;
```

6386

```
    it_ep_handle_t          ep;
```

6387

```
        it_dto_cookie_t      cookie;
```

6388

```
        it_dto_status_t      dto_status;
```

6389

```
        uint32_t             transferred_length;
```

6390

```
        it_dto_ud_flags_t    flags;
```

6391

```
        it_ud_ep_id_t        ud_ep_id;
```

6392

```
        it_path_t            src_path;
```

6393

```
} it_all_dto_cmpl_event_t;
```

6394

6395

6396

6397 **DESCRIPTION**

6398

The DTO Completion Event Stream, `IT_DTO_EVENT_STREAM`, generates Events for all Data Transfer Operations completions as well as RMR Bind and Unbind completions.

6399

6400

Unreliable Datagram Receive Completion Events provide additional data beyond that of the other DTO Completion Events. The additional data is large enough to warrant defining a much smaller Event structure for all other DTO operations usable by Consumers interested in conserving the memory footprint of their application.

6401

6402

6403

6404

With the exception of UD Receive completions, all DTO completions, including RMR Binds and Unbinds, can be represented by the `it_dto_cmpl_event_t` structure.

6405

6406

UD Receive completions require use of the `it_all_dto_cmpl_event_t` structure. Consumers wishing to receive UD Receive and Send Completion Events on one Simple EVD or wishing to handle all possible DTO completions with one Simple EVD must use the `it_all_dto_cmpl_event_t` structure or the encompassing [it_event_t](#) structure (see [it_event_t](#)). Failure to use the `it_all_dto_cmpl_event_t` structure or [it_event_t](#) structure for UD Receive Completion Events can result in program termination.

6407

6408

6409

6410

6411

6412

The `it_dto_cmpl_event_t` structure has the following members:

6413	<i>event_number</i>	Identifier of the Event type. Valid values:
6414		IT_DTO_SEND_CMPL_EVENT,
6415		IT_DTO_RC_RECV_CMPL_EVENT,
6416		IT_DTO_RDMA_WRITE_CMPL_EVENT,
6417		IT_DTO_RDMA_READ_CMPL_EVENT,
6418		IT_RMR_BIND_CMPL_EVENT
6419	<i>evd</i>	Handle for the Event Dispatcher where the Event was queued.
6420	<i>ep</i>	Handle for the Endpoint on which the DTO was posted.
6421	<i>cookie</i>	Cookie that the Consumer associated with the DTO at the post time.
6422		See it_dto_cookie_t for details.
6423	<i>dto_status</i>	Status of completed DTO.
6424	<i>transferred_length</i>	Length of transferred message.
6425		See it_dto_status_t for values and definition of <i>dto_status</i> .
6426		The <i>transferred_length</i> field indicates the amount of data transferred in Receive operations. The
6427		content of this field is undefined for Send, RDMA Read, RDMA Write, RMR Bind, and RMR
6428		Unbind operations. This field is also only valid if <i>dto_status</i> is IT_DTO_SUCCESS, otherwise
6429		the contents are undefined.
6430		The <i>it_all_dto_cmpl_event_t</i> structure has the following additional members:
6431	<i>flags:</i>	Flags indicating additional service specific information.
6432	<i>ud_ep_id</i>	Remote Endpoint ID from incoming datagram.
6433	<i>src_path</i>	Partial Source Path information from incoming datagram.
6434		The IT_DTO_UD_RECV_CMPL_EVENT <i>event_number</i> is an additional valid value only for
6435		the <i>it_all_dto_cmpl_event_t</i> structure or it_event_t structure.
6436		The <i>flags</i> parameter indicates whether or not the InfiniBand Global Routing Header (GRH) is
6437		present in the first 40 bytes of the message payload. If GRH is present, the IT_UD_IB_
6438		GRH_PRESENT bit will be set in <i>flags</i> . If the GRH is not present (IT_UD_IB_GRH_PRESENT
6439		bit cleared in <i>flags</i>), the first 40 bytes of the payload are undefined.
6440		For an IT_DTO_UD_RECV_CMPL_EVENT, the <i>transferred_length</i> field includes the length of
6441		the transferred message plus 40 bytes regardless of the IT_UD_IB_GRH_PRESENT bit value.
6442		The remote Endpoint ID, <i>ud_ep_id</i> , is derived from the incoming datagram. See
6443		it_ep_attributes_t for more details.
6444		Partial Source address information is returned in datagram Completion Event in the <i>src_path</i>
6445		structure element. See Application Usage, below.
6446		The <i>src_path</i> can hold more information than is returned in the Completion Event. The members
6447		of the it_path_t structure that are pertinent to a datagram Completion Event are listed in the table
6448		below. For each member, the corresponding Infiniband Datagram addressing information that
6449		the member corresponds to is also identified. For a detailed explanation of the semantics

6450 associated with the Datagram addressing information, see Chapter 11.4.2.1 Poll For Completion
 6451 in the Infiniband Architecture Release 1.1 specification.

it_path_t member	Unreliable Datagram Completion Addressing Information
ib.sl	Service level
ib.remote_port_lid	Source LID

6452

6453 IT_DTO_EVENT_STREAM Events may or may not cause Notification depending on the use of
 6454 DTO flags (see [it_dto_flags_t](#)). See [it_evd_create](#) for details of Notification.

6455 Default overflow behavior of an IT_DTO_EVENT_STREAM SEVD is overflow Notification
 6456 enabled. The behavior of the SEVD is equivalent to IT_EVD_OVERFLOW_DEFAULT cleared
 6457 and IT_EVD_OVERFLOW_NOTIFY set. Once an IT_DTO_EVENT_STREAM SEVD
 6458 overflows, it can not be rearmed. See [it_evd_create](#) for details of overflow detection.

6459 Overflow of an IT_DTO_EVENT_STREAM SEVD is catastrophic for the associated Endpoint
 6460 or Endpoints. Each Endpoint is transitioned to the IT_EP_STATE_NONOPERATIONAL state
 6461 as defined in [it_ep_state_t](#). The Endpoints are unrecoverable; [it_ep_free](#) must be called for all
 6462 Endpoints sharing the same SEVD on which the overflow occurred.

6463 **RETURN VALUE**

6464 None.

6465 **ERRORS**

6466 None.

6467 **APPLICATION USAGE**

6468 Within an IT_DTO_UD_RECV_CMPL_EVENT Event, the *src_path* member returned contains
 6469 insufficient information to identify the remote Endpoint. To resolve the remote Endpoint Path,
 6470 the user should pass *src_path* returned in this Event to [it_address_handle_create](#) with the
 6471 IT_AH_PATH_COMPLETE bit cleared. [it_address_handle_create](#) will complete the resolution
 6472 of the Path.

6473 **SEE ALSO**

6474 [it_post_send\(\)](#), [it_post_sendto\(\)](#), [it_post_rcv\(\)](#), [it_post_rcvfrom\(\)](#), [it_post_rdma_read\(\)](#),
 6475 [it_post_rdma_write\(\)](#), [it_dto_status_t](#), [it_dto_flags_t](#), [it_event_t](#), [it_evd_create\(\)](#), [it_evd_wait\(\)](#),
 6476 [it_address_handle_create\(\)](#), [it_ep_state_t](#), [it_ep_free\(\)](#), [it_ep_reset\(\)](#), [it_path_t](#), [it_dto_cookie_t](#)

6477

it_dto_flags_t

6478

NAME

6479 *it_dto_flags_t* – DTO flags for Send, Receive, RDMA Read, RDMA Write, RMR Bind and
 6480 RMR Unbind operations
 6481

SYNOPSIS

```

6482 #include <it_api.h>
6483
6484 typedef enum
6485 {
6486     /* If flag set, completion generates a local event */
6487     IT_COMPLETION_FLAG          = 0x01,
6488
6489     /* If flag set, completion cause local Notification */
6490     IT_NOTIFY_FLAG             = 0x02,
6491
6492     /* If flag set, receipt of DTO at remote will cause Notification at
6493     remote */
6494     IT_SOLICITED_WAIT_FLAG     = 0x04,
6495
6496     /* If flag set, DTO processing will not start if
6497     previously posted RDMA Reads are not complete. */
6498     IT_BARRIER_FENCE_FLAG     = 0x08,
6499 } it_dto_flags_t;
6500
```

DESCRIPTION

6501 *it_dto_flags* Flags for posted DTOs: Send, Receive, RDMA Read, RDMA Write, RMR
 6502 Bind and RMR Unbind.
 6503

6504 Values for *it_dto_flags* are constructed by a bitwise-inclusive OR of flags from the following
 6505 discussion.
 6506

6507 Any combination of the following may be used subject to Restrictions as noted:

IT_COMPLETION_FLAG

6509 If set, generate a Completion Event for this DTO, else do not generate a Completion Event
 6510 unless there is an error. If there is an error, the Completion Event will be generated with
 6511 Notification regardless of *IT_NOTIFY_FLAG* value.

6512 If not set, then the completion of a subsequent DTO on the same work queue of the same
 6513 Endpoint with this flag set or with error completion will indicate the successful
 6514 completion of prior DTO(s) with this flag cleared.

Restrictions

6515 *IT_COMPLETION_FLAG* may be set or cleared only for Send, RDMA Write, RDMA Read,
 6516 RMR Bind and RMR Unbind operations on a Reliable Connection Service Type, and may be
 6517 set or cleared only for Send operation on an Unreliable Datagram Service Type.
 6518

6519

6520 *IT_COMPLETION_FLAG* must be set for all Receive DTO operations on all Service Types
 6521 (Reliable Connection and Unreliable Datagram). Posting a Receive DTO operation with
 6522 *IT_COMPLETION_FLAG* cleared is an error.

6523 **IT_NOTIFY_FLAG**

6524 If set, generate Notification of completion of the DTO/RMR.

6525 **Restrictions**

6526 *IT_NOTIFY_FLAG* may be set or cleared on all DTO and RMR operations on a Reliable
 6527 Connected Service Type, and may be set or cleared on Send and Receive operations on an
 6528 Unreliable Datagram Service Type. It is an error to set *IT_NOTIFY_FLAG* if
 6529 *IT_COMPLETION_FLAG* is clear.

6530

6531 A completion will be generated with the Notification for a Receive DTO if the matching
 6532 received Send DTO had been posted at the remote with the *IT_SOLICITED_WAIT_FLAG*
 6533 set regardless of the *IT_NOTIFY_FLAG* of the posted Receive DTO.

6534 **IT_SOLICITED_WAIT_FLAG**

6535 If set, the Send DTO operation will request completion Notification for the matching Receive
 6536 on the other side of the Connection or, for Unreliable Datagram, for the matching Receive at
 6537 the remote datagram Endpoint.

6538 **Restrictions**

6539 *IT_SOLICITED_WAIT_FLAG* is supported only for Send operations for all Service Types. It
 6540 is an error to specify *IT_SOLICITED_WAIT_FLAG* on other operations.

6541

6542 If set, requests Notification of completion of the matching remote Receive DTO regardless of
 6543 the value of the *IT_NOTIFY_FLAG* on the Receive DTO.

6544 **IT_BARRIER_FENCE_FLAG**

6545 If set, then the DTO/RMR operation will not be started until all previously posted RDMA
 6546 Read requests to the Endpoint have been completed.

6547 **Restrictions**

6548 If the service does not support RDMA Read, it is an error to set this flag. Specifically, it is an
 6549 error to set *IT_BARRIER_FENCE_FLAG* on a DTO on UD service.

6550

6551 *IT_BARRIER_FENCE_FLAG* must be cleared for all Receive DTO operations on any
 6552 Service Types. Posting a Receive DTO operation with *IT_BARRIER_FENCE_FLAG* set is an
 6553 error.

6554 **EXTENDED DESCRIPTION**

6555 The following table lists all DTO and RMR operations and details the legal *it_dto_flags* values
 6556 on each.

6557

DTO or RMR operation	Legal it_dto_flags combinations
it_post_send	All possible combinations subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.
it_post_sendto	IT_BARRIER_FENCE_FLAG may not be used. All other possible combinations of the remaining flags are legal subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.
it_post_rcv	IT_COMPLETION_FLAG must be specified. IT_BARRIER_FENCE_FLAG may not be used. IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations of the remaining flags are legal.
it_post_rcvfrom	IT_COMPLETION_FLAG must be specified. IT_BARRIER_FENCE_FLAG may not be used. IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations of the remaining flags are legal.
it_post_rdma_read	IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.
it_post_rdma_write	IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.
it_rmr_bind	IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.
it_rmr_unbind	IT_SOLICITED_WAIT_FLAG may not be used. All other possible combinations subject to the constraint that IT_NOTIFY_FLAG may only be set if IT_COMPLETION_FLAG is also set. All flags cleared is a legal value.

6558

6559

6560

The following table lists the DTOs on each Service Type on which each flag value is supported.

it_dto_flags_t Value	Supported DTO on RC	Supported DTO on UD
IT_COMPLETION_FLAG	Send, RDMA Read, RDMA Write, RMR Bind, RMR Unbind	Recvfrom
IT_NOTIFY_FLAG	Send, Recv, RDMA Read, RDMA Write, RMR Bind, RMR Unbind	Sendto, Recvfrom

IT_SOLICITED_WAIT_FLAG	Send	Sendto
IT_BARRIER_FENCE_FLAG	Send, RDMA Read, RDMA Write, RMR Bind, RMR Unbind	N/A

6561

6562

6563

As stated in the DESCRIPTION section, *IT_COMPLETION_FLAG* must be set on Recv and Recvfrom DTOs.

6564

6565

As stated in the DESCRIPTION section, *IT_BARRIER_FENCE_FLAG* must be cleared on Recv DTO for RC and must be cleared on Sendto DTO as well as cleared on Recvfrom DTO for UD.

6566

6567

6568

For the Infiniband transport, the following table maps the values in the *it_dto_flags_t* enumeration to their corresponding concepts as specified in Volume 1 of the Infiniband specification.

it_dto_flags_t Value	IB Concept
IT_COMPLETION_FLAG	May be implemented using Unsignalled Completions concept
IT_NOTIFY_FLAG	None but can be supported by Implementation.
IT_SOLICITED_WAIT_FLAG	Solicited Event
IT_BARRIER_FENCE_FLAG	Fence Indicator

6569

6570

6571

For the VIA transport, the following table maps the values in the *it_dto_flags_t* enumeration to their corresponding concepts as documented in the VIA specification.

6572

it_dto_flags_t Value	VIA Concept
IT_COMPLETION_FLAG	May be implemented by associating completion queues with work queues
IT_NOTIFY_FLAG	VipSendNotify, VipRecvNotify, VipCQNotify
IT_SOLICITED_WAIT_FLAG	Not applicable
IT_BARRIER_FENCE_FLAG	Queue Fence Bit

6573

RETURN VALUE

6574

None.

6575

ERRORS

6576

None.

6577

APPLICATION USAGE

6578

See Application Usage in [it_ep_state_t](#) for discussion of flushing DTO completions when the DTOs have *IT_COMPLETION_FLAG* cleared.

6579

6580 When posting Send DTOs with Completion Suppression (IT_COMPLETION_FLAG cleared) to
6581 an Endpoint, the Consumer is advised to enqueue at least one DTO with
6582 IT_COMPLETION_FLAG set in every *max_request_dtos* number of postings to the Endpoint,
6583 in order to preserve the capability to recover from failures.

6584 **SEE ALSO**

6585 [*it_post_send\(\)*](#), [*it_post_sendto\(\)*](#), [*it_post_rcv\(\)*](#), [*it_post_rcvfrom\(\)*](#), [*it_post_rdma_read\(\)*](#),
6586 [*it_post_rdma_write\(\)*](#), [*it_rmr_bind\(\)*](#), [*it_rmr_unbind\(\)*](#), [*it_dto_status_t*](#), [*it_dto_events*](#),
6587 [*it_ep_state_t*](#)

6588

it_dto_status_t

6589

6590 **NAME**6591 `it_dto_status_t` – definition of DTO and RMR completion status6592 **SYNOPSIS**

```

6593 #include <it_api.h>
6594
6595 typedef enum {
6596     IT_DTO_SUCCESS                = 0,
6597     IT_DTO_ERR_LOCAL_LENGTH      = 1,
6598     IT_DTO_ERR_LOCAL_EP         = 2,
6599     IT_DTO_ERR_LOCAL_PROTECTION = 3,
6600     IT_DTO_ERR_FLUSHED          = 4,
6601     IT_RMR_OPERATION_FAILED     = 5,
6602     IT_DTO_ERR_BAD_RESPONSE     = 6,
6603     IT_DTO_ERR_REMOTE_ACCESS    = 7,
6604     IT_DTO_ERR_REMOTE_RESPONDER = 8,
6605     IT_DTO_ERR_TRANSPORT        = 9,
6606     IT_DTO_ERR_RECEIVER_NOT_READY = 10,
6607     IT_DTO_ERR_PARTIAL_PACKET   = 11
6608 } it_dto_status_t;

```

6609 **DESCRIPTION**

6610 Any successfully initiated Data Transfer Operation (i.e. Send, Receive, RDMA Read, or RDMA
6611 Write) or RMR operation (i.e. RMR Bind or RMR Unbind) can return its completion status
6612 asynchronously via an Event enqueued on an SEVD. For some DTOs, the Consumer can
6613 control whether an Event is generated via the `IT_COMPLETION_FLAG` (see [it_dto_flags_t](#)). If
6614 an Event is generated, the completion status is contained in the `it_dto_status_t`.

6615 If the completion status is anything other than `IT_DTO_SUCCESS` for a Reliable Connected
6616 Endpoint, the Connection will be broken.

6617 The table below enumerates all of the allowed values for `it_dto_status_t`. For each value, a
6618 description of what the value means and the applicable operations on RC and on UD is shown.

it_dto_status_t Value	Description	Applicable RC Operations	Applicable UD Operations
IT_DTO_SUCCESS	The DTO completed successfully.	Send Recv RDMA Read RDMA Write RMR Bind RMR Unbind	Sendto Recvfrom
IT_DTO_ERR_LOCAL_LENGTH	The length of the incoming DTO was larger than max_dto_payload_size for the Endpoint	Recv	Recvfrom
IT_DTO_ERR_LOCAL_LENGTH	The length of the outgoing DTO was larger than max_dto_payload_size for the Endpoint.	Send RDMA Read RDMA Write	Sendto
IT_DTO_ERR_LOCAL_LENGTH	The total length of the buffers associated with a Receive DTO was too small to hold all the incoming data from a Send DTO	Recv	Recvfrom
IT_DTO_ERR_LOCAL_EP	An internal local Endpoint consistency error was detected while processing a DTO.	Send Recv RDMA Read RDMA Write RMR Bind RMR Unbind	Sendto Recvfrom

it_dto_status_t Value	Description	Applicable RC Operations	Applicable UD Operations
IT_DTO_ERR_LOCAL_PROTECTION	One of the segments in the DTO caused a protection violation when the DTO was processed. Possible causes for this error include the LMR in the segment wasn't valid, the range specified by the <i>addr</i> and <i>length</i> in the segment was outside the bounds of the LMR, the Protection Zone associated with the LMR didn't match the Protection Zone of the Endpoint that the DTO was posted to, or an attempt was made to access the LMR in a way that conflicted with its access permissions.	Send Recv RDMA Read RDMA Write	Sendto Recvfrom
IT_DTO_ERR_FLUSHED	The Endpoint entered the IT_EP_STATE_NONOPERATIONAL state before processing of the DTO could begin.	Send Recv RDMA Read RDMA Write RMR Bind RMR Unbind	Sendto Recvfrom

it_dto_status_t Value	Description	Applicable RC Operations	Applicable UD Operations
IT_RMR_OPERATION_FAILED	An RMR operation failed due to a protection violation. Possible causes for this error include LMR specified in the <i>it_rmr_bind</i> (or <i>it_rmr_unbind</i>) call was invalid, the range specified by the address and length in the call was outside the bounds of the LMR, the Protection Zones associated with the LMR, RMR and Endpoint to which the RMR operation was posted didn't match, or an attempt was made to grant access through the RMR that conflicted with the access allowed by either the LMR or the Endpoint.	RMR Bind RMR Unbind	N/A
IT_DTO_ERR_BAD_RESPONSE	The DTO operation that was posted to the Request Queue was responded to with an unexpected transport opcode	Send RDMA Read RDMA Write	N/A

it_dto_status_t Value	Description	Applicable RC Operations	Applicable UD Operations
IT_DTO_ERR_REMOTE_ACCESS	A protection violation was detected at the remote end when processing an RDMA DTO operation. Possible causes include a Protection Zone mismatch between the RMR and the Endpoint that is responding to the RDMA DTO operation, an attempt being made to do an RDMA Read or Write using an RMR that doesn't have those permissions enabled, or an attempt being made to do an RDMA Read or Write when the responding Endpoint doesn't have those permissions enabled.	RDMA Read RDMA Write	N/A
IT_DTO_ERR_REMOTE_RESPONDER	A DTO operation could not be completed at the remote end. Possible causes for this error include the remote Endpoint experiencing a condition causing an IT_DTO_ERR_LOCAL_EP error to be returned.	Send RDMA Read RDMA Write	N/A
IT_DTO_ERR_TRANSPORT	The underlying transport could not successfully transfer the data for the DTO operation. Possible causes for this error include the remote IA not responding, the DTO data was corrupted in the process of transmission, or the network fabric being used by the IA is broken.	Send Receive RDMA Read RDMA Write	N/A

it_dto_status_t Value	Description	Applicable RC Operations	Applicable UD Operations
IT_DTO_ERR_RECEIVER_NOT_READY	The DTO operation could not be processed because the responding side repeatedly indicated that it had no resources to do so.	Send RDMA Read RDMA Write	N/A
IT_DTO_ERR_PARTIAL_PACKET	The data delivered by the Receive DTO was truncated. The contents of the receiver's buffer are unspecified.	Receive	N/A

6619 **EXTENDED DESCRIPTION**

6620
6621
6622

For the Infiniband transport, the following table maps the values in the *it_dto_status_t* enumeration to their corresponding “Completion Return Status” values as specified in Volume 1, Chapter 11 of the Infiniband specification.

it_dto_status_t Value	IB “Completion Return Status” Name
IT_DTO_SUCCESS	Success
IT_DTO_ERR_LOCAL_LENGTH	Local Length Error
IT_DTO_ERR_LOCAL_EP	Local QP Operation Error
IT_DTO_ERR_LOCAL_PROTECTION	Local Protection Error
IT_DTO_ERR_FLUSHED	Work Request Flushed Error
IT_RMR_OPERATION_FAILED	Memory Window Bind Error
IT_DTO_ERR_BAD_RESPONSE	Bad Response Error
IT_DTO_ERR_REMOTE_ACCESS	Remote Access Error
IT_DTO_ERR_REMOTE_RESPONDER	Remote Operation Error
IT_DTO_ERR_TRANSPORT	Transport Retry Counter Exceeded
IT_DTO_ERR_RECEIVER_NOT_READY	RNR Retry Counter Exceeded
IT_DTO_ERR_PARTIAL_PACKET	(Not applicable to the IB transport.)

6623

6624
6625
6626

For the VIA transport, the following table maps the values in the *it_dto_status_t* enumeration to their corresponding bits in the Descriptor Control Segment “Status” field, as documented in the Appendix of the VIA specification.

it_dto_status_t Value	VIA “Status Bit” Name
IT_DTO_SUCCESS	Done

it_dto_status_t Value	VIA “Status Bit” Name
IT_DTO_ERR_LOCAL_LENGTH	Local Length Error
IT_DTO_ERR_LOCAL_EP	Local Format Error
IT_DTO_ERR_LOCAL_PROTECTION	Local Protection Error
IT_DTO_ERR_FLUSHED	Descriptor Flushed
IT_RMR_OPERATION_FAILED	(There is no operation corresponding to RMR Bind or RMR Unbind in VIA, but this error can still be returned from an IA that is utilizing the VIA transport. The Implementation synthesizes the RMR operation for VIA.)
IT_DTO_ERR_BAD_RESPONSE	(Not applicable to the VIA transport.)
IT_DTO_ERR_REMOTE_ACCESS	RDMA Protection Error
IT_DTO_ERR_REMOTE_RESPONDER	(Not applicable to the VIA transport.)
IT_DTO_ERR_TRANSPORT	Transport Error
IT_DTO_ERR_RECEIVER_NOT_READY	(Not applicable to the VIA transport.)
IT_DTO_ERR_PARTIAL_PACKET	Partial Packet Error

6627 **RETURN VALUE**

6628 None.

6629 **ERRORS**

6630 None.

6631 **SEE ALSO**

6632 [*it_post_send\(\)*](#), [*it_post_sendto\(\)*](#), [*it_post_rcv\(\)*](#), [*it_post_rcvfrom\(\)*](#), [*it_post_rdma_read\(\)*](#),
6633 [*it_post_rdma_write\(\)*](#), [*it_rmr_bind\(\)*](#), [*it_rmr_unbind\(\)*](#)

it_ep_attributes_t

```

6634
6635 NAME
6636     it_ep_attributes – Endpoint attributes

6637 SYNOPSIS
6638     #include <it_api.h>
6639
6640     typedef uint32_t    it_ud_ep_id_t;
6641     typedef uint32_t    it_ud_ep_key_t;
6642
6643     typedef enum {
6644         IT_EP_PARAM_ALL                = 0x00000001,
6645         IT_EP_PARAM_IA                 = 0x00000002,
6646         IT_EP_PARAM_SPIGOT             = 0x00000004,
6647         IT_EP_PARAM_STATE              = 0x00000008,
6648         IT_EP_PARAM_SERV_TYPE         = 0x00000010,
6649         IT_EP_PARAM_PATH               = 0x00000020,
6650         IT_EP_PARAM_PZ                 = 0x00000040,
6651         IT_EP_PARAM_REQ_SEVD          = 0x00000080,
6652         IT_EP_PARAM_RECV_SEVD         = 0x00000100,
6653         IT_EP_PARAM_CONN_SEVD         = 0x00000200,
6654         IT_EP_PARAM_RDMA_RD_ENABLE    = 0x00000400,
6655         IT_EP_PARAM_RDMA_WR_ENABLE    = 0x00000800,
6656         IT_EP_PARAM_MAX_RDMA_READ_SEG = 0x00001000,
6657         IT_EP_PARAM_MAX_RDMA_WRITE_SEG = 0x00002000,
6658         IT_EP_PARAM_MAX_IRD           = 0x00004000,
6659         IT_EP_PARAM_MAX_ORD           = 0x00008000,
6660         IT_EP_PARAM_EP_ID             = 0x00010000,
6661         IT_EP_PARAM_EP_KEY            = 0x00020000,
6662         IT_EP_PARAM_MAX_PAYLOAD       = 0x00040000,
6663         IT_EP_PARAM_MAX_REQ_DTO       = 0x00080000,
6664         IT_EP_PARAM_MAX_RECV_DTO      = 0x00100000,
6665         IT_EP_PARAM_MAX_SEND_SEG      = 0x00200000,
6666         IT_EP_PARAM_MAX_RECV_SEG      = 0x00400000
6667     } it_ep_param_mask_t;
6668
6669     /*
6670     * the it_ep_param_mask_t value in the comment beside or
6671     * following each attribute is the mask value used to select
6672     * the attribute in the it\_ep\_query and it\_ep\_modify calls
6673     */
6674     typedef struct {
6675         it_boolean_t    rdma_read_enable;
6676                         /* IT_EP_PARAM_RDMA_RD_ENABLE */
6677         it_boolean_t    rdma_write_enable;
6678                         /* IT_EP_PARAM_RDMA_WR_ENABLE */
6679         size_t          max_rdma_read_segments;
6680                         /* IT_EP_PARAM_MAX_RDMA_READ_SEG */
6681         size_t          max_rdma_write_segments;
6682                         /* IT_EP_PARAM_MAX_RDMA_WRITE_SEG */
6683         uint32_t        rdma_read_inflight_incoming;
6684                         /* IT_EP_PARAM_MAX_IRD */

```

