uDAPL:User Direct Access Programming Library

Version: 2.0

Date: January 5, 2007

DOCUMENT STATUS uDAPL-2.0 is the ratified DAT-Collaborative version of the uDAPL document with an addition of socket-based Connection Management for iWARP & IB, verb extensions semantic support, and errata to DAPL-1.2.

ABSTRACT The User Direct Access Programming Library (uDAPL) defines a single set of user-level APIs for all RDMA-capable Transports. The uDAPL mission is to define a Transport-independent and Platform-standard set of APIs that exploits RDMA capabilities, such as those present in IB, VI, and RDDP WG of IETF.

CHAPTER 1: INTRODUCTION

Over the last several years, multiple networks appeared that provide RDMA capabilities. Some of them define their own APIs and some of them do not define APIs at all. Users of these networks who develop applications that take advantage of the RDMA semantics want to have a common set of APIs for all of these networks. The work of the DAT Collaborative fills this need.

Application domains at which the DAT Collaborative-developed APIs are targeted are as follows:

1) DAFS
2) Heterogeneous clusters/databases
3) Homogeneous clusters/databases
4) Sockets that use RDMA capabilities (SDP)
5) Message Passing Interface (MPI)
6) SCSI RDMA Protocol (SRP)
7) iSCSI extensions for RDMA (iSER)

The DAT Collaborative is currently considering the following Transports that provide RDMA capabilities:

1) InfiniBand
2) VI Architecture
3) iWARP which is currently under development by RDDP WG of IETF for IP-based networks.

CHAPTER 2: TERMINOLOGY

This chapter defines the terms that the Direct Access Transport APIs are using.

Backlog A queue of incoming Connection Requests associated with a Connection Qualifier of an Interface Adapter. The size of the backlog specifies the upper bound on the number of pending Connection Request instances the Provider needs to support at any one time.

Barrier Fence An indicator that the posted operation to a connected Endpoint shall not be processed until prior potentially conflicting operations on the same connection are completed.

Class of Service An enumerator that can be used to specify the quality of network service requested for a connection.

Connect Request Handoff Passing a connect request from a uDAPL consumer to another DAT consumer strictly within the context of a single DAT provider instance and the same Interface Adapter on the same host.

Connection An association between a pair of Endpoints such that data of posted data-transfer operation requests of either Endpoint arrive at the other Endpoint of the Connection.

Connection Management A portion of DAT Provider (software, hardware, and the combination of the two) that is used for establishing, maintaining, and releasing connections.

Connection Qualifier A value that allows a Connection Manager to associate an incoming Connection request with the entity providing the service.

Consumer Context A Consumer-supplied value that can be associated with an instance of any DAT Object. It can be used to correlate the DAT Object with Consumer data structures.

Consumer Notification Object (CNO) A DAT IA-scope object that can be associated with a set of DAT Event Dispatchers so that under certain conditions (under consumer control), arrival of Notification Events on those Event Queues causes activation of the CNO. Consumers can wait for notification upon a CNO. Additionally, a CNO can pass through notifications to OS-dependent synchronization methods using an OS Wait Proxy Agent.

DAT Consumer An application that requires Direct Access Transport services by opening an Interface Adapter.

DAT Handle A programmatic constuct that represents the authorization of a Consumer to operate on a specific data structure internal to the DAT Provider.

DAT Provider The Provider of the Transport services for a Direct Access application.

Data Transfer Completion (DTC) The status of the completed data transfer operation.

Data Transfer Operation (DTO) The requested data movement transfer submitted to a DAT Provider.

DTO Cookie A Consumer-supplied identifier for a Data Transfer Operation that allows a Consumer to uniquely identify the DTO when it completes.

Dynamic Registry The information repository maintained on a per-process basis by the Registry. It reflects Providers actually loaded within the context of the process.

Endpoint The local part of a Connection that supports posting DTO requests and receives.

Event A structure or record that is delivered to the Consumer through an Event Dispatcher to provide notice of some kind. Types of Events include DTO completions, connection state changes, asynchronous errors, and software events generated by the Consumer.

Event Dispatcher A DAT Object that conceptually merges events from one or more Event Streams. These events can be dequeued by the Consumer directly via *dat_evd_dequeue* or *dat_evd_wait*. The Event Dispatcher is responsible for completion of transport-specific fetching and handshaking for the events it collects. Each event is delivered to the Consumer exactly once.

Event Stream A source of events for Event Dispatcher: DTO completions, Connection Requests for passive side, connection reject notifications for active side, connection establishment completion notifications, disconnect notifications, connection errors, Connection Request timeouts, channel adapter asynchronous errors, remote memory bind completion notifications, and Consumer-generated notifications. An Event Stream is the conduit between DAT Objects that generates events and Event Dispatchers that consume events.

Fabric A network with RDMA capabilities.

Host One or more Interface Adapters controlled by a single memory/CPU complex.

Interface Adapter (IA) A host resident device that transfers messages to and from the host memory associated with a specific Endpoint and a Fabric.

IA Address The Interface Adapter Address on the Fabric.

IOV The Input/Output Vector; an array of LMR Triplets that specifies the local buffer for a DTO or an RMR Bind.

kDAPL Provider The Provider of the Transport services for a kernel-level Direct Access application.

LMR Alignment A characteristic of an Interface Adapter that specifies the boundaries on which Local Memory Regions are actually enabled. If the granularity is 4 KB, then for each mapped region, actual registered memory starts and ends on the 4-KB boundary.

LMR Granularity A characteristic of an Interface Adapter that specifies the minimum size with which Local Memory Regions actually map local memory. If the granularity is 4 KB, each mapped region must be a multiple of 4 KB.

LMR Triplet A type used to specify a slice within a Local Memory Region. Each LMR Triplet specifies the LMR Context, the virtual address, and a size.

Local Memory Region Context (LMR Context) The Provider-generated handle for a Consumer-registered arbitrary size, contiguous virtual memory. The Consumer uses it to indicate the Memory Region for the local memory access operations. For example, it is used for DTOs on a Connection whose local Endpoint belongs to the Interface Adapter.

Local Memory Region Virtual Address (LMR VA) A Virtual Address that specifies the local memory address within a region of memory represented by the LMR Memory Region Context.

Memory Permission An indicator of what accesses are allowed to the memory (Local read/write, RDMA read/write).

Memory Protection An indicator of who can access the memory.

Memory Region An arbitrarily sized, virtually contiguous area of memory in the Consumer's address space that was registered, enabling Interface Adapter local access and, optionally, remote access.

Memory Registration The process of enabling Interface Adapter access to local memory by creating a Local Memory Region (LMR) and then optionally enabling remote access to any portion of it by creating one or more Remote Memory Regions (RMRs).

Notification Event All events are Notification Events, except for DTO and RMR completion events whose completion flag at the post specify Notification Suppression, or for Recv completions whose matching Send do not specify Solicited Wait. An arrival of a Notification Event triggers unblocking of a waiter on EVD if EVD is configured for Notification Events.

Non-Notification Event DTO and RMR completion events whose completion flag at the post does not specify Notification Suppression, or for Recv completions whose matching Send does not specify Solicited Wait. An arrival of a Non-Notification Event does not unblock a waiter on EVD if EVD is configured for Notification events.

Operation Types The Send, Receive, RDMA Read, or RDMA Write DTOs and RMR Binds.

OS Wait Proxy Agent An object that acts as a proxy for an OS-specific synchronization resource. Possibilities include a semaphore, a message queue, or a file descriptor. The proxy agent allows a CNO (Consumer Notification Object) to trigger the target resource without knowing the OS-specific methods for doing so. How consumers interact with the OS-specific resource is outside the scope of the uDAPL specification.

Port Qualifier A Network Identifier of the specific Endpoint that differentiates it from other Endpoints on the same IA Address. This is the identifier of an actual Endpoint. This contrasts with a Service Point Connection Qualifier, which can be used to request a connection or listen for Connection Requests.

Private Data The Consumer Data that is opaque to the CM that is passed between local and remote Consumers of a connection. Active side Connection request and Passive side Connection Acception support Consumer Private Data.

Consumers can use Private Data to “piggy-back” information over the CM message exchange.

Protection Zone A mechanism for association Endpoints and registered LMR and RMR memory of an Interface Adapter that defines protection for local and remote memory accesses by DTO operations.

Public Service Point A Persistent Service Point whose associated Connection Qualifier is advertised to other hosts, and which can support multiple Connection Requests.

RDMA Remote Direct Memory Access—The access of local memory by the remote Endpoint. There are two RDMA operations: RDMA Read and RDMA Write.

RDMA Memory Region Context (RMR Context) A representation for an arbitrarily sized, registered contiguous virtual space that belongs to an Interface Adapter so that it can support Remote DMA operations on the Connection whose local Endpoint belongs to the Interface Adapter.

RDMA Memory Region Target Address (RMR Target Address) The Address that specifies the memory address within a region of memory represented by RDMA Memory Region Context. The specification can be either by Interface Adapter Virtual Address or an offset from the start of the memory represented by the RMR Context.

Registry An active software component that is instantiated at most once per running process. It is responsible for routing *dat_ia_open()* calls, autoloading of Provider libraries, and accepting dynamic registration calls from Providers. The registry accesses information from the Static Registry and maintains the information in the Dynamic Registry.

Reliable Connection A connection type such that data of posted DTOs of either Endpoint of the Connection reliably arrives at the other Endpoint of the Connection uncorrupted in the absence of errors and in the order defined by the reliable connection ordering rules.

Remote Memory Region (RMR) A window that can be bound to a section of a Local Memory Region to enable remote accesses.

Reserved Service Point A Service Point whose associated Connection Qualifier is not advertised to other hosts. Its knowledge is only known by an application privately for peer-to-peer or application internally negotiated connections.

Request Queue (RQ) An internal opaque queue of a connected Endpoint on which DTO requests, DTO receives, and RMR Binds are posted. One RQ, which is commonly called Send Queue, collects send, RDMA Read, and RDMA Write DTO requests and RMR Binds. Another RQ, which is commonly called Receive Queue, collects receive DTOs.

RMR Alignment A characteristic of an Interface Adapter that specifies the boundaries on which Remote Memory Regions exported by this host are actually enabled.

RMR Bind The process of modifying what an RMR references to a local memory and permissions. The Bind specifies the referenced region and new permissions. Each Bind results in a new RMR Context, and invalidates previous RMR Contexts.

RMR Cookie A Consumer-supplied identifier for an RMR Bind operation that allows a Consumer to uniquely identify the RMR Bind when it completes.

RMR Granularity A characteristic of an Interface Adapter that specifies the minimum size with which Remote Memory Regions exported by this host are actually enabled. For an adapter that effectively had no windowing capability, the granularity is the size of local memory.

RMR Triplet A type used to specify a slice within a Remote Memory Region. Each RMR Triplet specifies the RMR Context, the target address, and a size.

Service Point A DAT Object associated with Connection Qualifiers that is generated in Connection Requests for Consumers or directly establishes connections for Consumers.

Shared Receive Queue (SRQ) A DAT Object that supplies DTO receives for multiple EPs. The Consumer supplies buffers to a single SRQ, rather than to the Receive Queue (RQ) of each EP. This allows pooling of available buffers across multiple EPs which can reduce the number of pre-committed buffers required.

Software Event An Event generated for an Event Dispatcher by the Consumer, as opposed to those generated by the Interface Adapter.

Solicited Wait A modifier for a send DTO request submitted to an Endpoint of the Connection. It specifies that the completion of the matching receive DTO on the remote side of the Connection generate a notification receive DTO Completion Event. All other receive DTO completions on that Connection complete with a non-notification Event.

Static Registry System-wide information repository used by the Registry to support auto-loading of Providers.

uDAPL Provider The Provider of the Transport services for a user-level Direct Access application.

Upper-Level Protocol (ULP) The higher level protocol applications that use DAT APIs. Examples of these are DAFS, SDP, SRP, and MPI.

Unsignaled Completion A modifier for a DTO or RMR Bind submitted to an Endpoint of the Connection that specifies that completion of the DTO or RMR Bind generates a non-notification Completion Event for an associated Event Dispatcher.

CHAPTER 3: MODEL

This chapter outlines the Direct Access Transport (DAT) Model.

The DAT Model is shown in Figure 1. There are two significant external interfaces to a DAT service Provider: One interface defines the boundary between the consumer of a set of transport services and the local transport Provider of these services. In the DAT model, this is the interface between the DAT Consumer and the uDAPL Provider.

The other interface defines the set of interactions between local and remote transport Providers that enables the local and remote Providers to offer a set of transport services between the local and remote transport consumers. In the DAT model, this is the set of interactions between a local uDAPL Provider and a remote uDAPL Provider that is visible to the local DAT Consumer and/or remote DAT Consumer.

This document defines the minimal set of required semantics for the interaction between uDAPL Providers that is visible to the local DAT Consumer. The transport protocol-specific details of the uDAPL Provider-to-uDAPL Provider interactions for specific transports is outside the scope of this document.These lower-level, transport-specific details are not defined here; it is expected that they are provided as part of the specification of a particular transport protocol (for example, IB, VI/TCP, FC-VI, and iWARP).

The DAT Collaborative's goal is to define the interface between uDAPL Provider and DAT Consumer. uDAPL defines the API for the kernel level when uDAPL Provider is within OS and below, while DAPL defines the API for the user level when DAT Consumer is completely within application space.

Each Interface Adapter is controlled by exactly one uDAPL Provider. Each uDAPL Provider can control multiple Interface Adapters. There can be multiple DAT Providers controlling disjoint sets of Interface Adapters on a host.

Figure 1, depicts the DAT framework and relationship between DAT and fabric protocols.

Figure 2 shows the DAT kernel API architecture model, including uDAPL Consumer, uDAPL Provider, OS, and Interface Adapter.

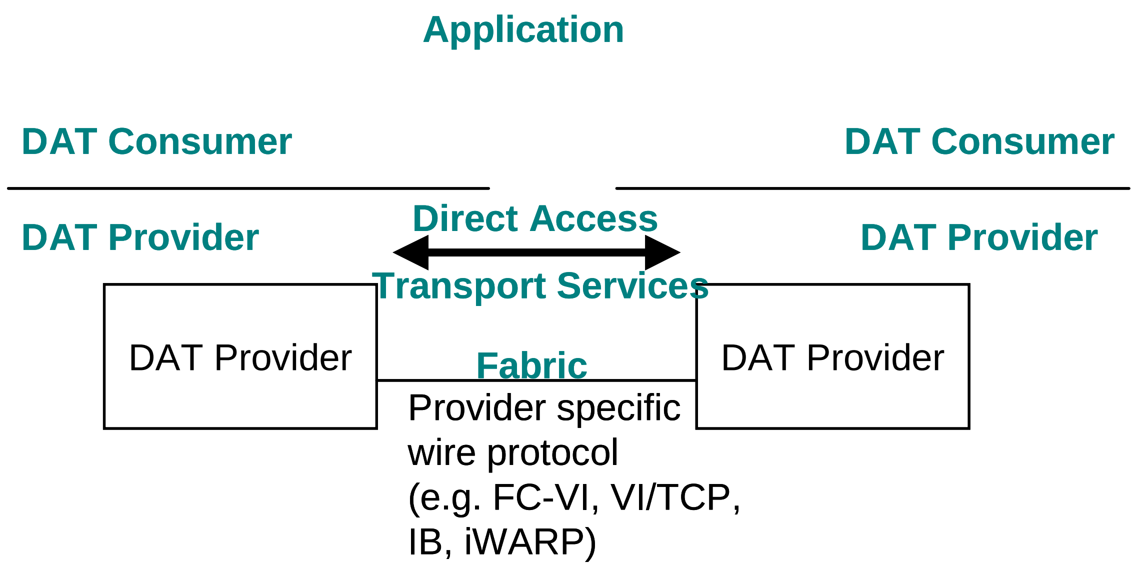


Figure 1 Direct Access Transport Framework

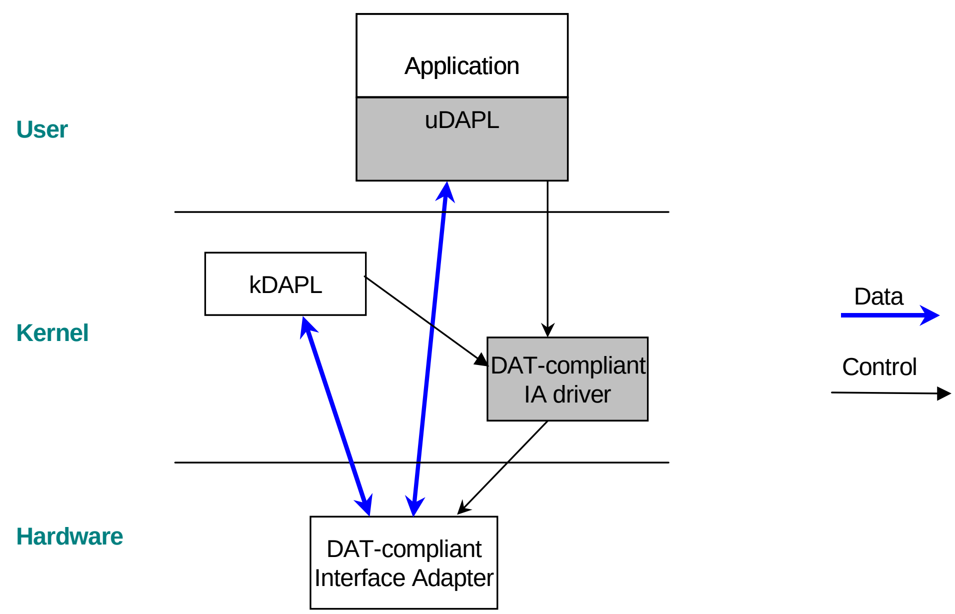


Figure 2 DAT User Architecture Model

CHAPTER 4: TRANSPORT REQUIREMENTS

This chapter states the transport requirements for DAT Provider-to-Provider interaction. The current version states the transport requirements for DAT:

1) DAT supports a connection that provides send-recv message transfers and RDMA Read and Write operations.
2) DAT supports reliable connection, which provides the following features:
    a) All data transfer operations submitted to the DAT Provider complete successfully in the absence of errors, with data delivered uncorrupted, in the order defined by ordering rules.
    b) Corruption of the data delivered to the Consumer (local one for RDMA Read) is detected as an error and reported to the Consumer.
    c) Data loss (inability to deliver data to the remote Endpoint of the Connection (from remote to local Endpoint for RDMA Read) is detected as an error and reported to the Consumer.
    d) Upon detection of an error, the connection is broken and all outstanding and in-progress data transfer operations are completed with an error.
    e) There is a one-to-one correspondence between send operations on one Endpoint of the Connection and recv operations on the other Endpoint of the Connection.
    f) There is no correspondence between RDMA operations on one Endpoint of the Connection and recv or send data transfer operation on the other Endpoint of the Connection.
    g) Data Transfer Operation Completion means that the Consumer can reclaim resources associated with the operation, including the memory that contains the data.
    h) Ordering rules:
        i) The data payload for the send operation matching a receive operation must be delivered into the receiver-indicated memory buffer without errors prior to the receive completion.
        ii) Receive operations on a Connection must be completed in the order of posting of their corresponding sends.

iii) Each RDMA Write operation posted on a Connection prior to a send operation must have its data payload delivered to the target memory region prior to the completion of the receive operation matching that send.

3) DAT supports multiple connections between the same or different pairs of nodes (client server pairs).

4) An RDMA Memory Region Context (RMR Context) supports RDMA operations for the set of DAT Connections that are associated with it. The association between a Connection and an RMR Context is established by the local Endpoint of the Connection where the Memory Region resides.

5) The same RMR Context can be associated with multiple connections. In addition, a connection can have multiple RMR Contexts associated with it.

6) The DAT Provider allows the DAT Consumer to create multiple RMR Contexts in the same memory.

7) DAT supports connection management, including the client-server connection establishment and the connection termination by either side of the Connection.

CHAPTER 5: USER-LEVEL API REQUIREMENTS

This chapter defines the requirements for the user-level Direct Access Transport APIs (uDAPL-2.0).

uDAPL-2.0 Consumer to Provider API requirements are:

5.1 LOCAL RESOURCE MODEL

1) Backward Compatibility
   a) A minor version upgrade guarantees that all uDAPL-compliant applications continue to work with, at most, recompiling.
   b) A major revision can deprecate functions.
   c) A support for a function can be dropped at a major revision after it is deprecated for a least one major revision.
   d) Errata and any revision can fix ambiguities of any previous version.
2) There is a one-to-one correspondence between the uDAPL Provider library and Interface Adapter:
   a) The uDAPL Provider library is open when the Interface Adapter is open.
   b) The uDAPL Provider library is closed when the Interface Adapter is closed.
   c) The Interface Adapter is the only DAT Object that can open and close the uDAPL Provider library.
   d) (Nonrequirement) uDAPL does not require a DAT Provider Object.
      i) uDAPL defines an Interface Adapter Object.
   e) uDAPL provides support for the Consumer to query Interface Adapter attributes.
   f) Closure of the open instance of an Interface Adapter is cascading and automatically closes all DAT Objects of the instance of the Interface Adapter.
   g) Closure of all other DAT Objects is restricted:
      i) Before closing a non-Interface Adapter DAT Object, the Consumer should make sure it is not in use (by some other DAT Object, including queues and in-progress and pending DTOs).

ii) uDAPL rejects an “improper” closure of a DAT Object that is still in use by other DAT Objects.

h) Multiple uDAPL Provider libraries can share binaries and loadable modules:

i) A single image of the uDAPL Provider library in the OS can be sufficient to open and close multiple Interface Adapters.

ii) A single Provider Library can present a single device as multiple Interface Adapters. For example, a Provider Library could present each port of an InfiniBand HCA as a distinct Interface Adapter.

3) uDAPL supports multiple Interface Adapters on a host:

a) (Nonrequirement) uDAPL is not required to support sharing of DAT Objects among Interface Adapters.

i) (Nonrequirement) uDAPL is not required to support sharing of Service Points across multiple Interface Adapters (waiting for incoming Connection Requests across multiple Interface Adapters)

ii) uDAPL shall support sharing of an OS Wait Proxy Agent across multiple providers.

iii) (Negative requirement) uDAPL provides no direct mechanism for a Consumer to wait for events from different Interface Adapters:

– A Consumer Notification Object (CNO) is valid only for Event Dispatchers from a single Interface Adapter.

– A uDAPL Consumer can use OS-specific synchronization methods to wait for Events from multiple Interface Adapters.

– uDAPL specifies OS Wait Proxy Agent for an Event Dispatcher CNO for invoking the OS-specific synchronization method with the Event Dispatcher Handle that caused the invocation.

– uDAPL Provider cannot claim exclusive ownership of the targeted OS-specific synchronization resource. The consumer must be free to feed notification of non-DAT Events to the same synchronization resource.

b) (Nonrequirement) uDAPL is not required to support sharing of DAT Objects among multiple open instances of the same Interface Adapter, except for Event Dispatcher for asynchronous errors:

i) uDAPL supports the API that allows the Consumer to request a shared or private Event Dispatcher for an open instance of Interface Adapter for asynchronous error notifications:

- uDAPL filters Asynchronous error notification to the instance of the open Interface Adapter that originated the error when it is known (for example, EVD overflow for the instance of IA).
- uDAPL delivers common asynchronous errors to the Event Dispatchers for asynchronous errors for all open instances of the Interface Adapter (for example, IA was disconnected).

c) (Nonrequirement) uDAPL Provider can define a default asynchronous error handler. which is “out of scope” for the current uDAPL Consumer. This is an assurance to the Consumer that there is a handler for asynchronous errors, but that its handler is not in an address space where this Consumer can access it.

i) The Provider default asynchronous error handler can be out of scope for a uDAPL Consumer.

ii) The Provider default asynchronous error handler can be in kernel space.

iii) All asynchronous errors specific to the Consumer open instance of an Interface Adapter shall be delivered to the default asynchronous error handler if the Consumer requested default asynchronous error handler.

d) uDAPL defines a uDAPL Provider library-independent method for the registration/deregistration of the uDAPL Provider Library and for discovery and enumeration of all DAT-capable Interface Adapters on a host:

i) DAT Collaborative provides APIs, source code, and headers for registration and deregistration of the uDAPL Provider library and enumeration of Interface Adapters:

- uDAPL Provider library registration, deregistration, and Interface Adapter enumeration can be platform (OS) dependent.
  - uDAPL Provider library registration consists of naming a Provider-specific library and creating an entry in the list of Interface Adapters on the host.
  - uDAPL supports uDAPL Provider library dynamic registration, deregistration, and Interface Adapter discovery and enumeration, even if a Provider library is statically linked.
- There can be, at most, one uDAPL Provider library registration, deregistration, and Interface Adapter discovery and enumeration code running on a host.

- A provider-independent, OS specific, registry library implements provider-independent portions of *dat_ia_open* for Consumers.
  - The registry executes on a per-process basis.
  - The registry loads Providers into memory and initializes them for each Interface Adapter (IA) Name opened by a Consumer.
  - The registry does not load Providers into memory or initialize them until requested to do so by a Consumer.
  - The registry utilizes a system-wide configuration file, known as the static registry, to determine which Provider library to load given a supplied IA Name.
  - The static registry must be editable by the System Administrator, independent of Provider installation or uninstalls.
  - The static registry specifies a default Provider to load given an input key of the IA Name, the API version, and whether thread safety is required.
  - To support testing, the static registry must allow other versions of a Provider to be entered, and must allow for an override to specify an alternate Provider to be selected for a given key.
  - All code within a Consumer must be compiled with the same version of the DAT API. The static registry rejects *dat_ia_open* calls with conflicting version information.
  - The registry must support Providers compiled with different versions of the dat.h. This includes both versions that are both earlier and later than the Consumer's version. This allows for separate distribution of Consumer executables and Provider libraries.
- The registry can be OS-dependent. The location of the static registry can vary by OS, as can the method of dynamically loading a Provider library.
- uDAPL Registration of DAT Provider APIs is out of scope of the uDAPL Provider Library.

e) uDAPL supports Polymorphic interfaces without explicitly specifying an Interface Adapter for each API function call.

i) The uDAPL Provider library has a function table for uDAPL defined APIs:

- The function table can include a pointer to the Provider or transport-specific extensions:

- This ensures that multiple vendor extension definitions do not collide on a host.

ii) Each uDAPL Provider library-created Object for which the Consumer has a handle has a pointer into the Provider library function table.

iii) The DAT Collaborative provides platform-independent macros that redirect uDAPL function calls to the proper Provider library function table:

- uDAPL Providers shall include these macros with their uDAPL Provider library.

4) Each uDAPL Consumer on a host opens the uDAPL Provider library.

a) Multiple uDAPL Consumers share the same address space, name space, and uDAPL Provider libraries.

b) The uDAPL Provider library can be shared between multiple uDAPL Consumers.

c) The uDAPL Provider library (Interface Adapter) can be opened and closed multiple times.

5) uDAPL supports multiple Interface Adapters on a host:

a) uDAPL Consumers shall share memory protections for multiple host processors and memory.

b) uDAPL shall provide OS ability to schedule uDAPL Consumer processes/threads on any available processor according to OS policy.

6) (Nonrequirement) uDAPL is not required to support Reliable Datagrams:

a) The DAT Collaborative will revisit Reliable Datagrams at a later time.

7) (Nonrequirement) The uDAPL Provider Library is not required to be thread safe:

a) If the platform convention is to be thread safe, the Consumer and the uDAPL Provider Library adhere to it.

b) uDAPL shall support the capability for Provider vendors to deploy both thread-safe and non-thread-safe libraries.

i) Vendors can use a naming convention for the Provider libraries that indicates whether the library is thread safe.

c) uDAPL shall support queryable Provider attributes that indicate whether a uDAPL provider library is thread safe.

d) uDAPL vendor shall document whether the Provider library is thread safe.

8) uDAPL shall provide support for multiple DAT Consumers using the same DAT Provider:

   a) uDAPL shall support passing connection among multiple uDAPL Consumers of the same DAT Provider

      i) uDAPL shall support Connection handoff (see Section 5.2, “Connection Management,” on page 34).

      ii) (Nonrequirement) uDAPL is not required to support Connect Request Redirection.

   b) (Nonrequirement) uDAPL is not required to support Object Sharing

      i) There is a need to be able to pass control of connected and active Endpoints between different uDAPL Consumers. The DAT Collaborative will revisit Object Sharing at a later time.

   c) (Negative Requirement) uDAPL shall not provide Process inheritance of DAT object for DAT Consumers

      i) A child process does not inherit any DAT objects over fork.

      ii) A child process shall not use any DAT handles of the parent process unless uDAPL specifically allows sharing of the handle of that object type between DAT Consumers.

      iii) Neither parent nor a child process shall crash over *exec* due to the opening or closing of a DAT library by a child, sibling, or parent.

      iv) A child process can become a DAT Consumer (uDAPL/kDAPL) of the same or different DAT Provider without any effect on the parent (with the potential exception of sharing underlying Provider and IA resources).

9) When a Consumer process terminates abnormally (without the chance to release the resources it was using), the uDAPL Provider shall free up the kernel resources and hardware adapter resources that were in use by that process.

   a) (Negative requirement) uDAPL Provider is not required to reclaim kernel and Interface Adapter resources immediately when the Consumer process crashes.

   b) uDAPL Provider should reclaim kernel and Interface Adapter resources of the abnormally terminated DAT Consumers before returning an “insufficient resources” failure.

      i) uDAPL Providers should reclaim resources during the Interface Adapter open and close operations.

ii) (Nonrequirement) uDAPL Providers are encouraged to reclaim resources (clean-up and garbage collection) at the finer granularity than only at the opening and closing of an Interface Adapter.

iii) Insufficient Resources error can be transitory. In addition to the time delay for the collecting resources of abnormally terminated DAT Consumers, the resources can be in use by other applications and DAT Consumers, or in the process of being cleaned up by normally terminating applications and DAT Consumers that use the shared Interface Adapter or Provider.

c) (Nonrequirement) uDAPL Provider is not responsible for identifying abnormally terminated DAT Consumers:

i) uDAPL Provider shall rely on the host Operating System for identifying abnormally terminated DAT Consumers.

ii) uDAPL Provider is not responsible for releasing resources that are the responsibility of the host OS to release.

d) When terminating normally, it is recommended that Consumer processes release all resources themselves. Consumers shall not rely on the Provider's abnormal-termination cleanup capabilities, because they might not clean up immediately or completely.

e) Upon process termination completion (as defined by the OS), all uDAPL process resources must be recovered by the uDAPL Provider:

i) Upon process termination, no access to the terminated process memory and other resources are allowed locally or remotely:

- To minimize the time window for remote accesses to terminating process memory, uDAPL implementation is encouraged to transition all Endpoints from the connected to the error state first, before doing resource recovery.
  - Consumer should not assume that the terminating process memory, including the shared one between the terminating process and others, is not modified by remote host via RDMA or previously posted local Recvs.
- The DAT Provider must work with the Host OS to ensure that all pending DTOs and RMR Contexts that enable access to consumer memory are invalidated or otherwise disposed of before process termination is completed. There must be NO possibility that a stale DTO or RMR Context enables access to physical memory that was reassigned to a new purpose.