```

6685         uint32_t    rdma_read_inflight_outgoing;
6686                 /* IT_EP_PARAM_MAX_ORD */
6687     } it_rc_only_attributes_t;
6688
6689     typedef struct {
6690         it_ud_ep_id_t    ud_ep_id;    /* IT_EP_PARAM_EP_ID */
6691         it_ud_ep_key_t   ud_ep_key;   /* IT_EP_PARAM_EP_KEY */
6692     } it_remote_ep_info_t;
6693
6694     typedef struct {
6695         it_remote_ep_info_t    ep_info;
6696
6697     } it_ud_only_attributes_t;
6698
6699     typedef union {
6700         it_rc_only_attributes_t    rc;
6701         it_ud_only_attributes_t    ud;
6702     } it_service_attributes_t;
6703
6704     typedef struct {
6705         size_t    max_dto_payload_size;    /* IT_EP_PARAM_MAX_PAYLOAD */
6706         size_t    max_request_dtos;       /* IT_EP_PARAM_MAX_REQ_DTO */
6707         size_t    max_recv_dtos;         /* IT_EP_PARAM_MAX_RECV_DTO */
6708         size_t    max_send_segments;     /* IT_EP_PARAM_MAX_SEND_SEG */
6709         size_t    max_recv_segments;     /* IT_EP_PARAM_MAX_RECV_SEG */
6710
6711         it_service_attributes_t    srv;
6712     } it_ep_attributes_t;

```

6713 DESCRIPTION

6714 `it_ep_attributes` List of Endpoint attributes. The `it_service_attributes_t` union
6715 elements are discriminated by `service_type` found in the
6716 [it_ep_param_t](#) structure in the [it_ep_query](#) man page. Mask values
6717 for query and modify of Endpoint attributes appear as comments to
6718 each attribute.

Attribute	Description	Service Type	Modifiable?
<code>max_dto_payload_size</code>	<p>Maximum message transfer size for the Endpoint. It specifies the maximum amount of payload data that Consumer will transfer in a single DTO Send or Receive message in either direction on the Endpoint.</p> <p>For RC only, it also specifies the maximum payload data size for RDMA Reads and Writes posted on the Endpoint.</p>	UD and RC	For RC, only when Endpoint is in the IT_EP_STATE_NONOPERATIONAL or in the IT_EP_STATE_UNCONNECTED states. For UD, only on creation.

Attribute	Description	Service Type	Modifiable?
max_request_dtos	Maximum number of outstanding Send, Sendto, RDMA Read, RDMA Write DTOs, RMR Bind and RMR Unbind operations combined that a Consumer can submit to the Endpoint. If the Consumer attempts to post more than this number of request DTOs simultaneously, an error will be returned from the it_post_send , it_post_rdma_read , etc., routines.	UD and RC	Subject to the setting of the <i>resizable_work_queue</i> field in the it_ia_info_t for this IA.
max_rcv_dtos	Maximum number of outstanding Recv or Recvfrom DTOs that a Consumer can submit to the Endpoint. If the Consumer attempts to post more than this number of Receive DTOs simultaneously, an error will be returned from the it_post_rcv or it_post_rcvfrom routines.	UD and RC	Subject to the setting of the <i>resizable_work_queue</i> field in the it_ia_info_t for this IA.
max_send_segments	Maximum number of data segments for a local buffer that the Consumer specifies for a posted Send or Sendto DTO for the Endpoint.	UD and RC	Only on creation.
max_rcv_segments	Maximum number of data segments for a local buffer that the Consumer specifies for a posted Recv or Recvfrom DTO for the Endpoint.	UD and RC	Only on creation.
ud_ep_id	Local Endpoint ID for this Endpoint.	UD only	Never – this is a READ-ONLY attribute.
ud_ep_key	Local Endpoint key for this Endpoint.	UD only	In any state.
rdma_read_enable	Flag allowing Consumer to enable or disable incoming RDMA Read operations on this Endpoint.	RC only	In any state.
rdma_write_enable	Flag allowing Consumer to enable or disable incoming RDMA Write operations on this Endpoint.	RC only	In any state.

Attribute	Description	Service Type	Modifiable?
max_rdma_read_segments	Maximum number of data segments for a local buffer that the Consumer specifies for a posted RDMA Read DTO for the Endpoint.	RC only	Only on creation.
max_rdma_write_segments	Maximum number of data segments for a local buffer that the Consumer specifies for a posted RDMA Write DTO for the Endpoint.	RC only	Only on creation.
rdma_read_inflight_incoming	Maximum number of incoming RDMA Reads from the remote side of the connected Endpoint that can be outstanding simultaneously.	RC only	When Endpoint is in the IT_EP_STATE_UNCONNECTED or IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state.
rdma_read_inflight_outgoing	Maximum number of outgoing RDMA Reads of the connected Endpoint that can be outstanding simultaneously.	RC only	When Endpoint is in the IT_EP_STATE_UNCONNECTED or IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state.
ep_state	The current state of the Endpoint.	UD and RC	Never – this is a READ-ONLY attribute.

6719

6720

6721

6722

6723

Since the Implementation is at liberty to allocate more resources than requested by the Consumer, the Consumer is advised to use [it_ep_query](#) to determine the Implementation-assigned values. All guarantees and warnings are with respect to the Implementation-assigned values.

6724

6725

Exceeding *max_request_dtos* or *max_rcv_dtos* using the post DTO and post RMR operations will result in the post operation returning an error or completing in error.

6726

6727

Posting more RDMA Read operations than specified in *rdma_read_inflight_outgoing* is not an error and will have no adverse effects.

6728

RETURN VALUE

6729

None.

6730

ERRORS

6731

None.

6732

FUTURE DIRECTIONS

6733

Some new Service Types may be added in the future.

6734 **SEE ALSO**
6735 [*it_ep_rc_create\(\)*](#), [*it_ep_ud_create\(\)*](#), [*it_ep_query\(\)*](#), [*it_ep_modify\(\)*](#), [*it_ia_info_t*](#)

it_ep_state_t

6736

6737 **NAME**

6738 it_ep_state_t – RC and UD Endpoint state type definition

6739 **SYNOPSIS**

```

6740 #include <it_api.h>
6741
6742 typedef enum
6743 {
6744     IT_EP_STATE_UNCONNECTED                = 0,
6745     IT_EP_STATE_ACTIVE1_CONNECTION_PENDING = 1,
6746     IT_EP_STATE_ACTIVE2_CONNECTION_PENDING = 2,
6747     IT_EP_STATE_PASSIVE_CONNECTION_PENDING = 3,
6748     IT_EP_STATE_CONNECTED                  = 4,
6749     IT_EP_STATE_NONOPERATIONAL             = 5
6750 } it_ep_state_rc_t;
6751
6752 typedef enum
6753 {
6754     IT_EP_STATE_UD_NONOPERATIONAL          = 0,
6755     IT_EP_STATE_UD_OPERATIONAL             = 1
6756 } it_ep_state_ud_t;
6757
6758 typedef union
6759 {
6760     it_ep_state_rc_t      rc;
6761     it_ep_state_ud_t      ud;
6762 } it_ep_state_t;

```

6763 **DESCRIPTION**

6764 The following table identifies and describes the RC Endpoint states. For each state, the table lists
 6765 the API routines that can be legally applied to an RC Endpoint in that state.

6766 Whenever an Endpoint transitions its state, at most one Event is generated for that transition.
 6767 The Endpoints state transitions before the Communication Management Message Event is
 6768 enqueued. This guarantees that the Consumer can only dequeue Events after the state transition
 6769 has occurred. Subsequent state transitions and their related Events will occur regardless of
 6770 whether the Consumer is dequeuing the Events.

6771

Reliable Connection Endpoint states	Description of state	Allowed calls
IT_EP_STATE_UNCONNECTED	The Endpoint is not Connected to another nor is there a pending Connection Establishment related to the Endpoint. The Endpoint is available to be used in a Connection Establishment. When an Endpoint is first created by calling	<i>it_ep_accept</i> , <i>it_ep_connect</i> , <i>it_ep_free</i> , <i>it_ep_modify</i> , <i>it_ep_query</i> , <i>it_get_consumer_context</i> , <i>it_get_handle_type</i> , <i>it_post_rcv</i> , <i>it_set_consumer_context</i>

	<i>it ep rc create</i> it is in this state.	
IT_EP_STATE_ACTIVE1_CONNECTION_PENDING	The Active side Endpoint has initiated a Connection Establishment.	<i>it ep free</i> , <i>it ep modify</i> , <i>it ep query</i> , <i>it get consumer context</i> , <i>it get handle type</i> , <i>it post rcv</i> , <i>it set consumer context</i> , <i>it ep disconnect</i>
IT_EP_STATE_ACTIVE2_CONNECTION_PENDING	The Active side Endpoint has initiated a Connection Establishment and has received an IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT Event because the Passive side has accepted the Connection Request. This state is only used in three-way Connection establishments.	<i>it ep accept</i> , <i>it ep free</i> , <i>it ep modify</i> , <i>it ep query</i> , <i>it get consumer context</i> , <i>it get handle type</i> , <i>it post rcv</i> , <i>it reject</i> , <i>it set consumer context</i> , <i>it ep disconnect</i>
IT_EP_STATE_PASSIVE_CONNECTION_PENDING	The Passive side Consumer has called <i>it ep accept</i> in response to the IT_CM_REQ_CONN_REQUEST_EVENT Event.	<i>it ep free</i> , <i>it ep modify</i> , <i>it ep query</i> , <i>it get consumer context</i> , <i>it get handle type</i> , <i>it post rcv</i> , <i>it set consumer context</i> , <i>it ep disconnect</i>
IT_EP_STATE_CONNECTED	The Endpoint is Connected and is ready for all types of Data Transfer and Bind Operations.	<i>it ep disconnect</i> , <i>it ep free</i> , <i>it ep modify</i> , <i>it ep query</i> , <i>it get consumer context</i> , <i>it get handle type</i> , <i>it post rdma read</i> , <i>it post rdma write</i> , <i>it post rcv</i> , <i>it post send</i> , <i>it rmr bind</i> , <i>it rmr unbind</i> , <i>it set consumer context</i>
IT_EP_STATE_NONOPERATIONAL	The Endpoint is in the process of disconnecting. Any pending Data Transfer Operations on the Endpoint will be flushed. Any well-formed operation	<i>it ep disconnect</i> , <i>it ep free</i> , <i>it ep modify</i> , <i>it ep query</i> , <i>it ep reset</i> , <i>it get consumer context</i> ,

	subsequently posted in this state will complete with Flushed or error Status.	<i>it_get_handle_type</i> , <i>it_post_rdma_read</i> , <i>it_post_rdma_write</i> , <i>it_post_rcv</i> , <i>it_post_send</i> , <i>it_rmr_bind</i> , <i>it_rmr_unbind</i> , <i>it_set_consumer_context</i>
--	---	---

6772

6773

6774

6775

The following table identifies the RC Endpoint state transitions.

State	Event	Transition to
IT_EP_STATE_UNCONNECTED	<i>it_ep_connect</i> called	IT_EP_STATE_ACTIVE1_CONNECTION_PENDING
	<i>it_ep_accept</i> called	IT_EP_STATE_PASSIVE_CONNECTION_PENDING
IT_EP_STATE_ACTIVE1_CONNECTION_PENDING	Completion of two-way Connection Establishment	IT_EP_STATE_CONNECTED
	IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT enqueued	IT_EP_STATE_ACTIVE2_CONNECTION_PENDING
	Local or Remote error, or <i>it_reject</i> called on Remote side	IT_EP_STATE_NONOPERATIONAL
	<i>it_ep_disconnect</i> called	IT_EP_STATE_NONOPERATIONAL
IT_EP_STATE_ACTIVE2_CONNECTION_PENDING	Completion of three-way Connection Establishment	IT_EP_STATE_CONNECTED
	Local or Remote error	IT_EP_STATE_NONOPERATIONAL
	<i>it_ep_disconnect</i> called	IT_EP_STATE_NONOPERATIONAL
IT_EP_STATE_PASSIVE_CONNECTION_PENDING	Completion of two-way Connection Establishment	IT_EP_STATE_CONNECTED
	Local or Remote error, or <i>it_reject</i> called on Active side	IT_EP_STATE_NONOPERATIONAL
	<i>it_ep_disconnect</i> called	IT_EP_STATE_NONOPERATIONAL
IT_EP_STATE_CONNECTED	Local Error, or <i>it_ep_disconnect</i> called, or Remote error or Remote disconnect	IT_EP_STATE_NONOPERATIONAL
IT_EP_STATE_NONOPERATIONAL	<i>it_ep_reset</i> called	IT_EP_STATE_UNCONNECTED

6776

6777
6778
6779

The following table identifies and describes the UD Endpoint states. For each state, the table lists the API routines that can be legally applied to a UD Endpoint in that state.

Unreliable Datagram Endpoint states	Description of state	Allowed calls
IT_EP_STATE_UD_NONOPERATIONAL	Any pending Data Transfer Operations on the Endpoint will be flushed. Any well-formed operation subsequently posted in this state will complete with a Flushed status.	<i>it_ep_free</i> , <i>it_ep_modify</i> , <i>it_ep_query</i> , <i>it_get_consumer_context</i> , <i>it_get_handle_type</i> , <i>it_post_rcvfrom</i> , <i>it_post_sendto</i> , <i>it_set_consumer_context</i> , <i>it_ep_reset</i>
IT_EP_STATE_UD_OPERATIONAL	Data Transfer Operations can be posted to the Endpoint. When an Endpoint is first created by calling <i>it_ep_ud_create</i> it is in this state.	<i>it_ep_free</i> , <i>it_ep_modify</i> , <i>it_ep_query</i> , <i>it_get_consumer_context</i> , <i>it_get_handle_type</i> , <i>it_post_rcvfrom</i> , <i>it_post_sendto</i> , <i>it_set_consumer_context</i> , <i>it_ep_reset</i>

6780

6781
6782
6783
6784
6785
6786
6787

An Endpoint in any state can be destroyed by calling [*it_ep_free*](#). However, calling [*it_ep_free*](#) may result in pending Completion Events for the Endpoint being silently discarded by the Implementation. Once a Reliable Connected Endpoint is referenced by either [*it_ep_connect*](#) or [*it_ep_accept*](#) if for any reason the Connection is not established the Endpoint will transition into the IT_EP_STATE_NONOPERATIONAL state from any state. If a Connection is established and then the Connection is broken for any reason, the Endpoint will transition into the IT_EP_STATE_NONOPERATIONAL state.

6788

IT_EP_STATE_UNCONNECTED

6789
6790
6791

When Endpoints are created they are in the IT_EP_STATE_UNCONNECTED state. Only Receive Data Transfer Operations can be posted to an unconnected Endpoint. An Endpoint must be in this state to be used in either an [*it_ep_connect*](#) or an [*it_ep_accept*](#) call.

6792

IT_EP_STATE_ACTIVE1_CONNECTION_PENDING

6793
6794
6795

Once an Active side Endpoint is referenced by a Connection Establishment it transitions into the IT_EP_STATE_ACTIVE1_CONNECTION_PENDING state. Receive Data Transfer Operations may be posted in this state.

6796
6797
6798
6799
6800
6801

In the case of two-way Connection Establishment, the IT_EP_STATE_ACTIVE1_CONNECTION_PENDING state is transient, and the Endpoint will transition to the IT_EP_STATE_CONNECTED state once the Passive side accepts the Connection. If the Passive side rejects the Connection, the Active side will receive an IT_CM_MSG_CONN_PEER_REJECT_EVENT Event and the Endpoint will transition into the IT_EP_STATE_NONOPERATIONAL state.

6802
6803
6804

In the case of three-way Connection Establishment, the Active side Endpoint will transition to IT_EP_STATE_ACTIVE2_CONNECTION_PENDING when an IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT Event is enqueued for the Active side Consumer. If the Active

6805 Consumer calls [*it_reject*](#) after processing the IT_CM_MSG_CONN_ACCEPT_
 6806 ARRIVAL_EVENT Event, the Endpoint will transition into the IT_EP_STATE_
 6807 NONOPERATIONAL state. If the Passive side rejects the Connection, the Active side will
 6808 receive an IT_CM_MSG_CONN_PEER_REJECT_EVENT Event and the Endpoint will
 6809 transition into the IT_EP_STATE_NONOPERATIONAL state.

6810 **IT_EP_STATE_ACTIVE2_CONNECTION_PENDING**

6811 In the case of three-way Connection Establishment, the Endpoint will transition to the
 6812 IT_EP_STATE_CONNECTED state when the Active side Consumer successfully calls
 6813 [*it_ep_accept*](#), or the Endpoint will transition to the IT_EP_STATE_NONOPERATIONAL state
 6814 if the Active side Consumer successfully calls [*it_reject*](#).

6815 In the case of two-way Connection Establishment, this state is transient, and the Endpoint will
 6816 transition to the IT_EP_STATE_CONNECTED state when the Connection is successfully
 6817 established.

6818 **IT_EP_STATE_PASSIVE_CONNECTION_PENDING**

6819 The Passive side Endpoint transitions into this state when the Consumer calls [*it_ep_accept*](#) for
 6820 three-way Connection Establishment. In this state only Receive Data Transfer Operations can be
 6821 posted. From this state the Endpoint will transition into the IT_EP_STATE_CONNECTED state
 6822 when Connection Establishment completes successfully.

6823 **IT_EP_STATE_CONNECTED**

6824 This state is entered when Connection Establishment completes. Upon transition to this state, the
 6825 Implementation delivers an IT_CM_MSG_CONN_ESTABLISHED_EVENT Event. All types
 6826 of Data Transfer and Bind Operations can be posted and will be processed in this state.

6827 If either the Local or Remote Consumer disconnects, the Endpoint will transition into the
 6828 IT_EP_STATE_NONOPERATIONAL state, and the Implementation will deliver an
 6829 IT_CM_MSG_CONN_DISCONNECT_EVENT Event.

6830 Local or Remote errors (e.g. protection violations) also cause the Endpoint to transition to the
 6831 IT_EP_STATE_NONOPERATIONAL state, and the Implementation will deliver an
 6832 IT_CM_MSG_CONN_BROKEN_EVENT Event.

6833 The transition out of the IT_EP_STATE_CONNECTED state is surfaced by either the
 6834 IT_CM_CONN_DISCONNECT_EVENT Event or the IT_CM_MSG_CONN_BROKEN_
 6835 EVENT Event, but never both.

6836 **IT_EP_STATE_NONOPERATIONAL**

6837 In this state no requests will be processed by the Endpoint and any well-formed requests posted
 6838 will generate Completions with a failing Completion Status. The Endpoint will remain in this
 6839 state until [*it_ep_reset*](#) is used to put the Endpoint back into the
 6840 IT_EP_STATE_UNCONNECTED state.

6841 **IT_EP_STATE_UD_OPERATIONAL**

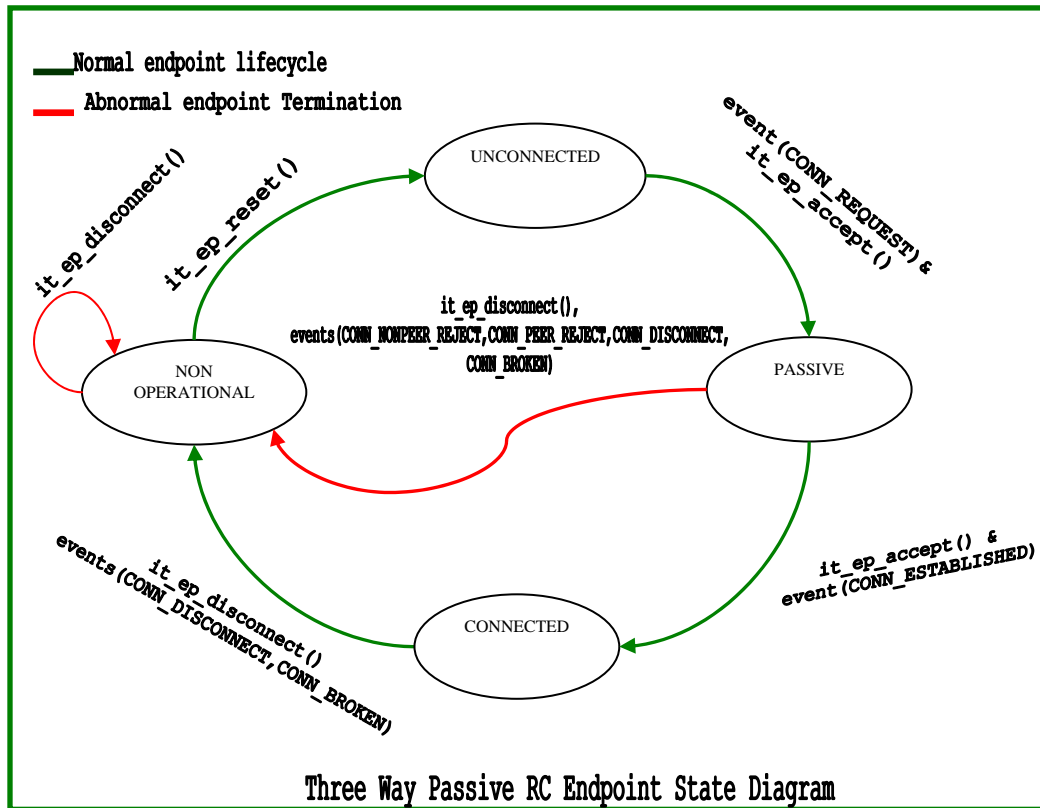
6842 In this state requests will be processed by the Endpoint.

6843 **IT_EP_STATE_UD_NONOPERATIONAL**

6844
6845
6846

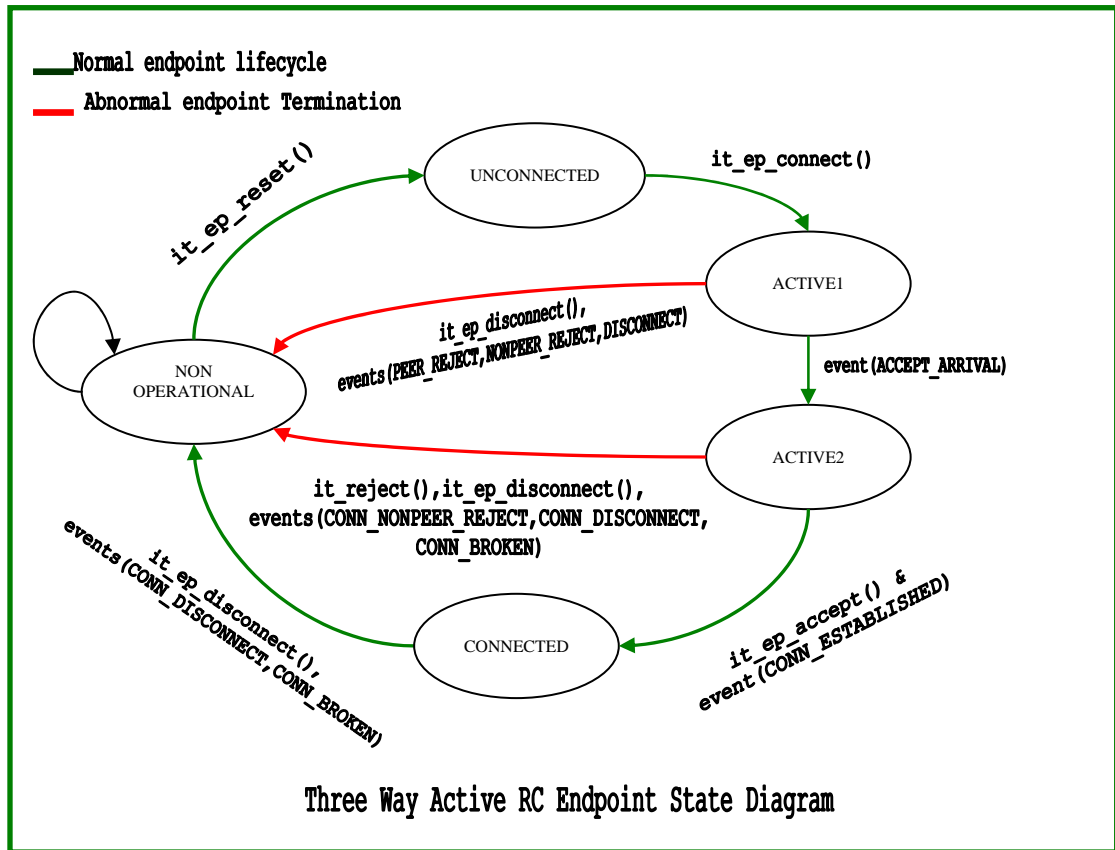
In this state no requests will be processed by the Endpoint and any well-formed requests posted will generate Completions with a failing Completion Status. Once an Unreliable Datagram Endpoint enters this state it can only be destroyed with *it_ep_free*.

6847 **EXTENDED DESCRIPTION**
6848



6849
6850

Figure 1 : Three Way Passive RC Endpoint State Diagram



6851
6852

Figure 2 : Three Way Active RC Endpoint State Diagram

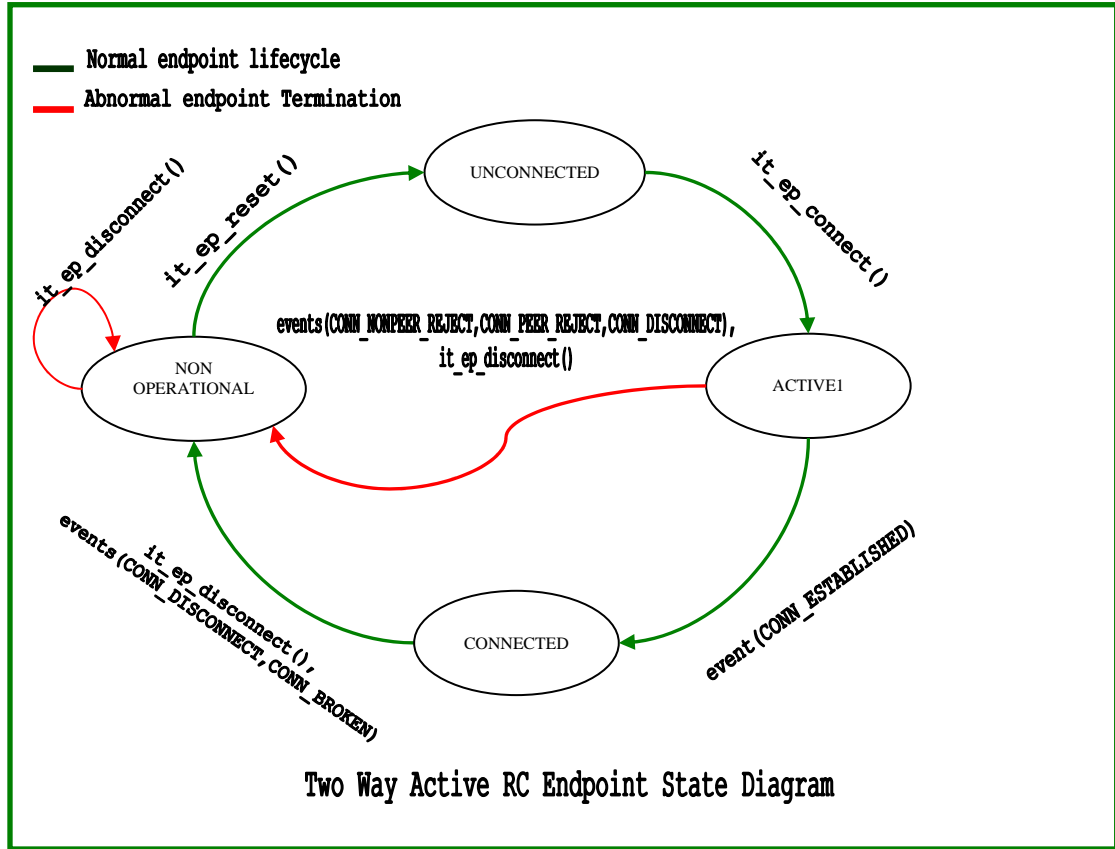
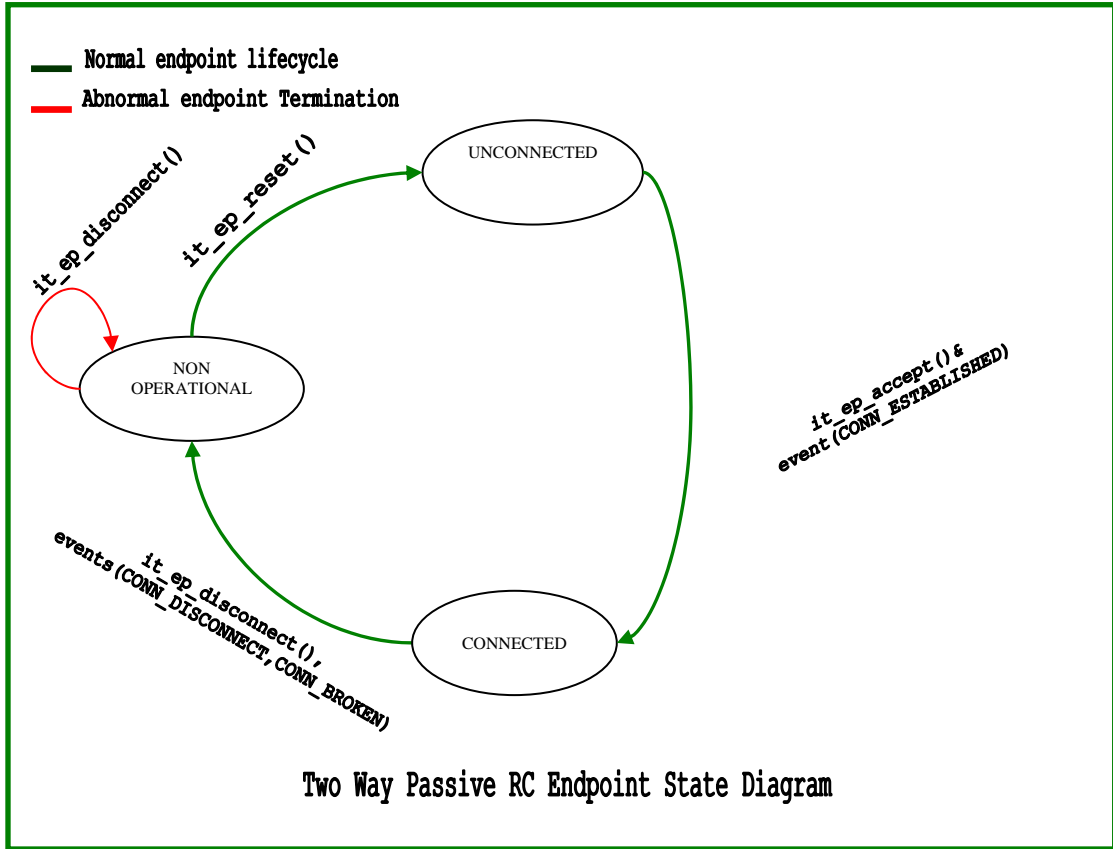


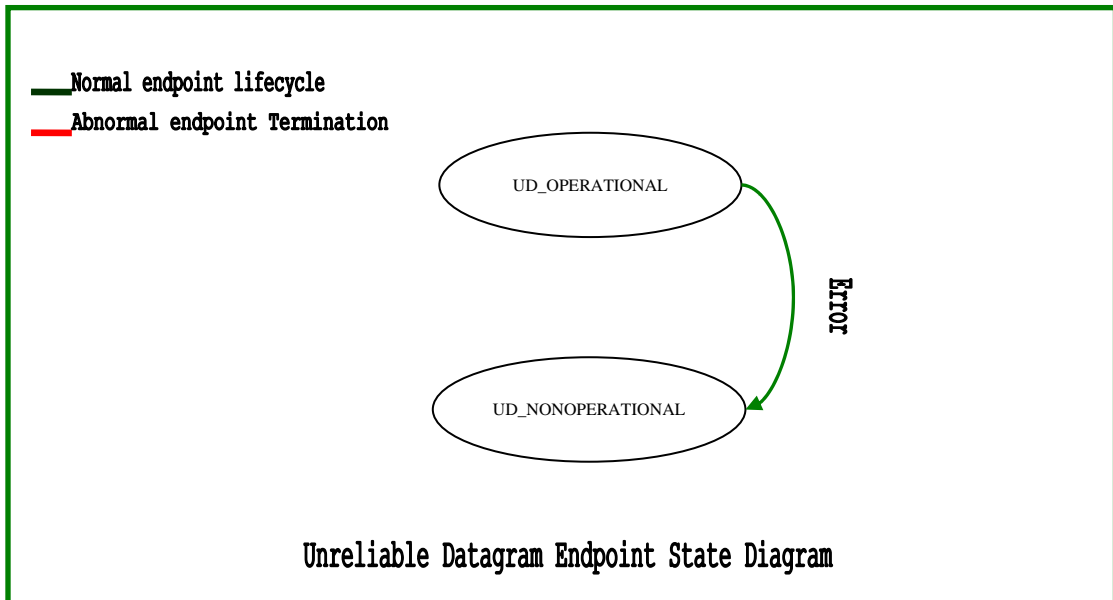
Figure 3 : Two Way Active RC Endpoint State Diagram

6853
6854



6855
6856

Figure 4 : Two Way Passive RC Endpoint State Diagram



6857
6858

Figure 5 : Unreliable Datagram Endpoint State Diagram

6859 **RETURN VALUE**

6860 None.

6861 **ERRORS**

6862 None.

6863 **APPLICATION USAGE**

- 6864 1. If the Consumer cares about the Completion Status of posted Data Transfer or Bind
 6865 Operations after an Endpoint transitions into the IT_EP_STATE_NONOPERATIONAL
 6866 state then the Consumer can Post Send and Receive DTOs to the Endpoint to serve as
 6867 markers in the Endpoint associated EVD. The API guarantees that any posting in
 6868 IT_EP_STATE_NONOPERATIONAL state will be immediately flushed to the EVDs. The
 6869 Consumer can reap Completions from the associated EVDs until Completions for the marker
 6870 DTOs are returned. This way the Consumer can be guaranteed that all Completions
 6871 associated with the Endpoint have been reaped.
- 6872 2. The Consumer is responsible for coordinating the use of functions that free a Connection
 6873 Establishment Identifier (cn_est_id) such as [*it_ep_accept*](#), [*it_reject*](#), [*it_ep_disconnect*](#) and
 6874 [*it_handoff*](#). The behavior of functions that are passed as invalid Connection Establishment
 6875 Identifier is indeterminate.
- 6876 3. The Consumer should be aware that the delivery of Private Data to the Remote Endpoint
 6877 specified in calls to [*it_ep_accept*](#), [*it_reject*](#) and [*it_ep_disconnect*](#) is unreliable and should be
 6878 used accordingly.

6879 **SEE ALSO**6880 [*it_ep_accept\(\)*](#), [*it_reject\(\)*](#), [*it_ep_disconnect\(\)*](#), [*it_handoff\(\)*](#), [*it_ep_reset\(\)*](#)

it_event_t

6881

6882 **NAME**

6883

it_event – definition of Event data structures

6884 **SYNOPSIS**

6885

```
#include <it_api.h>
```

6886

6887

```
#define IT_EVENT_STREAM_MASK      0xff000
```

6888

```
#define IT_TIMEOUT_INFINITE      ((uint64_t)(-1))
```

6889

```
typedef enum
```

6890

```
{
```

6891

```
    /* DTO Completion Event Stream */
```

6892

```
    IT_DTO_EVENT_STREAM                = 0x00000,
```

6893

```
    IT_DTO_SEND_CMPL_EVENT            = 0x00001,
```

6894

```
    IT_DTO_RC_RECV_CMPL_EVENT        = 0x00002,
```

6895

```
    IT_DTO_UD_RECV_CMPL_EVENT        = 0x00003,
```

6896

```
    IT_DTO_RDMA_WRITE_CMPL_EVENT     = 0x00004,
```

6897

```
    IT_DTO_RDMA_READ_CMPL_EVENT     = 0x00005,
```

6898

```
    IT_RMR_BIND_CMPL_EVENT           = 0x00006,
```

6899

6900

```
    /*
```

6901

```
     * Communication Management Request Event Stream
```

6902

```
     */
```

6903

```
    IT_CM_REQ_EVENT_STREAM            = 0x01000,
```

6904

```
    IT_CM_REQ_CONN_REQUEST_EVENT     = 0x01001,
```

6905

```
    IT_CM_REQ_UD_SERVICE_REQUEST_EVENT = 0x01002,
```

6906

6907

```
    /*
```

6908

```
     * Communication Management Message Event Stream
```

6909

```
     */
```

6910

```
    IT_CM_MSG_EVENT_STREAM            = 0x02000,
```

6911

```
    IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT = 0x02001,
```

6912

```
    IT_CM_MSG_CONN_ESTABLISHED_EVENT  = 0x02002,
```

6913

```
    IT_CM_MSG_CONN_DISCONNECT_EVENT   = 0x02003,
```

6914

```
    IT_CM_MSG_CONN_PEER_REJECT_EVENT  = 0x02004,
```

6915

```
    IT_CM_MSG_CONN_NONPEER_REJECT_EVENT = 0x02005,
```

6916

```
    IT_CM_MSG_CONN_BROKEN_EVENT       = 0x02006,
```

6917

```
    IT_CM_MSG_UD_SERVICE_REPLY_EVENT   = 0x02007,
```

6918

6919

```
    /* Asynchronous Affiliated Event Stream */
```

6920

```
    IT_ASYNC_AFF_EVENT_STREAM         = 0x04000,
```

6921

```
    IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE = 0x04001,
```

6922

```
    IT_ASYNC_AFF_EP_FAILURE           = 0x04002,
```

6923

```
    IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE = 0x04003,
```

6924

```
    IT_ASYNC_AFF_EP_LOCAL_ACCESS_VIOLATION = 0x04004,
```

6925

```
    IT_ASYNC_AFF_EP_REQ_DROPPED      = 0x04005,
```

6926

```
    IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION = 0x04006,
```

6927

```
    IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA = 0x04007,
```

6928

```
    IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION = 0x04008,
```

6929

6930

```
    /* Asynchronous Non-Affiliated Event Stream */
```

6931

```

6932         IT_ASYNC_UNAFF_EVENT_STREAM           = 0x08000,
6933         IT_ASYNC_UNAFF_IA_CATASTROPHIC_ERROR  = 0x08001,
6934         IT_ASYNC_UNAFF_SPIGOT_ONLINE          = 0x08002,
6935         IT_ASYNC_UNAFF_SPIGOT_OFFLINE         = 0x08003,
6936         IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE   = 0x08004,
6937
6938         /* Software Event Stream */
6939         IT_SOFTWARE_EVENT_STREAM              = 0x10000,
6940         IT_SOFTWARE_EVENT                    = 0x10001,
6941
6942         /* AEVD Notification Event Stream */
6943         IT_AEVD_NOTIFICATION_EVENT_STREAM     = 0x20000,
6944         IT_AEVD_NOTIFICATION_EVENT           = 0x20001
6945     } it_event_type_t;
6946
6947     typedef struct {
6948         it_event_type_t event_number;
6949         it_evd_handle_t evd;
6950     } it_any_event_t;
6951
6952     typedef union
6953     {
6954         /*
6955          * The following two union elements are
6956          * available for programming convenience.
6957          *
6958          * The event_number may be used to determine the
6959          * it_event_type_t of any Event. it_any_event_t
6960          * allows the EVD to be determined as well.
6961          */
6962         it_event_type_t      event_number;
6963         it_any_event_t       any;
6964
6965         /*
6966          * The remaining union elements correspond to
6967          * the various it_event_type_t types.
6968          */
6969
6970         /*
6971          * The following two Event structures
6972          * support the IT_DTO_EVENT_STREAM Event Stream.
6973          *
6974          * it_dto_cmpl_event_t supports
6975          * only the following events:
6976          *     IT_DTO_SEND_CMPL_EVENT
6977          *     IT_DTO_RC_RECV_CMPL_EVENT
6978          *     IT_DTO_RDMA_WRITE_CMPL_EVENT
6979          *     IT_DTO_RDMA_READ_CMPL_EVENT
6980          *     IT_RMR_BIND_CMPL_EVENT
6981          *
6982          * it_all_dto_cmpl_event_t supports all
6983          * possible DTO and RMR events:
6984          *     IT_DTO_SEND_CMPL_EVENT
6985          *     IT_DTO_RC_RECV_CMPL_EVENT

```

```

6986      *      IT_DTO_UD_RECV_CMPL_EVENT
6987      *      IT_DTO_RDMA_WRITE_CMPL_EVENT
6988      *      IT_DTO_RDMA_READ_CMPL_EVENT
6989      *      IT_RMR_BIND_CMPL_EVENT
6990      */
6991      it_dto_cmpl_event_t      dto_cmpl;
6992      it_all_dto_cmpl_event_t  all_dto_cmpl;
6993
6994  /*
6995  * The following two Event structures
6996  * support the IT_CM_REQ_EVENT_STREAM Event
6997  * stream:
6998  *
6999  * it_conn_request_event_t supports:
7000  *      IT_CM_REQ_CONN_REQUEST_EVENT
7001  *
7002  * it_ud_svc_request_event_t supports:
7003  *      IT_CM_REQ_UD_SERVICE_REQUEST_EVENT
7004  */
7005      it_conn_request_event_t      conn_req;
7006      it_ud_svc_request_event_t    ud_svc_request;
7007
7008  /*
7009  * The following two Event structures
7010  * support the IT_CM_MSG_EVENT_STREAM Event
7011  * stream:
7012  *
7013  * it_connection_event_t supports:
7014  *      IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT
7015  *      IT_CM_MSG_CONN_ESTABLISHED_EVENT
7016  *      IT_CM_MSG_CONN_PEER_REJECT_EVENT
7017  *      IT_CM_MSG_CONN_NONPEER_REJECT_EVENT
7018  *      IT_CM_MSG_CONN_DISCONNECT_EVENT
7019  *      IT_CM_MSG_CONN_BROKEN_EVENT
7020  *
7021  * it_ud_svc_reply_event_t supports:
7022  *      IT_CM_MSG_UD_SERVICE_REPLY_EVENT
7023  */
7024      it_connection_event_t      conn;
7025      it_ud_svc_reply_event_t    ud_svc_reply;
7026
7027  /*
7028  * it_affiliated_event_t supports
7029  * the following Event Stream:
7030  *      IT_ASYNC_AFF_EVENT_STREAM
7031  */
7032      it_affiliated_event_t      aff_async;
7033
7034  /*
7035  * it_unaffiliated_event_t supports
7036  * the following Event Stream:
7037  *      IT_ASYNC_UNAFF_EVENT_STREAM
7038  */
7039      it_unaffiliated_event_t    unaff_async;

```

```

7040
7041      /*
7042      * it_software_event_t supports
7043      * the following Event Stream:
7044      *     IT_SOFTWARE_EVENT_STREAM
7045      */
7046      it_software_event_t      sw;
7047
7048      /*
7049      * it_aevd_notification_event_t supports
7050      * the following Event Stream:
7051      *     IT_AEVD_NOTIFICATION_EVENT_STREAM
7052      */
7053      it_aevd_notification_event_t aevd_notify;
7054  } it_event_t;

```

7055 DESCRIPTION

7056 The *it_event_t* defines the format for Events for IT-APIs. Each Event consists of a Handle to the
7057 EVD where the Event has been queued, and Event type identifier with the Event type specific
7058 Event data.

7059 Events for a Simple EVD can be fed from only a single Event Stream type.

7060 Multiple Event numbers that can be on the same Event Stream type form an Event group. The
7061 Event data formats for all Event types of the same Event Stream type are defined in one or more
7062 separate man page(s) specific to each Event group.

7063 RETURN VALUE

7064 None.

7065 ERRORS

7066 None.

7067 APPLICATION USAGE

7068 The Consumer allocates an *it_event_t* object and passes it into the [it_evd_wait](#) or [it_evd_dequeue](#)
7069 calls in order to retrieve Events. The *it_event_t* object is a union of all possible Event Stream
7070 data types, thus it is the size of the largest possible Event type.

7071 If the Consumer wishes to conserve memory and use only the minimally-sized Event data
7072 structures found in the *it_event_t* union, they are free to. Use of *it_event_t* structures that are too
7073 small for an Event Stream may cause program termination.

7074 The Consumer may use the IT_EVENT_STREAM_MASK to convert from an *it_event_type_t*
7075 *event_number* to Event Stream by masking off the lower bits of the Event number.

7076 SEE ALSO

7077 [it_aevd_notification_event_t](#), [it_affiliated_event_t](#), [it_cm_msg_events](#), [it_cm_req_events](#),
7078 [it_dto_events](#), [it_software_event_t](#), [it_unaffiliated_event_t](#), [it_evd_wait\(\)](#), [it_evd_dequeue\(\)](#),
7079 [it_evd_create\(\)](#)

it_handle_t

7080

7081 **NAME**

7082

it_handle_t – enumeration and type definitions for IT Handles

7083 **SYNOPSIS**

7084

```
#include <it_api.h>
```

7085

7086

```
typedef enum {
    IT_HANDLE_TYPE_ADDR,
    IT_HANDLE_TYPE_EP,
    IT_HANDLE_TYPE_EVD,
    IT_HANDLE_TYPE_IA,
    IT_HANDLE_TYPE_LISTEN,
    IT_HANDLE_TYPE_LMR,
    IT_HANDLE_TYPE_PZ,
    IT_HANDLE_TYPE_RMR,
    IT_HANDLE_TYPE_UD_SVC_REQ
} it_handle_type_enum_t;
```

7097

7098

```
typedef void (*it_handle_t);
```

7099

```
#define IT_NULL_HANDLE ((it_handle_t) NULL)
```

7100

7101

```
typedef struct it_addr_handle_s (*it_addr_handle_t);
```

7102

```
typedef struct it_ep_handle_s (*it_ep_handle_t);
```

7103

```
typedef struct it_evd_handle_s (*it_evd_handle_t);
```

7104

```
typedef struct it_ia_handle_s (*it_ia_handle_t);
```

7105

```
typedef struct it_listen_handle_s (*it_listen_handle_t);
```

7106

```
typedef struct it_lmr_handle_s (*it_lmr_handle_t);
```

7107

```
typedef struct it_pz_handle_s (*it_pz_handle_t);
```

7108

```
typedef struct it_rmr_handle_s (*it_rmr_handle_t);
```

7109

```
typedef struct it_ud_svc_req_handle_s (*it_ud_svc_req_handle_t);
```

7110 **DESCRIPTION**

7111

The *it_handle_type_enum_t* associates an enumerated value with each type of Handle used in the API Implementation. The enumeration is used to describe the type of a Handle returned by [it_get_handle_type](#).

7112

7113

7114

The table below defines the relationship of IT-API Handle types and the associated *it_handle_type_enum_t* value.

7115

it_handle type	Returned it_handle_type_enum value
it_addr_handle_t	IT_HANDLE_TYPE_ADDR
it_ep_handle_t	IT_HANDLE_TYPE_EP
it_evd_handle_t	IT_HANDLE_TYPE_EVD
it_ia_handle_t	IT_HANDLE_TYPE_IA
it_listen_handle_t	IT_HANDLE_TYPE_LISTEN
it_lmr_handle_t	IT_HANDLE_TYPE_LMR

it_pz_handle_t	IT_HANDLE_TYPE_PZ
it_rmr_handle_t	IT_HANDLE_TYPE_RMR
it_ud_svc_req_handle_t	IT_HANDLE_TYPE_UD_SVC_REQ

7116 **RETURN VALUE**

7117 None.

7118 **ERRORS**

7119 None.

7120 **SEE ALSO**7121 [it_get_handle_type\(\)](#)

7122

it_ia_info_t

7123

7124 **NAME**

7125 `it_ia_info_t` – encapsulates all Interface Adapter attributes and Spigot information

7126 **SYNOPSIS**

```

7127 #include <it_api.h>
7128
7129 /* Enumerates all the transport types supported by the API. */
7130 typedef enum {
7131
7132     /* InfiniBand Native Transport */
7133     IT_IB_TRANSPORT          = 1,
7134
7135     /* VIA host Interface using IP transport, supporting
7136        only the Reliable Delivery reliability level */
7137     IT_VIA_IP_TRANSPORT     = 2,
7138
7139     /* VIA host Interface, using Fibre Channel transport, supporting
7140        only the Reliable Delivery reliability level*/
7141     IT_VIA_FC_TRANSPORT     = 3,
7142
7143     /* Vendor-proprietary Transport */
7144     IT_VENDOR_TRANSPORT     = 1000
7145
7146 } it_transport_type_t;
7147
7148
7149 /* Transport Service Type definitions. */
7150 typedef enum {
7151
7152     /* Reliable Connected Transport Service Type */
7153     IT_RC_SERVICE           = 0x1,
7154
7155     /* Unreliable Datagram Transport Service Type */
7156     IT_UD_SERVICE          = 0x2,
7157
7158 } it_transport_service_type_t;
7159
7160
7161 /* The following structure describes an Interface Adapter Spigot */
7162 typedef struct {
7163
7164     /* Spigot identifier */
7165     size_t                spigot_id;
7166
7167     /* Maximum sized Send operation for the RC service
7168        on this Spigot. */
7169     size_t                max_rc_send_len;
7170
7171     /* Maximum sized RDMA Read/Write operation for the RC service on
7172        this Spigot. */
7173     size_t                max_rc_rdma_len;

```

```

7174
7175     /* Maximum sized Send operation for the UD service
7176        on this Spigot. */
7177     size_t                                max_ud_send_len;
7178
7179     /* Indicates whether the Spigot is online or offline.
7180        An IT_TRUE value means online. */
7181     it_boolean_t                          spigot_online;
7182
7183     /* A mask indicating which Connection Qualifier types this
7184        IA supports for input to it_ep_connect and
7185        it_ud_service_request_handle_create. The bits in the mask are
7186        an inclusive OR of the values for Connection Qualifier types
7187        that this IA supports. */
7188     it_conn_qual_type_t                   active_side_conn_qual;
7189
7190     /* A mask indicating which Connection Qualifier types this to
7191        it_listen_create. The bits in the mask are an inclusive OR of
7192        the values for Connection Qualifier types that this IA
7193        supports. */
7194     it_conn_qual_type_t                   passive_side_conn_qual;
7195
7196
7197     /* The number of Network Addresses associated with Spigot */
7198     size_t                                num_net_addr;
7199
7200     /* Pointer to array of Network Address addresses. */
7201     it_net_addr_t*                       net_addr;
7202
7203 } it_spigot_info_t;
7204
7205 /* The following structure is used to identify the vendor associated
7206    with an IA that uses the IB transport*/
7207 typedef struct {
7208
7209     /* The NodeInfo:VendorID as described in chapter 14 of the
7210        IB spec. */
7211     uint32_t                              vendor : 24;
7212
7213     /* The NodeInfo:DeviceID as described in chapter 14 of the
7214        IB spec. */
7215     uint16_t                              device;
7216
7217     /* The NodeInfo:Revision as described in chapter 14 of the
7218        IB spec. */
7219     uint32_t                              revision;
7220 } it_vendor_ib_t;
7221
7222 /* The following structure is used to identify the vendor associated
7223    with an IA that uses a VIA transport*/
7224 typedef struct {
7225     /* The "Name" member of the VIP_NIC_ATTRIBUTES structure, as
7226        described in the VIA spec. */
7227     char                                  name[64];

```

```

7228
7229     /* The "HardwareVersion" member of the VIP_NIC_ATTRIBUTES structure,
7230     as described in the VIA spec. */
7231     unsigned long             hardware;
7232
7233     /* The "ProviderVersion" member of the VIP_NIC_ATTRIBUTES structure,
7234     as described in the VIA spec. */
7235     unsigned long             provider;
7236 } it_vendor_via_t;
7237
7238 /* The following structure is returned by the it_ia_query function */
7239 typedef struct {
7240
7241     /* Interface Adapter name, as specified in it_ia_create */
7242     char*                       ia_name;
7243
7244     /* The major version number of the latest version of the IT-API that
7245     this IA supports. */
7246     uint32_t                     api_major_version;
7247
7248     /* The minor version number of the latest version of the IT-API that
7249     this IA supports. */
7250     uint32_t                     api_minor_version;
7251
7252     /* The major version number for the software being used to control
7253     this IA. The IT-API imposes no structure whatsoever on this
7254     number; its meaning is completely IA-dependent. */
7255     uint32_t                     sw_major_version;
7256
7257     /* The minor version number for the software being used to control
7258     this IA. The IT-API imposes no structure whatsoever on this
7259     number; its meaning is completely IA-dependent. */
7260     uint32_t                     sw_minor_version;
7261
7262     /* The vendor associated with the IA. This information is useful
7263     if the Consumer wishes to do device-specific programming. This
7264     union is discriminated by transport_type. No vendor
7265     identification is provided for transports not listed below. */
7266     union {
7267
7268         /* Used if transport_type is IT_IB_TRANSPORT */
7269         it_vendor_ib_t             ib;
7270
7271         /* Used if transport_type is IT_VIA_IP_TRANSPORT or
7272         IT_VIA_FC_TRANSPORT */
7273         it_vendor_via_t           via;
7274
7275     } vendor;
7276
7277     /* The Interface Adapter and platform provide a data alignment hint
7278     to the Consumer to help the Consumer align their data transfer
7279     buffers in a way that is optimal for the performance of the IA.
7280     For example, if the best throughput is obtained by aligning
7281     buffers to 128-byte boundaries, dto_alignment_hint will have the

```

```

7282         value 128. The Consumer may choose to ignore the alignment hint
7283         without any adverse functional impact. (There may be an adverse
7284         performance impact.) */
7285     uint32_t                dto_alignment_hint;
7286
7287     /* The transport type (e.g. InfiniBand) supported by Interface
7288        Adapter. An Interface Adapter supports precisely one transport
7289        type. */
7290     it_transport_type_t     transport_type;
7291
7292     /* The Transport Service Types supported by this IA. This is
7293        constructed by doing an inclusive OR of the Transport Service
7294        Type values.*/
7295     it_transport_service_type_t supported_service_types;
7296
7297     /* Indicates whether work queues are resizable */
7298     it_boolean_t           resizable_work_queue;
7299
7300     /* Indicates whether the underlying transport used by this IA uses a
7301        three-way handshake for doing Connection establishment. Note
7302        that if the underlying transport supports a three-way handshake
7303        the Consumer can choose whether to use two handshakes or three
7304        when establishing the Connection. If the underlying transport
7305        supports a two-way handshake for establishing a Connection, the
7306        Consumer can only use two handshakes when establishing the
7307        Connection. */
7308     it_boolean_t           three_way_handshake_support;
7309
7310     /* Indicates whether Private Data is supported on Connection
7311        establishment or UD service resolution operations. */
7312     it_boolean_t           private_data_support;
7313
7314     /* Indicates whether the max_message_size field in the
7315        IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
7316     it_boolean_t           max_message_size_support;
7317
7318     /* Indicates whether the rdma_read_inflight_incoming field in the
7319        IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
7320     it_boolean_t           ird_support;
7321
7322     /* Indicates whether the rdma_read_inflight_outgoing field in the
7323        IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
7324     it_boolean_t           ord_support;
7325
7326     /* Indicates whether the IA generates IT_ASYNC_UNAFF_SPIGOT_ONLINE
7327        Events. See it_unaffiliated_event_t for details. */
7328     it_boolean_t           spigot_online_support;
7329
7330     /* Indicates whether the IA generates IT_ASYNC_UNAFF_SPIGOT_OFFLINE
7331        Events. See it_unaffiliated_event_t for details. */
7332     it_boolean_t           spigot_offline_support;
7333
7334     /* The maximum number of bytes of Private Data supported for the
7335        it_ep_connect routine. This will be less than or equal to

```