10) Provider memory allocation should not interfere with Consumer memory allocation.

    a) Provider should avoid any memory allocation/deallocation in the operations on the performance path.

    b) Provider should not do memory allocation from the OS memory heap that can be used by Consumer.

11) When a Consumer thread terminates abnormally or encounters a synchronous or asynchronous exception the Consumer may recover uDAPL resources by closing the interface adaptor handle (IA handle) associated with the uDAPL objects created by this thread. Providers guarantee cleanup after thread death or exception only when the consumer guarantees that DAT calls referring to the IA handle and its associated objects are serialized (only one DAT call on the IA handle or any of its descendent handles is active at a time).

    Providers must make the following guarantees:

    a) successful completion of the *dat_ia_close* operation indicates operations on uDAPL objects associated with different IA handles may continue unaffected.

    b) A return code of *DAT_INTERNAL_ERROR* from *dat_ia_close* indicates that unrecoverable damage to the provider has been detected and the Consumer must resort to process termination as described in 9) on page 32 to recover uDAPL resources.

12) uDAPL supports OS-independent and OS-dependent ways to unblock a waiter

    a) uDAPL Provider shall unblock a waiter (EVD or its associated CNO) when Software event is posted to the EVD.

    b) uDAPL Provider should unblock a waiter (EVD or its associated CNO) when an exception (interrupt or signal) happens.

        i) (non-Requirement) uDAPL Provider does not provide any guarantee for handling a race condition between exception occurence and wait on the EVD or associated CNO, or arriving of an event on EVD.

13) uDAPL defines which uDAPL calls are signal handler and exception handler safe.Provider can provide support for kDAPL APIs to privileged user mode Consumers.

14) A separate specification documents may define Transport-specific DAPL extensions.

5.2 CONNECTION MANAGEMENT

1) DAT supports connection management between a uDAPL Consumer and a remote Consumer of an RDMA Transport.

a) A remote Consumer can be uDAPL Consumer, kDAPL Consumer, or another application of the same RDMA Transport.

2) (Nonrequirement) uDAPL is not required to support synchronous functions for connection management:

a) If a uDAPL Consumer wants to block, it must block on the designated connection Event Dispatcher or the Consumer Notification Object (CNO) the Event Dispatcher triggers.

b) The DAT Collaborative will revisit synchronous connection management at a later time.

3) uDAPL supports the Connection Management API for two-stage connection establishment:

a) Stage 1: The active side requests a connection.

b) Stage 2: The passive side “listens” for Connection Requests and accepts, rejects or hands them off.

c) (Nonrequirement) uDAPL does not expose the “ready-to-use” (RTU) message from the active to passive side:

- Both active and passive sides are notified of the connection establishment completion by delivering the Connection Establishment Completion event to the Event Dispatchers specified by the Consumers on each side.

4) DAPL supports socket-like transport-independent connection management

a) Stage 1: The active side requests a connection from Endpoint that has the parameters required for a socket for connection set-up.

i) domain

ii) type

iii) protocol

iv) IP Address

v) port

b) Stage 2: The passive side “listens” for Connection Requests on the Service Point that has parameters required for a socket to listen on and either accepts or rejects the Connection Request

c) Both active and passive sides are notified of the connection establishment completion by delivering the Connection Establishment Completion event to the Connection Event Dispatchers specified by the Consumers for the Endpoints that got connected on each side.

5) Endpoint management:

a) uDAPL provides the API for Endpoint creation and destruction.

b) (Nonrequirement) Endpoint creation does not require binding it to an Interface Adapter port.

c) uDAPL binds an Endpoint to an IA port when a connection is established on the Endpoint.

6) Active/passive connection establishment model:

a) uDAPL provides the API for active-side explicit Endpoint creation (not bound to an IA port).

b) (Nonrequirement) uDAPL is not required to support active-side implicit Endpoint creation.

c) uDAPL provides the API for passive-side explicit and implicit Endpoint creation (not bound to IA port).

d) uDAPL provides the API for passive-side support to listen for a Connection Request on a Connection Qualifier–Service Point Object.

e) (Nonrequirement) uDAPL is not required to provide support for active-side canceling of pending Connection Requests.

f) uDAPL supports the following attributes for active-side requests for a connection:

- Explicit Local Endpoint
- Either
  - Remote host Interface Adapter address and Remote host Connection Qualifier,
  - Remote IP Address, Port and Protocol
- Private Data
- Timeout

g) If the timeout expired prior to connection establishment, uDAPL indicates it to the Consumer as follows:

i) For the active side of the Connection Request, uDAPL generates an event to the connection event's Event Dispatcher of the Local Endpoint, requesting a connection that indicates that the requested connection has not been established due to the timeout expiration.

ii) (Nonrequirement) uDAPL is not required to notify the passive side of the connection establishment at the time the timeout expires.

iii) uDAPL keeps the pending Connection Request instance (passive side) for the requested connection valid until the Consumer accepts or rejects the pending Connection Request, the Consumer closes the Interface Adapter instance, or it fails.

- (Nonrequirement) uDAPL is not required to notify the Consumer on the passive side of the Connection Request of the timeout expiration when the passive side Consumer rejects the pending Connection Request, closes the Interface Adapter, or fails.
- uDAPL notifies the passive side Consumer of the timeout expiration when the Consumer accepts the pending Connection Request. The accept operation for the pending Connection Request returns success and destroys the pending Connection Request. The Provider generates a Connection Completion Error that indicates that the connection establishment cannot be completed.

h) (Negative requirement) uDAPL ensures that a failure to establish connection for any reason should have no effect on the posted Data Transfer Operations.

i) uDAPL supports Private Data size of at least 64 bytes:

   i) Providers can support a larger transport-agreed, specific Private Data size:

      - Transport-specific private data size is documented by the Provider and must be specified in the Interface Adapter Private Data Size parameter.
      - All Providers for that Transport are required to support private data size of at least the Interface Adapter Private Data Size parameter.

j) (Nonrequirement) uDAPL is not required to provide support for binding Local Endpoint to a specific Interface Adapter port.

k) uDAPL provides the API for the passive-side Consumer:

   i) uDAPL provides the API for the passive-side Consumer to listen on a Connection Qualifier with a Backlog:

      - uDAPL maps the backlog to the size of an Event Dispatcher queue.

   ii) (Nonrequirement) uDAPL is not required to provide support for multiple listeners outstanding on the same Connection Qualifier:

      - uDAPL returns an error when the Consumer tries to create a Service Point on a Connection Qualifier that is in use by kDAPL, uDAPL, or any other interface or protocol for the Interface Adapter.

   iii) uDAPL provides support for consolidating Connection Request arrival notifications arrived on multiple Connection Qualifiers of an Interface Adapter into a single Event Dispatcher queue.

iv) uDAPL enforces that DAT Consumers can only get a Connection Request arrival notification from an Event Dispatcher.

v) uDAPL enforces that the Consumer gets, at most, one Connection Request arrival notification per Connection Request arrival.

- Connection Request arrival notification is delivered to, at most, one Event Dispatcher.
- There is, at most, one active CNO per Connection Request arrival notification.

vi) DAT Endpoints can only be connected using DAT APIs.

vii) uDAPL provides the API for the passive-side Consumer to accept a Connection Request with explicit/implicit Endpoints with Private Data.

- uDAPL provides a single operation for a Connection Request accept, regardless of explicit or implicit associated local Endpoints, and regardless of what type of Service Point the Connection Requests are delivered on.

viii) uDAPL provides the API for the passive-side Consumer to reject Connection Requests.

- (Nonrequirement) uDAPL is not required to provide support for Private Data for Connection rejection.
- DAPL supports providing Consumer private data for rejection for remote peer.
  - DAPL does not provide any guarantee that reject private data will reach remote peer.

ix) For iWARP transports that adhere to the MPA protocol the Consumer acceptance or rejection of the Connection Request is mapped into MPA reject frame.

- The rejection of the Connection Request is mapped into MPA rejection bit.
- The MPA frame rejection bit can only be used for peer reject.

l) Connection parameters

i) uDAPL Providers cannot change Endpoint and Connection attributes during connection establishment.

ii) uDAPL provides default QoS class of service (best effort).

iii) uDAPL provides the capability for Consumers to specify the transport-specific QoS.

iv) Active-side Connection Request calls have an attribute for specifying a multipathing request, if it is supported.

m) The passive-side uDAPL Provider can support redirect without any exposure of redirection in the uDAPL API or any participation by DAT Consumers:
   i) DAT Consumers are not involved in redirect of a Connection Request.
   ii) Active side has an alternative Connection Request call that specifies remote side as the existing connection, which means "duplicate this connection—connect to the same host/IA."
      – The duplicate connection call also duplicates the multipathing of the duplicating connection. Any fabric failures should effect both connections the same way: either both are up or both are down.

n) Connection Handoff
   i) Passive-side uDAPL Provider shall support Connection Request handoff:
      – uDAPL shall support connect request handoff between uDAPL Consumers to other uDAPL Consumers using the same uDAPL Provider on the same Interface Adapter on the same node.
      – A handoff specifies an alternate Connection Qualifier.
      – uDAPL shall provide a mechanism for Consumers to hand off Connect Requests by reposting these to the DAT Provider in association with a new Connection Qualifier.
      – The target Consumer (the Consumer to which the connection is being handed) shall receive the Connect Request on its Connection Qualifier as if it were received directly from the originator (the active side).

o) (Nonrequirement) uDAPL is not required to provide support for synchronous connection establishment.
   i) If a uDAPL Consumer wants to block, it must block on the designated connection Event Dispatcher or the CNO the Event Dispatcher triggers.

7) uDAPL provides the API for Disconnect:
   a) Passive and active sides can break a connection.
   b) uDAPL supports abrupt Disconnect.
      - Upon receiving a request for a Connection termination, the uDAPL Provider breaks the connection and completes all outstanding and in-progress DTOs with an error if they were not previously completed successfully before notifying the DAT Consumer about breaking the Connection.
   c) uDAPL supports (fenced) Graceful Disconnect:

- Upon receiving a request for a graceful Connection termination, the uDAPL Provider does not accept any new—and completes all outstanding and in-progress—Send, RDMA Read and Write DTOs, and RMR Binds before starting the Disconnect and notifying the DAT Consumer about breaking the Connection.

8) uDAPL supports the following Connection events:

   a) Connection Establishment Completion event to indicate Connection establishment completion (both active and passive sides):
      - Passive-side Private Data of the Connection acceptance is passed to the Consumer in the Connection Establishment Completion event.

   b) Connection Request Arrival Notification event to indicate the Connection Request arrival for the passive side:
      - Active-side Private Data of the Connection Request is passed to the Consumer in the Connection Request Arrival Notification event.

   c) Peer Connection Rejection Arrival Notification event to indicate the connection rejection for the active side.

   d) Non-Peer Connection Rejection Arrival Notification event to indicate the non-peer Connection rejection for the active side. This includes all reasons for not establishing the Connection, except timeout and peer reject. For example, remote host is not reachable, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors.

   e) Connection Completion Error Notification event to indicate the inability to complete the Connection establishment for the passive side. This notification is returned in response to the passive Consumer accepting the Connection Request. Examples of the cause of this error are Transport errors and timeout expiration on the active side.

   f) Disconnect Completion Notification event to indicate that a Connection is broken by a Disconnect Request for both the requested side and target side.

   g) Broken Connection event to indicate that a connection is broken and it is not due to the Consumer Disconnect on either side of the connection.

   h) Timeout Expired event to indicate that the timeout for the Connection Request expired before Connection establishment. This event is only possible on the active side of the connection establishment.

i) Remote host Unreachable event to indicate that Provider cannot reach remote host or that remote host does not respond to Connection Request.

9) DAPL supports the following Reliability model:

a) All DTOs submitted to the DAPL Provider are completed successfully in the absence of errors, with data delivered uncorrupted, in the order defined by the ordering rules.

b) Corruption of the data delivered to the DAT Consumer is detected as an error and reported to the Consumer.

c) Data loss (inability to deliver data to the remote Endpoint of the Connection, or to the local Endpoint for RDMA Read) is detected as an error and reported to the Consumer.

d) Upon detection of an error, the Connection is broken and all outstanding and in-progress DTOs are completed with an error.

e) There is a one-to-one correspondence between send DTOs on one Endpoint of the Connection and Recv DTOs on the other Endpoint of the Connection.

f) There is no correspondence between RDMA DTOs on one Endpoint of the Connection and Recv or Send DTOs on the other Endpoint of the Connection.

g) DTO Completion means that the Consumer can reclaim local resources associated with the DTO, including a local buffer that was specified for the DTO.

h) Delivery Ordering Rules:

   i) The data payload for the send DTO matching a receive DTO is delivered into the receive-indicated buffer memory prior to the receive DTO completion.

   ii) Receive DTOs on a connection are completed in the order of posting of their corresponding sends.

   iii) Each RDMA write DTO posted on a connection prior to a send DTO posted to the same connection has its data payload delivered to the memory specified by RMR Context and RMR Target Address of the RDMA Write DTO prior to the completion of the Receive DTO matching that send.

i) Completion Ordering Rules:

   i) The data payload of a DTO is delivered into the receive- or RDMA-indicated buffer prior to the DTO completion.

   ii) All Send and RDMA Write DTOs posted to a connection are completed in the order posted.

   iii) RDMA Read DTOs posted to a connection are completed in the order posted.

iv) RDMA Read DTOs can be completed out of order with respect to Send and RDMA Write DTOs posted to the same connection.

v) All Recv DTOs posted to a connection are completed in the order posted.

vi) No order relationship between completions of Recv DTOs and all other DTOs on the same connection.

vii) All Send, RDMA Read and RDMA Write DTO completions on a connection generate DTO completion Events into the same "Event Stream."

viii) All Recv DTO completions on a connection generate DTO completion Events into the same "Event Stream."

j) DTO Processing Rules:

i) All Send, RDMA Read, and RDMA Write DTOs posted to a connection start being processed in the order posted.

ii) All Recv DTOs posted to a connection start being processed in the order posted.

iii) There can be multiple outstanding DTOs of the same or different type on the same connection.

iv) If the Fence is specified for a Send, RDMA Read, or RDMA Write DTO, that DTO (and all following DTOs) cannot start being processed until all previously posted RDMA Read operations are completed.

5.3 DATA TRANSFER OPERATIONS INITIATION

uDAPL provides the API for DTO initiation with the following characteristics:

1) uDAPL supports the following Buffer Representations for DTOs:

a) The Local Buffer is represented by a Consumer IOV (I/O Vector) with each element being a data segment represented by an LMR_ triplet:

- LMR Context
- LMR Virtual Address
- Length

b) uDAPL traverses the Local Buffer in the following logical order:

i) Data segments are in the order defined by IOV.

ii) Each data segment is traversed from the start of the segment in linear order.

iii) There is, at most, one data segment (A) of the Local Buffer whose data is partially transferred/filled.

iv) Data of all data segments preceding A in the IOV order are fully transferred/filled.

v) (Nonrequirement) uDAPL is not required to touch any of the data segments following A in the IOV order.

vi) The order data transfer in/out of a memory buffer is not specified:

- Transfer of all DTO data is finalized prior to DTO completion.

c) IOV in a posted DTO is under uDAPL Provider control; Consumers cannot modify its content or the memory it refers to until they get control of it back:

i) The Provider must return control of IOV to the Consumer when the DTO it is specified for is completed.

ii) The Provider can return control of IOV to the Consumer at the return of the post DTO:

- The Provider documents and indicates through the Provider Attributes its support for this behavior.

d) Remote Buffer is represented by a single RMR_triplet:

- RMR Context
- RMR Target Address
- Length

e) The result of the RDMA DTO accessing remote memory that is being accessed by its local Consumer is not defined and the content of any remote memory accessed by the RDMA DTO is also undefined:

i) Coherency between operations on local memory and RDMA DTO operations on the same memory is defined by the local host system architecture.

f) uDAPL supports byte alignment for local and remote DTO buffers:

i) In any case, uDAPL Providers shall document a hint about what the optimal alignment is for DTO buffers for system performance per platform.

- (Advice) uDAPL Providers are advised to keep the “optimal alignment hint” as close to the byte alignment as possible and are recommended to have it smaller than the uDAPL-1.0-defined “optimal alignment hint” constant.
- Providers shall provide the “optimal alignment hint” as an attribute of Provider.

- (Advice) uDAPL Consumers are advised to keep DTO buffers aligned to the “optimal alignment hint” constant boundary for the portable performance characteristics across multiple Providers.
- For uDAPL-1.0, the “optimal alignment hint” constant is 256 bytes.

g) (Nonrequirement) uDAPL does not define explicit Scatter/Gather List DTO Objects, and uDAPL Providers are not required to support SGL Objects.

h) DAPL shall expose maximum IOV segments for local buffers for Send/Recv, RDMA Read, and RDMA Write as separate IA attributes.

i) (Nonrequirement) uDAPL is not required to support passing parameters for DTO initiation and DTO completion in any way other than in-line.

j) uDAPL allows data segments to overlap on the “send” side of a DTO.

k) Behavior for overlapping data segments for the “recv” side of a DTO is undefined.

l) For send, recv, and RDMA initiator/local side DTOs, uDAPL supports more than a single data segment (number of IOV elements).

m) (Nonrequirement) uDAPL does not define the API for canceling posted DTOs, and uDAPL Providers are not required to support it.

n) DAPL shall support an RDMA Read specifying a local data sink using an RMR Context rather than an LMR Context.

   i) DAT shall define a Provider attribute to indicate when use of the RMR Context as the sink of an RDMA Read is required to prevent exposing the RMR Context of the LMR Context to the wire.

   ii) DAPL shall support an RDMA Read specifying a local data sink using a single RMR Context.

   iii) (Nonrequirement) DAPL does not define the API for RDMA Read specifying a local data sink as a collection of an RMR Contexts.

2) uDAPL supports send DTOs.

3) uDAPL supports receive DTOs.

4) uDAPL supports RDMA Read DTOs.

5) DAT Provider shall support RDMA Read flow control.

a) The number of RDMA Read in progress simultaneously over a connection shall not exceed Consumer-requested EP attribute value.

6) uDAPL supports RDMA Write DTOs.

7) Multiple RDMA operations can be in progress simultaneously even over the same connection:

   a) The result of multiple RDMA DTOs accessing the same remote memory simultaneously is not defined.

8) DAPL should support Shared Receive Queues

   a) (Negative requirement) If IA does not support SRQ then DAPL Provider should not emulate it.

      i) SRQ support is intended to reflect support from the underlying HCA/RNIC.

   b) DAPL Provider shall document and report through Provider attribute if it supports SRQ.

      i) (Negative Requirement) All specific SRQ features are also optional to implement. DAPL does not require an “all or nothing” implementation of SRQ features.

      ii) DAPL Provider shall document and report through Provider attributes all the SRQ options it supports.

   c) An SRQ buffer can be dequeued by any Endpoints that uses SRQ. DAPL does not provide any guarantee on the order of SRQ Recv buffer de-allocation by Endpoints

      i) All buffers in SRQ should have the same IOV size and buffer length.

         – (Nonrequirement) DAPL Provider is not required to check that all posted Recv buffers have the same IOV size and length.

      ii) (Nonrequirement) DAPL provides no API to allow an Endpoint to seek an optimally sized buffer based on a size of the actual received message.

   d) DAPL shall provide a method to create an EP associated with an SRQ.

   e) (Nonrequirement) DAT does not provide any method to disassociate an EP from an SRQ.

      i) EP destruction disassociates EP from SRQ.

      ii) DAPL shall not allow an SRQ to be destroyed while it is still referenced by an EP.

- The implicit destruction due to abrupt *dat_ia_close*, for example implicitly for process termination, will destroy an SRQ and all its associated Endpoints.

f) DAPL shall provide a method allowing Consumers to post Recv buffers directly to the SRQ rather than to the EP, when the EP was created with an SRQ.

   i) (Negative Requirement) DAPL shall not allow a Recv buffer to be posted to an EP that is associated with a SRQ.

g) DAPL shall provide a method for the Consumer to receive notification when the number of available buffers on an SRQ falls below a Consumer-specified “low watermark” threshold.

   i) DAPL Provider support of “low watermark” threshold is optional.

   ii) (Negative Requirement) DAPL shall not define any method for permanently enabling a low watermark threshold in order to prevent the Consumer from being swamped with low watermark events.

   - Each “low watermark” threshold arming shall generate at most one asynchronous (warning) event on IA asynchronous EVD.
   - DAPL shall provide the Consumer an explicit method for arming the “low watermark” threshold.

h) DAPL shall define a method for the Consumer to guard against any single Endpoint holding too many uncompleted Recv buffers.

   i) Without such a guard a remote peer could drain the Shared Receive Queue of available buffers, potentially causing an error on other Endpoints associated with the same SRQ.

   ii) DAT shall provide a “soft high watermark” for local interfaces that support generating an asynchronous warning event to the Consumer when an Endpoint's span of allocated buffers exceeds the Consumer-specified threshold. If supported, detected violations will be reported as asynchronous events to the asynch_evd for the Consumer to act upon.

   iii) DAPL Provider support of “soft high watermark” threshold is optional.

   iv) (Negative Requirement) DAPL shall not define any method for permanently enabling a low watermark threshold in order to prevent the Consumer from being swamped with soft high watermark events.

   - Each “soft high watermark” threshold arming shall generate at most one asynchronous (warning) event on IA asynchronous EVD.

- DAPL shall provide the Consumer an explicit method for arming the soft high watermark threshold.

i) DAT shall support “hard high watermark” that supports automatic termination of a connection that exceed the Consumer- specified threshold.

  i) DAPL Provider support of “hard high watermark” threshold is optional.

j) (Nonrequirement) DAPL does not define an algorithm that is required to be used by Provider to allocate buffers from a SRQ when messages are received out-of-order.

k) DAPL shall provide a method that allows the Consumer to query the number of buffers allocated for each Endpoint, and the span of message numbers that those buffers would represent if the messages are to be numbered sequentially by the sender.

  i) The support for query is optional.

l) All buffers in SRQ must have the same Protection Zone.

m) DAPL shall support “split PZ” mode operation with SRQs so the SRQ associated Endpoints can have different PZs, while using a common PZ for the buffers to be placed in the SRQ.

  i) DAPL shall allow different RMRs to be valid on SRQ associated Endpoint

  ii) In order to use “split PZ” Consumers must use a different LMR to Send, RDMA Write that matches the EP than to Receive that must match the SRQ.

  iii) The “split PZ” support is optional.

n) (Nonrequirement) DAPL does not define a method of discharging buffers from an SRQ other than transferring them to an Endpoint upon message arrival or deleting the entire SRQ.

o) DAPL shall provide a method for the Consumer to request resizing of a SRQ.

  i) The Provider must ensure that SRQ resizing do not result in the loss of any buffers currently assigned to the SRQ.

p) The default attributes of all DAT objects effected by SRQ functionality is identical to SRQ support not being present. This ensure backwards compatibility to pre-DAPL-1.2 Consumers.

q) DAPL shall support return of SRQ specific attributes of EP for *dat_ep_query* when EP is associated with SRQ.

9) uDAPL provides Consumer support for specifying a “cookie” per DTO initiation.

10) Prior to Connection establishment, uDAPL only allows Receive operations to be posted to a local Endpoint:

   a) uDAPL does not allow send, RDMA Read, or RDMA Write operations to be posted to a local Endpoint prior to Connection establishment.

11) (Nonrequirement) uDAPL does not define semantic nor the API for Atomic operations and uDAPL Providers are not required to support it.

   a) IB specific DAPL extension can define API and semantic for Atomic operations.

12) (Nonrequirement) DAPL does not define semantic nor the API for RDMA Write with Immediate data operations and DAPL Providers are not required to support it.

   a) Transport-specific DAPL extension can define semantic and API for RDMA Write with Immediate data operations.

13) uDAPL supports Notification Suppression for completion event of DTOs, RMR and LMR asynchronous registration and invalidation.

14) uDAPL-1.1 and above supports per Endpoint for Receive Completions:

   a) Locally controlled Notification Suppression.

   b) Remotely controlled notification suppression via matching Send Solicited Wait.

   c) (Nonrequirement) Provider is not required to support both a) and b) simultaneously.

15) uDAPL can support Solicited Wait:

   a) A uDAPL Provider that does not support Solicited Wait natively in the Transport does not emulate it:

      i) A uDAPL Provider documents and reports though Interface Adapter Provider attributes if it supports Solicited Wait.

16) uDAPL supports Completion Suppression.

   a) The Consumer is required to provide a “cookie” even for DTOs with Completion Suppression.

17) uDAPL supports Barrier Fence.

18) uDAPL supports uDAPL Consumer-provided “cookies” per DTO initiation:

   a) The Consumer “cookie” is opaque to the Provider:

      i) uDAPL does not rely on the uniqueness of the Consumer provided “cookie” per DTO.

b) uDAPL supports a single type definition for Consumer "cookies" that can support the following:

   i) Index/integer

   ii) uDAPL Consumer handle

   iii) Pointer

   iv) 64 bits

5.4 DATA TRANSFER OPERATION COMPLETIONS

1) uDAPL provides support for consolidating DTO Completion notifications for DTO invocations submitted to multiple RQs of the same or different connected Endpoints into a single queue (Event Dispatcher Queue).

2) uDAPL generates a Completion event per posted DTO:

   a) uDAPL provides the API for the Consumer to indicate that a Provider does not generate the successful Completion notification for the posted DTO (suppressing DTO Completion notification).

3) uDAPL enforces that the uDAPL Consumer gets, at most, one Completion notification for a DTO:

   a) DTO Completion notification is delivered to, at most, one Event Dispatcher.

   b) There is, at most, one active CNO for the Completion notification of a DTO.

4) uDAPL enforces that uDAPL Consumers can only get DTO Completion notifications from Event Dispatchers.

5) uDAPL returns in the DTO Completion event the Consumer "cookie" specified by the Consumer at the DTO initiation.

6) uDAPL shall provide support for consolidation of completeness across multiple DAT Providers only through the use of OS Wait Proxy Agent representing a shared OS-specific synchronization mechanism.

7) uDAPL should enforce that there is, at most, one CNO active per Completion event. Providers are encouraged to mask false wakeups.

8) DAPL-2.0 and above shall provide in Completion Event for DTO the operation type of completed DTO.

5.5 MEMORY MANAGEMENT

uDAPL provides the API for Memory Management with the following semantic characteristics:

1) All memory registrations are per Interface Adapter.

2) The uDAPL Consumer must ensure that all its Interface Adapter registered memory is accessible by the Interface Adapter it opened.

3) All uDAPL Objects' memory is owned by the uDAPL Provider:

    a) Opaque handles for all uDAPL Objects: RMR, LMR, IA, Endpoint, Service Point, Protection Zone, Event Dispatcher.

    b) Event streams memory owned by the Provider.

    c) All queues (or their simulations for Consumers) are owned by the Provider and are opaque to Consumers. Consumers cannot reorder or delete entries or even see the queues (with their ordering) at all:

        i) (Nonrequirement) Actual queues are not required for uDAPL Providers.

        ii) Consumers can post entries to some queue types through methods on an Object that encompass the queue:

            – The Consumer can post Software events to an Event Dispatcher.

            – The Consumer can post DTOs to an Endpoint:

                – The Consumer can post send, RDMA Read, and RDMA Write DTOs to a connected Endpoint only.

        iii) Consumers can take entries from some queue types through methods on an Object that encompass the queue:

            – The Consumer can dequeue events from an Event Dispatcher.

        iv) uDAPL provides support for resizing the Event Dispatcher queue:

            – (Nonrequirement) uDAPL is not required to provide any visibility for Consumers into new allocated resources or extended current ones for the resized queue.

            – uDAPL preserves the content of the resized queue.

4) uDAPL provides the API for the following Memory Registration:

    a) LMR for local access by the uDAPL Provider:

        i) uDAPL provides support for enabling and disabling per LMR instance:

            – Local Read access

            – Local Write access

        ii) uDAPL provides support for memory protection specification per LMR instance - Protection Zone attribute.

iii) (Negative Requirement) uDAPL shall not support any mechanism for a Consumer to register memory that it does not already have validly mapped under the host OS and DAT memory privileges.

iv) DAPL supports Consumer ability to use as virtual address for the registered LMR either:

- process Virtual Addresses
- memory region 0-bazed addresses

v) uDAPL supports the Consumer capability to create multiple LMRs referencing the same memory:

- The same virtual addresses
- Registered by another LMR

vi) uDAPL shall enable the Consumer to register memory shared with other uDAPL Consumers of the same uDAPL Provider. The Provider should avoid duplication of hardware resources when registering memory shared by multiple uDAPL Consumers.

- uDAPL Consumer shall open the same Provider library of the Interface Adapter that is identified by the common IA_Name.
- (Negative Requirement) uDAPL is not required to support different Consumers reporting varying sizes for the same shared region.
- The shared region shall be identifiable by a unique Consumer cookie.
- uDAPL Consumers that share a memory region should register that memory on the host using platform-specific methods outside the DAT.
  - uDAPL Provider can rely on the uDAPL Consumer registering the shared memory region on the host so the Provider can register the physical memory for that region with an Interface Adapter once for all DAT Consumers sharing the region.
  - The shared memory region can have different Protection Zones and different DAT Consumers.
- uDAPL shall support multiple Consumers identifying the same shared region by the unique cookie, with identical sizes, without requiring the Consumers to apriori designate which one is actually the first to register the region (peer-to-peer shared memory model).
- (Nonrequirement) Different uDAPL Consumers are not required to have the same virtual address for the shared region.