```

7336         IT_MAX_PRIV_DATA. */
7337     size_t                connect_private_data_len;
7338
7339     /* The maximum number of bytes of Private Data supported for the
7340        it_ep_accept routine. This will be less than or equal to
7341        IT_MAX_PRIV_DATA. */
7342     size_t                accept_private_data_len;
7343
7344     /* The maximum number of bytes of Private Data supported for the
7345        it_reject routine. This will be less than or equal to
7346        IT_MAX_PRIV_DATA. */
7347     size_t                reject_private_data_len;
7348
7349     /* The maximum number of bytes of Private Data supported for the
7350        it_ep_disconnect routine. This will be less than or equal to
7351        IT_MAX_PRIV_DATA. */
7352     size_t                disconnect_private_data_len;
7353
7354     /* The maximum number of bytes of Private Data supported for the
7355        it_ud_service_request_handle_create routine. This will be
7356     less than or
7357     equal to IT_MAX_PRIV_DATA. */
7358     size_t                ud_req_private_data_len;
7359
7360     /* The maximum number of bytes of Private Data supported for the
7361        it_ud_service_reply routine. This will be less than or equal to
7362        IT_MAX_PRIV_DATA. */
7363     size_t                ud_rep_private_data_len;
7364
7365     /* Specifies the number of Spigots associated with this Interface
7366        Adapter */
7367     size_t                num_spigots;
7368
7369     /* An array of Spigot information data structures. The array
7370        contains num_spigots elements. */
7371     it_spigot_info_t*     spigot_info;
7372
7373     /* The Handle for the EVD that contains the affiliated async Event
7374        Stream. If no EVD contains the Affiliated Async Event Stream,
7375        this member will have the distinguished value IT_NULL_HANDLE */
7376     it_evd_handle_t      affiliated_err_evd;
7377
7378     /* The Handle for the EVD that contains the Unaffiliated Async Event
7379        Stream. If no EVD contains the Unaffiliated Async Event Stream,
7380        this member will have the distinguished value IT_NULL_HANDLE */
7381     it_evd_handle_t      unaffiliated_err_evd;
7382
7383 } it_ia_info_t;

```

7384 DESCRIPTION

7385 The *it_ia_info_t* structure is returned by the [it_ia_query](#) routine.

7386 The *it_ia_info_t* structure specifies the capabilities and attributes of an Interface Adapter. It also
7387 identifies the Interface Adapter's Spigots and their Network Addresses.

7388 Spigot identifiers are required inputs to the [it_get_pathinfo](#) and [it_listen_create](#) routines. Spigot
 7389 Network Addresses may be used in the advertisement of local-Consumer-provided services to
 7390 remote Consumers.

7391 The IA can be in one of two states: enabled or disabled. An IA will be in the enabled state
 7392 when it is created (via [it_ia_create](#)). Normally an IA will remain in the enabled state, but if it
 7393 encounters a catastrophic error it will move into the disabled state. The Implementation
 7394 guarantees that no unreported data corruption has occurred as a result of the IA entering the
 7395 disabled state. If the Consumer calls any API routine other than [it_ia_free](#) while the IA is in the
 7396 disabled state, neither the IA nor any of the Implementation data structures associated with it
 7397 will be modified in any way. Instead, the routine will return the error code
 7398 IT_ERR_IA_CATASTROPHE, and none of the output parameters from the routine will be
 7399 valid. Once an IA has entered the disabled state the only recovery action that the Consumer can
 7400 perform is to free the IA.

7401 **RETURN VALUE**

7402 None.

7403 **ERRORS**

7404 None.

7405 **FUTURE DIRECTIONS**

7406 Quality of Service control for VIA and other transports may be added in the future.

7407 **SEE ALSO**

7408 [it_ia_query\(\)](#), [it_get_pathinfo\(\)](#), [it_listen_create\(\)](#)

it_lmr_triplet_t

7409
 7410 **NAME**
 7411 `it_lmr_triplet_t` – structure describing a DTO buffer in a Local Memory Region

7412 **SYNOPSIS**
 7413 `#include <it_api.h>`
 7414
 7415 `typedef struct {`
 7416 `it_lmr_handle_t lmr;`
 7417 `void *addr;`
 7418 `it_length_t length;`
 7419 `} it_lmr_triplet_t;`

7420 **DESCRIPTION**
 7421 The `it_lmr_triplet_t` structure describes a local Source or Destination buffer segment for Data
 7422 Transfer Operations. Its members are defined as follows:

7423 `lmr` Handle of LMR in which the local buffer resides.
 7424 `addr` Starting address of the local buffer segment.
 7425 `length` Length in bytes of the local buffer segment.

7426 **RETURN VALUE**
 7427 None.

7428 **ERRORS**
 7429 None.

7430 **APPLICATION USAGE**
 7431 The LMR Triplet is an input parameter to all of the DTO operations, such as [*it_post_send*](#).

7432 **SEE ALSO**
 7433 [*it_post_send\(\)*](#), [*it_post_sendto\(\)*](#), [*it_post_rcv\(\)*](#), [*it_post_rcvfrom\(\)*](#), [*it_post_rdma_write\(\)*](#),
 7434 [*it_post_rdma_read\(\)*](#).

it_net_addr_t

7435

7436 **NAME**

7437

it_net_addr_t – encapsulates all supported Network Address types

7438 **SYNOPSIS**

7439

```
#include <it_api.h>
```

7440

7441

```
/* Enumerates all the possible Network Address types supported
   by the API. */
```

7442

7443

```
typedef enum {
```

7444

7445

```
    /* IPv4 address */
```

7446

```
    IT_IPV4 = 0x1,
```

7447

7448

```
    /* IPv6 address */
```

7449

```
    IT_IPV6 = 0x2,
```

7450

7451

```
    /* InfiniBand GID */
```

7452

```
    IT_IB_GID = 0x3,
```

7453

7454

```
    /* VIA Network Address */
```

7455

```
    IT_VIA_HOSTADDR = 0x4
```

7456

```
} it_net_addr_type_t;
```

7457

7458

```
/* Defines the Network Address format for a VIA "host address".
   The API has a fixed upper bound on the maximum sized VIA
   address it will support*/
```

7459

7460

7461

7462

```
#define IT_MAX_VIA_ADDR_LEN    64
```

7463

7464

```
typedef struct {
```

7465

7466

```
    /* The number of bytes in the array below that are
       significant */
```

7467

7468

```
    uint16_t                len;
```

7469

7470

```
    /* VIA host address, which is an array of bytes */
```

7471

```
    unsigned char           hostaddr[IT_MAX_VIA_ADDR_LEN];
```

7472

7473

```
} it_via_net_addr_t;
```

7474

7475

```
/* This defines the Network Address format for the InfiniBand
   GID, which is just an IPv6 address. */
```

7476

7477

```
typedef struct in6_addr      it_ib_gid_t;
```

7478

7479

```
/* This describes a Network Address suitable for input to several
   routines in the API. */
```

7480

7481

```
typedef struct {
```

7482

7483

```
    /* The discriminator for the union below. */
```

7484

```
    it_net_addr_type_t      addr_type;
```

7485


```

7486
7487         union {
7488
7489             /* IPv4 address, in network byte order */
7490             struct in_addr          ipv4;
7491
7492             /* IPv6 address, in network byte order */
7493             struct in6_addr         ipv6;
7494
7495             /* InfiniBand GID, in network byte order */
7496             it_ib_gid_t            gid;
7497
7498             /* VIA Network Address. */
7499             it_via_net_addr_t      via;
7500
7501         } addr;
7502
7503     } it_net_addr_t;

```

7504 **DESCRIPTION**

7505 The *it_net_addr_t* type is used by several routines in the API to encapsulate a Network Address
7506 associated with a Spigot on an IA. The *it_net_addr_t* is the name for a Spigot when it is being
7507 accessed remotely. (When it is being accessed locally, it is named by a Spigot identifier, not by
7508 an *it_net_addr_t*.) Each Spigot on an IA has at least one *it_net_addr_t* associated with it. A
7509 Spigot can have more than one Network Address associated with it, and these Network
7510 Addresses can be of different types. (For example, an InfiniBand HCA might have both an IPv4
7511 address and an InfiniBand GID associated with one of its Spigots.) The set of Network
7512 Addresses that can be used to refer to a Spigot on an IA can be determined using the [it_ia_query](#)
7513 routine.

7514 In order to aid Consumers in writing portable applications that span platforms with different
7515 native byte orders, all Network Addresses that are supported by the API with the exception of
7516 the VIA “host address” are required to be input to the API in network byte order, and will be
7517 output from the API in network byte order. (The VIA “host address” is defined to be an array of
7518 bytes, and hence is not affected by which native byte order a platform uses.)

7519 **RETURN VALUE**

7520 None.

7521 **ERRORS**

7522 None.

7523 **SEE ALSO**

7524 [it_get_pathinfo\(\)](#), [it_ia_query\(\)](#)

it_path_t

7525

7526 **NAME**

7527

it_path_t – describes the Path between a pair of Spigots

7528 **SYNOPSIS**

7529

```
#include <it_api.h>
```

7530

7531

```
/* This is the remote component of the Path information for the
   InfiniBand transport */
```

7532

```
typedef struct {
```

7533

7534

```
    /* Partition Key, as defined in the REQ message for the IB
       CM protocol */
```

7535

```
    uint16_t                partition_key;
```

7536

7537

```
    /* Path Packet Payload MTU, as defined in the REQ message
       for the IB CM protocol */
```

7538

```
    uint8_t                 path_mtu : 4;
```

7539

7540

7541

```
    /* PacketLifeTime, as defined in the PathRecord in IB
       specification. This field is useful for Consumers that
       wish to use timeout values other than the default ones
       for doing Connection establishment. */
```

7542

```
    uint8_t                 packet_lifetime : 6;
```

7543

7544

7545

```
    /* Local Port LID, as defined in the REQ message for the IB
       CM protocol. The low order bits of this value also
       constitute the "Source Path Bits" that are used to
       create an Address Handle. */
```

7546

```
    uint16_t                local_port_lid;
```

7547

7548

```
    /* Remote Port LID, as defined in the REQ message for the
       IB CM protocol. This is also the "Destination LID" used
       to create an Address Handle. */
```

7549

```
    uint16_t                remote_port_lid;
```

7550

7551

7552

```
    /* Local Port GID in network byte order, as defined in the
       REQ message for the IB CM protocol. This is also used to
       determine the appropriate "Source GID Index" to be used
       when creating an Address Handle. */
```

7553

```
    it_ib_gid_t            local_port_gid;
```

7554

7555

```
    /* Remote Port GID in network byte order, as defined in the
       REQ message for the IB CM protocol. This is also the
       "Destination GID or MGID" used to create an Address
```

7556

```
    Handle. */
```

7557

```
    it_ib_gid_t            remote_port_gid;
```

7558

7559

```
    /* Packet Rate, as defined in the REQ message for the IB CM
       protocol. This is also the "Maximum Static Rate" to be
       used when creating an Address Handle. */
```

7560

```
    uint8_t                 packet_rate : 6;
```

7561

7562

7563

7564

7565

7566

7567

7568

7569

7570

7571

7572

7573

7574

7575

```

7576
7577 /* SL, as defined in the REQ message for the IB CM
7578    protocol. This is also the "Service Level" to be used
7579    when creating an Address Handle. */
7580 uint8_t          sl : 4;
7581
7582 /* Subnet Local, as defined in the REQ message for the IB
7583    CM protocol. When creating an Address Handle, setting
7584    this bit causes a GRH to be included as part of any
7585    Unreliable Datagram sent using the Address Handle. */
7586 uint8_t          subnet_local : 1;
7587
7588 /* Flow Label, as defined in the REQ message for the IB CM
7589    protocol. This is also the "Flow Label" to be used when
7590    creating an Address Handle. This is only valid if
7591    subnet_local is clear. */
7592 uint32_t         flow_label : 20;
7593
7594 /* Traffic Class, as defined in the REQ message for the IB
7595    CM protocol. This is also the "Traffic Class" to be
7596    used when creating an Address Handle. This is only
7597    valid if subnet_local is clear. */
7598 uint8_t          traffic_class;
7599
7600 /* Hop Limit, as defined in the REQ message for the IB CM
7601    protocol. This is also the "Hop Limit" to be used when
7602    creating an Address Handle. This is only valid if
7603    subnet_local is clear. */
7604 uint8_t          hop_limit;
7605
7606 } it_ib_net_endpoint_t;
7607
7608 /* This is the remote component of the Path information for the
7609    VIA transport */
7610 typedef it_via_net_addr_t it_via_net_endpoint_t;
7611
7612 /* This is the Path data structure used by several routines in
7613    the API */
7614 typedef struct {
7615
7616     /* Identifier for the Spigot to be used on the local IA
7617        Note that this data structure is always used in a
7618        Context where the IA associated with the Spigot can be
7619        deduced. */
7620     size_t          spigot_id;
7621
7622     /* The transport-independent timeout parameter for how long
7623        to wait, in microseconds, before timing out a Connection
7624        establishment attempt using this Path. The timeout
7625        period for establishing a Connection
7626        can only be specified on the Active side; the timeout
7627        period can not be changed on the Passive side. */
7628     uint64_t        timeout;
7629

```

```

7630     /* The remote component of the Path */
7631     union {
7632
7633         /* For use with InfiniBand */
7634         it_ib_net_endpoint_t         ib;
7635
7636         /* For use with VIA */
7637         it_via_net_endpoint_t        via;
7638
7639     } remote;
7640
7641 } it_path_t;

```

7642 DESCRIPTION

7643 The *it_path_t* type is used by several routines in the API to encapsulate a Path between two
7644 Spigots. The *it_path_t* contains a local Spigot identifier, a remote Spigot address, and a
7645 specification of all information that determines the properties of the Path that messages will take
7646 between the two Spigots. The local Spigot to be used is identified in a transport-independent
7647 manner, but the remote Spigot and the Path to that Spigot are specified in a transport-dependent
7648 manner.

7649 The *it_path_t* structure also contains a *timeout* member. This *timeout* member defines how long
7650 the local Consumer is willing to wait for a response to its attempt to establish a Connection when
7651 the *it_path_t* is used with the [it_ep_connect](#) routine. If the Consumer retrieves the Path using the
7652 [it_get_pathinfo](#) routine, the Implementation will provide a *timeout* that should be sufficiently
7653 long to establish a Connection under most circumstances, and so the Consumer should have no
7654 need to modify this value. If the Consumer chooses to provide their own value for the *timeout*
7655 member, the Consumer should take care to choose a value that is compatible with any
7656 underlying transport-specific timeout values governing Connection establishment that may be
7657 present in the transport-specific portion of the *it_path_t*. Choosing a value of *timeout* that is
7658 incompatible with the transport-specific timeout values governing Connection establishment will
7659 result in an error being returned from the [it_ep_connect](#) routine.

7660 All data contained within the *it_path_t* data structure appears in host byte order unless otherwise
7661 noted in the comments associated with the members of the data structure. The contents of the
7662 *path_t* returned from the [it_get_pathinfo](#) routine are only valid on the node on which the routine
7663 was invoked.

7664 RETURN VALUE

7665 None.

7666 ERRORS

7667 None.

7668 SEE ALSO

7669 [it_get_pathinfo\(\)](#), [it_ep_connect\(\)](#), [it_address_handle_create\(\)](#),
7670 [it_ud_service_request_handle_create\(\)](#)

it_software_event_t

7671

7672 NAME7673 `it_software_event_t` – Software Event type**7674 SYNOPSIS**

```

7675 #include <it_api.h>
7676
7677 typedef struct {
7678     it_event_type_t           event_number;
7679     it_evd_handle_t          evd;
7680     void                      *data;
7681 } it_software_event_t;

```

7682 DESCRIPTION

7683 *event_number* Identifier of the Event type. Valid values:
7684 IT_SOFTWARE_EVENT

7685 *evd* Handle for the Event Dispatcher where the Event was queued.

7686 *data* The pointer that the Consumer furnished to the [it_evd_post_se](#)
7687 routine.

7688 An IT_SOFTWARE_EVENT_STREAM Event is generated when the Consumer calls the
7689 [it_evd_post_se](#) routine to post a Software Event.

7690 The IT_SOFTWARE_EVENT_STREAM Event Stream supports Events with the *event_number*
7691 IT_SOFTWARE_EVENT (see [it_event_t](#)).

7692 The Software Event type just passes back the same pointer that the Consumer furnished to the
7693 [it_evd_post_se](#) routine.

7694 All Events on an IT_SOFTWARE_EVENT_STREAM SEVD cause Notification. See
7695 [it_evd_create](#) for details of Notification.

7696 The Software Event type EVD does not overflow. Instead, [it_evd_post_se](#) will generate an
7697 immediate error if the Consumer attempts to queue a software Event on a full Software Event
7698 EVD.

7699 RETURN VALUE

7700 None.

7701 ERRORS

7702 None.

7703 SEE ALSO7704 [it_evd_post_se\(\)](#), [it_evd_create\(\)](#), [it_evd_wait\(\)](#), [it_event_t](#)

it_status_t

7705

7706 **NAME**

7707

it_status_t – definition of IT-API call return status

7708 **SYNOPSIS**

7709

```
#include <it_api.h>
```

7710

```
typedef enum {
```

7711

```
    IT_SUCCESS = 0,
```

7712

```
    IT_ERR_ABORT,
```

7713

```
    IT_ERR_ACCESS,
```

7714

```
    IT_ERR_ADDRESS,
```

7715

```
    IT_ERR_AEVD_NOT_ALLOWED,
```

7716

```
    IT_ERR_ASYNC_AFF_EVD_EXISTS,
```

7717

```
    IT_ERR_ASYNC_UNAFF_EVD_EXISTS,
```

7718

```
    IT_ERR_CANNOT_RESET,
```

7719

```
    IT_ERR_CONN_QUAL_BUSY,
```

7720

```
    IT_ERR_EP_TIMEWAIT,
```

7721

```
    IT_ERR_EVD_BUSY,
```

7722

```
    IT_ERR_EVD_QUEUE_FULL,
```

7723

```
    IT_ERR_FAULT,
```

7724

```
    IT_ERR_IA_CATASTROPHE,
```

7725

```
    IT_ERR_INTERRUPT,
```

7726

```
    IT_ERR_INVALID_ADDRESS,
```

7727

```
    IT_ERR_INVALID_AEVD,
```

7728

```
    IT_ERR_INVALID_AH,
```

7729

```
    IT_ERR_INVALID_ETIMEOUT,
```

7730

```
    IT_ERR_INVALID_CM_RETRY,
```

7731

```
    IT_ERR_INVALID_CN_EST_FLAGS,
```

7732

```
    IT_ERR_INVALID_CN_EST_ID,
```

7733

```
    IT_ERR_INVALID_CONN_EVD,
```

7734

```
    IT_ERR_INVALID_CONN_QUAL,
```

7735

```
    IT_ERR_INVALID_CONVERSION,
```

7736

```
    IT_ERR_INVALID_DTO_FLAGS,
```

7737

```
    IT_ERR_INVALID_EP,
```

7738

```
    IT_ERR_INVALID_EP_ATTR,
```

7739

```
    IT_ERR_INVALID_EP_KEY,
```

7740

```
    IT_ERR_INVALID_EP_STATE,
```

7741

```
    IT_ERR_INVALID_EP_TYPE,
```

7742

```
    IT_ERR_INVALID_EVD,
```

7743

```
    IT_ERR_INVALID_EVD_STATE,
```

7744

```
    IT_ERR_INVALID_EVD_TYPE,
```

7745

```
    IT_ERR_INVALID_FLAGS,
```

7746

```
    IT_ERR_INVALID_HANDLE,
```

7747

```
    IT_ERR_INVALID_IA,
```

7748

```
    IT_ERR_INVALID_LENGTH,
```

7749

```
    IT_ERR_INVALID_LISTEN,
```

7750

```
    IT_ERR_INVALID_LMR,
```

7751

```
    IT_ERR_INVALID_LTIMEOUT,
```

7752

```
    IT_ERR_INVALID_MAJOR_VERSION,
```

7753

```
    IT_ERR_INVALID_MASK,
```

7754

```
    IT_ERR_INVALID_MINOR_VERSION,
```

7755

```

7756     IT_ERR_INVALID_NAME ,
7757     IT_ERR_INVALID_NETADDR ,
7758     IT_ERR_INVALID_NUM_SEGMENTS ,
7759     IT_ERR_INVALID_PDATA_LENGTH ,
7760     IT_ERR_INVALID_PRIVS ,
7761     IT_ERR_INVALID_PZ ,
7762     IT_ERR_INVALID_QUEUE_SIZE ,
7763     IT_ERR_INVALID_RECV_EVD ,
7764     IT_ERR_INVALID_RECV_EVD_STATE ,
7765     IT_ERR_INVALID_REQ_EVD ,
7766     IT_ERR_INVALID_REQ_EVD_STATE ,
7767     IT_ERR_INVALID_RETRY ,
7768     IT_ERR_INVALID_RMR ,
7769     IT_ERR_INVALID_RNR_RETRY ,
7770     IT_ERR_INVALID_RTIMEOUT ,
7771     IT_ERR_INVALID_SGID ,
7772     IT_ERR_INVALID_SLID ,
7773     IT_ERR_INVALID_SOFT_EVD ,
7774     IT_ERR_INVALID_SOURCE_PATH ,
7775     IT_ERR_INVALID_SPIGOT ,
7776     IT_ERR_INVALID_THRESHOLD ,
7777     IT_ERR_INVALID_UD_STATUS ,
7778     IT_ERR_INVALID_UD_SVC ,
7779     IT_ERR_INVALID_UD_SVC_REQ_ID ,
7780     IT_ERR_LMR_BUSY ,
7781     IT_ERR_MISMATCH_FD ,
7782     IT_ERR_NO_CONTEXT ,
7783     IT_ERR_NO_PERMISSION ,
7784     IT_ERR_PAYLOAD_SIZE ,
7785     IT_ERR_PDATA_NOT_SUPPORTED ,
7786     IT_ERR_PZ_BUSY ,
7787     IT_ERR_QUEUE_EMPTY ,
7788     IT_ERR_RANGE ,
7789     IT_ERR_RESOURCES ,
7790     IT_ERR_RESOURCE_IRD ,
7791     IT_ERR_RESOURCE_LMR_LENGTH ,
7792     IT_ERR_RESOURCE_ORD ,
7793     IT_ERR_RESOURCE_QUEUE_SIZE ,
7794     IT_ERR_RESOURCE_RECV_DTO ,
7795     IT_ERR_RESOURCE_REQ_DTO ,
7796     IT_ERR_RESOURCE_RRSEG ,
7797     IT_ERR_RESOURCE_RSEG ,
7798     IT_ERR_RESOURCE_RWSEG ,
7799     IT_ERR_RESOURCE_SSEG ,
7800     IT_ERR_TIMEOUT_EXPIRED ,
7801     IT_ERR_TOO_MANY_POSTS ,
7802     IT_ERR_WAITER_LIMIT
7803 } it_status_t;

```

7804 DESCRIPTION

7805 Most IT-API function calls return *it_status_t* on function completion. IT_SUCCESS indicates
7806 that an IT-API operation was invoked successfully; otherwise the return code indicates the
7807 reason for failure. See each individual man page for the meaning of a return code in the Context
7808 of the function.

7809 Some API function calls are used to initiate asynchronous operations. For those function calls, a
 7810 return value of IT_SUCCESS indicates only that the operation was successfully initiated; it does
 7811 not indicate that it was successfully completed. To determine if an asynchronous operation was
 7812 successfully completed, the Completion Event for the asynchronous operation should be
 7813 examined.

7814 **RETURN VALUE**
 7815 None.

7816 **ERRORS**
 7817 None.

7818 **SEE ALSO**
 7819 [it_address_handle_create\(\)](#), [it_address_handle_free\(\)](#), [it_address_handle_modify\(\)](#),
 7820 [it_address_handle_query\(\)](#), [it_convert_net_addr\(\)](#), [it_ep_accept\(\)](#), [it_ep_connect\(\)](#),
 7821 [it_ep_disconnect\(\)](#), [it_ep_free\(\)](#), [it_ep_modify\(\)](#), [it_ep_query\(\)](#), [it_ep_rc_create\(\)](#), [it_ep_reset\(\)](#),
 7822 [it_ep_ud_create\(\)](#), [it_evd_create\(\)](#), [it_evd_dequeue\(\)](#), [it_evd_free\(\)](#), [it_evd_modify\(\)](#),
 7823 [it_evd_post_se\(\)](#), [it_evd_query\(\)](#), [it_evd_wait\(\)](#), [it_get_consumer_context\(\)](#),
 7824 [it_get_handle_type\(\)](#), [it_get_pathinfo\(\)](#), [it_handoff\(\)](#), [it_ia_create\(\)](#), [it_ia_free\(\)](#), [it_ia_query\(\)](#),
 7825 [it_listen_create\(\)](#), [it_listen_free\(\)](#), [it_listen_query\(\)](#), [it_lmr_create\(\)](#), [it_lmr_free\(\)](#),
 7826 [it_lmr_modify\(\)](#), [it_lmr_query\(\)](#), [it_lmr_sync_rdma_read\(\)](#), [it_lmr_sync_rdma_write\(\)](#),
 7827 [it_post_rdma_read\(\)](#), [it_post_rdma_write\(\)](#), [it_post_rcv\(\)](#), [it_post_rcvfrom\(\)](#), [it_post_send\(\)](#),
 7828 [it_post_sendto\(\)](#), [it_pz_create\(\)](#), [it_pz_free\(\)](#), [it_pz_query\(\)](#), [it_reject\(\)](#), [it_rmr_bind\(\)](#),
 7829 [it_rmr_create\(\)](#), [it_rmr_free\(\)](#), [it_rmr_query\(\)](#), [it_rmr_unbind\(\)](#), [it_set_consumer_context\(\)](#),
 7830 [it_ud_service_reply\(\)](#), [it_ud_service_request\(\)](#), [it_ud_service_request_handle_create\(\)](#),
 7831 [it_ud_service_request_handle_free\(\)](#), [it_ud_service_request_handle_query\(\)](#), [it_dto_events](#)

it_unaffiliated_event_t

7832

7833 **NAME**7834 `it_unaffiliated_event_t` – Unaffiliated Asynchronous Event type7835 **SYNOPSIS**

```

7836 #include <it_api.h>
7837
7838 typedef struct {
7839     it_event_type_t      event_number;
7840     it_evd_handle_t      evd;
7841     it_ia_handle_t       ia;
7842
7843     size_t               spigot_id;
7844 } it_unaffiliated_event_t;

```

7845 **DESCRIPTION**

7846 *event_number* Identifier of the Event type. Valid values:
7847 IT_ASYNC_UNAFF_SPIGOT_ONLINE,
7848 IT_ASYNC_UNAFF_SPIGOT_OFFLINE,
7849 IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE

7850 *evd* Handle for the Event Dispatcher where the Event was queued.

7851 *ia* The Handle for the IA that experienced the Unaffiliated Event.

7852 *spigot_id* The identifier for the Spigot that changed state on the IA. Valid only
7853 for the IT_ASYNC_UNAFF_SPIGOT_ONLINE and
7854 IT_ASYNC_UNAFF_SPIGOT_OFFLINE Events.

7855 IT_ASYNC_UNAFF_EVENT_STREAM Events are generated when an Unaffiliated
7856 Asynchronous Event occurs. There are several types of Unaffiliated Asynchronous Events, and
7857 each type is identified by *event_number*. The Consumer asks for Unaffiliated Asynchronous
7858 Events to be delivered when it creates an EVD for the Unaffiliated Asynchronous Event Stream
7859 using the [it_evd_create](#) call.

7860 The following table maps the values in the Unaffiliated Asynchronous Events [it_event_type_t](#)
7861 enumeration to a transport independent description.

it_event_type_t value	Generic Event Description
IT_ASYNC_UNAFF_SPIGOT_ONLINE	A Spigot on the IA that was previously offline is now online. The Implementation will only generate this Event if the <i>it_ia_info.spigot_online_event_support</i> value is IT_TRUE.
IT_ASYNC_UNAFF_SPIGOT_OFFLINE	A Spigot on the IA that was previously online is now offline. The Implementation will only generate this Event if the <i>it_ia_info.spigot_offline_event_support</i> value is

	IT_TRUE.
IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE	The API Implementation was unable to enqueue an entry into an Affiliated Asynchronous Event SEVD.

7862 **EXTENDED DESCRIPTION**

7863 For the Infiniband transport, the following table maps the values in the Unaffiliated
7864 Asynchronous Errors [it_event_type_t](#) enumeration to their corresponding “Unaffiliated
7865 Asynchronous Events” and “Unaffiliated Asynchronous Errors” as specified in Volume 1,
7866 Chapter 11 of the Infiniband specification.

it_event_type_t value	IB “Unaffiliated Asynchronous Event/Error” name
IT_ASYNC_UNAFF_SPIGOT_ONLINE	Port Active
IT_ASYNC_UNAFF_SPIGOT_OFFLINE	Port Error

7867 For the VIA transport, the following table maps the values in the Unaffiliated Asynchronous
7868 Errors [it_event_type_t](#) enumeration to their corresponding descriptions in the
7869 “VipErrorCallback” man page in the Appendix of the VIA specification.

it_event_type_t value	VIA “VipErrorCallback” name(s)
IT_ASYNC_UNAFF_SPIGOT_ONLINE	(Not applicable to the VIA transport.)
IT_ASYNC_UNAFF_SPIGOT_OFFLINE	(Not applicable to the VIA transport.)

7870 All Events on an IT_ASYNC_UNAFF_EVENT_STREAM SEVD cause Notification. See
7871 [it_evd_create](#) for details of Notification.
7872

7873 Overflow of an IT_ASYNC_UNAFF_EVENT_STREAM SEVD is not visible to the Consumer;
7874 all subsequent IT_ASYNC_UNAFF_EVENT_STREAM Events are silently dropped until the
7875 Consumer dequeues at least one Event from the EVD that contains the
7876 IT_ASYNC_UNAFF_EVENT_STREAM.

7877 When a Consumer has created more than one IA corresponding to an underlying physical
7878 adapter (say in different processes), then every Unaffiliated Event is replicated to every IA
7879 instance.

7880 **RETURN VALUE**

7881 None.

7882 **ERRORS**

7883 None.

7884 **SEE ALSO**

7885 [it_ia_create\(\)](#), [it_event_t](#), [it_evd_create\(\)](#), [it_evd_wait\(\)](#)

7886 A. Implementer's Guide

7887 The IT-API Standard does not prohibit any implementation from providing functionality beyond
 7888 that specified in the standard. However, we urge that implementations and authors of code using
 7889 IT-API avoid the "it_" prefix for any function name or data structure name not defined by the
 7890 standard. This is to preserve the option for future enhancement of IT-API without concern that a
 7891 new IT-API name will conflict with a name used in an existing Implementation or application.

7892 **it_address_handle_create**

7893 The Infiniband Create Address Handle verb doesn't take a Source GID as input; it takes a Source GID index.
 7894 The Implementation therefore needs to use the Query HCA verb to get access to the GID table associated
 7895 with the port identified by *spigot_id*, and match the input *ib.local_port_gid* field to an entry in the GID table
 7896 to determine the appropriate Source GID index to use.

7897 The Infiniband Create Address Handle verb doesn't take a Source LID as input, it takes the Source Path Bits
 7898 instead and uses them in conjunction with the Base LID to create the appropriate SLID to use. The
 7899 Implementation therefore needs to the Query HCA verb to retrieve the LMC associated with the port
 7900 identified by *spigot_id* to determine how many of the low order bits in the input *ib.local_port_lid* need to be
 7901 extracted as the Source Path Bits.

7902 When running over the InfiniBand transport, if the Consumer provides a Path to [it_address_handle_create](#)
 7903 that contains a P_Key that is not in the HCA's P_Key table, the Implementation shall return
 7904 IT_ERR_INVALID_SOURCE_PATH.

7905 **it_affiliated_event_t**

7906 Asynchronous Events should be copied from hardware resources into per-process software queues. The effect
 7907 of overflow of the software queue should be isolated to the owning process. When overflow of the Affiliated
 7908 Event EVD occurs, hardware resources should still be dequeued and discarded.

7909 IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE should be generated for overflow on all Simple EVDs
 7910 with the exception of the Affiliated and Unaffiliated Event EVDs.

7911 **it_cm_msg_events**

7912 IT_CM_MSG_CONN_BROKEN_EVENT Events should be synthesized by the Implementation from
 7913 asynchronous Events, etc., generated by the underlying transport. The Consumer has the option of ignoring
 7914 asynchronous Events (by not creating an EVD for the Affiliated or Unaffiliated Asynchronous Event
 7915 Streams) but still needs warning of state changes affecting their Endpoints.

7916 In general, all transport-specific Connection Management rejection Events not explicitly defined in this API
 7917 should be implemented as IT_CM_MSG_CONN_NONPEER_REJECT_EVENT with IT_CN_REJ_OTHER
 7918 reject reason code Events.

7919 When running over the InfiniBand transport, Implementations have the option to "chew up" a REJ that is
 7920 returned with reject reason code 1 ("No QP available"), 3 ("No resources available"), or 4 (Timeout) rather
 7921 than immediately posting an Event with status IT_CN_REJ_RESOURCES (for REJ codes 1 and 3) or

7922 IT_CN_REJ_OTHER (for REJ code 4). The Implementation may wait until the timeout specified by the
 7923 Consumer in the [it_path_t](#) structure input to [it_ep_connect](#) expires and then enqueue a non-peer reject Event
 7924 with an IT_CN_REJ_TIMEOUT status. Alternatively, within the specified timeout period the
 7925 Implementation may retry the Connection establishment attempt on the Consumer's behalf. If a Connection
 7926 could not be established within the Consumer-specified timeout period, the Implementation should enqueue a
 7927 non-peer reject Event with an IT_CN_REJ_TIMEOUT status after the timeout period has expired.

7928 When running on the IB transport, there are two different things that can signal the Implementation that it
 7929 should generate the IT_CM_MSG_CONN_ESTABLISHED_EVENT on the Passive side: receiving an RTU
 7930 message, or receiving a "Communication Established" Affiliated Asynchronous Event from the HCA. (Due
 7931 to inherent races in the IB Connection establishment process, it is also possible that both of these conditions
 7932 could be present.) If the Implementation receives an RTU message while the Endpoint is in the
 7933 IT_EP_STATE_PASSIVE_CONNECTION_PENDING state and within the timeout period advertised to the
 7934 Passive side in the REQ message, it should generate an IT_CM_MSG_CONN_ESTABLISHED_EVENT
 7935 Event and use the Private Data from the RTU as part of that Event. If the Implementation receives the
 7936 "Communication Established" Affiliated Asynchronous Event without receiving an RTU, the Implementation
 7937 should generate the IT_CM_MSG_CONN_ESTABLISHED_EVENT Event with a Private Data size of zero,
 7938 and when/if the RTU for the Connection subsequently arrives the Implementation should ignore it.

7939 There is a potential race condition in the three-way handshake Connection establishment method between the
 7940 Implementation generating the IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT Event and the
 7941 Consumer calling [it_ep_disconnect](#) (or [it_ep_free](#)).

7942 The conditions for the race arise when a Consumer has called [it_ep_connect](#), but before the Connection is
 7943 successfully established, the Consumer calls [it_ep_disconnect](#) or [it_ep_free](#) on the Endpoint. Within the time
 7944 frame of this single Connection attempt, the Implementation must order Events as follows.

7945 It is acceptable for the Implementation to generate and queue the IT_CM_MSG_CONN_
 7946 ACCEPT_ARRIVAL_EVENT Event to the SEVD prior to the IT_CM_MSG_CONN_DISCONNECT_
 7947 EVENT Event. But, when the IT_CM_MSG_CONN_DISCONNECT_EVENT Event is generated, the
 7948 Implementation must invalidate the *cn_est_id* found in the IT_CM_MSG_CONN_ACCEPT_
 7949 ARRIVAL_EVENT Event. Likewise, if the Consumer calls [it_ep_free](#), the Implementation must also
 7950 invalidate the *cn_est_id*. (Note that it is possible that the *cn_est_id* may have already been invalidated by a
 7951 Consumer call to [it_ep_accept](#), [it_reject](#) or [it_handoff](#).)

7952 On the other hand, if the Implementation has generated an IT_CM_MSG_CONN_DISCONNECT_EVENT
 7953 Event, and subsequently the Implementation receives indication that the Connection Request has been
 7954 accepted by the remote side, the Implementation shall not generate an IT_CM_MSG_CONN_ACCEPT_
 7955 ARRIVAL_EVENT Event. In this situation, the Implementation is still responsible for generating any
 7956 necessary transport-specific response to the arrived acceptance message.

7957 **it_conn_qual_t**

7958 When the IANA Port Number Connection Qualifier type is used with the VIA transport, the IANA Port
 7959 Number is mapped into a 2-byte VIA connection discriminator, with byte 0 of the connection discriminator
 7960 containing the upper 8 bits of the 16-bit IANA Port Number, and byte 1 containing the lower 8 bits of the 16-
 7961 bit IANA Port Number.

7962 When the IANA Port Number Connection Qualifier type is used with the InfiniBand transport, the IANA Port
 7963 Number is mapped into the 64-bit Service ID as follows:

Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7	Byte 8
--------	--------	--------	--------	--------	--------	--------	--------

0x10	0x000CE1	0x0	0x0	IANA Port Number
------	----------	-----	-----	------------------

7964 **it_dto_flags_t**

7965 The Implementation should attempt to support the use of IT_NOTIFY_FLAG on Receive DTOs. Where the
 7966 underlying transport does not support Receive DTO Notification Suppression it may be necessary for the
 7967 Implementation to generate Receive Notifications regardless of the setting of the IT_NOTIFY_FLAG on the
 7968 Receive DTOs.

7969 **it_ep_accept**

7970 In all case where a communication manager message changes the state of an Endpoint the Implementation
 7971 must first transition the Endpoint state before generating the Event. This closes a race condition where the
 7972 Consumer may see the Event and call a function expecting the Endpoint to be in the state that the Event
 7973 results in. For example, when the Consumer reaps the IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT
 7974 the Endpoint should be in the IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state and ready to be
 7975 accepted.

7976 **it_ep_attributes_t**

7977 The Implementation must allocate resources needed for the support of the Consumer-requested attribute. In
 7978 general, the Implementation can allocate more resources than requested by Consumer (a few exceptions are
 7979 noted in the next paragraph). Some of these resources may actually be allocated at Connection establishment
 7980 time for RC, but Connection establishment can not fail because resources requested by Consumer at Endpoint
 7981 creation time are not available. Connection establishment may fail when remote Endpoint of the Connection
 7982 does not have enough resources to match local Endpoint ones. For example, if the local Endpoint's attributes
 7983 have an *rdma_read_inflight_incoming* that is less than the remote Endpoint's *rdma_read_inflight_outgoing*,
 7984 the Connection establishment attempt will fail.

7985 The attributes that the Implementation must allocate in exactly the quantity that the Consumer specified are:

7986 *it_rc_only_attributes_t.rdma_read_inflight_outgoing*

7987 *it_ep_attributes_t.max_dto_payload_size*

7988 The reason *rdma_read_inflight_outgoing* is listed above is that for InfiniBand you can't establish a
 7989 Connection unless the ORD value for one side of the Connection is less than or equal to the IRD value for
 7990 other side.

7991 The reason *max_dto_payload_size* is listed above is that for VIA you can't establish a Connection unless the
 7992 passive side and the active side Endpoints have matching values for this attribute.

7993 Whether the *max_dto_payload_size* limit is actually enforced for data transfers is IA-specific. (it might not be
 7994 transport-specific; but it is not clear if all VIA Implementations do this checking.)

7995 An attempt to change *max_request_dtos* or *max_recv_dtos* for an Endpoint of an Interface Adapter whose
 7996 *it_ia_info_resizable_work_queue* is Clear must not be successful and IT_ERR_INVALID_EP_STATE is the
 7997 return value for this case.

7998 When running over the InfiniBand transport, the Implementation must set Signaling type to Selectable in
 7999 order to support [it_dto_flags](#).

8000 it_ep_connect

8001 When running over the InfiniBand transport, if the Consumer provides a Path to [it_ep_connect](#) that contains a
 8002 P_Key that is not in the HCA's P_Key table, the Implementation shall return IT_ERR_INVALID_
 8003 SOURCE_PATH.

8004 it_ep_disconnect

8005 If the EP is already in non-operation state, no messages or Events should be generated. In this case, the
 8006 transport level Disconnect Request should have been sent when the EP transitioned into the non-operational
 8007 state.

8008 When running over the InfiniBand transport, the Implementation should send CM DREQ (Disconnect
 8009 Request).

8010 it_ep_free

8011 The [it_ep_free](#) is equivalent to the destruction of the underlying transport Endpoint. Except as noted below
 8012 for the *cn_est_id*, the Implementation is at liberty to retain resources until such a time as it is capable of
 8013 freeing them. For IB this means that Completion Events may be left on the CQ after QP destruction, and CM
 8014 generated Events may be left on connect EVD.

8015 This call must destroy the *cn_est_id* associated with the Endpoint if it has not been destroyed before. If the
 8016 *cn_est_id* was destroyed it should not cause any problem for the Implementation. The [it_ep_free](#) call when
 8017 the Endpoint is in IT_EP_STATE_ACTIVE1_CONNECTION_PENDING may be racing with the
 8018 Implementation generating IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT that creates *cn_est_id*.

8019 When a remote Endpoint involved in Connection establishment is destroyed, locally the
 8020 IT_CM_MSG_CONN_NONPEER_REJECT_EVENT shall be generated with either IT_CN_REJ_OTHER,
 8021 or IT_CN_REJ_TIMEOUT reject_code reason.

8022 it_ep_rc_create

8023 The Implementation should ignore the *it_ep_rc_creation_flag_t* parameter on transports where the timewait
 8024 state is not applicable.

8025 it_ep_reset

8026 This operation must hide any internal Implementation waiting for timeout expiration that Endpoint may be in
 8027 due to [it_ep_disconnect](#) call during Connection set up.

8028 it_ep_state_t

8029 In all case where a communication manager message changes the state of an Endpoint the Implementation
 8030 must first transition the Endpoint state before generating the Event. This closes a race condition where the
 8031 Consumer may see the Event and call a function expecting the Endpoint to be in the state that the Event
 8032 results in. For example, when the Consumer reaps the IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT
 8033 the Endpoint should be in the IT_EP_STATE_ACTIVE2_CONNECTION_PENDING state and ready to be
 8034 accepted.

8035 The Connection establishment identifier (*conn_est_id*) object should not be destroyed by the Implementation
 8036 when Endpoint transition state occurs due to a communication manager message or error. They should only
 8037 be destroyed by explicit Consumer initiated functions such as [it_ep_accept](#), [it_reject](#) and [it_ep_disconnect](#).

8038 **it_evd_create**

8039 An IT-API Implementation should meet the following rules for EVD behavior.

8040 **Definitions:**

- 8041 1. An arriving (queued on SEVD) Event is a *notification event* if any one of the following is true:
- 8042 a. is an Event for a DTO with IT_NOTIFY_FLAG set when posted.
- 8043 b. is an Event for a Recv DTO where matching Send DTO was originally posted with
- 8044 IT_SOLICITED_WAIT_FLAG set (regardless of IT_NOTIFY_FLAG on Recv DTO).
- 8045 c. is the Nth Event to arrive where threshold is set to N.
- 8046 d. is an Event for a DTO completing in error regardless of IT_NOTIFY_FLAG and
- 8047 IT_COMPLETION_FLAG.
- 8048 e. is a non-DTO Completion Event

8049 The above are called *arriving notification events*.8050 The Events of type *a, b, d* and *e* are also called plain *notification events* and retain their *notification*
8051 *status*¹ on the SEVD queue.

- 8052 2. An arriving Event is a *non-notification event* if none of the above 1a. to 1e. criteria are met. These are
- 8053 called both *arriving non-notification events* and plain *non-notification events*.
- 8054 3. IT_THRESHOLD_DISABLED stands for no threshold or threshold == infinity.
- 8055 4. The desired semantic for IT-API is: *it_evd_wait* returns only:
- 8056 a. for SEVD - when there is a Notification Event of 1a,1b,1d, or 1e type (above) or when there
- 8057 are a number of Events on the SEVD equal to or more than the threshold on the SEVD
- 8058 queue.
- 8059 b. for AEVD - when any one of the associated SEVDs satisfies 4a.

8060 **Rules:**

- 8061 1. *it_evd_wait* **will** block² when:
- 8062 a. for SEVD - queue is empty.
- 8063 b. for AEVD - all associated SEVDs are empty.
- 8064 2. *it_evd_wait* **may** block if there are no *notification events* and number of Events is below the
- 8065 threshold:
- 8066 a. for SEVD - on the SEVD queue.
- 8067 b. for AEVD - on all associated SEVD queues.
- 8068 3. *it_evd_wait* **will** return if there is a *notification event*³ or the number of Events is greater than or equal

¹ InfiniBand does not retain the notion of *notification status* for types *a, b,* and *d* after arrival. Hence, Implementations that do not rely on this notion, and only rely on **arriving notification events** must be permitted. Events of type **e** have its own Event Stream types and its own SEVDs. Hence, Implementation can retain the **notification status** for them based on SEVD Event Stream type.

² This is really an Implementation issue. Semantically, Consumer does not care if it is blocked or not.

- 8069 to the *threshold*:
- 8070 a. for SEVD - on the SEVD queue.
- 8071 b. for AEVD - on any associated SEVDs.
- 8072 4. If threshold > 1, `evd_wait` **should** block⁴ if less than threshold number of Events and no *notification*
8073 *events* are⁵
- 8074 a. for SEVD - on the SEVD queue.
- 8075 b. for AEVD - on all SEVD queues of AEVD.
- 8076 5. Each *arriving notification event* **must** unblock at least one waiter⁶, but **should** unblock only one
8077 waiter.
- 8078 6. An *arriving notification event* **can** unblock as many waiters as there are Events available.
- 8079 7. [*it_evd_dequeue*](#) **will** return an Event if one exists regardless of waiters from
- 8080 a. for SEVD - the SEVD queue.
- 8081 b. for AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS` set – the AEVD queue of
8082 `IT_AEVD_NOTIFICATION_EVENTS`.
- 8083 c. for AEVD with `IT_EVD_DEQUEUE_NOTIFICATIONS` cleared - any of its SEVDs⁷.

8084 **Heuristic for wakeup of multiple waiters:**

8085 Rules 5 and 6 in the EVD Rules (above) require that multiple waiters be awakened on every Notification
8086 Event because of the possibility of Notification coalescing.

8087 The following heuristic is recommended: In the Notification Handler, check the queue length⁸, and wake up
8088 that many waiters and no more. This limits the number of threads that wake up, and reduces the number that
8089 wake up and find no Event because some other thread(s) has consumed them.

8090 The handler algorithm should be something like:

```
8091     evd_handler() {
8092         for (;;) {
8093             nToUnblock = min(nmore(), nwaiters());
```

³ This semantic mandates that events retain their notification status on the EVD (a “sticky notification” semantic).

However, the event that passes a threshold number of events should not be considered to be a notification event – only the number of events on the SEVD(s) should be considered at the time `it_evd_wait` is called (if any SEVD has threshold number of events on it, then `it_evd_wait` must return).

⁴ This terminology allows an Implementation that can support thresholds as well as other notification events in hardware to do so while not mandating that those Implementations that cannot use suboptimal schemes.

⁵ with `sevd_threshold = IT_THRESHOLD_DISABLED (= infinity)`, `it_evd_wait` will block if no notification event are on EVD. If a notification event is on the EVD, `it_evd_wait` will return each event until the notification event is dequeued - from then on, `it_evd_wait` *may* block.

⁶ Events that arrive when there are no blocked `evd_wait` *should* retain their *notification status*. Events that had `IT_SOLICITED_WAIT_FLAG` or `IT_NOTIFY_FLAG` set when originally posted or completions with errors that arrive at a time when no waiters are waiting will cause `it_evd_wait` to return when later called on the EVD. Implementation *can* be *over-eager* and return an `it_evd_wait` without checking the *notification status* of events on EVD

⁷ If there is an event on any of the feeding SEVDs, `it_evd_dequeue` must return it regardless of the notification status of the SEVD and even if the SEVD is disabled.

⁸ The queue length function is not Verbs compliant, but will be commonly available.


```

8094         for (n = 0; n != nToUnblock; ++n)
8095             unblockWaiter();
8096         if (nwaiters() == 0) break;
8097         rearmHandler();
8098         if (nmore() == 0) break;
8099     }
8100 }
8101

```

8102 **Additional issues:**

8103 An AEVD does not have a queue as such. Potentially, an AEVD can be implemented using a bit array with
8104 an entry for each feeding SEVD. An SEVD bit "set" for an AEVD with IT_EVD_DEQUEUE_
8105 NOTIFICATIONS set means that the SEVD Handle can be returned in an IT_AEVD_NOTIFICATION_
8106 EVENT Event from the AEVD wait or dequeue. The SEVD bit "set" for an AEVD with
8107 IT_EVD_DEQUEUE_NOTIFICATIONS cleared means that the actual underlying SEVD Event can be
8108 returned from the AEVD wait or dequeue.

8109 Ideally, the SEVD bit is cleared as soon as the SEVD is not in the Notification criteria. But it can be cleared
8110 as soon as the SEVD becomes empty or, instead, only when both it becomes empty and there is direct (waiter
8111 on SEVD) or indirect waiter (AEVD waiter or FD) on SEVD.

8112 If SEVD is enabled, then an arriving SEVD Event that causes SEVD to reach Notification criteria or maintain
8113 Notification criteria of the SEVD will set the SEVD bit for the AEVD.

8114 It is recommended that the Implementation employ a "starvation-free" algorithm in returning Events via an
8115 AEVD from underlying SEVDs. That is, the Implementation should ensure forward progress on all SEVDs
8116 feeding an AEVD.

8117 For IT_EVD_DEQUEUE_NOTIFICATION set, the Implementation should eventually select every feeding
8118 SEVD that has reached Notification status when generating the next AEVD Notification Event.

8119 For IT_EVD_DEQUEUE_NOTIFICATION cleared, the Implementation of [it_evd_wait](#) should eventually
8120 select the first Event from every feeding SEVD that has reached Notification status.

8121 When the requested stream type is IT_ASYNC_AFF_EVENT_STREAM or IT_ASYNC_UNAFF_EVENT_
8122 STREAM, the Implementation creates the requested EVD, and fills in the appropriate evd field in the
8123 associated [it_ia_info_t](#) structure for that IA.

8124 The Implementation shall not provide to the Consumer Un-affiliated and Affiliated Events that happened
8125 before Consumer created SEVDs for Unaffiliated and Affiliated Event Stream. The Implementation can
8126 provide Events that happened during SEVD creation time.

8127 **it_evd_dequeue**

8128 The Implementation should abide by the EVD rules defined in the [it_evd_create](#) section of this Implementers
8129 Guide.

8130 All synchronization issues between multiple waiters and/or dequeue-ers from the same Event Dispatcher
8131 simultaneously are left to Implementation.

8132 Implementation is not required to check that the *Event* structure that Consumer provides is sufficient to hold a
8133 returned Event.

8134 **it_evd_post_se**

8135 All the synchronization issues between multiple Consumer Contexts trying to post software Events to an
8136 Event Dispatcher instance simultaneously are left to the Implementation.

8137 **it_evd_wait**

8138 The Implementation should abide by the EVD rules defined in the [it_evd_create](#) section of this Implementers
8139 Guide.

8140 All synchronization issues between multiple waiters and/or dequeue-ers from the same Event Dispatcher
8141 simultaneously are left to Implementation.

8142 Implementation is not required to check that the *event* structure that Consumer provides is sufficient to hold a
8143 returned Event.

8144 **it_event_t**

8145 Each Event Stream has a designated contiguous range of Event numbers with common most-significant bits
8146 (as masked by IT_EVENT_STREAM_MASK) representing the Event Stream.

8147 **it_handle_t**

8148 The definition of all object Handles is Implementation specific, but that all object Handles can be typecast to
8149 [it_handle_t](#) and vice versa.

8150 **it_handoff**

8151 Once the IA is open, and until all processes close it, the *ia_name* and Spigot identifier need to reference the
8152 same objects for all processes that are referencing the IA.

8153 **it_ia_create**

8154 The Implementation shall not provide to the Consumer Un-affiliated and Affiliated Events that happened
8155 before Consumer created SEVDs for Unaffiliated and Affiliated Event Stream. The Implementation can
8156 provide Events that happened during SEVD creation time.

8157 **it_ia_free**

8158 The Implementation should free the IT Objects in the reverse order of their construction in order to guarantee
8159 that all underlying transport resources will be successfully freed.

8160 **it_ia_info_t**

8161 The 1000+ number range in the *it_transport_type_t* is intended to facilitate the short-term prototyping efforts
8162 of vendors who are developing new transports that work with the IT-API. A vendor that wishes to productize
8163 their prototyping effort should contact the ICSC in order to be assigned a permanent transport number in the
8164 < 1000 range. The Implementation is not responsible for ensuring that two different vendors utilizing the
8165 1000+ range of transport numbers do not collide.

8166 For the IB transport, when [it_ep_disconnect](#) is called there are two possible underlying CM messages that the
8167 Private Data could travel with: DREQ, or REJ. The value of *disconnect_private_data_len* for IB should be
8168 the lesser of the maximum Private Data supported in a DREQ message and the maximum Private Data
8169 supported in a REJ message.

8170 **it_lmr_create**

8171 If the EVD where the CM Request Events stream (IT_CM_REQ_EVENT_STREAM) is routed on the
 8172 passive side is full, then the Active side shall receive an IT_CM_MSG_CONN_NONPEER_REJECT_
 8173 EVENT Event with a reason code of IT_CN_REJ_TIMEOUT.

8174 **it_lmr_create**

8175 [it_lmr_create](#) can be implemented on the InfiniBand transport by using the Register Memory Region Verb.
 8176 To allow subsequent calls to [it_rmr_bind](#) using this LMR, the Implementation must include the Enable
 8177 Memory Window Binding input modifier, and also the Enable Local Write Access modifier. The latter is
 8178 necessary because the subsequent [it_rmr_bind](#) call may request remote write access, and the InfiniBand
 8179 Architecture specifies that local access is a pre-requisite for remote access. Note that the Register Memory
 8180 Region Verb implicitly enables local access; so on InfiniBand transport the Consumer's setting of the
 8181 IT_PRIV_LOCAL_READ flag in the privs argument is effectively ignored.

8182 The IT_LMR_FLAG_SHARED option can be implemented using the InfiniBand Register Memory Region
 8183 and Register Shared Memory Region Verbs. The Implementation should search for a matching LMR. If a
 8184 match is found, call the Register Shared Memory Region Verb; if not, call the Register Memory Region Verb.
 8185 The Implementation should protect against race conditions between multiple Consumers and avoid calling
 8186 Register Memory Region multiple times for the same memory region.

8187 See the advice under [it_rmr_bind](#) for comments concerning *rmr_context* and byte order.

8188 **it_lmr_free**

8189 Beware that determining when it is permissible to unlock physical memory is tricky. The Implementation
 8190 must handle multiple LMRs with overlapping ranges, LMRs on different IAs with overlapping ranges, and
 8191 LMRs created in overlapping regions of shared memory by different Consumers. In addition, any of these
 8192 LMRs may have been created with the IT_LMR_FLAG_SHARED flag set.

8193 **it_lmr_query**

8194 [it_lmr_query](#) can be implemented on the InfiniBand transport by using the Query Memory Region Verb. The
 8195 *actual_addr* and *actual_length* fields in params should be derived from the Actual Remote Protection Bounds
 8196 returned by the Verb, because the Consumer will be most concerned with the degree of exposure of the
 8197 region to remote Consumers. In addition, the Actual Remote Protection Bounds will be contained within the
 8198 Actual Local Protection Bounds, because the remote bounds are rounded to 4K byte boundaries, and local
 8199 bounds are rounded to page boundaries. This means the Consumer may safely use the returned *actual_addr*
 8200 and *actual_length* as both local and remote bounds.

8201 **it_lmr_sync_rdma_read**8202 **it_lmr_sync_rdma_write**

8203 There is no defined error code for the case where some portion of the *local_segments* array lies outside the
 8204 Consumer's valid address space. It is expected that the Implementation will signal the application with the
 8205 appropriate platform-dependent signal in this case, as would happen for any dereference of an invalid pointer.

8206

8207 **it_rmr_bind**

8208 The *rmr_context* returned by *it_rmr_bind* is defined to be returned in network byte order, e.g. in big endian
 8209 format. The intent is that no further reordering of the bytes of the *rmr_context* will be performed by either
 8210 the local or remote Consumer, or the local or remote Implementation. When the remote Consumer passes the
 8211 *rmr_context* to a call such as *it_post_rdma_write*, the Implementation of [it_post_rdma_write](#) will put the first
 8212 byte of *rmr_context* on the network wire first, the second byte of *rmr_context* second, and so on. Thus, the
 8213 Implementation of [it_rmr_bind](#) should return the *rmr_context* in the byte order that the IA expects to see on
 8214 the wire, so that the IA may correctly interpret the incoming *rmr_context*.

8215 **it_rmr_query**

8216 In order to return correct values for the *bound*, *lmr*, *addr*, *length*, and *privs* attributes of params, the
 8217 Implementation must track the completion status of Bind and Unbind operations, and the parameters
 8218 associated with those operations. To do so, the Implementation can replace the Consumer's cookie passed to
 8219 [it_rmr_bind](#) or [it_rmr_unbind](#) with an Implementation cookie that points to a structure. This structure stores
 8220 the original Consumer cookie and all relevant parameters to the Bind or Unbind operation. In
 8221 *it_evd_dequeue*, the Implementation would peek at the operation type of the dequeued Completion Event. If
 8222 the type is a Bind operation (note that InfiniBand does not distinguish this from an Unbind operation), and the
 8223 completion status is successful, then extract the Implementation cookie, and restore the Consumer's cookie.
 8224 Read the structure, and copy the parameters to storage associated with the RMR object. The next
 8225 [it_rmr_query](#) call can obtain its parameters from this storage.

8226 The *rmr_context* attribute could be handled similarly, or the Query Memory Region Verb could be used for
 8227 InfiniBand transport.

8228 **it_post_rdma_read**

8229 The Implementation should avoid resource allocation as part of [it_post_rdma_read](#) to ensure that this
 8230 operation is non-blocking and thread safe. This operation can not fail due to insufficient resources. All
 8231 resource allocation required must be done at Endpoint creation time to ensure that all necessary resources are
 8232 available at post time.

8233 The Implementation should support zero-copy data transfers and kernel bypass for the RDMA Read
 8234 operation.

8235 **it_post_rdma_write**

8236 The Implementation should avoid resource allocation as part of [it_post_rdma_write](#) to ensure that this
 8237 operation is non-blocking and thread safe. This operation can not fail due to insufficient resources. All
 8238 resource allocation required must be done at Endpoint creation time to ensure that all necessary resources are
 8239 available at post time.

8240 The Implementation should support zero-copy data transfers and kernel bypass for the RDMA Write
 8241 operation.

8242 **it_post_recv**

8243 The Implementation should avoid resource allocation as part of [it_post_recv](#) to ensure that this operation is
 8244 non-blocking and thread safe. This operation can not fail due to insufficient resources. All resource allocation
 8245 required must be done at Endpoint creation time to ensure that all necessary resources are available at post
 8246 time.

- 8247 The Implementation should support zero-copy data transfers and kernel bypass for the Receive operation.
- 8248 **it_post_recvfrom**
- 8249 The Implementation should avoid resource allocation as part of *it_post_recvfrom* to ensure that this operation
8250 is non-blocking and thread safe. This operation can not fail due to insufficient resources. All resource
8251 allocation required must be done at Endpoint creation time to ensure that all necessary resources are available
8252 at post time.
- 8253 The Implementation should support zero-copy data transfers and kernel bypass for the ReceiveFrom
8254 operation.
- 8255 **it_post_send**
- 8256 The Implementation should avoid resource allocation as part of *it_post_send* to ensure that this operation is
8257 non-blocking and thread safe. This operation can not fail due to insufficient resources. All resource allocation
8258 required must be done at Endpoint creation time to ensure that all necessary resources are available at post
8259 time.
- 8260 The Implementation should support zero-copy data transfers and kernel bypass for the Send operation.
- 8261 **it_post_sendto**
- 8262 The Implementation should avoid resource allocation as part of *it_post_sendto* to ensure that this operation is
8263 non-blocking and thread safe. This operation can not fail due to insufficient resources. All resource allocation
8264 required must be done at Endpoint creation time to ensure that all necessary resources are available at post
8265 time.
- 8266 For details on handling IT_ERR_INVALID_AH under InfiniBand see compliance statement o10-2.1.1
8267 (IBTA 1.1 vol 1).
- 8268 The Implementation should support zero-copy data transfers and kernel bypass for the SendTo operation.
- 8269 **it_ud_service_request_handle_create**
- 8270 The Implementation only needs to validate the components of the *it_path_t* structure that refer to local
8271 entities, specifically *spigot_id*, *local_port_lid*, and *local_port_gid*.
- 8272 When running over the InfiniBand transport, if the Consumer provides a Path to
8273 [*it_ud_service_request_handle_create*](#) that contains a P_Key that is not in the HCA's P_Key table, the
8274 Implementation shall return IT_ERR_INVALID_SOURCE_PATH.
- 8275 **it_unaffiliated_event_t**
- 8276 Asynchronous Events should be copied from hardware resources into per-process software queues. The effect
8277 of overflow of the software queue should be isolated to the owning process. When overflow of the Affiliated
8278 Event EVD occurs, hardware resources should still be dequeued and discarded.
- 8279 In the case of Unaffiliated Events, the underlying Implementation should copy every Event into each process-
8280 specific queue.
- 8281

8282 B. Header Files

8283 B.1 it_api.h