- uDAPL Providers can use shared objects to fulfill shared memory requirements.
- uDAPL Providers can clone the original LMR.
  - Sharable LMRs shall not be modifiable even if non-shared LMRs are modifiable.

vii) DAPL supports platforms with non-coherent memory in areas between
- cpu cache and memory
- I/O controller cache and memory

viii) DAPL shall enable the Consumer to preserve a consistent view of memory content on platforms with non-coherent memory.
- DAPL Providers shall inform Consumers through Provider attribute values that synchronization is required for RDMA Read and Write.
- if proper synchronization step is not performed by the Consumer, data coherency becomes undefined.
- I/O controller cache and memory

ix) DAPL shall support remote invalidation of LMR and RMR
  - Invalidation requires that the remotely invalidated LMR or RMR can not be used for remote RDMA operations
  - (Non-Requirement) Remote invalidation does not require that the underlying memory of LMR or RMR is no longer mapped or pinned by the Provider

b) Remote Memory Region (RMR) within an LMR for the RDMA operations:

i) uDAPL provides support for enabling and disabling per RMR instance:
- RDMA Read
- RDMA Write

ii) DAPL enables Consumers to scope memory protection specification for RMRs with one of the following:
- Protection zone based
- Endpoint based

iii) DAPL Provider must support at least one type of RMR protection:
- Protection zone based
- Endpoint based

iv) DAPL Provider must specify via Provider attribute which of RMR protection scope types it supports.

v) uDAPL provides (lightweight) operation for binding RMR within its associated LMR Context:

- Bind of RMR can change RMR's memory region within the LMR context.
- Bind of RMR creates an RMR Context.
- Bind of RMR makes previous RMR Contexts of the RMR invalid.
- Binding of RMR is an asynchronous operation.
    - Completion notification of an RMR Bind is reported as an event of an Event Dispatcher of an Endpoint to which RMR Bind is posted to.
    - A new RMR Context is returned synchronously and makes previous RMR Contexts invalid.
    - A failure of the asynchronous part of the RMR Bind breaks the connection of the Endpoint it was posted to.
    - RMR Bind is a barrier operation for the Endpoint Connection. uDAPL does not start any of the Send, RDMA Read, RDMA Write, or RMR Bind operations posted to the Endpoint after the RMR Bind until it is completed successfully.
    - The Consumer does not provide new RMR Context until RMR Bind is completed. The Consumer can use barrier behavior of the RMR Bind and post a DTO to transfer new RMR Context to a remote Consumer to the Endpoint to which RMR Bind posted to.
    - Consumer should not post an RMR bind of the same RMR until the previous one completes.
- RMR Bind must be lightweight:
    - Less overhead than memory registration for RMR (RMR creation).
    - Is suitable for per RDMA DTO basis use.

vi) uDAPL provides RMR Context for binded RMR that is suitable for sharing with a remote Consumer:

- uDAPL only allows RDMA operations to succeed if the local Endpoint of the connection of the RDMA operation and RMR of a valid RMR Context used for specification of RDMA DTO buffer have an identical Protection Zone attribute.
- uDAPL enforces that RMR Context is valid until the associated RMR has been rebound or destroyed.

- uDAPL supports the use of the same RMR Context for RDMA operations on multiple connections.
- A Connection is capable of supporting RDMAs with multiple RMR Contexts whose associated RMRs share the same Protection Zone.

vii) (Nonrequirement) uDAPL can support the uDAPL Consumer capability to create multiple RMRs referencing the same LMR.

c) (Negative requirement) uDAPL should not allow the following:

i) Interface Adapter to access memory outside the Consumer-registered one (remote Consumer to read or write into local Consumer memory on the physical pages portions of which have been registered by the Interface Adapter).

ii) Consumer access to Interface Adapter pages outside the portion of the pages registered for that Consumer.

5) DAT Provider shall expose RDMA Transport requirement for RDMA Write memory privilege for RDMA Read accessible memory.

5.6 ERROR DETECTION AND NOTIFICATION

uDAPL provides Error Detection and Error Notification.

1) uDAPL provides an Event Dispatcher for Asynchronous Error Notification associated with an Interface Adapter.

2) If a catastrophic error is reported through an Asynchronous Error Event Dispatcher, the behavior of the Interface Adapter after that is not defined:

a) uDAPL is required to support closure of the Interface Adapter even in the presence of a catastrophic error.

5.7 EVENT MODEL

uDAPL provides the API for the following Notification Model:

1) Ability to consolidate all notifications into a single queue (ordered event queue - virtual):

a) An Event Dispatcher Object that provides consolidation of DAT Notification events into a single ordered queue.

b) Support for multiple Consumer-created Event Dispatchers for the same Interface Adapter that can all work in parallel.

c) uDAPL supports the Consumer capability to specify the minimum length of the Event Dispatcher event queue.

i) uDAPL supports the Consumer capability to resize the Event Dispatcher queue:

- (Nonrequirement) uDAPL is not required to support the Consumer capability to shrink the Event Dispatcher queue such that existing events in it have to be dropped.

d) The order of events of an individual Event Stream is preserved by Event Dispatcher.

e) (Nonrequirement) uDAPL is not required to provide any ordering of events among multiple Event Streams of a single Event Dispatcher except the Event Streams corresponding to a single connection, as specified:

i) A Connection Establishment event precedes any DTO Completion events.

ii) All DTO Completion events that, for successful completions, should precede the Disconnect event.

iii) No order between DTO Completion events that are completed with an error and the Disconnect event.

f) To the extent that it is possible for the uDAPL Provider to efficiently determine the true time ordering of events on different Event Streams, it should preserve that order when dispatching events.

2) If the queue of an Event Dispatcher is full then DAPL shall generate an Event Dispatcher overflow error that is delivered to the Asynchronous Error EVD of the Interface Adapter

a) DAPL is allowed to report a single overflow error for multiple overflows of the same Event Dispatcher (sticky overflow error).

b) If the queue of the Asynchronous Error EVD of the Interface Adapter is full then the last reported error shall be Event Dispatcher overflow catastrophic error of itself.

3) Sticky CNO–persistent CNO

a) At most, one CNO can be registered per Event Dispatcher.

b) uDAPL shall allow Consumer to specify zero or one OS Wait Proxy Agent per CNO.

c) A CNO delivers a notice that an Event Dispatcher that feeds it has had a Notification Event, and the identity of one Event Dispatcher for which that is true.

i) (Negative requirement) A CNO does not have to identify all its feeding Event Dispatchers that had Notification Events.

ii) A CNO can consolidate multiple notifications that occur faster than it can unblock a waiter into a single notification.

iii) (Negative requirement) The CNO does not deliver events. Consumers must dequeue the events from the Event Dispatcher themselves.

iv) When there are multiple waiters on a CNO, at least one is awakened by a notification. uDAPL Provider has the freedom to choose a waiter for notification delivery.

v) There can be a short, nondeterministic time from reception of a notification and unblocking of a waiter. During this time, other notifications can be consolidated. During this time, a fresh call to *dat_cno_wait* can consume the notification.

d) The same CNO can be registered for multiple Event Dispatchers of the same Interface Adapter.

e) The CNO remains bound to an Event Dispatcher until it is explicitly unbound, by replacing it with another CNO or by setting CNO to NULL.

f) (Negative requirement) Association between a CNO and an Event Dispatcher is persistent and CNO does not have to be reregistered during each dispatch by the Consumer.

g) uDAPL shall provide a Consumer with an ability to get the next event from the Event Dispatcher queue.

4) All transport-specific interactions with the Event Stream for the event triggered CNO shall be completed prior to signaling the CNO or prior to the event being placed on the Event Dispatcher queue.

5) The scope of the Event Dispatcher is a single Interface Adapter:

a) The same CNO can be registered with multiple Event Dispatchers of the same Interface Adapter.

b) CNOs of different Interface Adapters can unblock a common OS Wait Proxy Agent.

c) uDAPL shall support associating the same OS Wait Proxy Agent with CNOs of the same or different Interface Adapters.

6) The Event Dispatcher cannot be destroyed or freed when it has Event Streams associated with it:

a) Event Dispatchers of an Interface Adapter are destroyed when the Interface Adapter is closed.

b) The uDAPL Consumer is not required to drain the Event Dispatcher queue prior to a destruction of the Event Dispatcher.

7) uDAPL supports the capability of the uDAPL Consumer to generate software notification events (Software events):

a) uDAPL does not allow the Consumer to generate or mask uDAPL events/errors/notifications:

- The Provider Library can support a capability to mask Software events as uDAPL events for the debugging library:

- If the Provider Library supports masking of Software events as uDAPL events, it defines a separate method on Event Dispatcher with a different prefix than defined for uDAPL or uDAPL operations.

b) (Nonrequirement) There is no ordering between Consumer-generated Software events posted to an Event Dispatcher and other events of the Event Dispatcher.

c) Software events form an independent Event Stream.

d) Software events are notification events.

e) Posting of a Software event cannot cause the Event Dispatcher queue overflow

i) An attempt to post a Software event that causes an overflow is reported to a Consumer synchronously and the Software event is not being posted to the Event Dispatcher.

ii) An attempt to post a Software event that causes an overflow for an Event Dispatcher does not generate the EVD overflow error and hence, is not reported on the Asynchronous Error Event Dispatcher.

8) uDAPL supports Threshold parameter for EVD waiters.

a) (Nonrequirement) Provider is not required to support Threshold and Notification Suppression for the same EVD simultaneously.

9) All non DTOs, RMR and LMR asynchronous registration and invalidation completion events are notification events.

a) Whether or not DTOs, RMR and LMR asynchronous registration and invalidation completion events are notification events or not is controlled by posted operation completion flags or for Recv operation Solicited Wait flag of matching Send operation.

10) DAPL supports the capability Extended Object to generate events (Extension events):

a) DAPL Extension can provide an ability to generate either notification or non-notification events

b) DAPL Extension define ordering between Extension events posted to an Event Dispatcher and other events of the Event Dispatcher.

i) DAT Provider attribute defines whether or not Provider supports merging Extended event stream with other event streams on an Event Dispatcher.

c) Extended events form an independent Event Stream.

d) Extension event data is passed as pointer to a buffer

i) Buffer must be accessible from the Consumer space

ii) Provider shall ensure that the buffer is accessible from Consumer space.

e) Posting of an Extended event may cause the Event Dispatcher queue overflow

i) Provider shall define the overflow behavior of the extension event stream.

5.8 NAME SERVICE

uDAPL relies on the following Name Service requirements:

1) A name service exists that translates host names to IP addresses, and vice versa.
2) DAT supports the IPv6 format.
    a) DAT allows the use of encapsulation of IPv4, GIDs, or other 128-bit transport addresses into IPv6.
3) The name must be consistent across all fabrics for local name resolution.
4) There is no other transport dependency or local Interface Adapter dependency.
    a) There is no requirement for IPoverIB.
5) Any nonlocal IP name must be reserved.
    a) Global IP name space must be respected.
6) DAT does not require any new name service mechanism on a platform.
    a) Consumer can use any existing host-provided name-service mechanisms and APIs that provide IPv6 resolution.
7) The naming assignment must be consistent. All name-resolution APIs must resolve the name and address information consistently on the platform for a given host.

5.9 HIGH AVAILABILITY (HA)

1) DAPL optionally supports High Availability
2) DAPL optionally supports two connection models
    a) single path model
    b) multipaths model
        i) (non requirement) DAPL does not require to expose 2 path or any specific number of paths connection model
3) DAPL optionally supports Active-Passive model of multipathing

4) DAPL Provider can provide load balancing to balance active path assigment for multiple connections that share the same underlying paths
    a) DAPL Provider attribute specifies whether or not load-balancing is supported
5) (non Requirement) DAPL does not provide support for non-A/P multipathing model
6) (non Requirement) DAPL does not gurantee HA across heterogeneous IAs
7) DAPL supports 2 models of High Availability
    a) Consumer level HA
        i) DAPL provides support for Consumer doing multipathing at the application level.
            – DAPL does not impose any restriction to the HA model Consumer want to do:
                – Hot Standby
                – Parallel Connections
                – Cold Stanby
                – New EP and connection creation upon a connection failure
                – disconnected EP is reconnected on failure
        ii) DAPL Provider exposes individual HW as IA
            – Each RDMA capable HW: HCA, RNIC
            – Each port of RDMA capable HW
            – DAPL provides an API for Consumer to find out whether 2 IAs are two ports of the same HCA, RNIC
                – DAT Registry supports a query which reports whether or not 2 IAs share resources.
        iii) DAPL Provider hides all redirection for dat_ep_dup_connect but guarantees that the new connection reaches the same destination and Connection Qualifier as the duplicated one got its connection request delivered
            – (Non Requirement) DAPL Provider does not guarantee load balancing between duplicated and new connection
            – (Non Requirement) DAPL Provider does not required to guarantee that original and duplicate connections share the same path(s)
    b) Provider delivered HA. DAPL defines the following Provider HA models:
        i) DAPL supports Service Persistent model

- IA_Address and Connection Qualifier are persistent across failures. The remote IA_address can be virtualized and represent different hosts, different IAs or different ports of the IA.

ii) (non-Requirement) DAPL does not have to support RMR_contexts persistency model

iii) (Non Requirement) DAPL does not have to support Session Persistency

- DAPL 2.0 does not define the concept of the session
- All EPs of a session failover together

iv) DAPL supports EP level consistency

- DAPL Provider maintains a connection
- DAPL Provider reports active path migration to a Consumer according to a) DAPL exposes the following abstract events on page 61

v) DAPL supports Transport level consistency

- All failures are handled by the underlying transport
  - For example, IB APM, IP SCTP provide path migration for Transport level connection persistency
- Transport level consistency guarantee an EP connection persistency but does not guarantee that the connection IA_address and Connection Qualifier will be persistent across faults

c) DAPL Provider HA model guaranties the following connection (EP) persistency

i) EP state is maintain in the presense of a fault

- A failure which does not result in loss of physical connection between local and remote hosts preserves the state of the EPs on both ends of the connection across the fault and potentially active path migration
- A failure of the last physical path connecting local and remote hosts results in the connection failure and EP transitioning into disconnected state

ii) Faults that do not result in path migration do not have any impact on any existing DAT objects with exception that some connections may transition from multipathed state to single path state.

iii) Path migration, including transport level path migration, does not impact posted, completed and in-progress DTOs

iv) Path migration guarantee that all LMRs, RMRs, LMR_contexts, RMR_context remain valid and operational.

        v) DAPL Provider guarantees the ordering of DTO completions and processing in the presence of path migration

            – DAPL provider guarantees that all RDMA operation complete remotely and are in remote Consumer memory before Recv completion that matches the Send that followed RDMA DTOs has been delivered to Remote Consumer.

    d) DAPL Provider can provide any combination of the following multipathing and path migration policies:

        – IP address multipathing. DAPL Provider provides multiple routes between two endpoints of a connection

            – DAPL Provider guarantees that all created DAT objects can still be used in the presence of faults.

        – Inter-NIC migration. Migration between homogeneous RNICs, HCAs.

            – DAPL Provider guarantees that all created DAT objects are inter-NIC scoped.

        – Intra-NIC migration. Physical port migration between different ports of the same RNIC, HCA.

            – DAPL Provider guarantees that all created DAT objects are intra-NIC and intra-port scoped.

        – DAPL Provider attribute specifies which of the above 3 models are supported

            – DAPL provides the same migration policy support for all EPs

            – (non Requirement) DAPL EP connect calls do not support Consumer specification which of the multipathing policy to use for the connection

8) Provider notifies Consumer about path migration while maintaning the state of the EP as defined below in a) DAPL exposes the following abstract events on page 61

9) DAPL HA defines a new EVD stream that is associated with IA and is used for delivery of HA events

    a) DAPL exposes the following abstract events

        i) connection is down to a single path

        ii) connection now has more than one path

        iii) Broken connection, connection is down to 0 paths, is NOT delivered to the HA event stream

    b) broken connection event are delivered to the EP connect EVD

    c) IA HA event stream is not associated with any EVD by default

d) Consumer can associate IA HA event stream with IA asynchronous EVD using EVD_modify_event_streams

e) IA HA event stream can only be delivered to IA asynchronous EVD.

f) DAPL does not deliver any remote HA event on local IA HA event stream unless it also impacts the local side

   i) DAPL Provider can optionally deliver any network cloud events that impact existing physical paths of a connection

      – The delivered event is per physical path and not per DAT connection

10) DAPL supports a Consumer ability to specify whether a requested multipathing connection must have more than a single path or if it can accept a connection in initially degraded mode of a single path.

   a) DAPL EP connect calls mutlipathing argument shall support the following 3 values:

      i) no multipathing requested

      ii) mutlipathing required. Do not create connection if only a single path is available

      iii) multipathing requested. Connection can be established in degraded mode.

         – If connection is established in degraded mode an event will be delivered to the HA event stream, if it had not been delived before, indicating that only a single path is available

11) DAPL support Consumer ability accept connection (dat_cr_accept) with the same multipathing flags as EP connect calls

12) DAPL Provider supports Consumer ability to listen on all underlying HW resources based on the HA model it provides.

13) (Negative requirement) DAPL does not provide Consumer ability for Multipathing hint for SRQ.

14) DAPL Provider maintains the multipathing state of the connection: single path, more than one path.

15) DAPL does not migrate a connection from one path to another for multipathed connection if it is not transmission related.

   a) The bad data faults (checksum) on one connection shall not cause path migration for other connections

      i) bad data will cause the multipathing connection failure

      ii) DAPL Provider shall handle transfer level data faults (checksum causes data retransmission)

CHAPTER 6: uDAPL-2.0 API

This chapter defines the user-level DAT API for uDAPL.

6.1 API CONVENTIONS

The DAT API conventions are as follows:

1. All OUT parameters are passed as pointers and hence have "*" in front of the parameter.
2. All character string passing are of the type of char*, where * is not part of the type but is in front of a parameter.
3. INOUT parameter is used by DAT as defined by C.
4. Integer masks are used for Query and Modify routines to request specific parameters and attributes.
5. Handles are used for objects. The Handles are pointers.
6. IN parameters that are passed as pointers are explicitly marked “const”.

The DAT API is a set of methods that apply to DAT Objects. These types are as follows:

- DAT_IA: An open instance of an Interface Adapter (IA)
- DAT_PZ: A Protection Zone
- DAT_LMR: A Local Memory Region
- DAT_RMR: A Remote Memory Region
- DAT_EP: A Local Endpoint
- DAT_PSP: A Public Service Point
- DAT_RSP: A Reserved Service Point
- DAT_EVD: An Event Dispatcher
- DAT_CR: A Connection Request
- DAT_SRQ: A Shared Receive Queue
- DAT_CNO: A Consumer Notification Object
- DAT_CSP: A Common Service Point

6.1.1 NAMESPACE

For the ANSI C mapping of DAT, all global symbols defined by DAT at compile or link time begin with “DAT_” or “dat_”. No other package on a host can use these symbols.

Except as specifically noted, kDAPL and uDAPL have the same namespace. The same types and methods are defined for almost all

Consumer activities. Exceptions will be noted within this document. Where both a uDAPL- and a kDAPL-specific method exist for a given operation, the latter will be identified with a “k” prefix in the “verb” part of the name. For example, the kDAPL-specific method for EVD creation is *dat_evd_kcreate*, while the uDAPL-specific method is *dat_evd_create*.

6.1.2 MEMORY SPACE

All DAT Objects are allocated by the Provider from the Provider memory. All DAT Objects are opaque to a Consumer. Except as noted, DAT Objects are not transferable between Providers or Consumers. There are no exceptions for uDAPL-1.0, uDAPL-1.1, uDAPL-1.2 and uDAPL-2.0.

All DAT Object handles are in the Consumer address space and are not transferable between DAT Consumers even of the same Provider. The only constructs that are transferable are RMR_CONTEXT and DAT_OS_WAIT_PROXY_AGENT.

6.1.3 THREAD, SIGNAL AND EXCEPTION HANDLER SAFETY AND BLOCKING DEFINITIONS

6.1.3.1 THREAD SAFETY DEFINITIONS

“Thread safe” and “non-thread safe” are terms that apply unambiguously to a library. A “thread safe” library is one that can have any number of threads executing within it without regard to what functions those threads call. A “non-thread safe” library is one in which the behavior of having multiple threads of execution within it is undefined.

However, it is confusing to call a routine “thread safe” or “not-thread safe,” because thread safety is implicitly about interactions between routines. If there is a thread of execution within a routine called thread safe, and a thread of execution within a routine called non-thread safe, what is the result? The answer is not obvious because the definitions of “thread safe” and “not-thread safe” with respect to routines has not been specified.

For the uDAPL library, the terms “thread safe” and “non-thread safe” are defined with respect to routines as follows. Note that in what follows “can be called” translates to “the results of calling this function are well-defined” and “cannot be called” translates to “the results of calling this function are not well defined”. The Provider does not enforce the thread safety restrictions described in this document. If the Consumer violates them, the behavior is not defined.

- A routine is "thread safe" if that routine
  - Can be called without imposing any restrictions on routines called by other threads in the system.
  - Can be called without regard to what other routines currently have threads of execution within them.
- A routine is "not-thread safe" if the routine cannot be called if any other in-progress non-thread safe routine shares any of its primary arguments. Almost all routines have a single primary argument which

is the first argument in its signature. The single exception is *dat_rmr_bind*, which has as a primary argument both its first argument (*dat_rmr_handle*) and the Endpoint (*dat_ep_handle*) on which it is called.

This definition explicitly allows simultaneous non-thread safe calls on objects that are “linked” (for example, an EVD and the CNO that it references), so long as no primary object is explicitly shared between the routines.

A Provider is “thread safe” if all routines within the Provider marked as having Provider-dependent thread safety are thread safe. A Provider is “not-thread safe” if all routines within the Provider marked as having Provider-dependent thread safety are not thread safe.

Note that uDAPL only allows these two types of Provider libraries: “thread safe” and “non-thread safe.”

6.1.3.2 Signal and Exception Handler Safety Definitions

It is confusing to call a routine “Signal and Exception handler safe” or “not-safe,” because Signal and Exception handler safety is implicitly about interactions between routines. If there is a thread of execution within a routine called Signal and Exception handler safe, and a thread of execution within a routine called non-Signal and Exception handler safe, what is the result? The answer is not obvious because the definitions of “Signal and Exception handler safe” and “not-Signal and Exception handler safe” with respect to routines has not been specified.

For the uDAPL library, the terms “Signal and Exception handler safe” and “non-Signal and Exception handler safe” are defined with respect to routines as follows. Note that in what follows “can be called” translates to “the results of calling this function are well-defined” and “cannot be called” translates to “the results of calling this function are not well defined”. The Provider does not enforce the Signal and Exception handler safety restrictions described in this document. If the Consumer violates them, the behavior is not defined.

- A routine is "Signal and Exception handler safe" if that routine
  - Can be called in Signal and Exception handler without imposing any restrictions on routines called by other threads and handlers in the system.
  - Can be called without regard to what other routines currently have threads of execution within them.
- A routine is "not-Signal and Exception handler safe" if the routine cannot be called in a handler if any other in-progress non-Signal and Exception handler safe routine shares any of its primary arguments. Almost all routines have a single primary argument which is the first argument in its signature. The single exception is *dat_rmr_bind*, which has as a primary argument both its first argument (*dat_rmr_handle*) and the Endpoint (*dat_ep_handle*) on which it is called.

This definition explicitly allows simultaneous non-Signal and Exception handler safe calls on objects that are “linked” (for example, an EVD and the CNO that it references), so long as no primary object is explicitly shared between the routines.

6.1.3.3 DESIGN PRINCIPLES

- All functions must be thread Signal and Exception handler safe unless it is explicitly noted otherwise.
- Some functions are explicitly noted as Provider-optional thread safe because enforcing thread safety might significantly impact either these function's performance or the performance of functions with which they synchronize. These functions are either on a performance-critical path, or might, in a reasonable implementation, need to be synchronized with performance-critical path functions.
- Some functions are explicitly noted as Provider-optional Signal and Exception handler safe. These include functions that are not expected to be needed by Consumer to use in handler, like connection management and query functions.
- All functions that require memory allocation and freeing are blocking. All functions that Consumer expect to use in handlers are non-blocking. The remaining functions are Provider-dependent.
- If Provider claims to be thread safe, then all functions, except object destructors but including all above noted optional functions, must be thread safe.
- If Provider claims to be Signal and Exception handler safe, then all Provider-dependent functions are Signal and Exception handler safe. Thus, also means that all these functions are non-blocking. For uDAPL Provider Signal and Exception handler safety is subject to caveats of Section 6.9, “Operating System Specific Notes,” on page 303.

More detailed design principles are listed below, and object destruction is discussed in the next section:

- Noncritical path routines should be thread safe for Consumer convenience.
- Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread Signal and Exception handler safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
    - *dat_ep_get_status* is explicitly described as not doing heroic synchronization measures and hence (implicitly) not necessarily returning a coherent state. Hence the logic of the above bullet does not apply to it.

- Because *dat_evd_wait* is defined as thread safe, it is always acceptable to call *dat_evd_post_se* on an EVD that has a waiter on it, even if *dat_evd_post_se* is non-thread safe.
- The connection related EVD calls can involve modifying the state related to DTO posting (a critical path operation). Hence, making those calls thread Signal and Exception handler safe might require locking on the critical path.
- *dat_rmr_bind* is a critical path operation; it should not be thread safe.
- All Post operations must be Signal and Exception (subject to caveats of Section 6.9, "Operating System Specific Notes," on page 303) handler safe. Even if DTO and RMR post routines are not thread safe, threads can be present in both the Request and Recv queues of an Endpoint simultaneously. Note that this does not allow multiple threads posting to the Request queue, or multiple threads posting to the Recv queue of an Endpoint. *dat_rmr_bind* is a Request queue post operation.
  - More precisely, in a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.
  - *dat_rmr_bind* is special from a thread safety point of view. If this routine is non-thread safe, then it cannot be called simultaneously with any non-thread safe routines operating on the Endpoint as their primary argument. This restriction is in addition to the standard non-thread safety restriction prohibiting multiple calls with the RMR as the primary object (first argument). This restriction has the exception described in the above bullet; *dat_rmr_bind* can be called simultaneously with *dat_ep_post_recv* on the same Endpoint.

6.1.3.4 OBJECT DESTRUCTION

DAT explicitly disallows operate/destroy races completely. No routine (including a thread-safe and Signal and Exception handler safe routine) can be called while one of the objects on which it is operating is being destroyed, and an object destruction routine cannot be called while another routine is operating on that object, regardless of the thread safety of that other routine. Note that this makes object destruction routines exclusive with all other routines acting on the same object, whether thread safe or not, and whether that object is primary to the other routine or not. uDAPL Consumers cannot call any object destruction routine simultaneously with any other routine that operates on that object in any fashion. If this restriction is violated, the consequences are undefined (and are likely to be severe).

For two potentially blocking routines *dat_evd_wait* and *dat_cno_wait*, there is a recommendation to unblock a waiting thread before object destruction. For EVD, it is to use *dat_evd_set_unwaitable* (see 6.3.4.7 DAT_EVD_Set_Unwaitable on page 132 and 6.3.4.2 DAT_EVD_Free on page 128), and for CNO, it is to use *dat_evd_post_se* (see *6.3.2.3.1 Usage on page 120*).

From the Consumer's perspective, *dat_cr_accept*, *dat_cr_reject*, and *dat_cr_handoff* are object destruction routines; the CR is not available to the Consumer after calling these routines. Hence, it is not permissible to use the CR in one of these routines simultaneously with its use in any other routine, or to use the CR in any other routine after it has been passed to one of these routines.

6.1.3.5 SAFETY SPECIFICATION

For each of the uDAPL routines, it is defined as being either thread safe, non-thread safe, or that its thread safety is Provider-dependent. Provider-dependent routines take their thread safety from the is_thread_safe boolean in the DAT_PROVIDER_INFO structure (see 8.2.2.1 DAT_Registry_List_Providers on page 310).