```

8284
8285     #include "it_api_os_specific.h"
8286
8287     #define IN
8288     #define OUT
8289
8290     typedef enum {
8291         IT_SUCCESS = 0,
8292         IT_ERR_ABORT,
8293         IT_ERR_ACCESS,
8294         IT_ERR_ADDRESS,
8295         IT_ERR_AEVD_NOT_ALLOWED,
8296         IT_ERR_ASYNC_AFF_EVD_EXISTS,
8297         IT_ERR_ASYNC_UNAFF_EVD_EXISTS,
8298         IT_ERR_CANNOT_RESET,
8299         IT_ERR_CONN_QUAL_BUSY,
8300         IT_ERR_EP_TIMEWAIT,
8301         IT_ERR_EVD_BUSY,
8302         IT_ERR_EVD_QUEUE_FULL,
8303         IT_ERR_FAULT,
8304         IT_ERR_IA_CATASTROPHE,
8305         IT_ERR_INTERRUPT,
8306         IT_ERR_INVALID_ADDRESS,
8307         IT_ERR_INVALID_AEVD,
8308         IT_ERR_INVALID_AH,
8309         IT_ERR_INVALID_ETIMEOUT,
8310         IT_ERR_INVALID_CM_RETRY,
8311         IT_ERR_INVALID_CN_EST_FLAGS,
8312         IT_ERR_INVALID_CN_EST_ID,
8313         IT_ERR_INVALID_CONN_EVD,
8314         IT_ERR_INVALID_CONN_QUAL,
8315         IT_ERR_INVALID_CONVERSION,
8316         IT_ERR_INVALID_DTO_FLAGS,
8317         IT_ERR_INVALID_EP,
8318         IT_ERR_INVALID_EP_ATTR,
8319         IT_ERR_INVALID_EP_KEY,
8320         IT_ERR_INVALID_EP_STATE,
8321         IT_ERR_INVALID_EP_TYPE,
8322         IT_ERR_INVALID_EVD,
8323         IT_ERR_INVALID_EVD_STATE,
8324         IT_ERR_INVALID_EVD_TYPE,
8325         IT_ERR_INVALID_FLAGS,
8326         IT_ERR_INVALID_HANDLE,
8327         IT_ERR_INVALID_IA,

```

8328 IT_ERR_INVALID_LENGTH,
 8329 IT_ERR_INVALID_LISTEN,
 8330 IT_ERR_INVALID_LMR,
 8331 IT_ERR_INVALID_LTIMEOUT,
 8332 IT_ERR_INVALID_MAJOR_VERSION,
 8333 IT_ERR_INVALID_MASK,
 8334 IT_ERR_INVALID_MINOR_VERSION,
 8335 IT_ERR_INVALID_NAME,
 8336 IT_ERR_INVALID_NETADDR,
 8337 IT_ERR_INVALID_NUM_SEGMENTS,
 8338 IT_ERR_INVALID_PDATA_LENGTH,
 8339 IT_ERR_INVALID_PRIVS,
 8340 IT_ERR_INVALID_PZ,
 8341 IT_ERR_INVALID_QUEUE_SIZE,
 8342 IT_ERR_INVALID_RECV_EVD,
 8343 IT_ERR_INVALID_RECV_EVD_STATE,
 8344 IT_ERR_INVALID_REQ_EVD,
 8345 IT_ERR_INVALID_REQ_EVD_STATE,
 8346 IT_ERR_INVALID_RETRY,
 8347 IT_ERR_INVALID_RMR,
 8348 IT_ERR_INVALID_RNR_RETRY,
 8349 IT_ERR_INVALID_RTIMEOUT,
 8350 IT_ERR_INVALID_SGID,
 8351 IT_ERR_INVALID_SLID,
 8352 IT_ERR_INVALID_SOFT_EVD,
 8353 IT_ERR_INVALID_SOURCE_PATH,
 8354 IT_ERR_INVALID_SPIGOT,
 8355 IT_ERR_INVALID_THRESHOLD,
 8356 IT_ERR_INVALID_UD_STATUS,
 8357 IT_ERR_INVALID_UD_SVC,
 8358 IT_ERR_INVALID_UD_SVC_REQ_ID,
 8359 IT_ERR_LMR_BUSY,
 8360 IT_ERR_MISMATCH_FD,
 8361 IT_ERR_NO_CONTEXT,
 8362 IT_ERR_NO_PERMISSION,
 8363 IT_ERR_PAYLOAD_SIZE,
 8364 IT_ERR_PDATA_NOT_SUPPORTED,
 8365 IT_ERR_PZ_BUSY,
 8366 IT_ERR_QUEUE_EMPTY,
 8367 IT_ERR_RANGE,
 8368 IT_ERR_RESOURCES,
 8369 IT_ERR_RESOURCE_IRD,
 8370 IT_ERR_RESOURCE_LMR_LENGTH,
 8371 IT_ERR_RESOURCE_ORD,
 8372 IT_ERR_RESOURCE_QUEUE_SIZE,
 8373 IT_ERR_RESOURCE_RECV_DTO,
 8374 IT_ERR_RESOURCE_REQ_DTO,
 8375 IT_ERR_RESOURCE_RRSEG,
 8376 IT_ERR_RESOURCE_RSEG,
 8377 IT_ERR_RESOURCE_RWSEG,
 8378 IT_ERR_RESOURCE_SSEG,
 8379 IT_ERR_TIMEOUT_EXPIRED,
 8380 IT_ERR_TOO_MANY_POSTS,
 8381 IT_ERR_WAITER_LIMIT

```

8382     } it_status_t;
8383
8384     typedef uint32_t it_rmr_context_t;
8385
8386     #ifdef IT_32BIT
8387         typedef uint32_t it_length_t; /* a 32-bit platform */
8388
8389     #else
8390
8391         typedef uint64_t it_length_t; /* a 64-bit platform */
8392
8393     #endif
8394
8395     typedef enum {
8396         IT_PRIV_NONE           = 0x0001,
8397         IT_PRIV_READ_ONLY     = 0x0002,
8398         IT_PRIV_REMOTE_READ   = 0x0004,
8399         IT_PRIV_REMOTE_WRITE  = 0x0008,
8400         IT_PRIV_REMOTE        = 0x0010,
8401         IT_PRIV_ALL           = 0x0020,
8402         IT_PRIV_DEFAULT      = 0x0040
8403     } it_mem_priv_t;
8404
8405     typedef enum {
8406         IT_LMR_FLAG_NONE       = 0x0001,
8407         IT_LMR_FLAG_SHARED    = 0x0002,
8408         IT_LMR_FLAG_NONCOHERENT = 0x0004
8409     } it_lmr_flag_t;
8410
8411     typedef uint64_t it_ud_svc_req_identififier_t;
8412
8413     typedef uint64_t it_cn_est_identififier_t;
8414
8415     /* it_boolean_t.txt */
8416
8417     typedef enum {
8418         IT_FALSE = 0,
8419         IT_TRUE  = 1
8420     } it_boolean_t;
8421
8422     /* it_handle_t.txt */
8423
8424     typedef enum {
8425         IT_HANDLE_TYPE_ADDR,
8426         IT_HANDLE_TYPE_EP,
8427         IT_HANDLE_TYPE_EVD,
8428         IT_HANDLE_TYPE_IA,
8429         IT_HANDLE_TYPE_LISTEN,
8430         IT_HANDLE_TYPE_LMR,
8431         IT_HANDLE_TYPE_PZ,
8432         IT_HANDLE_TYPE_RMR,
8433         IT_HANDLE_TYPE_UD_SVC_REQ
8434     }
8435 
```



```

8436     } it_handle_type_enum_t;
8437
8438     typedef void *          it_handle_t;
8439     #define IT_NULL_HANDLE ((it_handle_t) NULL)
8440
8441     typedef struct it_addr_handle_s      * it_addr_handle_t;
8442     typedef struct it_ep_handle_s        * it_ep_handle_t;
8443     typedef struct it_evd_handle_s        * it_evd_handle_t;
8444     typedef struct it_ia_handle_s         * it_ia_handle_t;
8445     typedef struct it_listen_handle_s     * it_listen_handle_t;
8446     typedef struct it_lmr_handle_s        * it_lmr_handle_t;
8447     typedef struct it_pz_handle_s         * it_pz_handle_t;
8448     typedef struct it_rmr_handle_s        * it_rmr_handle_t;
8449     typedef struct it_ud_svc_req_handle_s * it_ud_svc_req_handle_t;
8450
8451
8452     /* it_conn_qual_t.txt */
8453
8454     /* Enumerates all the possible Connection Qualifier types
8455        supported by the API. */
8456     typedef enum {
8457
8458         /* IANA (TCP/UDP) Port Number */
8459         IT_IANA_PORT = 0x1,
8460
8461         /* InfiniBand Service ID, as described in section 12.7.3 of
8462            Volume 1 of the InfiniBand specification. */
8463         IT_IB_SERVICEID = 0x2,
8464
8465         /* VIA Connection Discriminator */
8466         IT_VIA_DISCRIMINATOR = 0x4
8467
8468     } it_conn_qual_type_t;
8469
8470     /* Defines the Connection Qualifier format for a VIA
8471        "connection discriminator".
8472        The API imposes a fixed upper bound on the discriminator size. */
8473
8474     #define IT_MAX_VIA_DISC_LEN      64
8475
8476     typedef struct {
8477
8478         /* The total number of bytes in the array below */
8479         /* that are significant */
8480         uint16_t          len;
8481
8482         /* VIA connection discriminator, which is an array of bytes */
8483         unsigned char     discriminator[IT_MAX_VIA_DISC_LEN];
8484
8485     } it_via_discriminator_t;
8486
8487     /* This defines the Connection Qualifier for InfiniBand,
8488        which is the 64-bit Service ID */
8489     typedef uint64_t      it_ib_serviceid_t;

```

```

8490
8491 /* This describes a Connection Qualifier suitable for input to
8492 several routines in the API. */
8493 typedef struct {
8494
8495     /* The discriminator for the union below. */
8496     it_conn_qual_type_t      type;
8497
8498     union {
8499
8500         /* IANA Port Number, in network byte order */
8501         uint16_t             port;
8502
8503         /* InfiniBand Service ID, in network byte order */
8504         it_ib_serviceid_t   serviceid;
8505
8506         /* VIA connection discriminator. */
8507         it_via_discriminator_t discriminator;
8508
8509     } conn_qual;
8510 } it_conn_qual_t;
8511
8512 /* it_context_t.txt */
8513
8514 typedef union {
8515     void *                ptr;
8516     uint64_t              index;
8517 } it_context_t;
8518
8519 /* it_dto_cookie_t.txt */
8520
8521 typedef uint64_t it_dto_cookie_t;
8522
8523 /* it_dto_status_t.txt */
8524
8525 typedef enum {
8526     IT_DTO_SUCCESS                = 0,
8527     IT_DTO_ERR_LOCAL_LENGTH       = 1,
8528     IT_DTO_ERR_LOCAL_EP          = 2,
8529     IT_DTO_ERR_LOCAL_PROTECTION   = 3,
8530     IT_DTO_ERR_FLUSHED           = 4,
8531     IT_RMR_OPERATION_FAILED       = 5,
8532     IT_DTO_ERR_BAD_RESPONSE       = 6,
8533     IT_DTO_ERR_REMOTE_ACCESS      = 7,
8534     IT_DTO_ERR_REMOTE_RESPONDER   = 8,
8535     IT_DTO_ERR_TRANSPORT          = 9,
8536     IT_DTO_ERR_RECEIVER_NOT_READY = 10,
8537     IT_DTO_ERR_PARTIAL_PACKET     = 11,
8538 } it_dto_status_t;
8539
8540 /* it_dto_flags_t.txt */
8541
8542
8543

```

```

8544
8545 typedef enum
8546 {
8547     /* If flag set, completion generates a local Event */
8548     IT_COMPLETION_FLAG          = 0x01,
8549
8550     /* If flag set, completion cause local Notification */
8551     IT_NOTIFY_FLAG             = 0x02,
8552
8553     /* If flag set, receipt of DTO at remote will cause
8554        Notification at remote */
8555     IT_SOLICITED_WAIT_FLAG     = 0x04,
8556
8557     /* If flag set, DTO processing will not start if
8558        previously posted RDMA Reads are not complete. */
8559     IT_BARRIER_FENCE_FLAG     = 0x08,
8560 } it_dto_flags_t;
8561
8562 /* it_net_addr_t.txt */
8563
8564 /* Enumerates all the possible Network Address types supported
8565    by the API. */
8566 typedef enum {
8567
8568     /* IPv4 address */
8569     IT_IPV4 = 0x1,
8570
8571     /* IPv6 address */
8572     IT_IPV6 = 0x2,
8573
8574     /* InfiniBand GID */
8575     IT_IB_GID = 0x3,
8576
8577     /* VIA Network Address */
8578     IT_VIA_HOSTADDR = 0x4
8579 } it_net_addr_type_t;
8580
8581
8582 /* Defines the Network Address format for a VIA "host address".
8583    The API has a fixed upper bound on the maximum sized VIA
8584    address it will support */
8585
8586 #define IT_MAX_VIA_ADDR_LEN      64
8587
8588 typedef struct {
8589
8590     /* The number of bytes in the array below that are significant */
8591     uint16_t          len;
8592
8593     /* VIA host address, which is an array of bytes */
8594     unsigned char     hostaddr[IT_MAX_VIA_ADDR_LEN];
8595
8596 } it_via_net_addr_t;
8597

```

```

8598     /* This defines the Network Address format for the InfiniBand
8599     GID, which is just an IPv6 address. */
8600     typedef struct in6_addr          it_ib_gid_t;
8601
8602     /* This describes a Network Address suitable for input to several
8603     routines in the API. */
8604     typedef struct {
8605
8606         /* The discriminator for the union below. */
8607         it_net_addr_type_t          addr_type;
8608
8609         union {
8610
8611             /* IPv4 address, in network byte order */
8612             struct in_addr          ipv4;
8613
8614             /* IPv6 address, in network byte order */
8615             struct in6_addr         ipv6;
8616
8617             /* InfiniBand GID, in network byte order */
8618             it_ib_gid_t            gid;
8619
8620             /* VIA Network Address. */
8621             it_via_net_addr_t       via;
8622
8623         } addr;
8624     } it_net_addr_t;
8625
8626
8627     /* it_ia_info_t.txt */
8628
8629     /* Enumerates all the transport types supported by the API. */
8630     typedef enum {
8631
8632         /* InfiniBand Native Transport */
8633         IT_IB_TRANSPORT = 1,
8634
8635         /* VIA host Interface using IP transport, supporting
8636         only the Reliable Delivery reliability level */
8637         IT_VIA_IP_TRANSPORT = 2,
8638
8639         /* VIA host Interface, using Fibre Channel transport, supporting
8640         only the Reliable Delivery reliability level*/
8641         IT_VIA_FC_TRANSPORT = 3,
8642
8643         /* Vendor-proprietary Transport */
8644         IT_VENDOR_TRANSPORT = 1000
8645     } it_transport_type_t;
8646
8647
8648     /* Transport Service Type definitions. */
8649     typedef enum {
8650
8651

```

```

8652
8653     /* Reliable Connected Transport Service Type */
8654     IT_RC_SERVICE = 0x1,
8655
8656     /* Unreliable Datagram Transport Service Type */
8657     IT_UD_SERVICE = 0x2,
8658
8659 } it_transport_service_type_t;
8660
8661
8662 /* The following structure describes an Interface Adapter Spigot */
8663 typedef struct {
8664
8665     /* Spigot identifier */
8666     size_t                spigot_id;
8667
8668     /* Maximum sized Send operation for the RC service on
8669     this Spigot. */
8670     size_t                max_rc_send_len;
8671
8672     /* Maximum sized RDMA Read/Write operation for the RC service on
8673     this Spigot. */
8674     size_t                max_rc_rdma_len;
8675
8676     /* Maximum sized Send operation for the UD service on
8677     this Spigot. */
8678     size_t                max_ud_send_len;
8679
8680     /* Indicates whether the Spigot is online or offline.  A IT_TRUE
8681     value means online. */
8682     it_boolean_t         spigot_online;
8683
8684     /* A mask indicating which Connection Qualifier types
8685     this IA supports for input to it_ep_connect and
8686     it_ud_service_request_handle_create. The bits in the
8687     mask are an inclusive OR of the values for Connection
8688     Qualifier types that this IA supports. */
8689     it_conn_qual_type_t  active_side_conn_qual;
8690
8691     /* A mask indicating which Connection Qualifier types this to
8692     it_listen_create. The bits in the mask are an inclusive OR
8693     of the values for Connection Qualifier types that this IA
8694     supports. */
8695     it_conn_qual_type_t  passive_side_conn_qual;
8696
8697
8698     /* The number of Network Addresses associated with Spigot */
8699     size_t                num_net_addr;
8700
8701     /* Pointer to array of Network Address addresses. */
8702     it_net_addr_t*       net_addr;
8703
8704 } it_spigot_info_t;
8705

```

```

8706      /* The following structure is used to identify the vendor associated
8707         with an IA that uses the IB transport*/
8708      typedef struct {
8709
8710         /* The NodeInfo:VendorID as described in chapter 14
8711            of the IB spec. */
8712         uint32_t          vendor : 24;
8713
8714         /* The NodeInfo:DeviceID as described in chapter 14
8715            of the IB spec. */
8716         uint16_t         device;
8717
8718         /* The NodeInfo:Revision as described in chapter 14
8719            of the IB spec. */
8720         uint32_t         revision;
8721     } it_vendor_ib_t;
8722
8723     /* The following structure is used to identify the vendor
8724        associated with an IA that uses a VIA transport*/
8725     typedef struct {
8726         /* The "Name" member of the VIP_NIC_ATTRIBUTES structure,
8727            as described in the VIA spec. */
8728         char              name[64];
8729
8730         /* The "HardwareVersion" member of the VIP_NIC_ATTRIBUTES
8731            structure, as described in the VIA spec. */
8732         unsigned long     hardware;
8733
8734         /* The "ProviderVersion" member of the VIP_NIC_ATTRIBUTES
8735            structure, as described in the VIA spec. */
8736         unsigned long     provider;
8737     } it_vendor_via_t;
8738
8739     /* The following structure is returned by the it_ia_query function */
8740     typedef struct {
8741
8742         /* Interface Adapter name, as specified in it_ia_create */
8743         char*             ia_name;
8744
8745         /* The major version number of the latest version of the
8746            IT-API that this IA supports. */
8747         uint32_t          api_major_version;
8748
8749         /* The minor version number of the latest version of the
8750            IT-API that this IA supports. */
8751         uint32_t          api_minor_version;
8752
8753         /* The major version number for the software being used to
8754            control this IA. The IT-API imposes no structure whatsoever
8755            on this number; its meaning is completely IA-dependent. */
8756         uint32_t          sw_major_version;
8757
8758         /* The minor version number for the software being used to
8759            control this IA. The IT-API imposes no structure whatsoever

```

```

8760         on this number; its meaning is completely IA-dependent. */
8761 uint32_t             sw_minor_version;
8762
8763 /* The vendor associated with the IA. This information is useful
8764    if the Consumer wishes to do device-specific programming. This
8765    union is discriminated by transport_type. No vendor
8766    identification is provided for transports not listed below. */
8767 union {
8768
8769     /* Used if transport_type is IT_IB_TRANSPORT */
8770     it_vendor_ib_t         ib;
8771
8772     /* Used if transport_type is IT_VIA_IP_TRANSPORT or
8773        IT_VIA_FC_TRANSPORT */
8774     it_vendor_via_t       via;
8775
8776 } vendor;
8777
8778 /* The Interface Adapter and platform provide a data alignment hint
8779    to the Consumer to Help the Consumer align their data transfer
8780    buffers in a way the is optimal for the performance of the IA.
8781    For example, if the best throughput is obtained by aligning
8782    buffers to 128-byte boundaries, dto_alignment_hint will have the
8783    value 128. The Consumer may choose to ignore the alignment hint
8784    without any adverse functional impact. (There may be an adverse
8785    performance impact.) */
8786 uint32_t             dto_alignment_hint;
8787
8788 /* The transport type (e.g. InfiniBand) supported by Interface
8789    Adapter. An Interface Adapter supports precisely one transport
8790    type. */
8791 it_transport_type_t   transport_type;
8792
8793 /* The Transport Service Types supported by this IA. This is
8794    constructed by doing an inclusive OR of the Transport Service
8795    Type values.*/
8796 it_transport_service_type_t supported_service_types;
8797
8798 /* Indicates whether work queues are resizable */
8799 it_boolean_t         resizable_work_queue;
8800
8801 /* Indicates whether the underlying transport used by this IA uses
8802    a three-way handshake for doing Connection establishment. Note
8803    that if the underlying transport supports a three-way handshake
8804    the Consumer can choose whether to use two handshakes or three
8805    when establishing the Connection. If the underlying transport
8806    supports a two-way handshake for establishing a Connection, the
8807    Consumer can only use two handshakes when establishing the
8808    Connection. */
8809 it_boolean_t         three_way_handshake_support;
8810
8811 /* Indicates whether Private Data is supported on Connection
8812    establishment or UD service resolution operations. */
8813 it_boolean_t         private_data_support;

```

```

8814
8815 /* Indicates whether the max_message_size field in the
8816 IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
8817 it_boolean_t          max_message_size_support;
8818
8819 /* Indicates whether the rdma_read_inflight_incoming field
8820 in the IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
8821 it_boolean_t          ird_support;
8822
8823 /* Indicates whether the rdma_read_inflight_outgoing field
8824 in the IT_CM_REQ_CONN_REQUEST_EVENT is valid for this IA. */
8825 it_boolean_t          ord_support;
8826
8827 /* Indicates whether the IA generates IT_ASYNC_UNAFF_SPIGOT_ONLINE
8828 Events. See it_unaffiliated_event_t for details. */
8829 it_boolean_t          spigot_online_support;
8830
8831 /* Indicates whether the IA generates IT_ASYNC_UNAFF_SPIGOT_OFFLINE
8832 Events. See it_unaffiliated_event_t for details. */
8833 it_boolean_t          spigot_offline_support;
8834
8835 /* The maximum number of bytes of Private Data supported for the
8836 it_ep_connect routine. This will be less than or equal to
8837 IT_MAX_PRIV_DATA. */
8838 size_t                connect_private_data_len;
8839
8840 /* The maximum number of bytes of Private Data supported for the
8841 it_ep_accept routine. This will be less than or equal to
8842 IT_MAX_PRIV_DATA. */
8843 size_t                accept_private_data_len;
8844
8845 /* The maximum number of bytes of Private Data supported for the
8846 it_reject routine. This will be less than or equal to
8847 IT_MAX_PRIV_DATA. */
8848 size_t                reject_private_data_len;
8849
8850 /* The maximum number of bytes of Private Data supported for the
8851 it_ep_disconnect routine. This will be less than or equal to
8852 IT_MAX_PRIV_DATA. */
8853 size_t                disconnect_private_data_len;
8854
8855 /* The maximum number of bytes of Private Data supported for the
8856 it_ud_service_request_handle_create routine. This will be less
8857 than or equal to IT_MAX_PRIV_DATA. */
8858 size_t                ud_req_private_data_len;
8859
8860 /* The maximum number of bytes of Private Data supported for the
8861 it_ud_service_reply routine. This will be less than or equal to
8862 IT_MAX_PRIV_DATA. */
8863 size_t                ud_rep_private_data_len;
8864
8865 /* Specifies the number of Spigots associated with this Interface
8866 Adapter */
8867 size_t                num_spigots;

```



```

8868
8869     /* An array of Spigot information data structures.  The array
8870        contains num_spigots elements. */
8871     it_spigot_info_t*      spigot_info;
8872
8873     /* The Handle for the EVD that contains the affiliated async Event
8874        Stream.  If no EVD contains the Affiliated Async Event Stream,
8875        this member will have the distinguished value IT_NULL_HANDLE */
8876     it_evd_handle_t       affiliated_err_evd;
8877
8878     /* The Handle for the EVD that contains the unaffiliated async Event
8879        Stream.  If no EVD contains the Unaffiliated Async Event Stream,
8880        this member will have the distinguished value IT_NULL_HANDLE */
8881     it_evd_handle_t       unaffiliated_err_evd;
8882
8883 } it_ia_info_t;
8884
8885
8886 /* it_lmr_triplet_t.txt */
8887
8888 typedef struct {
8889     it_lmr_handle_t      lmr;
8890     void                 *addr;
8891     it_length_t          length;
8892 } it_lmr_triplet_t;
8893
8894 /* it_path_t.txt */
8895
8896 /* This is the remote component of the Path information for the
8897     InfiniBand transport */
8898 typedef struct {
8899
8900     /* Partition Key, as defined in the REQ message for the IB
8901        CM protocol */
8902     uint16_t              partition_key;
8903
8904     /* Path Packet Payload MTU, as defined in the REQ message
8905        for the IB CM protocol */
8906     uint8_t                path_mtu : 4;
8907
8908     /* PacketLifeTime, as defined in the PathRecord in IB
8909        specification.  This field is useful for Consumers that
8910        wish to use timeout values other than the default ones
8911        for doing Connection establishment. */
8912     uint8_t                packet_lifetime : 6;
8913
8914     /* Local Port LID, as defined in the REQ message for the IB
8915        CM protocol.  The low order bits of this value also
8916        constitute the "Source Path Bits" that are used to
8917        create an Address Handle. */
8918     uint16_t               local_port_lid;
8919
8920     /* Remote Port LID, as defined in the REQ message for the
8921        IB CM protocol.  This is also the "Destination LID" used

```

```

8922         to create an Address Handle. */
8923         uint16_t             remote_port_lid;
8924
8925         /* Local Port GID in network byte order, as defined in the
8926            REQ message for the IB CM protocol. This is also used to
8927            determine the appropriate "Source GID Index" to be used
8928            when creating an Address Handle. */
8929         it_ib_gid_t         local_port_gid;
8930
8931         /* Remote Port GID in network byte order, as defined in the
8932            REQ message for the IB CM protocol. This is also the
8933            "Destination GID or MGID" used to create an Address
8934            Handle. */
8935         it_ib_gid_t         remote_port_gid;
8936
8937         /* Packet Rate, as defined in the REQ message for the IB CM
8938            protocol. This is also the "Maximum Static Rate" to be
8939            used when creating an Address Handle. */
8940         uint8_t             packet_rate : 6;
8941
8942         /* SL, as defined in the REQ message for the IB CM
8943            protocol. This is also the "Service Level" to be used
8944            when creating an Address Handle. */
8945         uint8_t             sl : 4;
8946
8947         /* Subnet Local, as defined in the REQ message for the IB
8948            CM protocol. When creating an Address Handle, setting
8949            this bit causes a GRH to be included as part of any
8950            Unreliable Datagram sent using the Address Handle. */
8951         uint8_t             subnet_local : 1;
8952
8953         /* Flow Label, as defined in the REQ message for the IB CM
8954            protocol. This is also the "Flow Label" to be used when
8955            creating an Address Handle. This is only valid if
8956            subnet_local is clear. */
8957         uint32_t            flow_label : 20;
8958
8959         /* Traffic Class, as defined in the REQ message for the IB
8960            CM protocol. This is also the "Traffic Class" to be
8961            used when creating an Address Handle. This is only
8962            valid if subnet_local is clear. */
8963         uint8_t             traffic_class;
8964
8965         /* Hop Limit, as defined in the REQ message for the IB CM
8966            protocol. This is also the "Hop Limit" to be used when
8967            creating an Address Handle. This is only valid if
8968            subnet_local is clear. */
8969         uint8_t             hop_limit;
8970
8971     } it_ib_net_endpoint_t;
8972
8973     /* This is the remote component of the Path information for the
8974        VIA transport */
8975     typedef it_via_net_addr_t it_via_net_endpoint_t;

```

```

8976
8977 /* This is the Path data structure used by several routines in
8978 the API */
8979 typedef struct {
8980
8981     /* Identifier for the Spigot to be used on the local IA
8982     Note that this data structure is always used in a
8983     Context where the IA associated with the Spigot can be
8984     deduced. */
8985     size_t                spigot_id;
8986
8987     /* The transport-independent timeout parameter for how long
8988     to wait, in microseconds, before timing out a Connection
8989     establishment attempt using this Path. The timeout
8990     period for establishing a Connection
8991     can only be specified on the Active side; the timeout
8992     period can not be changed on the Passive side. */
8993     uint64_t             timeout;
8994
8995     /* The remote component of the Path */
8996     union {
8997
8998         /* For use with InfiniBand */
8999         it_ib_net_endpoint_t    ib;
9000
9001         /* For use with VIA */
9002         it_via_net_endpoint_t    via;
9003
9004     } remote;
9005
9006 } it_path_t;
9007
9008
9009 /* it_ep_attributes_t.txt */
9010
9011 typedef uint32_t    it_ud_ep_id_t;
9012 typedef uint32_t    it_ud_ep_key_t;
9013
9014 typedef enum {
9015     IT_EP_PARAM_ALL                = 0x00000001,
9016     IT_EP_PARAM_IA                 = 0x00000002,
9017     IT_EP_PARAM_SPIGOT             = 0x00000004,
9018     IT_EP_PARAM_STATE              = 0x00000008,
9019     IT_EP_PARAM_SERV_TYPE          = 0x00000010,
9020     IT_EP_PARAM_PATH               = 0x00000020,
9021     IT_EP_PARAM_PZ                 = 0x00000040,
9022     IT_EP_PARAM_REQ_SEVD           = 0x00000080,
9023     IT_EP_PARAM_RECV_SEVD         = 0x00000100,
9024     IT_EP_PARAM_CONN_SEVD         = 0x00000200,
9025     IT_EP_PARAM_RDMA_RD_ENABLE    = 0x00000400,
9026     IT_EP_PARAM_RDMA_WR_ENABLE    = 0x00000800,
9027     IT_EP_PARAM_MAX_RDMA_READ_SEG = 0x00001000,
9028     IT_EP_PARAM_MAX_RDMA_WRITE_SEG = 0x00002000,
9029     IT_EP_PARAM_MAX_IRD           = 0x00004000,

```

```

9030         IT_EP_PARAM_MAX_ORD           = 0x00008000,
9031         IT_EP_PARAM_EP_ID             = 0x00010000,
9032         IT_EP_PARAM_EP_KEY            = 0x00020000,
9033         IT_EP_PARAM_MAX_PAYLOAD       = 0x00040000,
9034         IT_EP_PARAM_MAX_REQ_DTO       = 0x00080000,
9035         IT_EP_PARAM_MAX_RECV_DTO      = 0x00100000,
9036         IT_EP_PARAM_MAX_SEND_SEG      = 0x00200000,
9037         IT_EP_PARAM_MAX_RECV_SEG      = 0x00400000
9038     } it_ep_param_mask_t;
9039
9040     /*
9041     * the it_ep_param_mask_t value in the comment beside or
9042     * following each attribute is the mask value used to select
9043     * the attribute in the it_ep_query and it_ep_modify calls
9044     */
9045     typedef struct {
9046         it_boolean_t    rdma_read_enable;
9047         /* IT_EP_PARAM_RDMA_RD_ENABLE */
9048         it_boolean_t    rdma_write_enable;
9049         /* IT_EP_PARAM_RDMA_WR_ENABLE */
9050         size_t          max_rdma_read_segments;
9051         /* IT_EP_PARAM_MAX_RDMA_READ_SEG */
9052         size_t          max_rdma_write_segments;
9053         /* IT_EP_PARAM_MAX_RDMA_WRITE_SEG */
9054         uint32_t        rdma_read_inflight_incoming;
9055         /* IT_EP_PARAM_MAX_IRD */
9056         uint32_t        rdma_read_inflight_outgoing;
9057         /* IT_EP_PARAM_MAX_ORD */
9058     } it_rc_only_attributes_t;
9059
9060     typedef struct {
9061         it_ud_ep_id_t    ud_ep_id;    /* IT_EP_PARAM_EP_ID */
9062         it_ud_ep_key_t   ud_ep_key;   /* IT_EP_PARAM_EP_KEY */
9063     } it_remote_ep_info_t;
9064
9065     typedef struct {
9066         it_remote_ep_info_t    ep_info;
9067     } it_ud_only_attributes_t;
9068
9069     typedef union {
9070         it_rc_only_attributes_t    rc;
9071         it_ud_only_attributes_t    ud;
9072     } it_service_attributes_t;
9073
9074     typedef struct {
9075         size_t    max_dto_payload_size;    /* IT_EP_PARAM_MAX_PAYLOAD */
9076         size_t    max_request_dtos;        /* IT_EP_PARAM_MAX_REQ_DTO */
9077         size_t    max_recv_dtos;          /* IT_EP_PARAM_MAX_RECV_DTO */
9078         size_t    max_send_segments;      /* IT_EP_PARAM_MAX_SEND_SEG */
9079         size_t    max_recv_segments;      /* IT_EP_PARAM_MAX_RECV_SEG */
9080     } it_service_attributes_t    srv;
9081
9082     } it_ep_attributes_t;
9083

```

```

9084
9085
9086      /* it_event_t.txt */
9087
9088      #define IT_EVENT_STREAM_MASK      0xff000
9089      #define IT_TIMEOUT_INFINITE      ((uint64_t)(-1))
9090
9091      typedef enum
9092      {
9093          /* DTO Completion Event Stream */
9094          IT_DTO_EVENT_STREAM          = 0x00000,
9095          IT_DTO_SEND_CMPL_EVENT       = 0x00001,
9096          IT_DTO_RC_RECV_CMPL_EVENT    = 0x00002,
9097          IT_DTO_UD_RECV_CMPL_EVENT    = 0x00003,
9098          IT_DTO_RDMA_WRITE_CMPL_EVENT = 0x00004,
9099          IT_DTO_RDMA_READ_CMPL_EVENT  = 0x00005,
9100          IT_RMR_BIND_CMPL_EVENT       = 0x00006,
9101
9102          /*
9103           * Communication Management Request Event Stream
9104           */
9105          IT_CM_REQ_EVENT_STREAM        = 0x01000,
9106          IT_CM_REQ_CONN_REQUEST_EVENT  = 0x01001,
9107          IT_CM_REQ_UD_SERVICE_REQUEST_EVENT = 0x01002,
9108
9109          /*
9110           * Communication Management Message Event Stream
9111           */
9112          IT_CM_MSG_EVENT_STREAM        = 0x02000,
9113          IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT = 0x02001,
9114          IT_CM_MSG_CONN_ESTABLISHED_EVENT = 0x02002,
9115          IT_CM_MSG_CONN_DISCONNECT_EVENT  = 0x02003,
9116          IT_CM_MSG_CONN_PEER_REJECT_EVENT = 0x02004,
9117          IT_CM_MSG_CONN_NONPEER_REJECT_EVENT = 0x02005,
9118          IT_CM_MSG_CONN_BROKEN_EVENT      = 0x02006,
9119          IT_CM_MSG_UD_SERVICE_REPLY_EVENT  = 0x02007,
9120
9121          /* Asynchronous Affiliated Event Stream */
9122          IT_ASYNC_AFF_EVENT_STREAM      = 0x04000,
9123          IT_ASYNC_AFF_SEVD_ENQUEUE_FAILURE = 0x04001,
9124          IT_ASYNC_AFF_EP_FAILURE        = 0x04002,
9125          IT_ASYNC_AFF_EP_BAD_TRANSPORT_OPCODE = 0x04003,
9126          IT_ASYNC_AFF_EP_LOCAL_ACCESS_VIOLATION = 0x04004,
9127          IT_ASYNC_AFF_EP_REQ_DROPPED    = 0x04005,
9128          IT_ASYNC_AFF_EP_RDMAW_ACCESS_VIOLATION = 0x04006,
9129          IT_ASYNC_AFF_EP_RDMAW_CORRUPT_DATA = 0x04007,
9130          IT_ASYNC_AFF_EP_RDMAR_ACCESS_VIOLATION = 0x04008,
9131
9132          /* Asynchronous Non-Affiliated Event Stream */
9133          IT_ASYNC_UNAFF_EVENT_STREAM    = 0x08000,
9134          IT_ASYNC_UNAFF_IA_CATASTROPHIC_ERROR = 0x08001,
9135          IT_ASYNC_UNAFF_SPIGOT_ONLINE    = 0x08002,
9136          IT_ASYNC_UNAFF_SPIGOT_OFFLINE   = 0x08003,
9137          IT_ASYNC_UNAFF_SEVD_ENQUEUE_FAILURE = 0x08004,

```

```

9138
9139     /* Software Event Stream */
9140     IT_SOFTWARE_EVENT_STREAM           = 0x10000,
9141     IT_SOFTWARE_EVENT                 = 0x10001,
9142
9143     /* AEVD Notification Event Stream */
9144     IT_AEVD_NOTIFICATION_EVENT_STREAM = 0x20000,
9145     IT_AEVD_NOTIFICATION_EVENT       = 0x20001
9146 } it_event_type_t;
9147
9148
9149 /* it_aevd_notification_event_t.txt */
9150
9151 typedef struct {
9152     it_event_type_t    event_number;
9153     it_evd_handle_t    aevd;
9154
9155     it_evd_handle_t    sevd;
9156 } it_aevd_notification_event_t;
9157
9158 /* it_affiliated_event_t.txt */
9159
9160 typedef struct {
9161     it_event_type_t    event_number;
9162     it_evd_handle_t    evd;
9163
9164     union {
9165         it_evd_handle_t    sevd;
9166         it_ep_handle_t     ep;
9167     } u;
9168 } it_affiliated_event_t;
9169
9170 /* it_cm_msg_events.txt */
9171
9172 #define IT_MAX_PRIV_DATA 256
9173
9174 typedef enum {
9175     IT_CN_REJ_OTHER           = 0,
9176     IT_CN_REJ_TIMEOUT        = 1,
9177     IT_CN_REJ_BAD_PATH       = 2,
9178     IT_CN_REJ_STALE_CONN     = 3,
9179     IT_CN_REJ_BAD_ORD        = 4,
9180     IT_CN_REJ_RESOURCES      = 5
9181 } it_conn_reject_code_t;
9182
9183 typedef struct {
9184     it_event_type_t          event_number;
9185     it_evd_handle_t          evd;
9186     it_cn_est_identifier_t   cn_est_id;
9187     it_ep_handle_t           ep;
9188     uint32_t                 rdma_read_inflight_incoming;
9189     uint32_t                 rdma_read_inflight_outgoing;
9190     it_path_t                dst_path;
9191     it_conn_reject_code_t    reject_reason_code;

```

```

9192         unsigned char          private_data[IT_MAX_PRIV_DATA];
9193         it_boolean_t           private_data_present;
9194     } it_connection_event_t;
9195
9196     typedef enum {
9197         IT_UD_SVC_EP_INFO_VALID           = 0,
9198         IT_UD_SVC_ID_NOT_SUPPORTED       = 1,
9199         IT_UD_SVC_REQ_REJECTED           = 2,
9200         IT_UD_NO_EP_AVAILABLE           = 3,
9201         IT_UD_REQ_REDIRECTED             = 4
9202     } it_ud_svc_req_status_t;
9203
9204     typedef struct {
9205         it_event_type_t             event_number;
9206         it_evd_handle_t             evd;
9207         it_ud_svc_req_handle_t      ud_svc;
9208         it_ud_svc_req_status_t      status;
9209         it_remote_ep_info_t         ep_info;
9210         it_path_t                   dst_path;
9211         unsigned char               private_data[IT_MAX_PRIV_DATA];
9212         it_boolean_t                 private_data_present;
9213     } it_ud_svc_reply_event_t;
9214
9215     /* it_cm_req_events.txt */
9216
9217     typedef struct {
9218         it_event_type_t             event_number;
9219         it_evd_handle_t             evd;
9220         it_cn_est_identifier_t       cn_est_id;
9221         it_conn_qual_t              conn_qual;
9222         it_net_addr_t               source_addr;
9223         size_t                       spigot_id;
9224         uint32_t                     max_message_size;
9225         uint32_t                     rdma_read_inflight_incoming;
9226         uint32_t                     rdma_read_inflight_outgoing;
9227         unsigned char               private_data[IT_MAX_PRIV_DATA];
9228         it_boolean_t                 private_data_present;
9229     } it_conn_request_event_t;
9230
9231     typedef struct {
9232         it_event_type_t             event_number;
9233         it_evd_handle_t             evd;
9234         it_ud_svc_req_identifier_t   ud_svc_req_id;
9235         it_conn_qual_t              conn_qual;
9236         it_net_addr_t               source_addr;
9237         size_t                       spigot_id;
9238         unsigned char               private_data[IT_MAX_PRIV_DATA];
9239         it_boolean_t                 private_data_present;
9240     } it_ud_svc_request_event_t;
9241
9242
9243     /* it_dto_events.txt */
9244
9245     typedef enum {

```

```

9246         IB_UD_IB_GRH_PRESENT = 0x01
9247     } it_dto_ud_flags_t;
9248
9249     typedef struct {
9250         it_event_type_t      event_number;
9251         it_evd_handle_t      evd;
9252         it_ep_handle_t       ep;
9253         it_dto_cookie_t      cookie;
9254         it_dto_status_t      dto_status;
9255         uint32_t             transferred_length;
9256     } it_dto_cmpl_event_t;
9257
9258     typedef struct {
9259         it_event_type_t      event_number;
9260         it_evd_handle_t      evd;
9261         it_ep_handle_t       ep;
9262         it_dto_cookie_t      cookie;
9263         it_dto_status_t      dto_status;
9264         uint32_t             transferred_length;
9265         it_dto_ud_flags_t    flags;
9266         it_ud_ep_id_t        ud_ep_id;
9267         it_path_t            src_path;
9268     } it_all_dto_cmpl_event_t;
9269
9270     /* it_software_event_t.txt */
9271
9272     typedef struct {
9273         it_event_type_t      event_number;
9274         it_evd_handle_t      evd;
9275         void                  * data;
9276     } it_software_event_t;
9277
9278     /* it_unaffiliated_event_t.txt */
9279
9280     typedef struct {
9281         it_event_type_t      event_number;
9282         it_evd_handle_t      evd;
9283         it_ia_handle_t       ia;
9284         size_t               spigot_id;
9285     } it_unaffiliated_event_t;
9286
9287     /* it_event_t.txt */
9288
9289     typedef struct {
9290         it_event_type_t      event_number;
9291         it_evd_handle_t      evd;
9292     } it_any_event_t;
9293
9294     typedef union
9295     {
9296         /*
9297         * The following two union elements are
9298         * available for programming convenience.
9299         */

```



```

9300     * The event_number may be used to determine the
9301     * it_event_type_t of any event. it_any_event_t
9302     * allows the evd to be determined as well.
9303     */
9304     it_event_type_t          event_number;
9305     it_any_event_t          any;
9306
9307     /*
9308     * The remaining union elements correspond to
9309     * the various it_event_type_t types.
9310     */
9311
9312     /*
9313     * The following two Event structures
9314     * support the IT_DTO_EVENT_STREAM Event Stream.
9315     *
9316     * it_dto_cmpl_event_t supports
9317     * only the following events:
9318     *     IT_DTO_SEND_CMPL_EVENT
9319     *     IT_DTO_RC_RECV_CMPL_EVENT
9320     *     IT_DTO_RDMA_WRITE_CMPL_EVENT
9321     *     IT_DTO_RDMA_READ_CMPL_EVENT
9322     *     IT_RMR_BIND_CMPL_EVENT
9323     *
9324     * it_all_dto_cmpl_event_t supports all
9325     * possible DTO and RMR events:
9326     *     IT_DTO_SEND_CMPL_EVENT
9327     *     IT_DTO_RC_RECV_CMPL_EVENT
9328     *     IT_DTO_UD_RECV_CMPL_EVENT
9329     *     IT_DTO_RDMA_WRITE_CMPL_EVENT
9330     *     IT_DTO_RDMA_READ_CMPL_EVENT
9331     *     IT_RMR_BIND_CMPL_EVENT
9332     */
9333     it_dto_cmpl_event_t          dto_cmpl;
9334     it_all_dto_cmpl_event_t     all_dto_cmpl;
9335
9336     /*
9337     * The following two Event structures
9338     * support the IT_CM_REQ_EVENT_STREAM Event
9339     * stream:
9340     *
9341     * it_conn_request_event_t supports:
9342     *     IT_CM_REQ_CONN_REQUEST_EVENT
9343     *
9344     * it_ud_svc_request_event_t supports:
9345     *     IT_CM_REQ_UD_SERVICE_REQUEST_EVENT
9346     */
9347     it_conn_request_event_t     conn_req;
9348     it_ud_svc_request_event_t   ud_svc_request;
9349
9350     /*
9351     * The following two Event structures
9352     * support the IT_CM_MSG_EVENT_STREAM Event
9353     * stream:

```

```

9354      *
9355      * it_connection_event_t supports:
9356      *     IT_CM_MSG_CONN_ACCEPT_ARRIVAL_EVENT
9357      *     IT_CM_MSG_CONN_ESTABLISHED_EVENT
9358      *     IT_CM_MSG_CONN_PEER_REJECT_EVENT
9359      *     IT_CM_MSG_CONN_NONPEER_REJECT_EVENT
9360      *     IT_CM_MSG_CONN_DISCONNECT_EVENT
9361      *     IT_CM_MSG_CONN_BROKEN_EVENT
9362      *
9363      * it_ud_svc_reply_event_t supports:
9364      *     IT_CM_MSG_UD_SERVICE_REPLY_EVENT
9365      */
9366      it_connection_event_t      conn;
9367      it_ud_svc_reply_event_t    ud_svc_reply;
9368
9369      /*
9370      * it_affiliated_event_t supports
9371      * the following Event Stream:
9372      *     IT_ASYNC_AFF_EVENT_STREAM
9373      */
9374      it_affiliated_event_t      aff_async;
9375
9376      /*
9377      * it_unaffiliated_event_t supports
9378      * the following Event Stream:
9379      *     IT_ASYNC_UNAFF_EVENT_STREAM
9380      */
9381      it_unaffiliated_event_t    unaff_async;
9382
9383      /*
9384      * it_software_event_t supports
9385      * the following Event Stream:
9386      *     IT_SOFTWARE_EVENT_STREAM
9387      */
9388      it_software_event_t        sw;
9389
9390      /*
9391      * it_aevd_notification_event_t supports
9392      * the following Event Stream:
9393      *     IT_AEVD_NOTIFICATION_EVENT_STREAM
9394      */
9395      it_aevd_notification_event_t    aevd_notify;
9396      } it_event_t;
9397
9398
9399      /* it_ep_state_t.txt */
9400      typedef enum
9401      {
9402          IT_EP_STATE_UNCONNECTED                = 0,
9403          IT_EP_STATE_ACTIVE1_CONNECTION_PENDING = 1,
9404          IT_EP_STATE_ACTIVE2_CONNECTION_PENDING = 2,
9405          IT_EP_STATE_PASSIVE_CONNECTION_PENDING = 3,
9406          IT_EP_STATE_CONNECTED                  = 4,
9407          IT_EP_STATE_NONOPERATIONAL             = 5

```

```

9408     } it_ep_state_rc_t;
9409
9410     typedef enum
9411     {
9412         IT_EP_STATE_UD_NONOPERATIONAL    = 0,
9413         IT_EP_STATE_UD_OPERATIONAL      = 1
9414     } it_ep_state_ud_t;
9415
9416     typedef union
9417     {
9418         it_ep_state_rc_t  rc;
9419         it_ep_state_ud_t  ud;
9420     } it_ep_state_t;
9421
9422
9423     /* it_dg_remote_ep_addr_t.txt */
9424
9425     typedef struct
9426     {
9427         it_addr_handle_t      addr;
9428         it_remote_ep_info_t   ep_info;
9429     } it_ib_ud_addr_t;
9430
9431     typedef enum
9432     {
9433         IT_DG_TYPE_IB_UD
9434     } it_dg_type_t;
9435
9436     typedef struct
9437     {
9438         it_dg_type_t          type; /* IT_DG_TYPE_IB_UD */
9439         union {
9440             it_ib_ud_addr_t   ud;
9441         } addr;
9442     } it_dg_remote_ep_addr_t;
9443
9444     typedef enum {
9445         IT_AH_PATH_COMPLETE = 0x1
9446     } it_ah_flags_t;
9447
9448     typedef enum {
9449         IT_ADDR_PARAM_ALL      = 0x0001,
9450         IT_ADDR_PARAM_IA      = 0x0002,
9451         IT_ADDR_PARAM_PZ      = 0x0004,
9452         IT_ADDR_PARAM_PATH    = 0x0008
9453     } it_addr_param_mask_t;
9454
9455     typedef struct {
9456         it_ia_handle_t          ia;          /* IT_ADDR_PARAM_IA */
9457         it_pz_handle_t          pz;          /* IT_ADDR_PARAM_PZ */
9458         it_path_t               path;        /* IT_ADDR_PARAM_PATH */
9459     } it_addr_param_t;
9460
9461     typedef struct {

```