The following Table 1 summarizes thread and Signal and Exception handler safety for each call.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ia_open	Thread safe	yes	no	
dat_ia_query	Thread safe	Provider-dependent	Provider-dependent	Noncritical path routines should be thread safe for Consumer convenience.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ia_close	Non-thread safe	yes	no	By the definitions given above, all object destruction are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_set_ consumer_context	Provider-dependent	Provider-dependent	Provider-dependent	
dat_get_ consumer_context	Provider-dependent	no	Provider-dependent	
dat_get_handle_ type	Provider-dependent	no	Provider-dependent	
dat_cno_create	Thread safe	yes	no	Noncritical path routines should be thread safe for Consumer convenience.
dat_cno_fd_create	Thread safe	yes	no	Noncritical path routines should be thread safe for Consumer convenience.
dat_cno_trigger	Thread safe	yes	no	

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_cno_free	Non-thread safe	yes	no	By the definitions given above, all object destruction are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_cno_modify_agent	Provider-dependent	Provider-dependent	Provider-dependent	
dat_cno_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot. Also, there isn't any obvious reason for *dat_cno_query* to be defined differently from *dat_evd_query*.
dat_cno_wait	Thread safe	yes	no	

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_cr_query	Thread safe	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
dat_cr_accept	Non-thread safe	Provider-dependent	Provider-dependent	*dat_cr_accept*, *dat_cr_reject*, and *dat_cr_handoff* are object destruction routines; the CR is not available to the Consumer after calling these routines. Hence, it is not permissible to use the CR in one of these routines simultaneously with its use in any other routine, or to use the CR in any other routine after it is passed to one of these routines.
dat_cr_reject	Non-thread safe	Provider-dependent	Provider-dependent	*dat_cr_accept*, *dat_cr_reject*, and *dat_cr_handoff* are object destruction routines; the CR is not available to the Consumer after calling these routines. Hence, it is not permissible to use the CR in one of these routines simultaneously with its use in any other routine, or to use the CR in any other routine after it is passed to one of these routines.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_cr_handoff	Non-thread safe	Provider-dependent	Provider-dependent	*dat_cr_accept*, *dat_cr_reject*, and *dat_cr_handoff* are object destruction routines; the CR is not available to the Consumer after calling these routines. Hence, it is not permissible to use the CR in one of these routines simultaneously with its use in any other routine, or to use the CR in any other routine after it is passed to one of these routines.
dat_evd_create	Thread safe	yes	no	Noncritical path routines should be thread safe for Consumer convenience.
dat_evd_free	Non-thread safe	yes	no	By the definitions given above, all object destruction are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_evd_wait	Thread safe	yes	no	Performance Critical operation.
dat_evd_dequeue	Thread safe	no	yes	Performance Critical operation.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_evd_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
dat_evd_modify_cno	Provider-dependent	Provider-dependent	Provider-dependent	
dat_evd_enable	Thread Safe	no	yes	
dat_evd_disable	Thread Safe	no	yes	
dat_evd_set_unwaitable	Thread Safe	no	yes	
dat_evd_clear_unwaitable	Thread Safe	no	yes	
dat_evd_resize	Provider-dependent	Provider-dependent	Provider-dependent	
dat_evd_post_se	Provider-dependent	no	yes	Because *dat_evd_wait* is defined as thread safe, it is always acceptable to call *dat_evd_post_se* on an EVD that has a waiter on it, even if *dat_evd_post_se* is non-thread safe.
dat_ep_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ep_free	Non-thread safe	yes	no	By the definitions given above, all object destruction are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_ep_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
dat_ep_modify	Provider-dependent	Provider-dependent	Provider-dependent	
dat_ep_connect	Provider-dependent	Provider-dependent	Provider-dependent	The connection related EVD calls might involve modifying the state related to DTO posting (a critical path operation). Hence, making those calls thread safe might require locking on the critical path.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ep_common_connect	Provider-dependent	Provider-dependent	Provider-dependent	The connection related EVD calls might involve modifying the state related to DTO posting (a critical path operation). Hence, making those calls thread safe might require locking on the critical path.
dat_ep_dup_connect	Provider-dependent	Provider-dependent	Provider-dependent	The connection related EVD calls might involve modifying the state related to DTO posting (a critical path operation). Hence, making those calls thread safe might require locking on the critical path.
dat_ep_disconnect	Provider-dependent	Provider-dependent	Provider-dependent	The connection related EVD calls might involve modifying state related to DTO posting (a critical path operation). Hence, making those calls thread safe might require locking on the critical path.
dat_ep_reset	Provider-dependent	Provider-dependent	Provider-dependent	The connection related EVD calls may involve modifying state related to DTO posting (a critical path operation). Hence making those calls thread safe may require locking on the critical path.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ep_post_send	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.
dat_ep_post_send_with_invalidate	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ep_post_recv	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.
dat_ep_post_rdma_read	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_ep_post_rdma_read_to_rmr	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.
dat_ep_post_rdma_write	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_send_with_invalidate*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, *dat_ep_post_rdma_read_to_rmr*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes.
dat_ep_get_status	Thread safe	no	yes	*at_ep_get_status* is explicitly described as not doing heroic synchronization measures and hence (implicitly) not necessarily returning a coherent state. Hence, the logic of the above bullet does not apply to it.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_lmr_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_lmr_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_lmr_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
dat_rmr_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_rmr_create_for_ep	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_rmr_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_rmr_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_rmr_bind	Provider-dependent	no	yes	Performance Critical operation. In a non-thread safe Provider, there is an exception to the general thread safety restrictions: there can be one thread executing in one of the routines *dat_ep_post_send*, *dat_ep_post_rdma_write*, *dat_ep_post_rdma_read*, or *dat_rmr_bind* at the same time as another thread is executing in *dat_ep_post_recv*. No more than one thread can execute in either of these classes. *dat_rmr_bind* is special from a thread safety point of view. If this routine is non-thread safe, it cannot be called simultaneously with any non-thread safe routines operating on the Endpoint as their primary argument. This restriction is in addition to the standard non-thread safety restriction prohibiting multiple calls with the RMR as the primary object (first argument). *dat_rmr_bind* can be called simultaneously with *dat_ep_post_recv* on the same Endpoint.
dat_psp_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_psp_create_any	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_psp_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_psp_query	Thread safe	Provider-dependent	Provider-dependent	Non-critical path routines should be thread safe for Consumer convenience.
dat_rsp_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_rsp_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_rsp_query	Thread safe	Provider-dependent	Provider-dependent	Non-critical path routines should be thread safe for Consumer convenience.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_csp_create_any	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_csp_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_csp_query	Thread safe	Provider-dependent	Provider-dependent	Non-critical path routines should be thread safe for Consumer convenience.
dat_pz_create	Thread safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_pz_free	Non-thread safe	yes	no	By the definitions given above, all object destructions are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_pz_query	Thread safe	Provider-dependent	Provider-dependent	Non-critical path routines should be thread safe for Consumer convenience.
dat_srq_create	Thread Safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_srq_set_lw	Provider Dependent	Provider-dependent	Provider-dependent	
dat_srq_free	Non-Thread Safe	yes	no	By the definitions given above, all object destruction are non-thread safe. They all are called with the only object argument being the object to be destroyed, which means that no other routines on that object can be in process simultaneously with them. In some sense, these routines are outside the regular scheme, because threads of execution being within them prohibits threads of execution in both thread safe and non-thread safe routines on the same objects.
dat_srq_query	Provider-dependent	Provider-dependent	Provider-dependent	Any query routine is presumed to provide a coherent snapshot of its object, and making that snapshot coherent might require locking both in the snapshot routine and in any routines that modify the object's state. Therefore, it is inappropriate to make a query routine thread safe unless that routine is specifically noted to not necessarily provide a coherent snapshot.
dat_srq_resize	Provider-dependent	Provider-dependent	Provider-dependent	

Table 1 uDAPL API calls safety specification.

uDAPL Call	Thread Safety	Blocking	Signal and Exception Handling safe	Notes
dat_srq_post_recv	Provider-dependent	no	yes	
dat_ep_create_with_srq	Thread Safe	yes	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_ep_recv_query	Provider-dependent	Provider-dependent	Provider-dependent	
dat_ep_set_watermark	Provider-dependent	Provider-dependent	Provider-dependent	
dat_lmr_sync_rdma_read	Provider-dependent	Provider-dependent	yes	
dat_lmr_sync_rdma_write	Provider-dependent	Provider-dependent	yes	
dat_registry_list_providers	Thread safe	Provider-dependent	no	Non-critical path routines should be thread safe for Consumer convenience.
dat_registry_providers_related	Provider-dependent	Provider-dependent	no	
dat_strerror	Thread safe	no	yes	Non-critical path routines should be thread safe for Consumer convenience.

6.2 Local Resources Management

6.2.1 Interface Adapter

6.2.1.1 DAT_IA_Open

Synopsis:

```
DAT_RETURN
    dat_ia_open (
    IN const DAT_NAME_PTR    ia_name_ptr,
    IN     DAT_COUNT         async_evd_min_qlen,
    INOUT  DAT_EVD_HANDLE    *async_evd_handle,
    OUT    DAT_IA_HANDLE     *ia_handle
    )
```

Parameters:

ia_name_ptr:	Symbolic name for the IA to be opened. The name should be defined by the Provider registration.
async_evd_min_qlen:	Minimum length of the Asynchronous Event Dispatcher queue.
async_evd_handle:	Pointer to a handle for an Event Dispatcher for asynchronous events generated by the IA. This parameter can be *DAT_EVD_ASYNC_EXISTS* to indicate that there is already EVD for asynchronous events for this Interface Adapter or *DAT_HANDLE_NULL* for a Provider to generate EVD for it.
ia_handle:	Handle for an open instance of a DAT IA. This handle is used with other functions to specify a particular instance of the IA.

Description: *dat_ia_open* opens an IA by creating an IA instance. Multiple instances (opens) of an IA can exist.

The value of *DAT_HANDLE_NULL* for *async_evd_handle* (**async_evd_handle* == *DAT_HANDLE_NULL)* indicates that the default Event Dispatcher is created with the requested *async_evd_min_qlen*. The *async_evd_handle* returns the handle of the created Asynchronous Event Dispatcher. The first Consumer that opens an IA must use *DAT_HANDLE_NULL* because no EVD can yet exist for the requested *ia_name_ptr*.

The Asynchronous Event Dispatcher (*async_evd_handle)* is created with no *CNO* (*DAT_HANDLE_NULL*). Consumers can change these values using *dat_evd_modify_cno*. The Consumer can modify parameters of the Event Dispatcher using *dat_evd_resize* and *dat_evd_modify_cno*.

The Provider is required to provide a queue size at least equal to *async_evd_min_qlen*, but is free to provide a larger queue size or dynamically enlarge the queue when needed. The Consumer can determine the actual queue size by querying the created Event Dispatcher instance.

If *async_evd_handle* is not *DAT_HANDLE_NULL*, the Provider does not create an Event Dispatcher for an asynchronous event and the Provider ignores the *async_evd_min_qlen* value. The *async_evd_handle* value passed in by the Consumer must be an asynchronous Event Dispatcher created for the same Provider (*ia_name_ptr*). The Provider does not have to check for the validity of the Consumer passed in *async_evd_handle.* It is the Consumer responsibility to guarantee that *async_evd_handle* is valid and for this Provider. How the *async_evd_handle* is passed between DAT Consumers is out of scope of the DAT specification. If the Provider determines that the Consumer-provided *async_evd_handle* is invalid, the operation fails and returns *DAT_INVALID_HANDLE*. The *async_evd_handle* remains unchanged, so the returned *async_evd_handle* is the same the Consumer passed in. All asynchronous notifications for the open instance of the IA are directed by the Provider to the Consumer passed in Asynchronous Event Dispatcher specified by *async_evd_handle.*

Consumer can specify the value of *DAT_EVD_ASYNC_EXISTS* to indicate that there exists an event dispatcher somewhere else on the host, in user or kernel space, for asynchronous event notifications. It is up to the Consumer to ensure that this event dispatcher is unique and unambiguous. A special handle may be returned for the Asynchronous Event Dispatcher for this scenario, *DAT_EVD_OUT_OF_SCOPE*, to indicate that there is a default Event Dispatcher assigned for this Interface Adapter, but that it is not in a scope where this Consumer may directly invoke it.

The Asynchronous Event Dispatcher is an Object of both the Provider and IA. Each Asynchronous Event Dispatcher bound to an IA instance is notified of all asynchronous events, such that binding multiple Asynchronous Event Dispatchers degrades performance by duplicating asynchronous event notifications for all Asynchronous Event Dispatchers. Also, transport and memory resources can be consumed per Event Dispatcher bound to an IA.

As with all Event Dispatchers, the Consumer is responsible for synchronizing access to the event queue.

dat_ia_open is synchronous and thread safe.

Valid IA names are obtained from *dat_registery_list_providers*, as defined by the Provider registration (see Section 8.2.2.1 on page 310).

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_NAME_NOT_FOUND	The specified IA name was not found in the list of registered Providers.
DAT_INVALID_HANDLE	Invalid DAT handle; *async_evd_handle* is invalid.

6.2.1.1.1 USAGE

dat_ia_open is the root method for the Provider, and, thus, all Objects. It is the root handle through which the Consumer obtains all other DAT handles. When the Consumer closes its handle, all its DAT Objects are released.

Consumers can also use *dat_ia_openv* (see 8.3.3 Version Support for IA open on page 317) that allows them to specify DAT version number as well as thread safety for the library to be used directly instead of relying on the uDAPL configuration header file (see 8.3.3 Version Support for IA open on page 317) for it.

6.2.1.1.2 RATIONALE

dat_ia_open is the workhorse method that provides an IA instance. It can also initialize the Provider library or do any other registry-specific functions.

6.2.1.1.3 MODEL IMPLICATIONS

dat_ia_open creates a unique handle for the IA to the Consumer. All further DAT Objects created for this Consumer reference this handle as their owner.

When IA is open the High Availability events are automatically associated with Asyncronous EVD.

dat_ia_open can use a reference count for the Provider Library to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

6.2.1.2 DAT_IA_CLOSE

Synopsis:

```
DAT_RETURN
   dat_ia_close (
   IN     DAT_IA_HANDLE      ia_handle,
   IN     DAT_CLOSE_FLAGS    ia_flags
   )
```

Parameters:

ia_handle: Handle for an instance of a DAT IA.

ia_flags: Flags for IA closure. Default value of *DAT_CLOSE_DEFAULT = DAT_CLOSE_ABRUPT _FLAG* represents abrupt closure of IA. See Table 2 for flag definitions.

Table 2 IA Closure Flag Definitions

Features	Definition	Description
Abrupt close	DAT_CLOSE_ABRUPT_FLAG	Abrupt cascading close of IA including all Consumer created DAT objects.
Graceful close	DAT_CLOSE_GRACEFUL_FLAG	Closure is successful only if all DAT objects created by the Consumer have been freed before the graceful closure call.

Description: *dat_ia_close* closes an IA (destroys an instance of the Interface Adapter).

The *ia_flags* specify whether the Consumer wants *abrupt* or *graceful* close.

The abrupt close does a phased, cascading destroy. All DAT Objects associated with an IA instance are destroyed. These include all the connection oriented Objects: public and reserved Service Points; Endpoints, Connection Requests, LMRs (including *lmr_contexts*), RMRs (including *rmr_contexts*), Event Dispatchers, CNOs, and Protection Zones. All waiters on all CNOs, including the OS Wait Proxy Agents, are unblocked with the *DAT_HANDLE_NULL* handle returns for an unblocking EVD. All direct waiters on all EVDs are also unblocked and return with *DAT_ABORT*.

The graceful close does a destroy only if the Consumer has done a cleanup of all DAT objects created by the Consumer with the exception of the asynchronous EVD. Otherwise, the operation does not destroy the IA instance and returns the *DAT_INVALID_STATE.*

If async EVD was created as part of the of *dat_ia_open*, *dat_ia_close* must destroy it. If *async_evd_handle* was passed in by the Consumer at *dat_ia_open*, this handle is not destroyed. This is applicable to both abrupt and graceful *ia_flags* values.

Because the Consumer did not create async EVD explicitly, the Consumer does not need to destroy it for graceful close to succeed.

It is illegal to use the destroyed handle in any subsequent operation.

dat_ia_close is synchronous and not thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations. The current *dat_ia_close* operation has failed but future attempts to close the IA may succeed. Since the IA handle may be partially torn down the IA handle, or any of its descendent objects, may only be used for subsequent invocations of *dat_ia_close*.
DAT_INTERNAL_ERROR	A consistency check failure prevents cleanly closing the IA and recovering the resources associated with the IA. Subsequent calls to *dat_ia_close* on *ia_handle* are guaranteed not to succeed. Remaining DAT resources associated with *ia_handle* can be recovered only when the process exits.

DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *ia_flags* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. IA instance has Consumer created objects associated with it.

6.2.1.2.1 USAGE

dat_ia_close is the root cleanup method for the Provider, and, thus, all Objects.

Consumers are advised to explicitly destroy all Objects they created prior to closing the IA instance, but can use this function to clean up everything associated with an open instance of IA. This allows the Consumer to clean up in case of errors.

Note that an abrupt close implies destruction of EVDs and CNOs. Just as with explicit destruction of an EVD or CNO, the Consumer should take care to avoid a race condition where a Consumer ends up attempting to wait on an EVD or CNO that has just been deleted.

The techniques described in *dat_cno_free* (See Section 6.3.2.3, "DAT_CNO_Free," on page 120) and *dat_evd_free* (See "DAT_EVD_Free" on page 128.) can be used for these purposes.If the Consumer desires to shut down the IA as quickly as possible, the Consumer can call *dat_ia_close(abrupt)* without unblocking CNO and EVD waiters in an orderly fashion. There is a slight chance that an invalidated DAT handle will cause a memory fault for a waiter. But this might be an acceptable behavior, especially if the Consumer is shutting down the process.

6.2.1.2.2 RATIONALE

No provision is made for blocking on event completion or pulling events from queues.

This is the general cleanup and last resort method for Consumer recovery. An implementation must provide for successful completion under all conditions, avoiding hidden resource leakage (dangling memory, zombie processes, and so on) eventually leading to a reboot of the operating system.

6.2.1.2.3 MODEL IMPLICATIONS

dat_ia_close deletes all Objects that were created using the IA handle.

dat_ia_close can decrement a reference count for the Provider Library that is incremented by *dat_ia_open* to ensure that the Provider Library cannot be removed when it is in use by a DAT Consumer.

DAT_INSUFFICIENT_RESOURCES indicates that a transient error has occurred and the Consumer may retry the close operation later. The Consumer may not use the handles of DAT objects which are descended from *ia_handle* after this return code from *dat_ia_close*. If the *DAT_CLOSE_GRACEFUL_FLAG* is set then *DAT_INSUFFICIENT_RESOURCES* may be returned only after *dat_ia_close* has verified that *ia_handle* is in the appropriate state for a graceful close with associated objects previously cleaned up by the Consumer.

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to the destroyed object.

6.2.1.3 INTERFACE ADAPTER ATTRIBUTES

The IA attributes are common to all open instances of the IA. DAT defines a method to query the IA attributes but does not define a method to modify them.

If IA is multiported, each port is presented to a Consumer as a separate IA. In the cases when the multiport IA provides special semantics that go beyond separate IA semantics, the Provider can present it as a single IA. An example of it is a dual port IA that automatically provides multipathing for each connection for failover support.

Adapter name:	The name of the IA controlled by the Provider. The same as *ia_name_ptr*.
Vendor name:	Vendor of IA hardware.
HW version major:	Major version of IA hardware.
HW version minor:	Minor version of IA hardware.
Firmware version major:	Major version of IA firmware.
Firmware version minor:	Minor version of IA firmware.
IA_address_ptr:	An address of the Interface Adapter.
Max EPs:	Maximum number of Endpoints that the IA can support. This covers all Endpoints in all states, including the ones used by the Providers, zero or more applications, and management.
Max DTOs per EP:	Maximum number of DTOs and RMR_ binds that any Endpoint can support for single direction. This means the maximum number of outstanding and in-progress Send, RDMA Read, RDMA Write DTOs, and RMR Binds at any one time for any Endpoint; and maximum number of outstanding and in-progress Receive DTOs at any one time for any Endpoint.

Max incoming RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the target.
Max outgoing RDMA Reads per EP:	Maximum number of RDMA Reads that can be outstanding per (connected) Endpoint with the IA as the originator.
Max EVDs:	Maximum number of Event Dispatchers that an IA can support. An IA cannot support Event Dispatcher directly, but indirectly by Transport-specific Objects, for example, Completion Queues for Infiniband™, VI and iWARP. The Event Dispatcher Objects can be shared among multiple Providers and similar Objects from other APIs, for example, Event Queues for DAPL.
Max EVD queue size:	Maximum size of the EVD queue supported by an IA.
Max IOV segments per non-RDMA DTO:	Maximum entries in an IOV list for non-RDMA DTOs that an IA supports. Notice that this number cannot be explicit but must be implicit to transport-specific Object entries. For example, for IB, it is the maximum number of scatter/gather entries per Work Request, and for VI it is the maximum number of data segments per VI Descriptor.
Max LMRs:	Maximum number of Local Memory Regions IA supports among all Providers and applications of this IA.
Max LMR block size:	Maximum contiguous block that can be registered by the IA.
Max LMR VA:	Highest valid virtual address within the context of an LMR. Frequently, IAs on 32-bit architectures only support 32-bit local virtual addresses.
Max PZs:	Maximum number of Protection Zones that the IA supports.
Max Message size:	Maximum message size supported by the IA.
Max RDMA size:	Maximum RDMA size supported by the IA.
Max RMRs:	Maximum number of RMRs an IA supports among all Providers and applications of this IA.

Max RMR target address:	Highest valid target address with the context of a local RMR. Frequently, IAs on 32-bit architectures only support 32-bit local virtual addresses.
Max SRQs:	Maximum number of Shared Received Queues that the IA can support.
Max EPs per SRQ:	Maximum number of EPs that can use a Shared Received Queue simultaneously.
Max Recv DTOs per SRQ:	Maximum number of Recv DTOs that a Shared Received Queue can support.
Max IOV segments for RDMA Read:	Maximum entries in an IOV list for RDMA Read DTO that an IA supports. For example for iWARP it should be 1, while for IB it should be the same as for other DTOs.
Max IOV segments for RDMA Write:	Maximum entries in an IOV list for RDMA Write DTO that an IA supports. For example for IB it should be the same as for other DTOs, while for iWARP it can be different.
Max incoming RDMA Reads:	Maximum number of inbound RDMA Reads that the HCA/RNIC can support. This covers all open instances of IA.
Max outgoing RDMA Reads:	Maximum number of outbound RDMA Reads that the HCA/RNIC can support. This covers all open instances of IA.
Max RDMA Reads per Endpoint IN guarantee:	Indicator whether or not maximum incoming RDMA Read resources are guaranteed per Endpoint. DAT_FALSE means that maximum incoming RDMA Read per EP is guaranteed and every Endpoint can get this maximum. DAT_TRUE means that maximum incoming RDMA Read resources are not guaranteed per Endpoint. This means that the incoming RDMA Read resources are shared between EPs at HCA/RNIC. So the number of incoming RDMA Read allocated to an Endpoint affects the number of incoming RDMA Read resources available for other Endpoints of all instances of IA on the HCA/RNIC.

Max RDMA Reads per Endpoint OUT guarantee:	Indicator whether or not maximum outgoing RDMA Read resources are guaranteed per Endpoint. DAT_FALSE means that maximum outgoing RDMA Read per EP is guaranteed and every Endpoint can get this maximum. DAT_TRUE means that maximum outgoing RDMA Read resources are not guaranteed per Endpoint. This means that the outgoing RDMA Read resources are shared between EPs at HCA/RNIC. So the number of outgoing RDMA Read allocated to an Endpoint affects the number of outgoing RDMA Read resources available for other Endpoints of all instances of IA on the HCA/RNIC.
ZB support:	Binary indicator of support of zero-based Virtual Addressing for LMR.
Extension interface:	The DAT_EXTENSION_INTERFACE can have of the three values: DAT_EXTENSION_IB, DAT_EXTENSION_IW, or DAT_EXTENSION_NONE. The last of them indicates that Provider does not support extensions. DAT Provider can support either IB or iWARP extension but not both.
Extension version:	The DAT_EXTENSION_VERSION indicates which version of the extension interface Provider supports.
Num transport attributes:	Number of transport-specific attributes
Transport-specific attributes:	Array of transport-specific attributes. Each entry has the format of *DAT_NAMED_ATTR*, which is a structure with two elements. The first element is the name of the attribute, and the second is the value of the attribute as a string.
Num vendor attributes:	Number of vendor-specific attributes
Vendor-specific attributes:	Array of vendor-specific attributes. Each entry has the format of *DAT_NAMED_ATTR*, which is a structure with two elements. The first element is the name of the attribute, and the second is the value of the attribute as a string.

6.2.1.3.1 MODEL IMPLICATIONS

If RDMA Read resources on RNIC/HCA are shared between Endpoints then it affects Consumes that rely on ability to create an Endpoint with prespecified number of incoming or outgoing RDMA Read. Consumers can address the situation by choosing the Provider that provide guaranteed number of RDMA Reads per Endpoint or by configuring the application to handle lack of the RDMA Read resources per Endpoint, for example, by pre-allocating Endpoints first, or responding to connection requests with smaller than requested RDMA Read support numbers. Most client-server based applications need to be able to handle a response to a connection request where a server is unable to allocated the requested RDMA Read incoming credits.

6.2.1.3.2 DAT Extensions Attributes

DAT Provider can support IB or iWARP extensions that are defined by separate documents. DAT_EXTENSION_INTERFACE and DAT_EXTENSION_VERSION attributes indicate which extension and which version of the extension Provider supports. Each extension defines their own transport specific attributes.

6.2.1.4 DAPL Provider Attributes

The list of Provider attributes. The Provider attributes are specific to the open instance of the IA. DAT defines a method to query Provider attributes but does not define a method to modify them.

Provider name:	Name of the Provider vendor.
Provider version major:	Major Version of uDAPL Provider.
Provider version minor:	Minor Version of uDAPL Provider.
DAPL API version major:	Major Version of uDAPL API supported.
DAPL API version minor:	Minor Version of uDAPL API supported.
LMR memory types supported:	Memory types that LMR Create supports for memory registration. This value is a union of LMR Memory Types DAT_MEM_TYPE_VIRTUAL, DAT_MEM_TYPE_LMR, and DAT_MEM_TYPE_SHARED_VIRTUAL that the Provider supports. All Providers must support the following Memory Types: DAT_MEM_TYPE_VIRTUAL, DAT_MEM_TYPE_SHARED_VIRTUAL, and DAT_MEM_TYPE_LMR.

IOV ownership:	An enumeration flag that specifies the ownership of the local buffer description (IOV list) after post DTO returns. The three values are as follows: DAT_IOV_CONSUMER indicates that the Consumer has the ownership of the local buffer description after a post returns, DAT_IOV_PROVIDER_NOMOD indicates that the Provider still has ownership of the local buffer description of the DTO when the post DTO returns, but the Provider does not modify the buffer description, and DAT_IOV_PROVIDER_MOD indicates that the Provider still has the ownership of the local buffer description of the DTO when the post DTO returns and that the Provider can modify the buffer description. In any case, the Consumer obtains ownership of the local buffer description after the DTO transfer is completed and the Consumer is notified through a DTO completion event.
QOS supported:	The union of the connection QOS supported by the Provider.
Completion flags supported:	Completion flag - DAT_COMPLETION_FLAGS values: DAT_COMPLETION_SUPPRESS_FLAG, DAT_COMPLETION_UNSIGNALLED_FLAG, DAT_COMPLETION_SOLICITED_WAIT_FLAG, and DAT_COMPLETION_BARRIER_FENCE_FLAG supported by the Provider.
Thread safety:	Provider Library thread safe or not. The Provider Library is not required to be thread safe.
Max private data size:	Maximum size of private data the Provider supports. This value is at least 64 bytes.
Multipathing support:	Capability of the Provider to support Multipathing for connection establishment.

EP creator for PSP:	Indicator about who can create an Endpoint for a Connection Request: Consumer - *DAT_PSP_CREATES_EP_NEVER*, Provider - *DAT_PSP_CREATES_EP_ALWAYS*, or both - *DAT_PSP_CREATES_EP_IFASKED*. It is used for Public Service Point creation.
PZ support:	Indicator of what kind of protection the Provider's PZ provides. - DAT_PZ_UNIQUE - Each Protection Zone is unique within the scope of the IA it was created, and it has been assigned exclusive use of some hardware/verb layer matching resource. For example, this means that the PZ has been assigned ownership of a Protection Domain ID for IB and iWARP. - DAT_PZ_SHARABLE - The Protection Zone has a uniquely assigned hardware layer resource, but may be shared with other processes. Sharing can be achieved as the result of administrative configuration, ability of a Provider to support sharing of pz_handles among processes and/or supplemental API calls that are outside the scope of DAT.
Optimal Buffer Alignment	Local and remote DTO buffer alignment for optimal performance on the Platform. The *DAT_OPTIMAL_ALIGNMENT* must be divisible by this attribute value. The maximum allowed value is *DAT_OPTIMAL_ALIGNMENT* (256).
EVD stream merging support	a 2D binary matrix where each row and column represents an event stream type. Each binary entry is 1 if the event streams of its row and column can fed to the same EVD, and 0 otherwise. More than two different event stream types can feed the same EVD if for each pair of the event stream types the entry is 1. Provider should support merging of all event stream types. Consumer should check this attribute before requesting an EVD that merges multiple event stream types.

SRQ support	Capability of Provider to support Shared Receive Queues. DAT_FALSE means that SRQs are not supported, DAT_TRUE means that SRQs are supported. DAPL versions 1.1 and earlier do not support SRQ.
SRQ Watermark Support	Indicator of which Watermark associated with SRQ capabilities are supported by the Provider. - 0x000 - No Watermarks are supported. - 0x001 - SRQ Low Watermark is supported. - 0x010 - Soft High Watermark for EP associated with SRQ is supported. - 0x0100 - Hard High Watermark for EP associated with SRQ is supported.
PZ mismatch for SRQ and EPs	Indicator whether or not the Provider supports different PZs for SRQ and EPs that use it. DAT_FALSE means that PZ must be the same for SRQ and EP and DAT_TRUE means that it can be different.
SRQ info support	Indicator of which SRQ info queries are supported by the Provider. - 0x01 - SRQ available Recv buffers information is supported. - 0x10 - SRQ outstanding Recv buffers information is supported. If supported that information is returned by *dat_srq_query* as *available_dto_count* and *outstanding_dto_count* in *dat_srq_param* structure.
EP Recv info support	Indicator of which EP Recv info queries are supported by the Provider. - 0x01 - SRQ buffers on EP information is supported. - 0x10 - information for the number of SRQ buffers needed by an EP to complete arriving messages is supported. If supported that information is returned by *dat_ep_recv_query* as *nbufs_allocated* and *bufs_alloc_span*.

LMR synchronization requirement	Binary indicator for the need to use synchronization calls in conjunction with RDMA operations. DAT_TRUE means that *dat_lmr_sync_rdma_read* and *dat_lmr_sync_rdma_write* are required on remote side of RDMA op initiator, and DAT_FALSE means that no synchronization calls are needed.
DTO asynchronous return guarantee	Boolean attribute for asynchronous return guarantee for Send and RDMA Write. The *DAT_TRUE* means that the asynchronous returns defined in Section 6.8.2 on page 295 are generated for Send and RDMA Write for each defined return value. *DAT_FALSE* means that not all return values can be generated.
RDMA Write req for RDMA Read	Boolean attribute that indicates whether RDMA Write is required for a buffer for RDMA Read accesses. *DAT_TRUE* means that both RDMA Write and RDMA Read privileges are required for a buffer for RDMA Read access. *DAT_FALSE* means that RDMA Write privileges are not required for RDMA Read buffer accesses.
RDMA Read LMR Context exposure	Boolean attribute that indicates whether use of the RMR Context as the sink of an RDMA Read is required to prevent exposing the RMR Context of the LMR Context to the wire.
RMR scopes supported	Attribute specifying whether Provider supports RMR scoped to PZ, single EP, or all types.
Signal and Exception handler safety:	Provider Library Signal and Exception handler safe or not. The Provider Library is not required to be Signal and Exception handler safe. Even when Provider library is not Signal and Exception handler safe some of the Provider-dependent functions can be Signal and Exception handler safe.
HA support	Boolean attribute that indicates whether Provider supports HA or not.

HA load balancing	Attribute specifying which load balancing for HA are provided. DAT_HA_LB_NONE indicates that no load balancing is provided. DAT_HA_LB_INTERCONN indicates that connection level load balancing is provided. That is Provider load balances DAT connections at connection establishment time over multiple available physical paths between the same pair of nodes. DAT_HA_LB_INTRACONN indicates that intra-connection level load balancing is provided. That is Provider layers DAT Connection over multiple physical paths and Provider load balances connection traffic over physical paths preserving DAT transport and API requirements.
Num provider attributes:	Number of Provider-specific attributes
Provider-specific attributes:	Array of Provider-specific attributes. Each entry has the format of *DAT_NAMED_ATTR*, which is a structure with two elements. The first element is the name of the attribute, and the second is the value of the attribute as a string.

6.2.1.5 DAT_IA_QUERY

Synopsis:

```
DAT_RETURN
   dat_ia_query (
   IN     DAT_IA_HANDLE            ia_handle,
   OUT    DAT_EVD_HANDLE           *async_evd_handle,
   IN     DAT_IA_ATTR_MASK         ia_attr_mask,
   OUT    DAT_IA_ATTR              *ia_attributes,
   IN     DAT_PROVIDER_ATTR_MASK   provider_attr_mask,
   OUT    DAT_PROVIDER_ATTR        *provider_attributes
   )
```

Parameters:

ia_handle: Handle for an open instance of an IA.

async_evd_handle: Handle for an Event Dispatcher for asynchronous events generated by the IA.

ia_attr_mask: Mask for the *ia_attributes*.

ia_attributes:	Pointer to a Consumer-allocated structure that the Provider fills with IA attributes.
provider_attr_mask:	Mask for the *provider_attributes*.
provider_attributes:	Pointer to a Consumer-allocated structure that the Provider fills with Provider attributes.

Description: *dat_ia_query* provides the Consumer with the IA parameters, as well as the IA and Provider attributes. Consumers pass in pointers to Consumer-allocated structures for the IA and Provider attributes that the Provider fills.

ia_attr_mask and *provider_attr_mask* allow the Consumer to specify which attributes to query. The Provider returns values for requested attributes. The Provider can also return values for any of the other attributes.

dat_ia_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter;
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* is invalid.

6.2.1.5.1 Usage

Consumer can specify *DAT_IA_FIELD_NONE* or *DAT_PROVIDER_FIELD_NONE* if only Provider attributes or IA attributes are requested, respectively.

dat_ia_query is synchronous and thread safe.

6.2.1.5.2 Rationale

6.2.1.5.3 Model Implications

6.2.2 Consumer Context

These two operations allow Consumers to associate and retrieve user context information about any DAT Object instance. The Consumer can supply/obtain a pointer to a data structure that is opaque to the Provider. Each DAT Object maintains a single Consumer context. This data has no semantic meaning to the Provider. The user must synchronize all access.

By default, all DAT Objects are created with a NULL value for the Consumer Context.

6.2.2.1 DAT_SET_CONSUMER_CONTEXT

Synopsis: `DAT_RETURN`

```
dat_set_consumer_context (
IN      DAT_HANDLE          dat_handle,
IN      DAT_CONTEXT         context
)
```

Parameters:

dat_handle: Handle for a DAT Object associated with *context*.

context: Consumer context to be stored within the associated *dat_handle*. The Consumer context is opaque to the uDAPL Provider. *NULL* represents no context.

Description:

dat_set_consumer_context associates a Consumer context with the specified *dat_handle*. The *dat_handle* can be one of the following handle types: *DAT_IA_HANDLE, DAT_EP_HANDLE, DAT_EVD_HANDLE, DAT_CR_HANDLE, DAT_RSP_HANDLE, DAT_PSP_HANDLE, DAT_CSP_HANDLE, DAT_PZ_HANDLE, DAT_LMR_HANDLE,* or *DAT_RMR_HANDLE, or DAT_CNO_HANDLE*.

Only a single Consumer context is provided for any *dat_handle*. If there is a previous Consumer context associated with the specified handle, the new context replaces the old one. The Consumer can disassociate the existing context by providing a NULL pointer for the *context*. The Provider makes no assumptions about the contents of *context*; no check is made on its value. Furthermore, the Provider makes no attempt to provide any synchronization for access or modification of the *context*.

dat_set_consumer_context is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *context* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *dat_handle* is invalid.

6.2.2.2 DAT_GET_CONSUMER_CONTEXT

Synopsis:

```
DAT_RETURN
   dat_get_consumer_context (
   IN      DAT_HANDLE          dat_handle,
   OUT     DAT_CONTEXT         *context
   )
```

Parameters:

dat_handle:	Handle for a DAT Object associated with the *context*.
context:	Pointer to Consumer-allocated storage where the current value of the *dat_handle* context will be stored.

Description: *dat_get_consumer_context* gets the Consumer context from the specified *dat_handle*. The *dat_handle* can be one of the following handle types: *DAT_IA_HANDLE, DAT_EP_HANDLE, DAT_EVD_HANDLE, DAT_CR_HANDLE, DAT_PSP_HANDLE, DAT_RSP_HANDLE, DAT_CSP_HANDLE, DAT_PZ_HANDLE, DAT_LMR_HANDLE,* or *DAT_RMR_HANDLE, or DAT_CNO_HANDLE*.

dat_get_consumer_context is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful. The Consumer context was successfully retrieved from the specified handle.
DAT_INVALID_HANDLE	Invalid DAT handle; *dat_handle* is invalid.

6.2.2.2.1 Usage

6.2.2.2.2 Rationale

This functionality is commonly used for directives by ULPs for asynchronous models of communication.

6.2.2.2.3 Model Implications

Two user context operations are generic for all DAT handles.

6.2.2.3 DAT_GET_HANDLE_TYPE

Synopsis:

```
DAT_RETURN
   dat_get_handle_type (
   IN     DAT_HANDLE         dat_handle,
   OUT    DAT_HANDLE_TYPE    *handle_type
   )
```

Parameters:

dat_handle:	Handle for DAT Object.
handle_type:	Type of the handle of *dat_handle*.

Description: *dat_get_handle_type* allows the Consumer to discover the type of a DAT Object using its handle. The *dat_handle* can be one of the following handles: *DAT_IA_HANDLE, DAT_EP_HANDLE, DAT_EVD_HANDLE, DAT_CR_HANDLE, DAT_PSP_HANDLE, DAT_RSP_HANDLE, DAT_CSP_HANDLE, DAT_PZ_HANDLE, DAT_LMR_HANDLE,* or *DAT_RMR_HANDLE*. The *handle_type* is one of the following handle types: *DAT_HANDLE_TYPE_IA, DAT_HANDLE_TYPE_EP, DAT_HANDLE_TYPE_EVD, DAT_HANDLE_TYPE_CR, DAT_HANDLE_TYPE_PSP, DAT_HANDLE_TYPE_RSP, DAT_HANDLE_TYPE_CSP, DAT_HANDLE_TYPE_PZ, DAT_HANDLE_TYPE_LMR, DAT_HANDLE_TYPE_RMR, or DAT_HANDLE_TYPE_CNO*.

dat_get_handle_type is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *dat_handle* is invalid.

6.2.2.3.1 USAGE

Consumers can use this operation to determine the type of Object being returned. This is needed for calling an appropriate query or any other operation on the Object handle.

6.2.2.3.2 RATIONALE

6.2.2.3.3 MODEL IMPLICATIONS

6.3 EVENT MANAGEMENT

6.3.1 EVENT MODEL

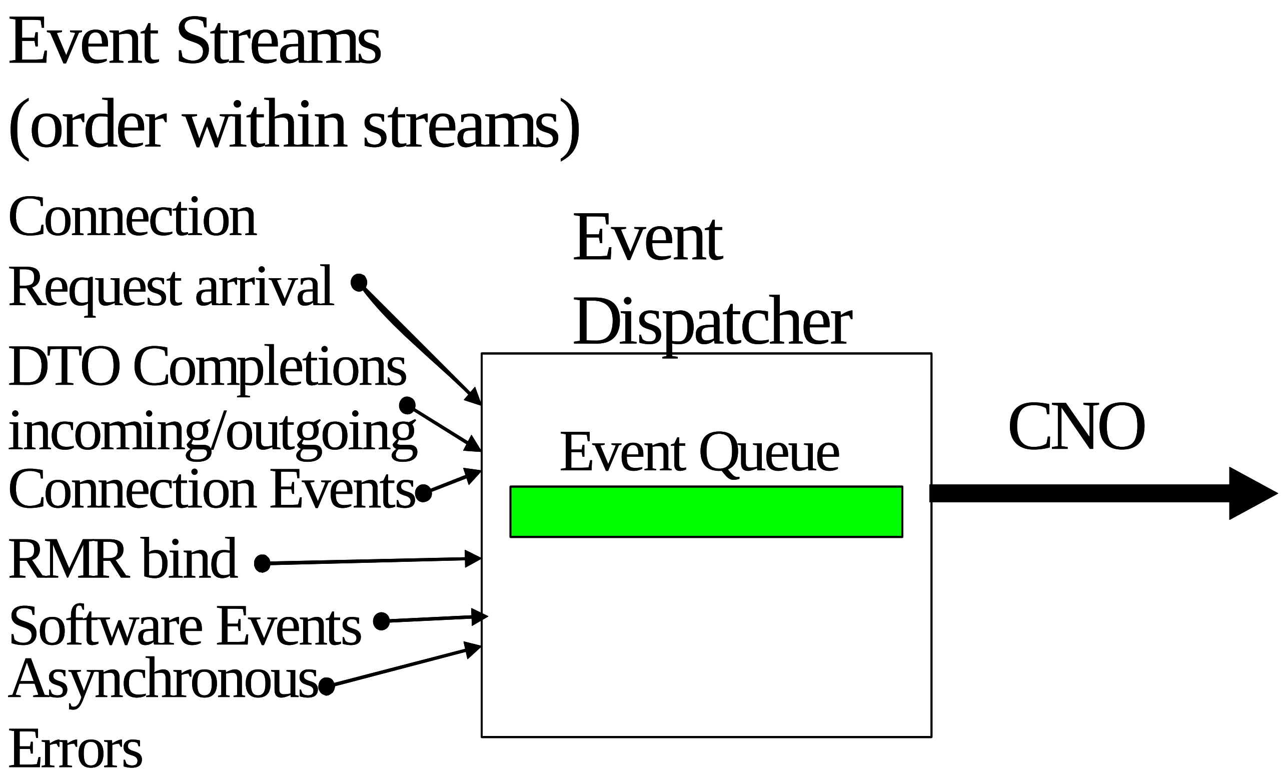


Figure 1 Event Model

uDAPL provides a common model for notifications and connection management events, data transfer completions, asynchronous errors, and all other notifications. These are logically grouped into Event Streams.

Event streams feed into Event Dispatchers, which provide queues to gather the events.

- An **Event Stream** is a source of notifications. For uDAPL, these include the following:
  - Data transfer completions
  - Connection Request arrivals
  - Connection events, including connection establishment completions, disconnect notifications, timed out, unreachable, and other connection events
  - Remote memory bind completions

- IA Asynchronous errors and events including HA events. Note that SRQ limit notifications are not neccessarily “errors”.
- Software (user) generated notications

- An **Event Dispatcher** merges events from one or more event streams into a single conceptual queue that the Consumer can dequeue. The Event Dispatcher is responsible for completing any transport-specific fetching and handshaking for the events it is reporting. Each event is dequeued exactly once.
- The Consumer can directly wait on a single Event Dispatcher that dequeues the first event from the Event Dispatcher queue. Wait optionally blocks the current thread. The Consumer controls blocking via *timeout* and *threshold* parameters, and by completion flags of posted DTOs and RMRs for their completion events. The Consumer cannot wait on an Unwaitable Event Dispatcher.
- A Consumer can wait for a Notification event or any event if an EVD is configured for *DAT_EVD_STATE_CONFIG_THRESHOLD* (These events are called Notifiable events.) on a set of Event Dispatchers from a single Provider using a **CNO**. The Consumer is awakened with a handle to an Event Dispatcher that had a notifiable event. The Consumer can then collect events from that or any other Event Dispatchers.
- A **CNO** can optionally trigger an OS-dependent interprocess communications method using an **OS Wait Proxy Agent**. At least one form of proxy agent is defined for each host OS. Providers can define additional variations.
- A specialized version of CNO that used File Descriptor for OS Wait Proxy Agent can be used by a Consumer. For that type of CNO Consumer can use poll or select functionality of File Descriptor to get unblocked when CNO triggers File Descriptor as OS Wait Proxy Agent. When File Descriptor returns from poll or select, Consumer does not know which EVD triggered the unblocking. Consumer can use *dat_cno_trigger* call which returns the latest EVD which triggered CNO. Notice that this may be not the EVD which caused FD to return.

Event Streams are identified in a transport independent fashion. Each Event Stream can be assigned to only one Event Dispatcher. Multiple Event Streams can be assigned to a single Event Dispatcher. Interactions between an Event Dispatcher and its associated Event Streams can be transport- and OS-dependent as well as Provider (IA)-specific.

uDAPL defines a NULL handle (*DAT_HANDLE_NULL*). In the Event Dispatcher case, it represents the NULL Event Dispatcher. Consumers can assign one or more Event Streams to the NULL Event Dispatcher if they choose to ignore the notifications. The NULL Event Dispatcher is not the same as a disabled Event Dispatcher. The NULL Event Dispatcher never overflows; all incoming events are just dropped. The NULL Event

Dispatcher does not support any Event Dispatcher operations except *dat_evd_query.*

In contrast, a disabled Event Dispatcher supports all Event Dispatcher operations, and incoming events are queued. The queue can overflow and, therefore, generate an overflow error that is reported on the IA Asynchronous error Event Dispatcher.

Consumers can also generate events. These events form a separate stream: the Software Event stream. They are always Notification events and cannot be masked as uDAPL Provider events (the assumption being that is not a reason for the Software Event Stream). Providers can support a separate operation on an Event Dispatcher that provides a Consumer the ability to mask Consumer events as uDAPL events, but they must not use DAT function name prefixes for the Consumer routines. This operation can be very useful for debugging.

Extension objects can also generate events. These events do not form a separate stream but use one of the regular event streams. But in order to simplify Consumer event logic a pseudo event type of DAT_EXTENSION_EVENT is defined.

An Event Dispatcher instance can only process Event Streams from a single IA. An IA can have multiple Event Dispatcher instances associated with it.

The High Availability events are associated with the Asynchronous IA EVD. Consumer should ensure that the Asynchronous IA EVD is sized appropriately for HA events.

The Event Dispatcher is responsible for ensuring that the Event Streams are emptied. Ensuring that this is done promptly is left to the market forces and/or later versions of this specification.

Ordering of events within an Event Stream must be preserved. Ordering of events between different Event Streams is defined as follows and only for events of the same connection:

- A Connection Establishment event precedes any Data Transfer Completion events.
- All pending data transfer completion events that are completed successfully should precede a Disconnect event.
- There is no order implied between pending Data Transfer Completion events that are completed with an error and a Disconnect event.

This ordering is guaranteed only if Provider supports merging of connection and DTO completion events.

An Event Dispatcher can have zero or one CNO associated with it. Each CNO can have zero or one OS Wait Agent Proxy or File Desciptors associated with it. The same CNO can be associated with any number of EVDs from zero to all EVDs of a single IA instance.

An Event Dispatcher supports DTO and RMR Completions whose notification are controlled by either local *DAT_COMPLETION_UNSIGNALLED_FLAG* or remote *DAT_COMPLETION_SOLICITED_WAIT_FLAG* of posted DTO or RMR, but not both. When Event Dispatcher is used by an Endpoint that is configured for Notification controlled completion, either locally or remotely via posted DTO/RMR, then arrival of non-notification events does not effect the waiter. If an Event Dispatcher supports DTO and RMR Completions whose notification are controlled by either local *DAT_COMPLETION_UNSIGNALLED_FLAG* or remote *DAT_COMPLETION_SOLICITED_WAIT_FLAG* of posted DTO or RMR and the Event Dispatcher also supports non DTO Completions event streams then an event arrival on any of these streams will unblock a waiter if one exists.

An Event Dispatcher supports *threshold* value other than one if the Event Dispatcher is not used by an Endpoint that is configured for Notification control either local *DAT_COMPLETION_UNSIGNALLED_FLAG* or remote *DAT_COMPLETION_SOLICITED_WAIT_FLAG*. Otherwise, an attempt to wait with *threshold* value other then 1 results in immediate error *DAT_INVALID_STATE.* If an Event Dispatcher supports DTO and RMR Completions whose notification are controlled by either local *DAT_COMPLETION_UNSIGNALLED_FLAG* or remote *DAT_COMPLETION_SOLICITED_WAIT_FLAG* of posted DTO or RMR, and the Event Dispatcher also supports non DTO Completions event streams, than an event arrival on any of these streams counts toward *threshold*.

Events of all event streams but Request and Recv completions are always Notification Events. The Notification vs. Non-Notification status of Request and Recv Completion events are controlled by the Consumer via EP attributes and posted Recv and Request *completion_flags.*

De-facto EVD is configured for either locally controlled Notification flag (*DAT_COMPLETION_UNSIGNALLED_FLAG)*, remotely controlled Notification flag *(DAT_COMPLETION_SOLICITED_WAIT_FLAG)* or *threshold* argument of *dat_evd_wait.*

When an Event Dispatcher has a blocked waiter, the following logic applies:

1) The first notification event that arrives (if EVD is used for EP completion streams, the EP attribute for that completion stream must be configured for notification events [*DAT_COMPLETION_UNSIGNALLED_FLAG* or *DAT_COMPLETION_SOLICITED_WAIT_FLAG* for Receive Completion Type for Receives; *DAT_COMPLETION_UNSIGNALLED_FLAG* for Request Completion Type for Send, RDMA Read, RDMA Write and RMR Bind]) unblocks the waiter. The **first** event is delivered to the waiter.

2) An event whose arrival (if EVD is not used by Endpoints or used only by EP completion streams that are configured for *DAT_EVD_STATE_CONFIG_THRESHOLD*) reaches *threshold* number of events on the EVD unblocks the waiter. The **first** event is delivered to the waiter.

3) If the timeout period expires without an event arrival that crosses *threshold*, the total number of queued events must still be checked. If the threshold is reached, the caller should be unblocked with DAT_SUCCESS.

4) Even if specified and enabled, the CNO is never triggered by an Event Dispatcher with a blocked waiter.

When an Event Dispatcher does not have a blocked waiter, the following logic applies:

1) The notified status is sticky. Once signaled, the Event Dispatcher remains signaled, allowing the next wait to unblock immediately. This is a notification status, not a count. No matter how many notification events are queued, the first unblock or dequeue clears the notification status.

2) Events are queued, whether in an Event Stream-specific stream (which can be a hardware resource) or by the Event Dispatcher itself.

3) If the Event Dispatcher is enabled and has a specified CNO, the CNO is triggered with the Event Dispatcher's handle.

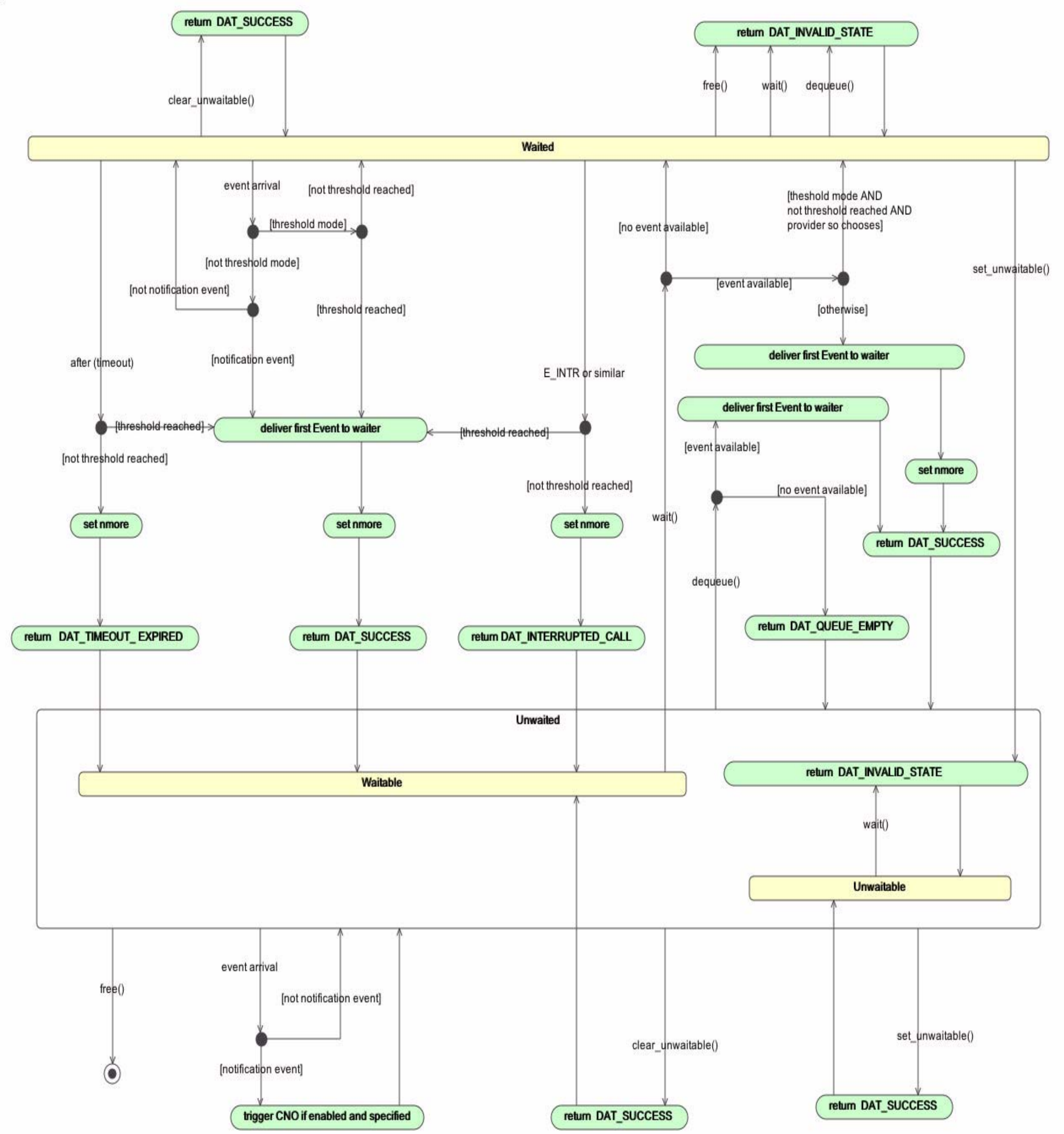
4) If there are Consumer threads waiting on it, the CNO unblocks one of them (which one is implementation-dependent) and passes the handle for this or any other Event Dispatcher that has an event.

5) Whether or not the CNO unblocked a waiter, it then triggers the associated OS Wait Proxy Agent(if there is one), passing the handle for this or any other Event Dispatcher that has an event on its event queue. Triggering an OS Wait Proxy Agent disassociates it from the CNO. It must be rearmed by the Consumer for each use.

6) If CNO has an associated File Descriptor then it if there is a thread polling or select on File Descriptor then CNO triggers return of the FD but not the delivery of the DAT_EVD_HANDLE as it would for OS Wait Proxy Agent.

   The semantic of the File Descriptor and its functionality is defined by the platform, with POSIX semantic expected. As with the generic OS Wait Proxy Agent, CNO must be rearm for the File Descriptor. The poll and select calls on the File Descriptor associated with the CNO rearm CNO for File Descriptor automatically. See Section 6.3.3.1, "File Descriptor," on page 125 for details.

Below is an EVD state transition diagram.



STATE: WAITED - The EVD was claimed by a waiter.

Action: *dat_evd_clear_unwaitable* - return *DAT_SUCCESS*, no change.

Action: *dat_evd_free* - return *DAT_INVALID_STATE*, no change.

Action: *dat_evd_wait* - return *DAT_INVALID_STATE*, no change.

Action: *dat_evd_dequeue* - return *DAT_INVALID_STATE*, no change.

Action: E_INTR or similar (Unix or equivalent semantic)

- if *threshold* is reached
  - deliver first Event to waiter
  - set *nmore*
  - return *DAT_SUCCESS*
  - transfer to Waitable state.
- else *threshold* is not reached
  - set *nmore*
  - return *DAT_INTERRUPTED_CALL*
  - transfer to Waitable state.

Action: *timeout* reached

- if *threshold* reached
  - deliver first Event to waiter
  - set *nmore*
  - return *DAT_SUCCESS*
  - transfer to Waitable state.
- else *threshold* is not reached
  - set *nmore*
  - return *DAT_TIMEOUT_EXPIRED*
  - transfer to Waitable state.

Action: *dat_evd_set_unwaitable*

  - return *DAT_INVALID_STATE*
  - transfer to Unwaitable state.

Action: event arrival

- if ([*DAT_EVD_STATE_CONFIG_THRESHOLD* mode] AND [*threshold* is reached]) OR (notification event) arrived
  - deliver first Event to waiter
  - set *nmore*
  - return *DAT_SUCCESS*
  - transfer to Waitable state
- else
  - no change

Comment: This was clarified by incorporating the entire behavior, whether uDAPL, verb layer, or hardware. Internal divisions are not shown here. Prior diagrams attempted a typical DAT/verb layer boundary.

STATE: UNWAITED (superstate for Waitable and Unwaitable)

Action: *dat_evd_free* - delete object.

Action: *dat_evd_clear_unwaitable*

- return *DAT_SUCCESS*
- transfer to Waitable state

Action: *dat_evd_set_unwaitable*

- return *DAT_SUCCESS*
- transfer to Unwaitable state

Action: *dat_evd_dequeue*

- if event available
  - deliver first Event
  - return *DAT_SUCCESS*
- otherwise
  - return *DAT_QUEUE_EMPTY*

Action: event arrival

- if notification event, kick CNO if enabled and specified

Action: *dat_evd_wait* -- from Waitable

- if (no events available) *OR* ([*DAT_EVD_STATE_CONFIG_THRESHOLD* mode] *AND* [*threshold* not reached] *AND* [Provider chooses to make this check])
  - transfer to Waited state
- otherwise
  - deliver first Event
  - set *nmore*
  - return *DAT_SUCCESS*

Action *dat_evd_wait* -- from Unwaitable

- return DAT_INVALID_STATE
- no state change

It is up to the Consumer to dequeue events from the Event Dispatcher after they are unblocked. Because there are might be events on the queue of any of the Event Dispatchers, it is the Consumer’s responsibility to check all the Event Dispatcher queues on which CNO was blocked.

There is a nondeterministic time span between an Event Dispatcher triggering a CNO and the waiting uDAPL Consumer being unblocked. Therefore, four Event Dispatchers triggering the same CNO in a short period could result in anywhere from one to four uDAPL Consumers being unblocked.

When a CNO is triggered before there is a waiter, it remembers that it was triggered. When the next Consumer waits on the CNO, it is immediately given the handle of an Event Dispatcher that has been notified, or any other Event Dispatcher that has an event.

Note for Provider: To the extent that it is possible for the uDAPL Provider to efficiently determine the true time ordering of events on different Event Streams, it should preserve that order when dispatching events.

Note to Consumer: Consumers are responsible for synchronizing dequeueing. When multiple Consumer threads are trying to dequeue an event from the same Event Dispatcher, the order is not defined.

Note for Provider: Provider should not wake up the Consumer prematurely when a *threshold* greater than 1 is requested.

Note for Consumer: Consumer should be able to tolerate eager behavior from the Provider. Although the Provider should not wake up the Consumer prematurely when *threshold* is set to a number greater than 1, it is allowed not to block if there are events on the EVD, even if their number is smaller than *threshold.* Consumer should use *threshold* for performance improvement, not as a semantic guarantee.

Note to Consumer: Consumers can use a simple polling model using the Event Dispatcher. This can be achieved when the Event Dispatcher instance does not have any CNO associated with it. Consumers use dequeue operation to get events from the Event Dispatcher. It is then up to the Consumer to ensure that the event queue does not overflow.

An event queue overflow generates an asynchronous error on the IA Event Dispatcher irrespective of the underlying RDMA Transport. Overflow of the Asynchronous Error Event Dispatcher is a catastrophic error; behavior of the Provider after that is undefined. The behavior of the Provider after it posts a catastrophic error is undefined. The Provider can consolidate multiple overflows of the same event queue into a single notification. In general, the Provider is free to consolidate multiple error notifications of the same type. Connections are not broken when an associated Event Dispatcher for the connection local Endpoint has the queue overflow condition. All cleanup of a queue overflow is left to the Consumer.

A connection whose DTO/RMR completion posting caused EVD to overflow will be broken. Consumer may not have any way to clean up the overflown EVD queue that supports DTO and/or RMR Bind completion. The EVD may no longer be usable at all, and access to events on it may no longer be supported. The only thing that Consumers can do is to destroy EVD, but first all connections that use it must be disconnected.

An EVD that only supports Connection Request, Connection events and/or Software events can never be made unusable by an overflow condition. Additionally, when an overflow would have occurred an EVD would handle it as described below based on event stream type.

A Software Event that would have caused an overflow is rejected synchronously. There is no overflow event, and the EVD remains operational.

The overflow behavior of events on the extension event stream is defined by the DAT extension document and DAT Provider. It is recommended that extension events should not cause overflow of their EVD but if HW is generating extension events that is not always possible. It is recommended that Providers use DAT event stream without utilizing extension event stream whenever possible. Consumers should read DAT extension specification as Provider documentation to find out extension event stream behavior.

if a Connection Request would have caused an EVD overflow, the Provider simply rejects the peer's request on its own. There is no overflow event, and the EVD remains operational.

Connection events from remote host are not rejected by the Provider. These include remote peer and non-peer rejections and disconnects. The requested action takes place. When connection events cannot be posted due to the overflow of the EVD, an overflow event is generated, however unlike DTO/RMR overflow events, the EVD is guaranteed to remain operational. Events that should be posted to the overflown EVD are lost. Once the space on the EVD queue becomes available the EVD performs as normal non-overflown EVD.

Events that are posted to the overflown queue (by the Provider or Consumer) are dropped by a Provider; they do not effect other events on the event queue of the Event Dispatcher.

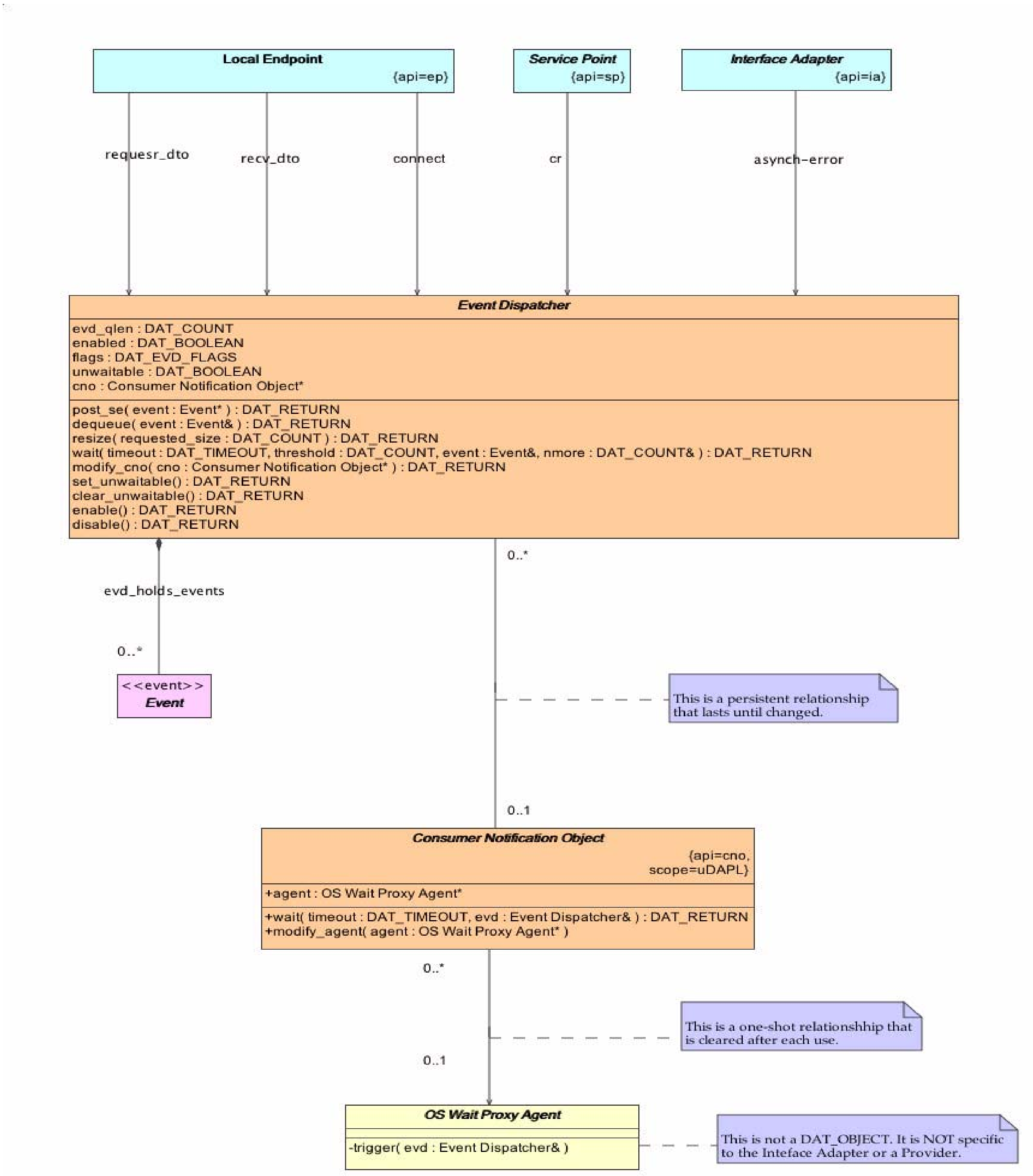
Note to Consumer: It is up to the Consumer to configure the Event Dispatcher and dequeue events fast enough to avoid an overflow condition.