```

9462
9463     /* Remote CM Response Timeout, as defined in the REQ
9464        message for the IB CM protocol */
9465     uint8_t             remote_cm_timeout : 5;
9466
9467     /* Local CM Response Timeout, as defined in the REQ
9468        message for the IB CM protocol */
9469     uint8_t             local_cm_timeout : 5;
9470
9471     /* Retry Count, as defined in the REQ message for the
9472        IB CM protocol */
9473     uint8_t             retry_count : 3;
9474
9475     /* RNR Retry Count, as defined in the REQ message for
9476        the IB CM protocol */
9477     uint8_t             rnr_retry_count : 3;
9478
9479     /* Max CM retries, as defined in the REQ message for
9480        the IB CM protocol */
9481     uint8_t             max_cm_retries : 4;
9482
9483     /* Local ACK Timeout, as defined in the REQ message
9484        for the IB CM protocol */
9485     uint8_t             local_ack_timeout : 5;
9486
9487 } it_ib_conn_attributes_t;
9488
9489 typedef struct {
9490
9491     /* VIA currently has no transport-specific connection
9492        attributes */
9493
9494 } it_via_conn_attributes_t;
9495
9496 typedef union {
9497     it_ib_conn_attributes_t     ib;
9498     it_via_conn_attributes_t    via;
9499 } it_conn_attributes_t;
9500
9501 typedef enum {
9502     IT_CONNECT_FLAG_TWO_WAY     = 0x0001,
9503     IT_CONNECT_FLAG_THREE_WAY  = 0x0002
9504 } it_cn_est_flags_t;
9505
9506 typedef struct {
9507     it_ia_handle_t             ia;           /* IT_EP_PARAM_IA */
9508     size_t                     spigot_id;  /* IT_EP_PARAM_SPIGOT */
9509     it_ep_state_t              ep_state;    /* IT_EP_PARAM_STATE */
9510     it_transport_service_type_t service_type; /* IT_EP_PARAM_SERV_TYPE */
9511     it_path_t                  dst_path;    /* IT_EP_PARAM_PATH */
9512     it_pz_handle_t             pz;         /* IT_EP_PARAM_PZ */
9513     it_evd_handle_t            request_sevd; /* IT_EP_PARAM_REQ_SEVD */
9514     it_evd_handle_t            recv_sevd;  /* IT_EP_PARAM_RECV_SEVD */

```

```

9516         it_evd_handle_t      connect_sevd; /* IT_EP_PARAM_CONN_SEVD */
9517         it_ep_attributes_t    attr;          /* see it_ep_attributes_t
9518                                         for mask flags for attr */
9519     } it_ep_param_t;
9520
9521     typedef enum {
9522         IT_EP_NO_FLAG          = 0x00,
9523         IT_EP_REUSEADDR       = 0x01
9524     } it_ep_rc_creation_flags_t;
9525
9526     #define IT_THRESHOLD_DISABLE 0
9527
9528     typedef enum {
9529         IT_EVD_DEQUEUE_NOTIFICATIONS = 0x01,
9530         IT_EVD_CREATE_FD             = 0x02,
9531         IT_EVD_OVERFLOW_DEFAULT      = 0x04,
9532         IT_EVD_OVERFLOW_NOTIFY      = 0x08,
9533         IT_EVD_OVERFLOW_AUTO_RESET   = 0x10
9534     } it_evd_flags_t;
9535
9536     typedef enum {
9537         IT_EVD_PARAM_ALL              = 0x0000001,
9538         IT_EVD_PARAM_IA              = 0x0000002,
9539         IT_EVD_PARAM_EVENT_NUMBER    = 0x0000004,
9540         IT_EVD_PARAM_FLAG            = 0x0000008,
9541         IT_EVD_PARAM_QUEUE_SIZE      = 0x0000010,
9542         IT_EVD_PARAM_THRESHOLD       = 0x0000020,
9543         IT_EVD_PARAM_AEVD_HANDLE     = 0x0000040,
9544         IT_EVD_PARAM_FD              = 0x0000080,
9545         IT_EVD_PARAM_BOUND           = 0x0000100,
9546         IT_EVD_PARAM_ENABLED         = 0x0000200,
9547         IT_EVD_PARAM_OVERFLOWED      = 0x0000400
9548     } it_evd_param_mask_t;
9549
9550     typedef struct {
9551         it_ia_handle_t    ia;                /* IT_EVD_PARAM_IA */
9552         it_event_type_t   event_number;     /* IT_EVD_PARAM_EVENT_NUMBER*/
9553         it_evd_flags_t    evd_flag;        /* IT_EVD_PARAM_FLAG */
9554         size_t            sevd_queue_size; /* IT_EVD_PARAM_QUEUE_SIZE */
9555         size_t            sevd_threshold; /* IT_EVD_PARAM_THRESHOLD */
9556         it_evd_handle_t   aevd;           /* IT_EVD_PARAM_AEVD_HANDLE*/
9557         int               fd;             /* IT_EVD_PARAM_FD */
9558         it_boolean_t      evd_bound;      /* IT_EVD_PARAM_BOUND */
9559         it_boolean_t      evd_enabled;    /* IT_EVD_PARAM_ENABLED */
9560         it_boolean_t      evd_overflowed; /* IT_EVD_PARAM_OVERFLOWED */
9561     } it_evd_param_t;
9562
9563     typedef struct {
9564
9565         /* Most recent major version number of the IT-API supported by the
9566         Interface */
9567         uint32_t    major_version;
9568
9569         /* Most recent minor version number of the IT-API supported by the

```

```

9570         Interface */
9571         uint32_t     minor_version;
9572
9573         /* The transport that the Interface uses, as defined in
9574            it_ia_info_t. */
9575         it_transport_type_t     transport_type;
9576
9577         /* The name of the Interface, suitable for input to it_ia_create.
9578            The name is a string of maximum length IT_INTERFACE_NAME_SIZE,
9579            including the terminating NULL character. */
9580         char    name[IT_INTERFACE_NAME_SIZE];
9581
9582     } it_interface_t;
9583
9584     typedef enum {
9585         IT_LISTEN_NO_FLAG                = 0x0000,
9586         IT_LISTEN_CONN_QUAL_INPUT        = 0x0001
9587     } it_listen_flags_t;
9588
9589     typedef enum {
9590         IT_LISTEN_PARAM_ALL                = 0x0001,
9591         IT_LISTEN_PARAM_IA_HANDLE         = 0x0002,
9592         IT_LISTEN_PARAM_SPIGOT_ID        = 0x0004,
9593         IT_LISTEN_PARAM_CONNECT_EVD      = 0x0008,
9594         IT_LISTEN_PARAM_CONN_QUAL        = 0x0010
9595     } it_listen_param_mask_t;
9596
9597     typedef struct {
9598         it_ia_handle_t     ia_handle;    /* IT_LISTEN_PARAM_IA_HANDLE */
9599         size_t             spigot_id;    /* IT_LISTEN_PARAM_SPIGOT_ID */
9600         it_evd_handle_t    connect_evd; /* IT_LISTEN_PARAM_CONNECT_EVD*/
9601         it_conn_qual_t     connect_qual; /* IT_LISTEN_PARAM_CONN_QUAL */
9602     } it_listen_param_t;
9603
9604     typedef enum {
9605         IT_LMR_PARAM_ALL                = 0x000001,
9606         IT_LMR_PARAM_IA                 = 0x000002,
9607         IT_LMR_PARAM_PZ                 = 0x000004,
9608         IT_LMR_PARAM_ADDR               = 0x000008,
9609         IT_LMR_PARAM_LENGTH             = 0x000010,
9610         IT_LMR_PARAM_MEM_PRIV           = 0x000020,
9611         IT_LMR_PARAM_FLAG               = 0x000040,
9612         IT_LMR_PARAM_SHARED_ID          = 0x000080,
9613         IT_LMR_PARAM_RMR_CONTEXT        = 0x000100,
9614         IT_LMR_PARAM_ACTUAL_ADDR        = 0x000200,
9615         IT_LMR_PARAM_ACTUAL_LENGTH      = 0x000400
9616     } it_lmr_param_mask_t;
9617
9618     typedef struct {
9619         it_ia_handle_t     ia;           /* IT_LMR_PARAM_IA */
9620         it_pz_handle_t     pz;           /* IT_LMR_PARAM_PZ */
9621         void                *addr;       /* IT_LMR_PARAM_ADDR */
9622         it_length_t        length;       /* IT_LMR_PARAM_LENGTH */
9623         it_mem_priv_t      privs;        /* IT_LMR_PARAM_MEM_PRIV */

```

```

9624     it_lmr_flag_t    flags;           /* IT_LMR_PARAM_FLAG */
9625     uint32_t        shared_id;       /* IT_LMR_PARAM_SHARED_ID*/
9626     it_rmr_context_t rmr_context;    /* IT_LMR_PARAM_RMR_CONTEXT */
9627     void            *actual_addr;    /* IT_LMR_PARAM_ACTUAL_ADDR */
9628     it_length_t     actual_length;   /*IT_LMR_PARAM_ACTUAL_LENGTH*/
9629 } it_lmr_param_t;
9630
9631 typedef uint64_t it_rdma_addr_t;
9632
9633 typedef enum {
9634     IT_PZ_PARAM_ALL   = 0x01,
9635     IT_PZ_PARAM_IA   = 0x02
9636 } it_pz_param_mask_t;
9637
9638 typedef struct {
9639     it_ia_handle_t ia; /* IT_PZ_PARAM_IA */
9640 } it_pz_param_t;
9641
9642 typedef enum {
9643     IT_RMR_PARAM_ALL           = 0x0000001,
9644     IT_RMR_PARAM_IA           = 0x0000002,
9645     IT_RMR_PARAM_PZ           = 0x0000004,
9646     IT_RMR_PARAM_BOUND        = 0x0000008,
9647     IT_RMR_PARAM_LMR          = 0x0000010,
9648     IT_RMR_PARAM_ADDR         = 0x0000020,
9649     IT_RMR_PARAM_LENGTH       = 0x0000040,
9650     IT_RMR_PARAM_MEM_PRIV     = 0x0000080,
9651     IT_RMR_PARAM_RMR_CONTEXT = 0x000100
9652 } it_rmr_param_mask_t;
9653
9654 typedef struct {
9655     it_ia_handle_t ia;           /* IT_RMR_PARAM_IA */
9656     it_pz_handle_t pz;          /* IT_RMR_PARAM_PZ */
9657     it_boolean_t   bound;       /* IT_RMR_PARAM_BOUND */
9658     it_lmr_handle_t lmr;        /* IT_RMR_PARAM_LMR */
9659     void *         addr;         /* IT_RMR_PARAM_ADDR */
9660     it_length_t    length;      /* IT_RMR_PARAM_LENGTH */
9661     it_mem_priv_t  privs;       /* IT_RMR_PARAM_MEM_PRIV */
9662     it_rmr_context_t rmr_context; /* IT_RMR_PARAM_RMR_CONTEXT */
9663 } it_rmr_param_t;
9664
9665 typedef enum {
9666     IT_UD_PARAM_ALL           = 0x00000001,
9667     IT_UD_PARAM_IA_HANDLE    = 0x00000002,
9668     IT_UD_PARAM_REQ_ID       = 0x00000004,
9669     IT_UD_PARAM_REPLY_EVD    = 0x00000008,
9670     IT_UD_PARAM_CONN_QUAL    = 0x00000010,
9671     IT_UD_PARAM_DEST_PATH    = 0x00000020,
9672     IT_UD_PARAM_PRIV_DATA    = 0x00000040,
9673     IT_UD_PARAM_PRIV_DATA_LENGTH = 0x00000080
9674 } it_ud_svc_req_param_mask_t;
9675
9676 typedef struct {
9677     it_ia_handle_t ia; /* IT_UD_PARAM_IA_HANDLE */

```

```

9678     uint32_t      request_id; /* IT_UD_PARAM_REQ_ID */
9679     it_evd_handle_t  reply_evd; /* IT_UD_PARAM_REPLY_EVD */
9680     it_conn_qual_t   conn_qual; /* IT_UD_PARAM_CONN_QUAL */
9681     it_path_t        destination_path; /* IT_UD_PARAM_DEST_PATH */
9682     unsigned char private_data[IT_MAX_PRIV_DATA];
9683                                     /* IT_UD_PARAM_PRIV_DATA */
9684     size_t private_data_length; /* IT_UD_PARAM_PRIV_DATA_LEN */
9685 } it_ud_svc_req_param_t;
9686
9687 it_status_t it_address_handle_create(
9688     IN          it_pz_handle_t      pz_handle,
9689     IN          const it_path_t      *destination_path,
9690     IN          it_ah_flags_t       ah_flags,
9691     OUT         it_addr_handle_t     *addr_handle
9692 );
9693
9694
9695 it_status_t it_address_handle_free(
9696     IN          it_addr_handle_t     addr_handle
9697 );
9698
9699 it_status_t it_address_handle_modify(
9700     IN          it_addr_handle_t     addr_handle,
9701     IN          it_addr_param_mask_t mask,
9702     IN          const it_addr_param_t *params
9703 );
9704
9705 it_status_t it_address_handle_query(
9706     IN          it_addr_handle_t     addr_handle,
9707     IN          it_addr_param_mask_t mask,
9708     OUT         it_addr_param_t     *params
9709 );
9710
9711 it_status_t it_convert_net_addr(
9712     IN          const it_net_addr_t* source_addr,
9713     IN          it_net_addr_type_t   addr_type,
9714     OUT         it_net_addr_t*      destination_addr
9715 );
9716
9717 it_status_t it_ep_accept(
9718     IN          it_ep_handle_t       ep_handle,
9719     IN          it_cn_est_identifiler_t cn_est_id,
9720     IN          const unsigned char   *private_data,
9721     IN          size_t                private_data_length
9722 );
9723
9724 it_status_t it_ep_connect(
9725     IN          it_ep_handle_t       ep_handle,
9726     IN          const it_path_t*     path,
9727     IN          const it_conn_attributes_t* conn_attr,
9728     IN          const it_conn_qual_t* connect_qual,
9729     IN          it_cn_est_flags_t    cn_est_flags,
9730     IN          const unsigned char* private_data,
9731     IN          size_t                private_data_length

```



```

9732     );
9733
9734     it_status_t it_ep_disconnect (
9735         IN          it_ep_handle_t          ep_handle,
9736         IN          const unsigned char     *private_data,
9737         IN          size_t                  private_data_length
9738     );
9739
9740     it_status_t it_ep_free(
9741         IN          it_ep_handle_t          ep_handle
9742     );
9743
9744     it_status_t it_ep_modify(
9745         IN          it_ep_handle_t          ep_handle,
9746         IN          it_ep_param_mask_t     mask,
9747         IN          const it_ep_attributes_t *ep_attr
9748     );
9749
9750     it_status_t it_ep_query(
9751         IN          it_ep_handle_t          ep_handle,
9752         IN          it_ep_param_mask_t     mask,
9753         OUT         it_ep_param_t          *params
9754     );
9755
9756     it_status_t it_ep_rc_create (
9757         IN          it_pz_handle_t          pz_handle,
9758         IN          it_evd_handle_t        request_sevd_handle,
9759         IN          it_evd_handle_t        rcv_sevd_handle,
9760         IN          it_evd_handle_t        connect_sevd_handle,
9761         IN          it_ep_rc_creation_flags_t flags,
9762         IN          const it_ep_attributes_t *ep_attr,
9763         OUT         it_ep_handle_t          *ep_handle
9764     );
9765
9766     it_status_t it_ep_reset(
9767         IN          it_ep_handle_t          ep_handle
9768     );
9769
9770     it_status_t it_ep_ud_create (
9771         IN          it_pz_handle_t          pz_handle,
9772         IN          it_evd_handle_t        request_sevd_handle,
9773         IN          it_evd_handle_t        rcv_sevd_handle,
9774         IN          const it_ep_attributes_t *ep_attr,
9775         IN          size_t                  spigot_id,
9776         OUT         it_ep_handle_t          *ep_handle
9777     );
9778
9779
9780     it_status_t it_evd_create (
9781         IN          it_ia_handle_t          ia_handle,
9782         IN          it_event_type_t        event_number,
9783         IN          it_evd_flags_t        evd_flag,
9784         IN          size_t                  sevd_queue_size,
9785         IN          size_t                  sevd_threshold,

```

```

9786         IN          it_evd_handle_t          aevd_handle,
9787         OUT         it_evd_handle_t          *evd_handle,
9788         OUT         int                       *fd
9789
9790     );
9791
9792     it_status_t it_evd_dequeue(
9793         IN          it_evd_handle_t          evd_handle,
9794         OUT         it_event_t              *event
9795
9796     );
9797
9798     it_status_t it_evd_free(
9799         IN          it_evd_handle_t          evd_handle
9800     );
9801
9802     it_status_t it_evd_modify(
9803         IN          it_evd_handle_t          evd_handle,
9804         IN          it_evd_param_mask_t     mask,
9805         IN          const it_evd_param_t     *params
9806     );
9807
9808     it_status_t it_evd_post_se(
9809         IN          it_evd_handle_t          evd_handle,
9810         IN          const void               *event
9811     );
9812
9813     it_status_t it_evd_query(
9814         IN          it_evd_handle_t          evd_handle,
9815         IN          it_evd_param_mask_t     mask,
9816         OUT         it_evd_param_t          *params
9817     );
9818
9819     it_status_t it_evd_wait(
9820         IN          it_evd_handle_t          evd_handle,
9821         IN          uint64_t                 timeout,
9822         OUT         it_event_t              *event,
9823         OUT         size_t                   *nmore
9824
9825     );
9826
9827     it_status_t it_get_consumer_context(
9828         IN          it_handle_t              handle,
9829         OUT         it_context_t             *context
9830     );
9831
9832     it_status_t it_get_handle_type(
9833         IN          it_handle_t              handle,
9834         OUT         it_handle_type_enum_t    *type_of_handle
9835     );
9836
9837     it_status_t it_get_pathinfo(
9838         IN          it_ia_handle_t           ia_handle,
9839         IN          size_t                   spigot_id,

```

```

9840         IN          const it_net_addr_t          *net_addr,
9841         IN OUT       size_t                      *num_paths,
9842         OUT          size_t                      *total_paths,
9843         OUT          it_path_t                   *paths
9844     );
9845
9846     it_status_t it_handoff(
9847         IN          const it_conn_qual_t          *conn_qual,
9848         IN          size_t                        spigot_id,
9849         IN          it_cn_est_identifier_t        cn_est_id
9850     );
9851
9852
9853     it_status_t it_ia_create(
9854         IN          const char                    *name,
9855         IN          uint32_t                     major_version,
9856         IN          uint32_t                     minor_version,
9857         OUT         it_ia_handle_t              *ia_handle
9858     );
9859
9860     it_status_t it_ia_free(
9861         IN          it_ia_handle_t              ia_handle
9862     );
9863
9864     void it_ia_info_free(
9865         IN          it_ia_info_t                *ia_info
9866     );
9867
9868
9869     it_status_t it_ia_query(
9870         IN          it_ia_handle_t              ia_handle,
9871         OUT         it_ia_info_t                **ia_info
9872     );
9873
9874     void it_interface_list(
9875         OUT         it_interface_t              *interfaces,
9876         IN OUT     size_t                      *num_interfaces,
9877         IN OUT     size_t                      *total_interfaces
9878     );
9879
9880     it_status_t it_listen_create(
9881         IN          it_ia_handle_t              ia_handle,
9882         IN          size_t                      spigot_id,
9883         IN          it_evd_handle_t            connect_evd,
9884         IN          it_listen_flags_t          flags,
9885         IN OUT     it_conn_qual_t            *conn_qual,
9886         OUT         it_listen_handle_t          *listen_handle
9887     );
9888
9889     it_status_t it_listen_free(
9890         IN          it_listen_handle_t          listen_handle
9891     );
9892
9893     it_status_t it_listen_query(

```

```

9894         IN             it_listen_handle_t     listen_handle,
9895         IN             it_listen_param_mask_t  mask,
9896         OUT            it_listen_param_t       *params
9897     );
9898
9899     it_status_t it_lmr_create(
9900         IN             it_pz_handle_t         pz_handle,
9901         IN             void                   *addr,
9902         IN             it_length_t           length,
9903         IN             it_mem_priv_t         privs,
9904         IN             it_lmr_flag_t         flags,
9905         IN             uint32_t               shared_id,
9906         OUT            it_lmr_handle_t       *lmr_handle,
9907         IN OUT         it_rmr_context_t      *rmr_context
9908     );
9909
9910     it_status_t it_lmr_free(
9911         IN             it_lmr_handle_t       lmr_handle
9912     );
9913
9914     it_status_t it_lmr_modify(
9915         IN             it_lmr_handle_t       lmr_handle,
9916         IN             it_lmr_param_mask_t   mask,
9917         IN             const it_lmr_param_t   *params
9918     );
9919
9920     it_status_t it_lmr_query(
9921         IN             it_lmr_handle_t       lmr_handle,
9922         IN             it_lmr_param_mask_t   mask,
9923         OUT            it_lmr_param_t       *params
9924     );
9925
9926     it_status_t it_lmr_sync_rdma_read(
9927         IN             const it_lmr_triplet_t *local_segments,
9928         IN             size_t                 num_segments
9929     );
9930
9931     it_status_t it_lmr_sync_rdma_write(
9932         IN             const it_lmr_triplet_t *local_segments,
9933         IN             size_t                 num_segments
9934     );
9935
9936
9937     it_status_t it_post_rdma_read (
9938         IN             it_ep_handle_t         ep_handle,
9939         IN             const it_lmr_triplet_t *local_segments,
9940         IN             size_t                 num_segments,
9941         IN             it_dto_cookie_t       cookie,
9942         IN             it_dto_flags_t        dto_flags,
9943         IN             it_rdma_addr_t        rdma_addr,
9944         IN             it_rmr_context_t      rmr_context
9945     );
9946
9947     it_status_t it_post_rdma_write (

```

```

9948         IN             it_ep_handle_t           ep_handle,
9949         IN             const it_lmr_triplet_t     *local_segments,
9950         IN             size_t                     num_segments,
9951         IN             it_dto_cookie_t           cookie,
9952         IN             it_dto_flags_t           dto_flags,
9953         IN             it_rdma_addr_t           rdma_addr,
9954         IN             it_rmr_context_t         rmr_context
9955     );
9956
9957     it_status_t it_post_recv(
9958         IN             it_ep_handle_t           ep_handle,
9959         IN             const it_lmr_triplet_t     *local_segments,
9960         IN             size_t                     num_segments,
9961         IN             it_dto_cookie_t           cookie,
9962         IN             it_dto_flags_t           dto_flags
9963     );
9964
9965     it_status_t it_post_recvfrom(
9966         IN             it_ep_handle_t           ep_handle,
9967         IN             const it_lmr_triplet_t     *local_segments,
9968         IN             size_t                     num_segments,
9969         IN             it_dto_cookie_t           cookie,
9970         IN             it_dto_flags_t           dto_flags
9971     );
9972
9973     it_status_t it_post_send(
9974         IN             it_ep_handle_t           ep_handle,
9975         IN             const it_lmr_triplet_t     *local_segments,
9976         IN             size_t                     num_segments,
9977         IN             it_dto_cookie_t           cookie,
9978         IN             it_dto_flags_t           dto_flags
9979     );
9980
9981     it_status_t it_post_sendto(
9982         IN             it_ep_handle_t           ep_handle,
9983         IN             const it_lmr_triplet_t     *local_segments,
9984         IN             size_t                     num_segments,
9985         IN             it_dto_cookie_t           cookie,
9986         IN             it_dto_flags_t           dto_flags,
9987         IN             const it_dg_remote_ep_addr_t *remote_ep_addr
9988     );
9989
9990     it_status_t it_pz_create(
9991         IN             it_ia_handle_t           ia_handle,
9992         OUT            it_pz_handle_t           *pz_handle
9993     );
9994
9995     it_status_t it_pz_free(
9996         IN             it_pz_handle_t           pz_handle
9997     );
9998
9999     it_status_t it_pz_query(
10000        IN             it_pz_handle_t           pz_handle,
10001        IN             it_pz_param_mask_t       mask,

```

```

10002         OUT          it_pz_param_t          *params
10003     );
10004
10005     it_status_t it_reject(
10006         IN          it_cn_est_identifier_t    cn_est_id,
10007         IN          const unsigned char      *private_data,
10008         IN          size_t                   private_data_length
10009     );
10010
10011     it_status_t it_rmr_bind(
10012         IN          it_rmr_handle_t          rmr_handle,
10013         IN          it_lmr_handle_t          lmr_handle,
10014         IN          void                     *addr,
10015         IN          it_length_t              length,
10016         IN          it_mem_priv_t            privs,
10017         IN          it_ep_handle_t           ep_handle,
10018         IN          it_dto_cookie_t          cookie,
10019         IN          it_dto_flags_t           dto_flags,
10020         OUT         it_rmr_context_t         *rmr_context
10021     );
10022
10023     it_status_t it_rmr_create(
10024         IN          it_pz_handle_t          pz_handle,
10025         OUT         it_rmr_handle_t         *rmr_handle
10026     );
10027
10028     it_status_t it_rmr_free(
10029         IN          it_rmr_handle_t         rmr_handle
10030     );
10031
10032     it_status_t it_rmr_query(
10033         IN          it_rmr_handle_t         rmr_handle,
10034         IN          it_rmr_param_mask_t     mask,
10035         OUT         it_rmr_param_t          *params
10036     );
10037
10038     it_status_t it_rmr_unbind(
10039         IN          it_rmr_handle_t         rmr_handle,
10040         IN          it_ep_handle_t           ep_handle,
10041         IN          it_dto_cookie_t          cookie,
10042         IN          it_dto_flags_t           dto_flags
10043     );
10044
10045     it_status_t it_set_consumer_context(
10046         IN          it_handle_t              handle,
10047         IN          it_context_t             context
10048     );
10049
10050     it_status_t it_ud_service_reply (
10051         IN          it_ud_svc_req_identifier_t  ud_svc_req_id,
10052         IN          it_ud_svc_req_status_t      status,
10053         IN          it_remote_ep_info_t         ep_info,
10054         IN          const unsigned char        *private_data,
10055         IN          size_t                      private_data_length

```

```

10056     );
10057
10058     it_status_t it_ud_service_request (
10059         IN             it_ud_svc_req_handle_t  ud_svc_handle
10060     );
10061
10062     it_status_t it_ud_service_request_handle_create (
10063         IN             const it_conn_qual_t     *conn_qual,
10064         IN             it_evd_handle_t         reply_evd,
10065         IN             const it_path_t         *destination_path,
10066         IN             const unsigned char     *private_data,
10067         IN             size_t                  private_data_length,
10068         OUT            it_ud_svc_req_handle_t  *ud_svc_handle
10069     );
10070
10071     it_status_t it_ud_service_request_handle_free (
10072         IN             it_ud_svc_req_handle_t  ud_svc_handle
10073     );
10074
10075     it_status_t it_ud_service_request_handle_query (
10076         IN             it_ud_svc_req_handle_t  ud_svc_handle,
10077         IN             it_ud_svc_req_param_mask_t  mask,
10078         OUT            it_ud_svc_req_param_t     *ud_svc_handle_info
10079     );
10080
10081

```

10081 B.2 it_api_os_specific.h

```

10082
10083     #include "/usr/include/sys/types.h"
10084     #include "/usr/include/netinet/in.h"
10085
10086     /* the following should have arrived from types.h */
10087     typedef unsigned char uint8_t;
10088     typedef unsigned short uint16_t;
10089     typedef unsigned int uint32_t;
10090     typedef unsigned long uint64_t;
10091
10092     /* defines */
10093
10094     #define IT_INTERFACE_NAME_SIZE 128
10095

```