The Consumer doesn’t know which memory to free and how to recover resources because the Provider owns the DAT Object’s memory. *dat_ia_close* must always be callable to do a shutdown and clean up resources. The Provider must ensure that all resources are cleaned up regardless of the overflow and other conditions.

The Consumer can only free the Event Dispatcher if it does not have any associated Event Streams, except for the Software Event Stream, Event Streams generated by the IA Object (asynchronous error Event Dispatcher and possibly high availability event stream) and potentially extension event stream, and if it does not have any waiting on it. When an Event Dispatcher is freed, all events on its event queue are lost.

When the IA is closed either by a Consumer or abnormally, all waiters on EVDs and CNOs are unblocked by the Provider. All waiters on all CNOs, including the OS Wait Proxy Agents, are unblocked with the NULL handle returns for an unblocking EVD. All direct waiters on all EVDs are also unblocked and are returned with *DAT_ABORT*.

The following diagram outlines the Event Dispatcher “class” diagram.


Local Endpoint
{api=ep}
Service Point
{api=sp}
Interface Adapter
{api=ia}
requesr_dto
recv_dto
connect
cr
asynch-error
Event Dispatcher
evd_qlen : DAT_COUNT
enabled : DAT_BOOLEAN
flags : DAT_EVD_FLAGS
unwaitable : DAT_BOOLEAN
cno : Consumer Notification Object*
post_se(event : Event*) : DAT_RETURN
dequeue(event : Event&) : DAT_RETURN
resize(requested_size : DAT_COUNT) : DAT_RETURN
wait(timeout : DAT_TIMEOUT, threshold : DAT_COUNT, event : Event&, nmore : DAT_COUNT&) : DAT_RETURN
modify_cno(cno : Consumer Notification Object*) : DAT_RETURN
set_unwaitable() : DAT_RETURN
clear_unwaitable() : DAT_RETURN
enable() : DAT_RETURN
disable() : DAT_RETURN
evd_holds_events
0..*
0..*
<<event>>
Event
This is a persistent relationship that lasts until changed.
0..1
Consumer Notification Object
{api=cno, scope=uDAPL}
+agent : OS Wait Proxy Agent*
+wait(timeout : DAT_TIMEOUT, evd : Event Dispatcher&) : DAT_RETURN
+modify_agent(agent : OS Wait Proxy Agent*)
0..*
This is a one-shot relationshhip that is cleared after each use.
0..1
OS Wait Proxy Agent
-trigger(evd : Event Dispatcher&)
This is not a DAT_OBJECT. It is NOT specific to the Inteface Adapter or a Provider.

6.3.1.1 uDAPL versus kDAPL Event Dispatchers

Note to Provider: Both uDAPL and kDAPL Event Dispatchers are serialization points that conceptually merge multiple Event Streams. They differ in how those Events are delivered to the Consumer and in how the Consumer is notified that Notifiable Events have been placed on the Event Dispatcher's queue.

uDAPL does not have any callback/UpCall capabilities. uDAPL Event Dispatchers define a method allowing the Consumer to block waiting for a *threshold* number of events on the queue, or for a notification event prior to a *timeout* expiration. A CNO object can play a similar role for uDAPL as an UpCall does for kDAPL, as notifiable events trigger a CNO and optionally an OS Proxy Wait Agent when an associated set of EVDs do not have waiters.

6.3.2 Consumer Notification Object

A CNO allows a Consumer to wait for a notification event on any of a large number of Event Dispatchers. A CNO is a DAT Object, and as such it has a scope of a single Interface Adapter. The association between a CNO and an EVD is persistent and remains in place until the Consumer explicitly changes it. However, a CNO can be configured to trigger an OS Wait Proxy Agent whenever it is triggered. This allows a Consumer to wait on a mix of events from multiple Interface Adapters and even non-DAT Events. This must be done in an OS-specific manner. uDAPL only specifies how the CNO triggers the OS-specific resource through the proxy agent. How the consumer actually waits on it is OS-specific. The invoking of the OS Wait Proxy Agent disassociates it from all CNOs it is associated with. Consumers can associate the same or another agent with a CNO using *dat_cno_modify_agent.* There can be, at most, one agent associated with the CNO instance. When there is no associated agent for the CNO, the query value for the Proxy agent is DAT_OS_WAIT_PROXY_AGENT_NULL. The special version of CNO which uses File Descriptor as OS Wait Proxy Agent follows the same rules. File Descriptor has a permanent association with the CNO which created it. The poll and select on a File Descriptor play the role of arming OS Wait Proxy Agent.

CNOs also support multiple concurrent waiters, even when the uDAPL Provider is not otherwise thread-safe. This allows a small number of "worker threads" to service a larger pool of Endpoints using one or more Event Dispatchers. The unblocked waiters do not get an event, but an EVD that has an event. This is in contrast to the EVD waiter, which gets an event from EVD and is a single waiter on the EVD. If there is a direct waiter on EVD, the EVD-associated CNO is not triggered until the waiter is unblocked.

Each trigger of the CNO object unblocks one of the CNO waiters and the OS Wait Proxy Agent if is associated with the CNO. DAT does not define an algorithm that defines which of multiple waiters of the CNO object is unblocked. This algorithm shall follow the convention of the platform.

The same CNO instance can be associated with multiple EVDs of the same instance of the Interface Adapter. This allows Consumers to funnel triggers from multiple EVDs to a single CNO.

6.3.2.1 DAT_CNO_CREATE

Synopsis:

```
DAT_RETURN
   dat_cno_create (
   IN     DAT_IA_HANDLE                 ia_handle,
   IN     DAT_OS_WAIT_PROXY_AGENT       agent,
   OUT    DAT_CNO_HANDLE                *cno_handle
   )
```

Parameters:

ia_handle:	Handle for an instance of DAT IA.
agent:	Pointer to an optional OS Wait Proxy Agent that is to be invoked whenever CNO is invoked. *DAT_OS_WAIT_PROXY_AGENT_NULL* indicates that there is no proxy agent.
cno_handle:	Handle for the created instance of CNO.

Description:

dat_cno_create creates a CNO instance. Upon creation, there are no Event Dispatchers feeding it.

agent specifies the proxy agent, which is OS-dependent and which is invoked when the CNO is triggered. After it is invoked, it is no longer associated with the CNO. The value of *DAT_OS_WAIT_PROXY_AGENT_NULL* specifies that no OS Wait Proxy Agent is associated with the created CNO.

Upon creation, the CNO is not associated with any EVDs, has no waiters and has, at most, one OS Wait Proxy Agent.

dat_cno_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* is invalid.

DAT_INVALID_PARAMETER	One of the parameters was invalid, out of range, or a combination of parameters was invalid. *agent*: is invalid.

6.3.2.2 DAT_CNO_FD_CREATE

Synopsis:

```
DAT_RETURN
   dat_cno_fd_create (
   IN     DAT_IA_HANDLE                 ia_handle,
   OUT    DAT_FD                        *os_fd,
   OUT    DAT_CNO_HANDLE                *cno_handle
   )
```

Parameters:

ia_handle:	Handle for an instance of DAT IA.
os_fd:	a file descriptor in Unix, i.e. struct pollfd or an equivalent object in other OSes.
cno_handle:	Handle for the created instance of CNO.

Description:

dat_cno_fd_create creates a CNO instance. Upon creation, there are no Event Dispatchers feeding it.

os_fd is a File Descriptor in Unix, i.e. struct pollfd or an equivalent object in other OSes that is always associates with the created CNO. Consumer can multiplex event waiting using UNIX poll or select functions.

Upon creation, the CNO is not associated with any EVDs, has no waiters and has the *os_fd* associated with it.

dat_cno_fd_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.3.2.2.1 USAGE

6.3.2.2.2 RATIONALE

6.3.2.2.3 MODEL IMPLICATIONS

6.3.2.3 DAT_CNO_FREE

Synopsis:

```
DAT_RETURN
   dat_cno_free (
   IN     DAT_CNO_HANDLE     cno_handle
   )
```

Parameters:

cno_handle:	Handle for an instance of the CNO.

Description:

dat_cno_free destroys a specified instance of the CNO.

A CNO cannot be deleted while it is referenced by an Event Dispatcher or while a thread is blocked on it.

It is illegal to use the destroyed handle in any subsequent operation.

dat_cno_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cno_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. CNO is in use by an EVD instance or there is a thread blocked on it.

6.3.2.3.1 USAGE

If there is a thread blocked in *dat_cno_wait*, the Consumer can do the following steps to unblock the waiter:

- Create a temporary EVD that accepts software events. It can be created in advance.
- For a CNO with the waiter, attach that EVD to the CNO and post the software event on the EVD.
- This unblocks the CNO.
- Repeat for other CNOs that have blocked waiters.
- Destroy the temporary EVD after all CNOs are destroyed and the EVD is no longer needed.

6.3.2.3.2 RATIONALE

6.3.2.3.3 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

The *dat_cno_free* for FD CNO will invoke an OS-specific *close* on the CNO associated File Descriptor. *close* system call follows the OS semantic. That is, it will block until all pending operations on the File Descriptor complete including any *poll, select* calls.

6.3.2.4 DAT_CNO_WAIT

Synopsis:

```
DAT_RETURN
   dat_cno_wait (
   IN     DAT_CNO_HANDLE          cno_handle,
   IN     DAT_TIMEOUT             timeout,
   OUT    DAT_EVD_HANDLE          *evd_handle
   )
```

Parameters:

cno_handle: Handle for an instance of CNO.

timeout: The duration to wait for a notification. The value *DAT_TIMEOUT_INFINITE* can be used to wait indefinitely.

evd_handle: Handle for an instance of EVD.

Description:

dat_cno_wait allows the Consumer to wait for notification events from a set of Event Dispatchers all from the same Interface Adapter. The Consumer blocks until notified or the timeout period expires.

It is important to remember that an Event Dispatcher that is disabled or in the “Waited” state does **not** deliver notifications. A uDAPL Consumer waiting directly upon an Event Dispatcher preempts the CNO.

All providers must support multiple waiters on a CNO, even if they are otherwise not considered “thread safe.”

The consumer can optionally specify a timeout, after which it is unblocked even if there are no notification events. On a timeout, *evd* is explicitly set to a NULL handle.

The returned *evd* handle is only a hint. Another Consumer can reap the Event before this Consumer can get around to checking the Event Dispatcher. Additionally, other Event Dispatchers feeding this CNO might have been notified. The Consumer is responsible for ensuring that all

EVDs feeding this CNO are polled regardless of whether they are identified as the immediate cause of the CNO unblocking.

All waiters on the CNO, including the OS Wait Proxy Agent if it is associated with the CNO, are unblocked with the NULL handle returns for an unblocking EVD *evd_handle* when the IA instance is destroyed or when all EVDs the CNO is associated with are freed.

dat_cno_wait is blocking and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cno_handle* is invalid.
DAT_QUEUE_EMPTY	The operation timed out without a notification.
DAT_INVALID_PARAMETER	One of the parameters was invalid or out of range, or a combination of parameters was invalid. *timeout* is invalid.
DAT_INTERRUPTED_CALL	[*Unix only*] The operation was interrupted by a signal.

6.3.2.4.1 Usage

Consumers can unblock a waiting thread by posting *dat_evd_post_se* to an EVD assosicated with CNO. This platform-independent method ensures that there is a notification event on the EVD queue. If there is a race between *dat_cno_wait* and *dat_evd_post_se* since there is a notification event the waiter will either be unblocked or it would not block at all. Consumer can either have a separate EVD feeding the CNO which support SE event stream or have an EVD on which it expect event to be configured so it can support SE event stream.

Consumer can use Platform-specific methods for unblocking a waiter that may result in DAT_INTERRUPTED_CALL. See the OS specific notes for more details (see “Operating System Specific Notes” on page 303).

6.3.2.4.2 Rationale

6.3.2.4.3 Model Implications

6.3.2.5 DAT_CNO_TRIGGER

Synopsis:

```
DAT_RETURN
   dat_cno_trigger (
   IN     DAT_CNO_HANDLE          cno_handle,
   OUT    DAT_EVD_HANDLE          evd_handle
   )
```

Parameters:

cno_handle:	Handle for an instance of CNO.
evd:	Handle for the latest EVD that triggered CNO.

Description: *dat_cno_trigger* returns the latest EVD that triggered the CNO.

dat_cno_trigger is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cno_handle* is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.3.2.6 DAT_CNO_MODIFY_AGENT

Synopsis:

```
DAT_RETURN
   dat_cno_modify_agent (
   IN    DAT_CNO_HANDLE                cno_handle,
   IN    DAT_OS_WAIT_PROXY_AGENT       agent
   )
```

Parameters:

cno_handle:	Handle for an instance of CNO.
agent:	Pointer to an optional OS Wait Proxy Agent to invoke whenever CNO is invoked. *DAT_OS_WAIT_PROXY_AGENT_NULL* indicates that there is no proxy agent.

Description: *dat_cno_modify_agent* modifies the OS Wait Proxy Agent associated with a CNO. If non-null, any trigger received by the CNO is also passed to the OS Wait Proxy Agent. This is in addition to any local delivery through the CNO. The Consumer can pass the value of *DAT_OS_WAIT_PROXY_AGENT_NULL* to disassociate the current Proxy agent from the CNO.

dat_cno_modify_agent is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cno_handle* is invalid.

DAT_INVALID_PARAMETER	One of the parameters was invalid, out of range, or a combination of parameters was invalid. *agent*: is invalid.

6.3.2.7 DAT_CNO_QUERY

Synopsis:

```
DAT_RETURN
   dat_cno_query (
   IN     DAT_CNO_HANDLE           cno_handle,
   IN     DAT_CNO_PARAM_MASK       cno_param_mask,
   OUT    DAT_CNO_PARAM            *cno_param
   )
```

Parameters:

cno_handle: Handle for the created instance of the Consumer Notification Object.

cno_param_mask: Mask for CNO parameters.

cno_param: Pointer to a Consumer-allocated structure that the Provider fills with CNO parameters.

Description:

dat_cno_query provides the Consumer parameters of the CNO. The Consumer passes in a pointer to the Consumer-allocated structures for CNO parameters that the Provider fills.

cno_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *cno_param_mask* requested parameters. The Provider can return values for any other parameters.

The return value of DAT_OS_WAIT_PROXY_AGENT_NULL indicates that there are no Proxy Agent associated with the CNO.

dat_cno_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *cno_param_mask* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *cno_handle* is invalid.

6.3.2.7.1 Usage

6.3.2.7.2 Rationale

6.3.2.7.3 Model Implications

6.3.3 OS Wait Proxy Agent

Creation of OS Wait Proxy Agents is Host OS- and/or Provider-dependent. The only DAT-defined operation on OS Wait Proxy Agents is that they can be triggered with a DAT_EVD_HANDLE by CNOs. Specific types of OS Wait Proxy Agents can have additional methods for waiting on them, or they can trigger an external OS-specific semaphore, message queue, or file descriptor instead.

OS Wait Proxy Agents are not DAT Objects. A consumer can use one created by one Provider when interfacing with another. However, each Provider is required to implement at least one class of OS Wait Proxy Agent to interface with a specific OS resource that will be defined by the DAT Collaborative for each OS. For any OS that has the equivalent of a Unix file descriptor, that mechanism is selected by default.

6.3.3.1 File Descriptor

The semantic of the File Descriptor and its functionality is defined by the platform, with POSIX semantic expected. As with the generic OS Wait Proxy Agent, CNO must be rearm for the File Descriptor. The *poll* and *select* calls on the File Descriptor associated with the CNO rearm CNO for File Descriptor automatically.

The File Descriptor used for CNO is *read* FD. The *poll* and *select* calls on CNO FD operate as defined by OS. If a thread that *poll* or *select* on the File Descriptor then CNO triggers return of the FD. Consumer should use *POLLIN* for the *poll* and *select* calls. For *select* the FD should be in the read list. On the return the FD event should be *POLLIN* under normal, non-error, situation.

The CNO FD supports only 4 operations: *select, poll, close* and *read.* When FD returns from CNO Consumer can *read* FD to get the latest EVD which triggered the CNO. So the CNO maintains the latest EVD which triggered it. The EVD which Consumer gets may not be the EVD which caused FD to return. And by the time Consumer checks the EVD for event, it may not have any. This is a typical race condition which multithreaded application needs to deal with. The Consumer can also call CNO directly (*dat_cno_trigger*), to find out which EVD triggered it last.

close destroys FD and destroy CNO association with it. Querying CNO after FD *close* will report that there is no FD or OS Wait Proxy Agent associated with CNO.

CNO FDs are not inherited across fork like any other uDAT objects.

6.3.4 EVENT DISPATCHER

6.3.4.1 DAT_EVD_CREATE

Synopsis:

```
DAT_RETURN
  dat_evd_create (
  IN     DAT_IA_HANDLE          ia_handle,
  IN     DAT_COUNT              evd_min_qlen,
  IN     DAT_CNO_HANDLE         cno_handle,
  IN     DAT_EVD_FLAGS          evd_flags,
  OUT    DAT_EVD_HANDLE         *evd_handle
  )
```

Parameters:

ia_handle: Handle for an instance of DAT IA.

evd_min_qlen: Minimum size of the queue for events.

cno_handle: Handle of the CNO to be triggered when there are notification Events for this Event Dispatcher and it is enabled and not directly wait blocked. The value of *DAT_HANDLE_NULL* specifies no CNO.

-

evd_flags: Flag for Event Dispatcher (See Appendix A.4). Default value is DAT_EVD_DEFAULT_FLAG. Table 3 lists the EVD flag definitions.

evd_handle: Handle for an instance of Event Dispatcher.

Table 3 EVD Flag Definitions

Event Stream	Definition	Description
Software	DAT_EVD_SOFTWARE_FLAG	Consumer can post software events to the EVD using *dat_evd_post_SE.*
Connection Requests	DAT_EVD_CR_FLAG	The EVD can be associated with the Public or Private Service Point to get connection requests.
Data Transfer Operations	DAT_EVD_DTO_FLAG	The EVD can be used for DTOs (*recv_evd, request_evd)* of any Endpoints.
Connections	DAT_EVD_CONNECTION_FLAG	The EVD can be used as *connect_evd* of any Endpoints.
RMR Bind	DAT_EVD_RMR_BIND_FLAG	The EVD can be used as *rmr_bind_evd* of any Endpoints.

Table 3 EVD Flag Definitions (Continued)

Event Stream	Definition	Description
Asynchronous events	DAT_EVD_ASYNC_FLAG	The EVD can be used for the Interface Adapter in *dat_ia_open* for *async_evd*.
Extension events	DAT_EVD_EXT_FLAG	The EVD can be used for Extension Objects.
All Provider Streams	DAT_EVD_DEFAULT_FLAG	The EVD can be used for any DAT objects except Software and Extension event streams. This is the union of DAT_EVD_CR_FLAG, DAT_EVD_DTO_FLAG, DAT_EVD_CONNECTION_FLAG, DAT_EVD_ASYNC_FLAG and DAT_EVD_RMR_BIND_FLAG.

Description: *dat_evd_create* creates an instance of Event Dispatcher. Upon creation, it does not have any Event Streams feeding events to it.

evd_min_qlen defines the size of the event queue that the Consumer requested. The Provider is required to provide a queue size of at least *evd_min_qlen,* but is free to provide a larger queue size (or provide dynamic queue enlargement when needed). The Consumer can determine the actual queue size by querying the created Event Dispatcher instance.

Specifying a *cno_handle* allows the Consumer to consolidate notifications from multiple Event Dispatchers from the same Interface Adapter to a single CNO. This can reduce the number of distinct waiting threads required for an application. Through the CNO, an OS Wait Proxy Agent can be used to enable waiting for notification events across multiple Interface Adapters or even waiting for non-DAT events.

evd_flags allows Consumers to specify what Event Streams can feed this EVD. Only a combination of merged event stream types supported by the Provider as specified by the *evd_stream_merging_supported* Provider attribute are allowed. A special constant *DAT_EVD_DEFAULT_FLAG* is defined that allows Consumers to use it for any Provider Event Streams but not for Consumer software events and extension events.

By default, the created EVD is enabled and waitable.

dat_evd_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle.

DAT_INVALID_PARAMETER	Invalid or out-of-range parameter or combination of parameters.

6.3.4.1.1 USAGE

Performance of *dat_evd_wait* may depend on the value of *evd_flags*. Consult your vendor for the optimization guidance.

For example, an EVD dedicated to DTO completions may have a better performance than EVD used for multiple event stream time. For example, a *request_evd* value of *DAT_EVD_DTO_FLAG* | *DAT_EVD_RMR_BIND_FLAG* or *DAT_EVD_DEFAULT_FLAG* & ~(*DAT_EVD_CR_FLAG* | *DAT_EVD_CONNECTION_FLAG* | *DAT_EVD_ASYNC_FLAG*) can produce higher performance for EVD operations such as *dat_evd_dequeue* and *dat_evd_wait*.

6.3.4.1.2 RATIONALE

6.3.4.1.3 MODEL IMPLICATIONS

Note to Consumer: The Consumer should check the Provider attribute for EVD stream merging support before creating an EVD that merges multiple event stream types. Because not all combinations of Event streams feeding the same EVD can be supported by the Provider, the Consumer should not rely on the ability to merge event streams of different types, except for DTO and RMR completions.

If Consumers require that an EVD be able to handle all event stream types, they should procure the Provider that provides this capability.

6.3.4.2 DAT_EVD_FREE

Synopsis:

```
DAT_RETURN
   dat_evd_free (
   IN     DAT_EVD_HANDLE     evd_handle
   )
```

Parameters:

evd_handle: Handle for an instance of the Event Dispatcher.

Description:

dat_evd_free destroys a specified instance of the Event Dispatcher.

All events on the queue of the specified Event Dispatcher are lost. The destruction of the Event Dispatcher instance does not have any effect on any DAT Objects that originated an Event Stream that had fed events to the Event Dispatcher instance. There should be no event streams feeding the Event Dispatcher except Software and Asynchronous errors event streams and no threads blocked on the Event Dispatcher when the EVD is being closed as at the time when it was created.

It is illegal to use the destroyed handle in any subsequent operation.

dat_evd_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.
DAT_INVALID_STATE	Invalid parameter. There are Event Streams associated with the Event Dispatcher feeding it.

6.3.4.2.1 Usage

Consumers are advised to destroy all Objects that originate Event Streams that feed an instance of the Event Dispatcher before destroying it. An exception to this rule is Event Dispatchers of an IA.

Freeing an IA automatically destroys all Objects associated with it directly and indirectly, including Event Dispatchers.

The Software event stream is destroyed when the EVD it feeds is destroyed.

The extension event stream is destroyed when the extension object that feeds it is destroyed.

6.3.4.2.2 Rationale

6.3.4.2.3 Model Implications

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.3.4.3 DAT_EVD_QUERY

Synopsis:

```
DAT_RETURN
   dat_evd_query (
   IN     DAT_EVD_HANDLE           evd_handle,
   IN     DAT_EVD_PARAM_MASK       evd_param_mask,
   OUT    DAT_EVD_PARAM            *evd_param
   )
```

Parameters:

evd_handle:	Handle for an instance of Event Dispatcher.
evd_param_mask:	Mask for EVD parameters.

evd_param: Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description: *dat_evd_query* provides to the Consumer parameters of the Event Dispatcher, including the state of the EVD (enabled/disabled). The Consumer passes in a pointer to the Consumer-allocated structures for EVD parameters that the Provider fills.

evd_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *evd_param_mask* requested parameters. The Provider can return values for any of the other parameters.

dat_evd_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameters; *evd_param_mask* is invalid.

6.3.4.3.1 USAGE

6.3.4.3.2 RATIONALE

6.3.4.3.3 MODEL IMPLICATIONS

6.3.4.4 DAT_EVD_MODIFY_CNO

Synopsis:

```
DAT_RETURN
   dat_evd_modify_cno (
   IN     DAT_EVD_HANDLE          evd_handle,
   IN     DAT_CNO_HANDLE          cno_handle
   )
```

Parameters:

evd_handle: Handle for an instance of the Event Dispatcher.

cno_handle: Handle for a CNO. The value of *DAT_NULL_HANDLE* specifies no CNO.

Description: *dat_evd_modify_cno* changes the associated CNO for the Event Dispatcher.

A Consumer can specify the value of *DAT_HANDLE_NULL* for *cno_handle* to associate not CNO with the Event Dispatcher instance.

Upon completion of the *dat_evd_modify_cno* operation, the passed IN new CNO is used for notification. During the operation, an event arrival can be delivered to the old or new CNO. If Notification is generated by EVD, it is delivered to the new or old CNO.

If the EVD is enabled at the time *dat_evd_modify_cno* is called, the Consumer must be prepared to collect a notification event on the EVD's old CNO as well as the new one. Checking immediately prior to calling *dat_evd_modify_cno* is not adequate. A notification could have been generated after the prior check and before the completion of the change.

The Consumer can avoid the risk of missed notifications either by temporarily disabling the EVD, or by checking the prior CNO after invoking this operation. The Consumer can disable EVD before a *dat_evd_modify_cno* call and enable it afterwards. This ensures that any notifications from the EVD are delivered to the new CNO only.

Note that even if this routine is used to disassociate a CNO from the EVD, events arriving on this EVD might cause waiters on that CNO to awaken after returning from this routine because of unblocking a CNO waiter already “in progress” at the time this routine is called. If this is the case, the events causing that unblocking are present on the EVD upon return from the *dat_evd_modify_cno* call and can be dequeued at that time.

dat_evd_modify_cno is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle.

6.3.4.4.1 Usage

6.3.4.4.2 Rationale

6.3.4.4.3 Model Implications

6.3.4.5 DAT_EVD_Enable

Synopsis:

```
DAT_RETURN
   dat_evd_enable(
   IN     DAT_EVD_HANDLE     evd_handle
   )
```

Parameters:

evd_handle: Handle for an instance of Event Dispatcher.

Description: *dat_evd_enable* enables the Event Dispatcher so that the arrival of an event can trigger the associated CNO. The enabling and disabling EVD has no effect on direct waiters on the EVD. However, direct waiters effectively take ownership of the EVD, so that the specified CNO is not triggered even if is enabled.

If the Event Dispatcher is enabled already, this operation is no-op.

dat_evd_enable is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.

6.3.4.6 DAT_EVD_DISABLE

Synopsis:

```
DAT_RETURN
   dat_evd_disable(
   IN     DAT_EVD_HANDLE     evd_handle
   )
```

Parameters:

evd_handle: Handle for an instance of Event Dispatcher.

Description:

dat_evd_disable disables the Event Dispatcher so that the arrival of an event does not affect the associated CNO.

If the Event Dispatcher is already disabled, this operation is no-op.

Note that events arriving on this EVD might cause waiters on the associated CNO to be awakened after the return of this routine because an unblocking a CNO waiter is already “in progress” at the time this routine is called or returned.

dat_evd_disable is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.

6.3.4.7 DAT_EVD_SET_UNWAITABLE

Synopsis:

```
DAT_RETURN
   dat_evd_set_unwaitable(
   IN     DAT_EVD_HANDLE     evd_handle
   )
```

Parameters:

evd_handle: Handle for an instance of Event Dispatcher.

Description: *dat_evd_set_unwaitable* transitions the Event Dispatcher into an unwaitable state. In this state, calls to *dat_evd_wait* return synchronously with a *DAT_INVALID_STATE* error, and threads already blocked in *dat_evd_wait* are awakened and return with a *DAT_INVALID_STATE* error without any further action by the Consumer. The actual state of the Event Dispatcher is accessible through *dat_evd_query* and is *DAT_EVD_UNWAITABLE* after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set unwaitable still trigger the CNO (if appropriate), and can be retrieved with *dat_evd_dequeue*. Because events can arrive normally on the EVD, the EVD might overflow; the Consumer is expected to protect against this possibility.

dat_evd_set_unwaitable is synchronous and thread-safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.

6.3.4.8 DAT_EVD_CLEAR_UNWAITABLE

Synopsis:

```
DAT_RETURN
   dat_evd_clear_unwaitable(
   IN     DAT_EVD_HANDLE     evd_handle
   )
```

Parameters:

evd_handle: Handle for an instance of Event Dispatcher.

Description: *dat_evd_clear_unwaitable* transitions the Event Dispatcher into a waitable state. In this state, calls to *dat_evd_wait* are permitted on the EVD. The actual state of the Event Dispatcher is accessible through *dat_evd_query* and is *DAT_EVD_WAITABLE* after the return of this operation.

This call does not affect a CNO associated with this EVD at all. Events arriving on the EVD after it is set waitable still trigger the CNO (if appropriate), and can be retrieved with *dat_evd_dequeue*.

dat_evd_clear_unwaitable is synchronous and thread-safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.

6.3.4.9 DAT_EVD_RESIZE

Synopsis:

```
DAT_RETURN
   dat_evd_resize(
   IN     DAT_EVD_HANDLE     evd_handle,
   IN     DAT_COUNT          evd_min_qlen
   )
```

Parameters:

evd_handle: Handle for an instance of Event Dispatcher.

evd_min_qlen: New number of events the Event Dispatcher event queue must hold.

Description: *dat_evd_resize* modifies the size of the event queue of Event Dispatcher.

Resizing of Event Dispatcher event queue should not cause any incoming or current events on the event queue to be lost. If the number of entries on the event queue is larger then the requested *evd_min_qlen*, the operation can return DAT_INVALID_STATE and not change an instance of Event Dispatcher.

dat_evd_resize is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *evd_min_qlen* is invalid.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_STATE	Invalid parameter. The number of entries on the event queue of the Event Dispatcher exceeds the requested event queue length.

6.3.4.9.1 USAGE

This operation is useful when the potential number of events that could be placed on the event queue changes dynamically.

6.3.4.9.2 RATIONALE

6.3.4.9.3 MODEL IMPLICATIONS

6.3.4.10 DAT_EVD_WAIT

Synopsis:

```
DAT_RETURN
   dat_evd_wait(
   IN     DAT_EVD_HANDLE     evd_handle,
   IN     DAT_TIMEOUT        timeout,
   IN     DAT_COUNT          threshold,
   OUT    DAT_EVENT          *event,
   OUT    DAT_COUNT          *nmore
   )
```

Parameters:

evd_handle: Handle for an instance of the Event Dispatcher.

timeout: The duration of time, in microseconds, that the Consumer is willing to wait for the event.

threshold: The number of events that should be on the EVD queue before the operation should return with DAT_SUCCESS. The *threshold* must be at least 1.

event: Pointer to the Consumer-allocated structure that the Provider fills with the event data.

nmore: The snapshot of the queue size at the time of the operation return.

Description:

dat_evd_wait removes the first event from the Event Dispatcher event queue and fills the Consumer-allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event type-specific parameters. The Consumer should allocate an event structure big enough to hold any event that the Event Dispatcher can deliver.

For all events, the Provider fills the *dat_event* that the Consumer allocates. Therefore, for all events, all fields of *dat_event* are OUT from the Consumer point of view. For *DAT_CONNECTION_REQUEST_EVENT*, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in *DAT_CR_ARRIVAL_EVENT_DATA.* That object is destroyed by the Provider as part of *dat_cr_accept, dat_cr_reject,* or *dat_cr_handoff.* The Consumer should not use *cr_handle* or any of its parameters, including *private_data*, after one of these operations destroys the Connection Request.

For *DAT_CONNECTION_EVENT_ESTABLISHED* for the Active side of connection establishment, the Provider returns the pointer for *private_* *data* and the *private_data_size.* For the Passive side, *DAT_CONNECTION_EVENT_ESTABLISHED* event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data.* The *private_data* is valid until the Active side Consumer destroys the connected Endpoint (*dat_ep_free*), or transitions the Endpoint into *Unconnected* state so it is ready for the next connection. So, while the Endpoint is in *Connected, Disconnect Pending,* or *Disconnected* state, the *private_data* of *DAT_CONNECTION_REQUEST_EVENT* is still valid for Active side Consumers.

Note to Provider: Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for *dat_ep_connect, dat_ep_dup_connect,* and *dat_cr_accept*. If the Consumer provides more data than the Provider and Transport can support (larger than IA Attribute of *max_private_data_size*), *DAT_INVALID_PARAMETER* is returned for that operation.

Note to Provider: Provider shall adhere to the memory allocation requirements stated in 5.1 Local Resource Model on page 13.

A Consumer that blocks doing a *dat_evd_wait* on an Event Dispatcher effectively takes exclusive ownership of that Event Dispatcher. Any other dequeue operation (*dat_evd_wait* or *dat_evd_dequeue*) on the Event Dispatcher is rejected with a DAT_INVALID_STATE error code.

The CNO associated with the *evd_handle* is not triggered upon event arrival if there is a Consumer blocked on *dat_evd_wait* on this Event Dispatcher.

The *timeout* allows the Consumer to restrict the amount of time it is blocked waiting for the event arrival. The value of *DAT_TIMEOUT_INFINITE* indicates that the Consumer waits indefinitely for an event arrival. Consumers should use extreme caution in using this value.

When *timeout* value is reached and the number of events on the EVD queue is below the *threshold* value, the operation fails and returns *DAT_TIMEOUT_EXPIRED*. In this case, no event is dequeued from the EVD and the return value for the *event* argument is undefined. However, an *nmore* value is returned that specifies the snapshot of the number of the events on the EVD queue that is returned.

If *timeout*=0 and if there are no *threshold* number of events available then *DAT_TIMEOUT_EXPIRED* is returned with *nmore* specified and *event* not defined. If *timeout*=0 and there are *threshold* number of events then *DAT_SUCCESS* is returned with both *nmore* and *event* defined. If *timeout*=0 the operation does not block regardless of the platform behavior for *timeout*=0.

The *threshold* allows the Consumer to wait for a requested number of event arrivals prior to waking the Consumer. If the value of the *threshold* is larger than the Event Dispatcher queue length, the operation fails with the return *DAT_INVALID_PARAMETER*. If a non-positive value is specified for *threshold*, the operation fails and returns *DAT_INVALID_PARAMETER*.

If EVD is used by an Endpoint for a DTO completion stream that is configured for a Consumer-controlled event Notification (*DAT_COMPLETION_UNSIGNALLED_FLAG* or *DAT_COMPLETION_SOLICITED_WAIT_FLAG* for Receive Completion Type for Receives; *DAT_COMPLETION_UNSIGNALLED_FLAG* for Request Completion Type for Send, RDMA Read, RDMA Write and RMR Bind), the *threshold* value must be 1. An attempt to specify some other value for *threshold* for this case results in *DAT_INVALID_STATE.*

The returned value of *nmore* indicates the number of events left on the Event Dispatcher queue after the *dat_evd_wait* returns. If the operation return value is *DAT_SUCCESS,* the *nmore* value is at least the value of (*threshold* -1). Notice that *nmore* is only a snapshot and the number of events can be changed by the time the Consumer tries to dequeue events via *dat_evd_wait* with timeout of zero or via *dat_evd_dequeue*.

For returns other than *DAT_SUCCESS*, *DAT_TIMEOUT_EXPIRED, and DAT_INTERRUPTED_CALL*, the returned value of *nmore* is undefined.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a *dat_evd_dequeue* or *dat_evd_wait* operation.

If the return value is neither *DAT_SUCCESS* nor *DAT_TIMEOUT_EXPIRED*, returned values of *nmore* and *event* are undefined. If the return value is *DAT_TIMEOUT_EXPIRED,* the return value of *event* is undefined, but the return value of *nmore* is defined. If the return value is *DAT_SUCCESS*, the return values of *nmore* and *event* are defined.

If this routine is called on an EVD in an unwaitable state, or if *dat_evd_set_unwaitable* is called on an EVD on which a thread is blocked in this routine, the routine returns with *DAT_INVALID_STATE*.

This operation is blocking and thread-safe. The ordering of events dequeued by overlapping calls to *dat_evd_wait* or *dat_evd_dequeue* is not specified.

Returns:

DAT_SUCCESS	The operation was successful. An event was returned to a Consumer.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.

DAT_INVALID_PARAMETER	Invalid parameter; *timeout* or *threshold* is invalid. For example, *threshold* is larger than the EVD's *evd_min_qlen.*
DAT_ABORT	The operation was aborted because IA was closed.
DAT_INVALID_STATE	One of the parameters was invalid for this operation. There is already a waiter on the EVD, or the EVD is in an unwaitable state.
DAT_TIMEOUT_EXPIRED	The operation timed out.
DAT_INTERRUPTED_CALL	[*Unix only*] The operation was interrupted by a signal.

6.3.4.10.1 USAGE

Consumers should be cautioned against using *threshold* combined with infinite *timeout*.
Consumers should not mix different models for control of unblocking a waiter. If the Consumer uses Notification Suppression or Solicited Wait to control the Notification events for unblocking a waiter, the *threshold* must be set to 1. If the Consumer uses *threshold* to control when a waiter is unblocked, *DAT_COMPLETION_UNSIGNALLED_FLAG* locally and *DAT_COMPLETION_SOLICITED_WAIT* remotely shall not be used. By default, all completions are Notification events.

Consumers can unblock a waiting thread by posting *dat_evd_post_se* to the EVD. This platform-independent method ensures that there is a notification event on the EVD queue. If there is a race between *dat_evd_wait* and *dat_evd_post_se,* since there is a notification event on the EVD, the waiter will either be unblocked or it would not block at all. This is subject to *threshold* value specified. In the worst case, Consumer may need to post *threshold* number of SEs. The EVD must be configured to support SE event stream.

Consumer can use Platform-specific methods for unblocking a waiter that may result in DAT_INTERRUPTED_CALL. See the OS specific notes for more details (see "Operating System Specific Notes" on page 303).

6.3.4.10.2 RATIONALE

6.3.4.10.3 MODEL IMPLICATIONS

Providers shall check for the number of events on the queue before returning the *DAT_TIMEOUT_EXPIRED* error. If the number of events on the EVD queue is equal to or larger than the specified *threshold* value, the operation must complete successfully.

Note for Provider: Provider should not wake up the Consumer prematurely when a *threshold* greater than 1 is used.

Note for Consumer: Consumer should be able to tolerate eager behavior from the Provider. Although the Provider should not wake up prematurely when *threshold* is set to a number greater than 1, it is allowed not to block if there are events on the EVD, even if their number is smaller than *threshold.* Consumers should use *threshold* for performance improvement, not as a semantic guarantee.

6.3.4.11 DAT_EVD_DEQUEUE

This operation is almost equivalent to *dat_evd_wait* with a *timeout of* 0 and a *threshold* of 1, except for error semantics and returns.

Synopsis:

```
DAT_RETURN
   dat_evd_dequeue(
   IN     DAT_EVD_HANDLE     evd_handle,
   OUT    DAT_EVENT          *event
   )
```

Parameters:

evd_handle: Handle for an instance of the Event Dispatcher.

event: Pointer to the Consumer-allocated structure that Provider fills with the event data.

Description: *dat_evd_dequeue* removes the first event from the Event Dispatcher event queue and fills the Consumer allocated *event* structure with event data. The first element in this structure provides the type of the event; the rest provides the event-type-specific parameters. The Consumer should allocate an *event* structure big enough to hold any event that the Event Dispatcher can deliver.

For all events the Provider fills the *dat_event* that the Consumer allocates. So for all events, all fields of *dat_event* are OUT from the Consumer point of view. For *DAT_CONNECTION_REQUEST_EVENT*, the Provider creates a Connection Request whose *cr_handle* is returned to the Consumer in *DAT_CR_ARRIVAL_EVENT_DATA.* That object is destroyed by the Provider as part of *dat_cr_accept, dat_cr_reject,* or *dat_cr_handoff.* The Consumer should not use *cr_handle* or any of its parameters, including *private_data,* after one of these operations destroys the Connection Request.

For *DAT_CONNECTION_EVENT_ESTABLISHED* for the Active side of connection establishment, the Provider returns the pointer for *private_data* and the *private_data_size.* For the Passive side, *DAT_CONNECTION_EVENT_ESTABLISHED* event *private_data* is not defined and *private_data_size* returns zero. The Provider is responsible for the memory allocation and deallocation for *private_data.* The *private_data* is valid until the Active side Consumer destroys the connected

Endpoint (*dat_ep_free*), or transitions the Endpoint into *Unconnected* state so it is ready for the next connection. So while the Endpoint is in *Connected, Disconnect Pending,* or *Disconnected* state, the *private_data* of *DAT_CONNECTION_REQUEST_EVENT* is still valid for Active side Consumers.

Note to Provider: Provider must pass to the Consumer the entire Private Data that the remote Consumer provided for *dat_ep_connect, dat_ep_dup_connect,* and *dat_cr_accept*. If the Consumer provides more data than the Provider and Transport can support (larger than the IA Attribute of *max_private_data_size*), *DAT_INVALID_PARAMETER* is returned for that operation.

Note to Provider: Provider shall adhere to the memory allocation requirements stated in 5.1 Local Resource Model on page 13.

The returned event that was posted from an Event Stream guarantees Consumers that all events that were posted from the same Event Stream prior to the returned event were already returned to a Consumer directly through a *dat_evd_dequeue* or *dat_evd_wait* operation.

This operation is nonblocking, synchronous, and thread-safe. The ordering of events dequeued by overlapping calls to *dat_evd_wait* or *dat_evd_dequeue* is not specified.

Returns:

DAT_SUCCESS	The operation was successful. An event was returned to a Consumer.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.
DAT_QUEUE_EMPTY	There are no entries on the Event Dispatcher queue.
DAT_INVALID_STATE	One of the parameters was invalid for this operation. There is already a waiter on the EVD.

6.3.4.11.1 USAGE

No matter how many contexts attempt to dequeue from an Event Dispatcher, each event is delivered exactly once. However, which Consumer receives which event is not defined. The Provider is not obligated to provide the first caller the first event unless it is the only caller. The Provider is not obligated to ensure that the caller receiving the first event executes earlier than contexts receiving later events.

Preservation of event ordering within an Event Stream is an important feature of the DAT Event Model. Consumers are cautioned that overlapping or concurrent calls to *dat_evd_dequeue* from multiple contexts can undermine this ordering information. After multiple contexts are involved, the Provider can only guarantee the order that it delivers events into the EVD. The Provider cannot guarantee that they are processed in the correct order.

Although calling *dat_evd_dequeue* does not cause a context switch, the Provider is under no obligation to prevent one. A context could successfully complete a dequeue, and then reach the end of its timeslice, before returning control to the Consumer code. Meanwhile, a context receiving a later event could be executing.

The Event ordering is preserved when dequeueing is serialized. Potential Consumer serialization methods include, but are not limited to, doing all dequeueing from a single context or protecting dequeueing via lock or semaphore.

6.3.4.11.2 RATIONALE

6.3.4.11.3 MODEL IMPLICATIONS

6.3.4.12 DAT_EVD_POST_SE

Synopsis:

```
DAT_RETURN
   dat_evd_post_se(
   IN        DAT_EVD_HANDLE       evd_handle,
   IN const  DAT_EVENT            *event
   )
```

Parameters:

evd_handle:	Handle for an instance of the Event Dispatcher.
event:	A pointer to a Consumer created Software Event.

Description:

dat_evd_post_se post Software event to the Event Dispatcher event queue. This is analogous to event arrival on the Event Dispatcher software Event Stream. The *event* that the Consumer provides shall adhere to the event format as defined in Appendix A.4. The first element in the event provides the type of the event (*DAT_SOFTWARE_EVENT*); the rest provides the event-type-specific parameters. These parameters are opaque to a Provider. Allocation and release of the memory referenced by the *event* pointer in a software event are the Consumer's responsibility.

There is no ordering between events from different Event Streams. All the synchronization issues between multiple Consumer contexts trying to post events to an Event Dispatcher instance simultaneously are left to a Consumer.

If the event queue is full, the operation is completed unsuccessfully and returns *DAT_QUEUE_FULL*. The *event* is not queued. The queue overflow condition does takes place and, therefore, the asynchronous Event Dispatcher is not effected.

dat_evd_post_se is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *event* is invalid.
DAT_QUEUE_FULL	The Event Dispatcher queue is full.

6.3.4.12.1 USAGE

6.3.4.12.2 RATIONALE

6.3.4.12.3 MODEL IMPLICATIONS

6.4 CONNECTION MANAGEMENT

uDAPL supports a client-server model for Connection establishment.

An active side (connection initiator) issues a request for a connection on a local Endpoint to a remote side. The remote side is identified by an IA Address and Connection Qualifier, or by IA Address that includes IP Address, port and IP protocol number.

A passive side (Connection Acceptor) creates a Service Point on a local IA with a specific Connection Qualifier, or by IA Address that includes IP Address, port and IP protocol number to listen for incoming Connection Requests.

6.4.1 INTERFACE ADAPTER ADDRESS

The Interface Adapter Address names an Interface Adapter, local or remote, that is used for connection management and Name Service. The format of the *dat_ia_address_ptr* follows the normal sockets programming practice of *struct sockaddr* *. DAT supports both IPv4 and IPv6 address families.

Allocation and initialization of DAT IA address structures must follow normal sockets programming procedures. The underlying type of the DAT IA address is the native *struct sockaddr* for each target operating system. In all cases, storage appropriate for the address family in use by the target Provider must be allocated.

For instance, when IPv6 addressing is in use, this should be allocated as *struct sockaddr_net6*. The sockaddr *sa_family* and, if present, *sa_len* fields must be initialized appropriately, as well as the address information.

When passed across the DAPL API, this storage is cast to the DAT_IA_ ADDRESS_PTR type. It is the responsibility of the callee to verify that the sockaddr contains valid data for the requested operation. It is always the responsibility of the caller to manage the storage.

Code Example for Linux

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <dat/udat.h>

struct sockaddr_in6 addr;
DAT_IA_ADDRESS_PTR ia_addr;
int status;

// Note: linux pton requires explicit encoding of IPv4 in
IPv6

   addr.sin6_family = AF_INET6;
   if (inet_pton(AF_INET6, "0:0:0:0:0:FFFF:192.168.0.1",
         &addr.sin6_addr) <= 0)
     return(-1); // Bad address or no address family support

   // initialize other necessary fields such as port, flow,
etc.

   ia_addr = (DAT_IA_ADDRESS_PTR) &addr;
   status = dat_ep_connect(ep_handle, ia_addr, conn_qual,
timeout, 0, NULL,
          qos, DAT_CONNECT_DEFAULT_FLAG);
```

6.4.1.1 PORT

The port allows a Consumer to use an IP port for Connection Management for all RDMA Transports. No mappings are needed for IP, and for IB the IBTA RDMA IP CM Service Annex defines support for the socket-based connection management including IP ports. DAPL does not expose Port separately but as part of Interface Adapter Address.

6.4.2 CONNECTION QUALIFIER

Connection Qualifier allows a DAT Provider to associate an incoming connection request with the entity providing the service. The Connection Qualifier provides functionality similar to the IB Service ID, TCP Port Number, or VI Discriminator.

Following are the mappings between DAT Connection Qualifiers and Transport-specific qualifiers:

1) For IB, the Connection Qualifier is a Service ID.
2) For IP, the least-significant 16 bits of the Connection Qualifier are used as a Port Number.
3) For VI, the least-significant 64 bits of the VI Discriminator are mapped into DAT Connection Qualifier. To ensure interoperability, the Consumer should use only the least-significant 64 bits.

6.4.3 COMMUNICATOR

The Communicator defines socket domain, type and protocol for DAT Endpoints and Common Service Points, and is used for connection management and Name Service. The Comminucator three fields formats follows the normal sockets programming practice of the platform. Communicator allows a socket-like transport-independent connection model.

Communicator is passed to the Common Service Point upon creation. It is also passed to the Endpoint upon creation or for Endpoint modification prior to the connection setup.

6.4.4 SERVICE POINT

A Service Point provides Consumers on the passive side the capability to listen for incoming Connection Requests and generate events upon their arrival. There is, at most, one Service Point listening on any given Connection Qualifier.

uDAPL defines the API for query parameters of Service Point instances, but it does not define an operation to modify Service Point parameters. Consumers can destroy and create a new instance with new desired parameters.

6.4.4.1 PUBLIC SERVICE POINT

The Public Service Point is a service point that allows the Consumer to listen on requests for connections arriving on a specified Connection Qualifier. The Public Service Point is used for client-server connection establishment. The Connection Qualifier for the Public Service Point is advertised by a Name Service.

The Consumer creates a Public Service Point that is a persistent listener for incoming Connection Requests. The Public Service Point can generate multiple Connection Request events. The number of outstanding Connection Requests generated by the Public Service Point is limited by the queue size of the associated Event Dispatcher. If the event queue of the associated Event Dispatcher is full and a Connection Request arrived, it is rejected with the appropriated return. If the associated Event Dispatcher is destroyed, all incoming Connection Requests are automatically rejected with the same return as if the event queue were full.

The Backlog of the Public Service Point is contained in the queue associated Event Dispatcher. The size of the backlog specifies the upper bound on the number of pending Connection Request instances Provider needs to support at any one time. If an Event Dispatcher is shared among multiple Public Service Points, the size of its queue is an upper bound for total Connection Requests for all of them combined. Consumers should avoid using the Event Dispatcher associated with Public Service Points for handling any other types of Event Streams.

DAT Consumers can destroy Public Service Points directly or indirectly by closing the IA.

6.4.4.1.1 DAT_PSP_CREATE

Synopsis:

```
DAT_RETURN
   dat_psp_create (
   IN     DAT_IA_HANDLE            ia_handle,
   IN     DAT_CONN_QUAL            conn_qual,
   IN     DAT_EVD_HANDLE           evd_handle,
   IN     DAT_PSP_FLAGS            psp_flags,
   OUT    DAT_PSP_HANDLE           *psp_handle
   )
```

Parameters:

ia_handle:	Handle for an instance of DAT IA.
conn_qual:	Connection Qualifier of the IA the Public Service Point shall be listening on.
evd_handle:	Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.
psp_flags:	Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of DAT_PSP_PROVIDER_FLAG indicates that the Consumer wants to get an Endpoint from the Provider; a value of DAT_PSP_CONSUMER_FLAG means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.
psp_handle:	Handle to an opaque Public Service Point.

Description: *dat_psp_create* creates a persistent Public Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

dat_psp_create is blocking. When the Public Service Point is created, DAT_SUCCESS is returned and *psp_handle* contains a handle to an opaque Public Service Point Object.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

psp_flags allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and *DAT_MODEL_NOT_SUPPORTED* is returned.

All Endpoints created by the Provider have *DAT_HANDLE_NULL* for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. Consumers should change Endpoint parameters, especially PZ and EVD, and are advised to change parameters for local accesses prior to the connection request acceptance with the Endpoint.

dat_psp_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* or *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *conn_qual* or *psp_flags* is invalid.
DAT_CONN_QUAL_IN_USE	The specified Connection Qualifier was in use.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.4.4.1.1.1 USAGE

Two uses of a Public Service Point are in the following sections.

Model 1 For this model, the Provider manipulates a pool of Endpoints for a Public Service Point. The Provider can use the same pool for more than one Public Service Point.

- The DAT Consumer creates a Public Service Point with a *flag* set to DAT_PSP_PROVIDER_FLAG.
- The Public Service Point does the following:

- Collects native transport information reflecting a received Connection Request
- Creates an instance of Connection Request
- Creates a Connection Request Notice (event) that includes the Connection Request instance (which includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint, and information about remote Endpoint)
- Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

  The Public Service Point is persistent and continues to listen for incoming requests for connection.

- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer can modify the provided local Endpoint to match the Connection Request and must either accept() or reject() the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a “connected” state and is fully usable for this connection, pending only any native transport mandated RTU (ready-to-use) messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in *Connected* state by a Connection Established Event on the Endpoint *connect_evd_ handle.*
- If rejected, control of the Local Endpoint is returned back to the Provider and its *ep_handle* is no longer usable by the Consumer.

Model 2 For this model, the Consumer manipulates a pool of Endpoints. Consumers can use the same pool for more than one Service Point.

- DAT Consumer creates a Public Service Point with a *flag* set to DAT_PSP_CONSUMER_FLAG.
- Public Service Point:
  - Collects native transport information reflecting a received Connection Request
  - Creates an instance of Connection Request
  - Creates a Connection Request Notice (event) that includes the Connection Request instance (which includes, among others, Public Service Point, its Connection Qualifier, Provider-generated Local Endpoint and information about remote Endpoint)

- Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

  The Public Service Point is persistent and continues to listen for incoming requests for connection.

- The Consumer creates a pool of Endpoints that it uses for accepting Connection Requests. Endpoints can be created and modified at any time prior to accepting a Connection Request with that Endpoint.
- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer can modify its local Endpoint to match the Connection Request and must either accept() or reject() the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a "connected" state and is fully usable for this connection, pending only any native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in *Connected* state by a Connection Established Event on the Endpoint *connect_evd_handle.*
- If rejected, the Consumer does not have to provide any Endpoint for the *dat_cr_reject*.

6.4.4.1.1.2 RATIONALE

Who can create an Endpoint per Connection Request might be a requirement of the underlying Transport. For example, for VI, the Consumer creates all Endpoints, for iWARP and IB allow both. Consumers can check the Provider attributes to determine which models are supported or read the Provider documentation that also provides this information.

The Provider is strongly encouraged to create an Endpoint that is immediately ready to accept a connection request (see advice to Providers in 6.6.4.2 Rationale on page 201).

6.4.4.1.1.3 MODEL IMPLICATIONS

Consumers cannot associate a pool of Consumer Endpoints with Service Points. They can do it manually by requesting that the Provider not generate local Endpoints for incoming Connection Requests. Then the Consumer can pick an Endpoint from its own pool of Endpoints for a connection.

For iWARP transport Provider establishes the TCP connection or SCTP association on its own. Any connection/association sockets supporting a PSP are not visible to the Consumer.

6.4.4.1.2 DAT_PSP_CREATE_ANY

Synopsis:

```
DAT_RETURN
    dat_psp_create_any (
```

```
IN    DAT_IA_HANDLE       ia_handle,
OUT   DAT_CONN_QUAL       *conn_qual,
IN    DAT_EVD_HANDLE      evd_handle,
IN    DAT_PSP_FLAGS       psp_flags,
OUT   DAT_PSP_HANDLE      *psp_handle
)
```

Parameters:

ia_handle: Handle for an instance of DAT IA.

conn_qual: Connection Qualifier of the IA the Public Service Point is listening on.

evd_handle: Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Public Service Point.

psp_flags: Flag that indicates whether the Provider or Consumer creates an Endpoint per arrived Connection Request. The value of DAT_PSP_PROVIDER_FLAG indicates that the Consumer wants to get an Endpoint from the Provider. A value of DAT_PSP_CONSUMER_FLAG means the Consumer does not want the Provider to provide an Endpoint for each arrived Connection Request.

psp_handle: Handle to an opaque Public Service Point.

Description: *dat_psp_create_any* creates a persistent Public Service Point that can receive multiple requests for connection, and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

dat_psp_create_any allocates an unused Connection Qualifier, creates a Public Service point for it, and returns both the allocated Connection Qualifier and the created Public Service Point to the Consumer.

Note to Provider: The allocated Connection Qualifier should be chosen from “nonprivileged” ports that are not currently used or reserved by any user or kernel Consumer or host ULP of the IA. The format of allocated Connection Qualifier returned is specific to IA transport type.

dat_psp_create_any is blocking. When the Public Service Point is created, DAT_SUCCESS is returned, *psp_handle* contains a handle to an opaque Public Service Point Object, and *conn_qual* contains the allocated Connection Qualifier. When return is not DAT_SUCCESS, *psp_handle* and *conn_qual* return values are undefined.

There is no explicit backlog for a Public Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

psp_flags allows Consumers to request that the Provider create an implicit Endpoint for each incoming Connection Request, or request that the Provider should not create one per Connection Request. If the Provider cannot satisfy the request, the operation shall fail and DAT_MODEL_NOT_SUPPORTED is returned.

All Endpoints created by the Provider have DAT_HANDLE_NULL for the Protection Zone and all Event Dispatchers. The Provider sets up Endpoint attributes to match the Active side connection request. Consumers should change Endpoint parameters, especially PZ and EVDs, and are advised to change parameters, like the ones for local accesses prior to the Connection Request acceptance with the Endpoint.

dat_psp_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* or *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *conn_qual* or *psp_flags* is invalid.
DAT_CONN_QUAL_UNAVAILABLE	No Connection Qualifiers available.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.4.4.1.2.4 USAGE

6.4.4.1.2.5 RATIONALE

Who can create an Endpoint per Connection Request might be a requirement of the underlying Transport. For example, for VI, the Consumer creates all Endpoints and IB allows both the Consumer and the Provider to create Endpoints. The Consumers can check the Provider attributes to determine which models are supported or read the Provider documentation that also provides this information.

The Provider is strongly encouraged to create an Endpoint that is immediately ready to accept a Connection Request (see advice to Providers in 6.6.4.2 Rationale on page 201).

For iWARP transport Provider establishes the TCP connection or SCTP association on its own. Any connection/association sockets supporting a PSP are not visible to the Consumer.

6.4.4.1.2.6 MODEL IMPLICATIONS

6.4.4.1.3 DAT_PSP_FREE

Synopsis:

```
DAT_RETURN
   dat_psp_free (
   IN     DAT_PSP_HANDLE     psp_handle
   )
```

Parameters:

psp_handle: Handle for an instance of the Public Service Point.

Description:

dat_psp_free destroys a specified instance of the Public Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point it had been listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation. But it must be consistent. This means that either a Connection Requested Event has been generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation shall have no effect on previously generated Connection Requested Events. This includes Connection Request instances and, potentially, Endpoint instances created by the Provider.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

It is illegal to use the destroyed handle in any subsequent operation.

dat_psp_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *psp_handle* is invalid.

6.4.4.1.3.7 USAGE

6.4.4.1.3.8 RATIONALE

6.4.4.1.3.9 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.4.4.1.4 DAT_PSP_QUERY

Synopsis:

```
DAT_RETURN
   dat_psp_query (
   IN     DAT_PSP_HANDLE          psp_handle,
   IN     DAT_PSP_PARAM_MASK      psp_param_mask,
   OUT    DAT_PSP_PARAM           *psp_param
   )
```

Parameters:

psp_handle:	Handle for an instance of Public Service Point.
psp_param_mask:	Mask for PSP parameters.
psp_param:	Pointer to a Consumer-allocated structure that Provider fills for Consumer-requested parameters.

Description:

dat_psp_query provides to the Consumer parameters of the Public Service Point. Consumer passes in a pointer to the Consumer allocated structures for PSP parameters that Provider fills.

psp_param_mask allows Consumers to specify which parameters they would like to query. The Provider will return values for *psp_param_mask* requested parameters. The Provider may return the value for any of the other parameters.

dat_psp_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *psp_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *psp_param_mask* is invalid.

6.4.4.1.4.10 USAGE

6.4.4.1.4.11 RATIONALE

6.4.4.1.4.12 MODEL IMPLICATIONS

6.4.4.2 COMMON SERVICE POINT

The Common Service Point is transport-independent analog of the Public Service Point. It allows the Consumer to listen on socket-equivalent for requests for connections arriving on a specified IP port instead of transport-dependent Connection Qualifier. An IA Address follows the platform conventions and provides among others the IP port to listen on. An IP port of the Common Service Point advertisement is supported by existing Ethernet infrastructure or DAT Name Service.

The Consumer creates a Commion Service Point that is a persistent listener for incoming Connection Requests. The Common Service Point can generate multiple Connection Request events. The number of outstanding Connection Requests generated by the Common Service Point is limited by the queue size of the associated Event Dispatcher. If the event queue of the associated Event Dispatcher is full and a Connection Request arrived, it is rejected with the appropriated return. If the associated Event Dispatcher is destroyed, all incoming Connection Requests are automatically rejected with the same return as if the event queue were full.

The Backlog of the Common Service Point is contained in the queue associated Event Dispatcher. The size of the backlog specifies the upper bound on the number of pending Connection Request instances Provider needs to support at any one time. If an Event Dispatcher is shared among multiple Common Service Points, the size of its queue is an upper bound for total Connection Requests for all of them combined. Consumers should avoid using the Event Dispatcher associated with Common Service Points for handling any other types of Event Streams.

DAT Consumers can destroy Common Service Points directly by using *dat_csp_free* or indirectly by closing the IA.

6.4.4.2.1 DAT_CSP_CREATE

Synopsis:

```
DAT_RETURN
   dat_csp_create (
   IN     DAT_IA_HANDLE           ia_handle,
   IN     DAT_COMM                *comm,
   IN     DAT_IA_ADDRESS_PTR      address,
   IN     DAT_EVD_HANDLE          evd_handle,
   OUT    DAT_CSP_HANDLE          *csp_handle
   )
```

Parameters:

ia_handle:	Handle for an instance of DAT IA.
comm:	Communicator of the CSP.
address:	IA Address to bind Common Service Point to.
evd_handle:	Event Dispatcher that provides the Connection Requested Events to the Consumer. The size of the event queue for the Event Dispatcher controls the size of the backlog for the created Common Service Point.

csp_handle:	Handle to an opaque Common Service Point.

Description:

dat_csp_create creates a persistent Common Service Point that can receive multiple requests for connection and generate multiple Connection Request instances that are delivered through the specified Event Dispatcher in Notification events.

comm allows Consumer to specify socket domain, type and protocol for the Service Point. The *comm* must follow the platform convention. That is the values of *domain* and *type* are required while *protocol* can be default of 0.

address allows Consumer to “bind” the Service Point to the specified address, including IP port to listen on. The *address* must follow the platform convension. For example, Consumer may specify NULL *address*, so that Provider assign it.

If the incoming connection request does not match the *comm* and *address* then the connection request is automatically rejected by the Provider without generating Connection Request object.

There is no explicit backlog for a Common Service Point. Instead, Consumers can control the size of backlog through the queue size of the associated Event Dispatcher.

dat_csp_create is blocking. When the Common Service Point is created, DAT_SUCCESS is returned and *csp_handle* contains a handle to an opaque Common Service Point Object.

dat_csp_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* or *evd_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *address*, *address_length*, *comm* or their combination is invalid.
DAT_PORT_IN_USE	The specified IP Port was in use.
DAT_COMM_NOT_SUPPORTED	The specified *comm* is not supported by the Provider.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.4.4.2.1.13 USAGE

The usage Common Service Point is analogous to model 2 of the PSP one.

Consumer manipulates a pool of Endpoints. Consumers can use the same pool for more than one Service Point.

- DAT Consumer creates a Common Service Point listening on specific IP port. Consumer can specify *comm* and *address* the same way it would be done for *socket.* This includes an ability to listen on a single port over a range of IP Addresses for a requested protocol.
- Consumer Service Point:
    - Collects native transport information reflecting a received Connection Request including requestor IP address and port, Service Point IP Address and port, and the protocol number.
    - Creates an instance of Connection Request
    - Creates a Connection Request Notice (event) that includes the Connection Request instance (which includes, among others, Common Service Point, IP port its listens on, and information about remote Endpoint)
    - Delivers the Connection Request Notice to the Consumer-specified target (CNO) *evd_handle*

    The Common Service Point is persistent and continues to listen for incoming requests for a connection.
- The Consumer creates a pool of Endpoints that it uses for accepting Connection Requests. Endpoints can be created and modified at any time prior to accepting a Connection Request with that Endpoint.
- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer can modify its local Endpoint to match the Connection Request and must either accept() or reject() the pending Connection Request.
- If accepted, the provided Local Endpoint is now in a “connected” state and is fully usable for this connection, pending only any native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in *Connected* state by a Connection Established Event on the Endpoint *connect_evd_handle.*
- If rejected, the Consumer does not have to provide any Endpoint for the *dat_cr_reject*.

6.4.4.2.1.14 RATIONALE

CSP allows Consumer to use the well-known connection model by providing an analog of the listening bound socket even for the RDMA Transport that does not have an analog for them. This allows Consumer to have a Connection Model which is Transport independent, yet which allows to use the whole machinery developed for Ethernet, including well-known ULP ports, port mappers, inetd and so on.

6.4.4.2.1.15 MODEL IMPLICATIONS

The model allows Consumer to associate an domain, type, protocol, IP address, and IP port to a listening point. Thus, the CSP may have a different IA_Address and port than its IA. Provider can restrict which values and combinations of these can be used by the Consumer. Provider may follow the platform conventions for it.

The underlying transport provide a mechanism to supply the exact information of the requestor that standard socket connection model provides. For iWARP this is done by the underlying TCP or SCTP layer, while for IB it is done by the RDMA IP CM Service Provider (which can be DAPL Provider itself). IB DAPL Provider converts IP port and protocol into Transport-specific Connection Qualifier using IBTA RDMA IP CM Service Annex specification.

Consumer can find an CSP parameters by quering it. If the requestor used *dat_ep_common_connect* than DAT_CR_PARAM will provide the IP Address and IP port to which remote endpoint is “bound” to and for for non-common model the IA Address and Port Qualifier of the remote IA otherwise. The protocol number of the requested connection can be extracted from the CSP which generated Connection Request.

The connection requests that the CSP gets are specific to the IA regardless whether or not Consumer specified wildcard for the local IP Address. Provider can restrict which values of *address* and *comm* it can handle.

Provider can assign the default IA Address of the IA and some port to the CSP if Consumer does not specify the *address.* Or Provider may require that the *address* is specified by the Consumer.

It is up to the Provider to ensure that Consumer requested *comm* and IA Address are valid and that IA can support them. For example, Provider can restrict what IP Addresses can be used by IA, via an IA IP address range defined by an administrator.

6.4.4.2.2 DAT_CSP_FREE

Synopsis:

```
DAT_RETURN
   dat_csp_free (
   IN     DAT_CSP_HANDLE     csp_handle
   )
```

Parameters:

csp_handle:	Handle for an instance of the Comon Service Point.

Description: *dat_csp_free* destroys a specified instance of the Common Service Point.

Any incoming Connection Requests for the *port* of the destroyed Service Point it had been listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation. But it must be consistent. This means that either a Connection Requested Event has been generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation shall have no effect on previously generated Connection Requested Events. This includes Connection Request instances and, potentially, Endpoint instances created by the Provider.

The behavior of this operation with creation of a Service Point on the same *port* at the same time is not defined. Consumers are advised to avoid this scenario.

It is illegal to use the destroyed handle in any subsequent operation.

dat_csp_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *csp_handle* is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.4.4.2.2.16 Usage

6.4.4.2.2.17 Rationale

6.4.4.2.2.18 Model Implications

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.4.4.2.3 DAT_CSP_Query

Synopsis:

```
DAT_RETURN
   dat_csp_query (
```

```
    IN      DAT_CSP_HANDLE          csp_handle,
    IN      DAT_CSP_PARAM_MASK      csp_param_mask,
    OUT     DAT_CSP_PARAM           *csp_param
    )
```

Parameters:

csp_handle:	Handle for an instance of Common Service Point.
csp_param_mask:	Mask for CSP parameters.
csp_param:	Pointer to a Consumer-allocated structure that Provider fills for Consumer-requested parameters.

Description: *dat_csp_query* provides to the Consumer parameters of the Common Service Point. Consumer passes in a pointer to the Consumer allocated structures for CSP parameters that Provider fills.

csp_param_mask allows Consumers to specify which parameters they would like to query. The Provider will return values for *csp_param_mask* requested parameters. The Provider may return the value for any of the other parameters.

dat_csp_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *csp_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *csp_param_mask* is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.4.4.2.3.19 USAGE

6.4.4.2.3.20 RATIONALE

6.4.4.2.3.21 MODEL IMPLICATIONS

6.4.4.3 RESERVED SERVICE POINT

The Reserved Service Point allows a Consumer to establish connection between a single Local Endpoint and a specific Remote Endpoint. A Reserved Service Point is used for peer-to-peer Connection establishment. The Connection Qualifier for the Reserved Service Point is not designed to be advertised by a Name Service. The Connection Qualifier of the Reserved Service Point is determined apriori through agreement by the application ahead of time, or by some other out-of-band approach.

The Consumer creates a Reserved Service Point explicitly with an associated Consumer-created Local Endpoint. A Reserved Service Point can only generate a single Connection Request. Subsequent remote requests for connections to the Reserved Service Point do not generate Connection Requests. The Provider shall generate the same responses to the remote side as if no Service Point is associated with the Connection Qualifier that the Reserved Service Point is associated with. Upon generation of the Connection Request, the Reserved Service Point is still valid and supports all DAT operations on its handle but do not generate any more Connection Requests. The Consumer should destroy the Reserved Service Point after it gets the Reserved Service Point Connection Request. After it is destroyed, the Consumer can create another Service Point on the same Connection Qualifier.

The Consumer shall not destroy the Endpoint associated with a Reserved Service Point. Instead The Consumer shall destroy the Reserved Service Point first, and then the Consumer can destroy or reuse the associated Endpoint.

6.4.4.3.1 DAT_RSP_CREATE

Synopsis:

```
DAT_RETURN
   dat_rsp_create (
   IN     DAT_IA_HANDLE            ia_handle,
   IN     DAT_CONN_QUAL            conn_qual,
   IN     DAT_EP_HANDLE            ep_handle,
   IN     DAT_EVD_HANDLE           evd_handle,
   OUT    DAT_RSP_HANDLE           *rsp_handle
   )
```

Parameters:

ia_handle:	Handle for an instance of DAT IA.
conn_qual:	Connection Qualifier of the IA the Reserved Service Point listens to.
ep_handle:	Handle for the Endpoint associated with the Reserved Service Point that is the only Endpoint that can accept a Connection Request on this Service Point. The value DAT_HANDLE_NULL requests the Provider to associate a Provider-created Endpoint with this Service Point.
evd_handle:	The Event Dispatcher to which an event of Connection Request arrival is generated for.
rsp_handle:	Handle to an opaque Reserved Service Point.

Description: *dat_rsp_create* creates a Reserved Service Point with the specified Endpoint that generates, at most, one Connection Request that is delivered to the specified Event Dispatcher in a Notification event.

dat_rsp_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle, evd_handle,* or *ep_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *conn_qual* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. For example, an Endpoint was not in the Unconnected state.
DAT_CONN_QUAL_IN_USE	Specified Connection Qualifier is in use.

6.4.4.3.1.22 USAGE

The usage of a Reserve Service Point is as follows:

- The DAT Consumer creates a Local Endpoint and configures it appropriately.
- The DAT Consumer creates a Reserved Service Point specifying the Local Endpoint.
- The Reserved Service Point does the following:
  - Collects native transport information reflecting a received Connection Request
  - Creates a Pending Connection Request
  - Creates a Connection Request Notice (event) that includes the Pending Connection Request (which includes, among others, Reserved Service Point Connection Qualifier, its Local Endpoint, and information about remote Endpoint)
  - Delivers the Connection Request Notice to the Consumer-specified target (CNO)*evd_handle*. The Local Endpoint is transitioned from *Reserved* to *Passive Connection Pending* state.
- Upon receiving a connection request, or at some time subsequent to that, the DAT Consumer must either accept() or reject() the Pending Connection Request.

- If accepted, the original Local Endpoint is now in a *Connected* state and fully usable for this connection, pending only native transport mandated RTU messages. This includes binding it to the IA port if that was not done previously. The Consumer is notified that the Endpoint is in a *Connected* state by a Connection Established Event on the Endpoint *connect_evd_handle.*
- If rejected, the Local Endpoint point transitions into *Unconnected* state. The DAT Consumer can elect to destroy it or reuse it for other purposes.

6.4.4.3.1.23 RATIONALE

6.4.4.3.1.24 MODEL IMPLICATIONS

For iWARP transport Provider establishes the TCP connection or SCTP association on its own. Any connection/association sockets supporting a PSP are not visible to the Consumer.

6.4.4.3.2 DAT_RSP_FREE

Synopsis:

```
DAT_RETURN
   dat_rsp_free (
   IN     DAT_RSP_HANDLE     rsp_handle
   )
```

Parameters:

rsp_handle: Handle for an instance of the Reserved Service Point.

Description:

dat_rsp_free destroys a specified instance of the Reserved Service Point.

Any incoming Connection Requests for the Connection Qualifier on the destroyed Service Point was listening on are automatically rejected by the Provider with the return analogous to the no listening Service Point.

The behavior of the Connection Requests in progress is undefined and left to an implementation. But it must be consistent. This means that either a Connection Requested Event was generated for the Event Dispatcher associated with the Service Point, including the creation of the Connection Request instance, or the Connection Request is rejected by the Provider without any local notification.

This operation shall have no effect on previously generated Connection Request Event and Connection Request.

The behavior of this operation with creation of a Service Point on the same Connection Qualifier at the same time is not defined. Consumers are advised to avoid this scenario.

For the Reserved Service Point, the Consumer-provided Endpoint reverts to Consumer control. Consumers shall be aware that due to a race condition, this Reserved Service Point might have generated a Connection Request Event and passed the associated Endpoint to a Consumer in it.

It is illegal to use the destroyed handle in any subsequent operation.

dat_rsp_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *rsp_handle* is invalid.

6.4.4.3.2.25 USAGE

6.4.4.3.2.26 RATIONALE

6.4.4.3.2.27 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.4.4.3.3 DAT_RSP_QUERY

Synopsis:

```
DAT_RETURN
   dat_rsp_query (
   IN     DAT_RSP_HANDLE          rsp_handle,
   IN     DAT_RSP_PARAM_MASK      rsp_param_mask,
   OUT    DAT_RSP_PARAM           *rsp_param
   )
```

Parameters:

rsp_handle:	Handle for an instance of Reserved Service Point.
rsp_param_mask:	Mask for RSP parameters.
rsp_param:	Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description: *dat_rsp_query* provides to the Consumer parameters of the Reserved Service Point. The Consumer passes in a pointer to the Consumer-allocated structures for RSP parameters that the Provider fills.

rsp_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *rsp_param_mask* requested parameters. The Provider can return values for any other parameters.

dat_rsp_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *rsp_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *rsp_param_mask* is invalid.

6.4.4.3.3.28 Usage

6.4.4.3.3.29 Rationale

6.4.4.3.3.30 Model Implications

6.4.5 Connection Request

Connection Request instance is created by the Provider upon Connection Request arrival on a Service Point. A handle for a Connection Request instance is passed to a Consumer through a Connection Request Event.

A Connection Request contains information about the requestor side of the connection, including the private data (see Appendix A.4).

A Connection Request can be queried but its parameters cannot be modified.

6.4.5.1 DAT_CR_Query

Synopsis:

```
DAT_RETURN
   dat_cr_query (
   IN     DAT_CR_HANDLE           cr_handle,
   IN     DAT_CR_PARAM_MASK       cr_param_mask,
   OUT    DAT_CR_PARAM            *cr_param
   )
```

Parameters:

cr_handle:	Handle for an instance of a Connection Request.
cr_param_mask:	Mask for Connection Request parameters.
cr_param:	Pointer to a Consumer-allocated structure that the Provider fills for Consumer-requested parameters.

Description: *dat_cr_query* provides to the Consumer parameters of the Connection Request. The Consumer passes in a pointer to the Consumer-allocated structures for Connection Request parameters that the Provider fills.

cr_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *cr_param_mask* requested parameters. The Provider can return values for any other parameters.

dat_cr_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cr_handle* handle is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *cr_param_mask* is invalid.

6.4.5.1.1 USAGE

The Consumer uses *dat_cr_query* to get information about requesting a remote Endpoint as well as a local Endpoint if it was allocated by the Provider for the arrived Connection Request. The local Endpoint is created if the Consumer used PSP with *DAT_PSP_PROVIDER*_FLAG as the value for *psp_flags*. For the remote Endpoint, *dat_cr_query* provides *remote_ia_address* and *remote_port_qual*. It also provides remote peer *private_data* and its size.

The *truncate_flag* in the Connection Request Arrival event specifies whether or not the arrived private data in Connection request object was truncated or not.

6.4.5.1.2 RATIONALE

6.4.5.1.3 MODEL IMPLICATIONS

6.4.5.2 DAT_CR_ACCEPT

Synopsis:

```
DAT_RETURN
   dat_cr_accept (
   IN     DAT_CR_HANDLE           cr_handle,
   IN     DAT_EP_HANDLE           ep_handle,
   IN     DAT_COUNT               private_data_size,
   IN const DAT_PVOID             private_data,
   IN     DAT_CONNECT_FLAGS       multipathing_flags
   )
```

Parameters:

cr_handle:	Handle to an instance of a Connection Request that the Consumer is accepting.
ep_handle:	Handle for an instance of a local Endpoint that the Consumer is accepting the Connection Request on. If the local Endpoint is specified by the Connection Request, the *ep_handle* shall be DAT_HANDLE_ NULL.
private_data_size:	Size of the *private_data*, which must be nonnegative.

private_data: Pointer to the private data that should be provided to the remote Consumer when the Connection is established. If *private_data_size* is zero, then *private_data* can be NULL.

multipathing_flags: Multipathing flags for the accepted connection. The default value is *DAT_CONNECT_DEFAULT_FLAG*, which is 0. See Table 4, “CR Accept Flag Definitions,” on page 165 for flag definitions.

Table 4 CR Accept Flag Definitions

Features	Definition/Bit	Value	Description
MultiPathing	DAT_MULTIPATH_FLAG least significant	0	Consumer does not request multipathing.
		1	Consumer requests multipathing.
		2	Consumer requires multipathing.

Description: *dat_cr_accept* establishes a Connection between the active remote side requesting Endpoint and the passive side local Endpoint. The local Endpoint is either specified explicitly by *ep_handle* or implicitly by a Connection Request. In the second case, *ep_handle* is DAT_HANDLE_NULL.

Consumers can specify private data that is provided to the remote side upon Connection establishment. If Consumer specifies more private data then Provider supports, see Provider *max_private_data_size* attribute, then operation fails synchronously without any effect on the local Endpoint, Pending Connection Request, private data, or remote Endpoint.

If the provided local Endpoint does not satisfy the requested Connection Request, the operation fails without any effect on the local Endpoint, Pending Connection Request, private data, or remote Endpoint.

The operation is asynchronous. The successful completion of the operation is reported through a Connection Event of type *DAT_CONNECTION_EVENT_ESTABLISHED* on the *connect_evd* of the local Endpoint.

If the Provider cannot complete the Connection establishment, the connection is not established and the Consumer is notified through a Connection Event of type *DAT_CONNECTION_EVENT_ACCEPT_COMPLETION_ERROR* on the *connect_evd* of the local Endpoint. It can be caused by the active side timeout expiration, transport error, or any other reason. If Connection is not established, Endpoint transitions into *Disconnected* state and all posted Recv DTOs are flushed to its *recv_evd_handle* (see 6.5.5 Endpoint States on page 110).

If the local Endpoint on which the connection is accepted does not have a Protection Zone defined, or if one of its EVDs is not defined, then the operation fails and returns *DAT_INVALID_EP_STATE*.

A connection can only be accepted on an Endpoint that is in Unconnected state if the Endpoint was not provided in the Connection Request, or on the Endpoint in Passive or Tentative Connection Pending states if the Endpoint was provided in the Connection Request. An attempt to accept a connection on an Endpoint in any other state fails and returns *DAT_INVALID_EP_STATE*.

This operation, if successful, also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any subsequent operation fails.

If the accepting Endpoint *comm, IA_Addreess* and *Connection Qualifier* are not defined then they inherits from the SP. If any of them are defined for the accepting Endpoint it must match one for the Service Point. When the connection is established the local Endpoint remote IA_Address and remote Connection Qualifier are filled.

multipathing_flags allows Consumer to specify multipathing information for the accepted connection. Consumer can request no multipathing, which is the default value. It can require multipathing, which means that connection should not be established if only a single path is available. Or multipathing can be requested, which means that multipathed connection can be established even if only a single path is available now.

dat_cr_accept is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cr_handle* or *ep_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *private_data_size* or *private_data* is invalid, out of range, or a combination of parameters was invalid. For example, accepting EP parameters, like *comm, local IA_Address,* or *local Connection Qualifier* do not match Service point ones.
DAT_INVALID_STATE	Parameter in an invalid state. For example, an Endpoint was not in the *Unconnected , Passive Connection Pending* or *Tentative Connection Pending* state.

DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. For example, the requested *multipathing* was not supported by the local Provider.

6.4.5.2.1 USAGE

Consumers should be aware that Connection establishment might fail in the following cases: If the accepting Endpoint has an outstanding RDMA Read outgoing attribute larger than the requesting remote Endpoint or outstanding RDMA Read incoming attribute, or if the outstanding RDMA Read incoming attribute is smaller than the requesting remote Endpoint or outstanding RDMA Read outgoing attribute.

Consumers should set the accepting Endpoint RDMA Reads as the target (incoming) to a number larger than or equal to the remote Endpoint RDMA Read outstanding as the originator (outgoing), and the accepting Endpoint RDMA Reads as the originator to a number smaller than or equal to the remote Endpoint RDMA Read outstanding as the target. DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. Consumer can use Private Data for it.

The behavior of the *dat_cr_accept* when an accepting EP incoming RDMA Read does not match requesting remote EP outgoing RDMA Read, or accepting EP outgoing RDMA Read does not match requesting EP incoming RDMA Read is equivalent to the underlying RDMA transport behavior. Specifically, for IB and iWARP the mismatch behavior for an established or establishing connection is not an error, but connection may be broken when RDMA read resources on one of its endpoint are exceeded.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider can change these default values during connection setup.

6.4.5.2.2 RATIONALE

6.4.5.2.3 MODEL IMPLICATIONS

The Provider cannot fail connection establishment because of insufficient resources to support the Provider-chosen outstanding RDMA Read attribute defaults for the Endpoint.

For iWARP/TCP transport if the Provider, either directly or indirectly via underlying NIC support, supports IETF MPA protocol it shall map the acceptance to the MPA Reply frame without rejection bit set.

For iWARP/SCTP the Provider maps the acceptance to a DDP Stream Session Accept message.

If Consumer specified more private data than local Provider supports the operations fails synchronously with DAT_INVALID_PARAMETER. If local Provider support the amount of private data but remote Provider cannot the remote Provider will pass the truncated private data to the Consumer and set the *truncate_flag* in the Connection Request Arrival event.

For the IB transport, Provider shall zero out transport specific private data fields beyond the Consumer provided private data. This ensures that remote Provider can detect the extra private data beyond what it can support.

For iWARP Providers that support IETF MPA both the size of the private data and the private data shall be mapped into MPA Request frame.

If the accepting Endpoint does not have *comm,* local *IA_Address* or local *Connection Qualifier* defines it inherits them from the Service Point on which the connection request arrived. For PSP and RSP which do not have *comm* defined the *comm* is inherited from the IA.

6.4.5.3 DAT_CR_REJECT

Synopsis:

```
DAT_RETURN
   dat_cr_reject (
   IN     DAT_CR_HANDLE      cr_handle,
   IN     DAT_COUNT          private_data_size,
   IN const DAT_PVOID        private_data
   )
```

Parameters:

cr_handle:	Handle to an instance of a Connection Request that the Consumer is rejecting.
private_data_size:	Size of the *private_data*, which must be nonnegative.
private_data:	Pointer to the private data that should be provided to the remote Consumer when the Connection is established. If *private_data_size* is zero, then *private_data* can be NULL.

Description:

dat_cr_reject rejects a Connection Request from the Active remote side requesting Endpoint. If the Provider passed a local Endpoint into a Consumer for the Public Service Point-created Connection Request, that Endpoint reverts to Provider Control. The behavior of an operation on that Endpoint is undefined. The local Endpoint that the Consumer provided for Reserved Service Point reverts to Consumer control, and the Consumer is free to use in any way it wants.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Established Event. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any subsequent operation fails.

dat_cr_reject is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *cr_handle* was invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *private_data_size* or *private_data* is invalid, out of range, or a combination of parameters was invalid.
DAT_INVALID_STATE	Parameter in an invalid state. For example, a CR Endpoint was not in the *Unconnected* , *Passive Connection Pending* or *Tentative Connection Pending* state.

6.4.5.3.1 Usage

6.4.5.3.2 Rationale

6.4.5.3.3 Model Implications

For iWARP/TCP transport if the Provider, either directly or indirectly via underlying NIC support, supports IETF MPA protocol it shall map the rejection to the MPA Reply frame with rejection bit set.

For iWARP/SCTP transport, the Provider maps the rejection to a DDP Stream Session Reject message.

Consumer should not rely on the remote peer getting the rejection private data. The remote side may be timeout out already, destroyed EP that requested connection, underlying transport connection has been teared down or any other reason.

If Consumer specified more private data than local Provider supports the operations fails synchronously with DAT_INVALID_PARAMETER. If local Provider support the amount of private data but remote Provider cannot the remote Provider will pass the truncated private data to the Consumer and set the *truncate_flag* in the Connection Request Arrival event.

For the IB transport, Provider shall zero out transport specific private data fields beyond the Consumer provided private data. This ensures that remote Provider can detect the extra private data beyond what it can support.

For iWARP Providers that support IETF MPA both the size of the private data and the private data shall be mapped into MPA Request frame.

6.4.5.4 DAT_CR_HANDOFF

Synopsis:

```
DAT_RETURN
   dat_cr_handoff (
   IN     DAT_CR_HANDLE      cr_handle,
   IN     DAT_CONN_QUAL      handoff
   )
```

Parameters:

cr_handle:	Handle to an instance of a Connection Request that the Consumer is handing off.
handoff:	Indicator of another Connection Qualifier on the same IA to which this Connection Request should be handed off.

Description: *dat_cr_handoff* hands off the Connection Request to another Service Point specified by the Connection Qualifier *handoff*.

The operation is synchronous. This operation also destroys the Connection Request instance. Use of the handle of the destroyed Connection Request in any subsequent operation fails.

dat_cr_handoff is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle. *cr_handle* was invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *handoff* is invalid.
DAT_INVALID_STATE	Parameter is in an invalid state. For example, a CR associated socket is in the state that cannot be handed off.

6.4.5.4.1 USAGE

Consumers are advised that support of this features by iWarp Providers is unlikely. When the connection is known to be over iWarp/TCP the

Consumer may be able to use socket CM to accomplish the same on the lower transport layer.

6.5 SHARED RECEIVE QUEUE

Shared Receive Queues provide Consumer the ability to share receive buffers among several endpoints.

6.5.1 DAT_SRQ_CREATE

Synopsis:

```
DAT_RETURN
   dat_srq_create (
   IN     DAT_IA_HANDLE         ia_handle,
   IN     DAT_PZ_HANDLE         pz_handle,
   IN     DAT_SRQ_ATTR          *srq_attr,
   OUT    DAT_SRQ_HANDLE        *srq_handle
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA to which the created SRQ belongs.
pz_handle	Handle for an instance of the Protection Zone.
srq_attr:	Pointer to a structure that contains Consumer-requested SRQ attributes.
srq_handle:	Handle for the created instance of a Shared Receive Queue.

Description:

dat_srq_create creates an instance of a Shared Receive Queue (SRQ) that is provided to the Consumer as *srq_handle*. If the value of DAT_RETURN is not *DAT_SUCCESS* then the value of *srq_handle* is not defined.

The created SRQ is unattached to any Endpoints.

Protection Zone *pz_handle* allows Consumers to control what local memory can be used for the Recv DTO buffers posted to the SRQ. Only memory referred to by LMRs of the posted Recv buffers that match the SRQ Protection Zone can be accessed by the SRQ.

The *srq_attributes* parameter specifies the initial attributes of the created SRQ. If the operation is successful, then created SRQ has a minimum queue size of *max_recv_dtos* and the number of entries on the posted Recv scatter list is at minimum equal to *max_recv_iov.* The created SRQ can have the queue size and support number of entries on post Recv buffers larger than requested. Consumer can query SRQ to find out the actual supported queue size and maximum Recv IOV.

Consumer must set *low_watermark* to *DAT_SRQ_LW_DEFAULT* to ensure that asynchronous event is not generated immediately, because there are no buffers in the created SRQ. Consumer should set Maximum Receive DTO attribute and Maximum number of elements in IOV for posted buffers as needed.

When an associated EP tries to get a buffer from SRQ and there are no buffers available, the behavior of the EP is the same as when there are no buffers on the EP Recv Work Queue.

dat_srq_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* or *pz_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; One of the requested SRQ attributes was invalid or a combination of attributes is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.5.1.1 USAGE

SRQ is created by Consumer prior to creation of EPs that will use it. Some Providers may restrict whether multiple EPs that share a SRQ can have different Protection Zones. Check *srq_ep_pz_difference_support* Provider attribute for it. The EPs that use SRQ may or may not use the same *recv_evd*.

Since a Recv buffer of SRQ can be used by any EP that is using SRQ, Consumer should ensure that posted Recv buffers are large enough to receive an incoming message on any of the EPs.

If Consumer does not want to get an asynchronous event when the number of buffers in SRQ falls below the Low Watermark they should leave its value as *DAT_SRQ_LW_DEFAULT.* If Consumers do want to get a notification they can set the value to the desired one by calling *dat_srq_set_lw*.

6.5.1.2 RATIONALE

SRQ allows Consumer to use fewer than the maximum Recv buffers for each connection. If Consumer can upper bound the number of incoming messages over all connections whose local EP is using SRQ, instead of posting this maximum for each connection Consumer can post them for all connections on SRQ. For example, the maximum utilized link bandwidth divided over the message size can be used for an upper bound.

6.5.1.3 Model Implications

Depending on the underlying Transport one or more messages can arrive simultaneously on an EP that is using SRQ. Thus, the same EP can have multiple Recv buffers in its possession without these buffers being on SRQ or *recv_evd*.

Since Recv buffers can be used by multiple connections of the local EPs that are using SRQ, the completion order of the Recv buffers is no longer guaranteed even when they use the same *recv_evd*. However, for each connection the Recv buffers completion order is guaranteed to be in the order of the posted matching Sends to the other end of the connection. There is no ordering guarantee that Receive buffers will be returned in the order they were posted even if there is only a single connection (Endpoint) associated with the SRQ. There is no ordering guarantee between different connections or between different *recv_evd*.

6.5.2 Shared Receive Queue States

The list of SRQ States and allowed SRQ operations is as follows:

SRQ State	Description	Allowed Operations
Operational	SRQ is fully functional and associated EPs can dequeue Recv buffers from it.	*dat_srq_free*, *dat_srq_set_lw*, *dat_srq_query*, *dat_srq_resize*, *dat_srq_post_recv*, *dat_ep_create_with_srq*, *dat_ep_recv_query*, *dat_ep_set_watermark.*
Error	SRQ is non-functional and associated EPs cannot dequeue Recv buffers from it. The only way to recover is to destroy all EPs associated with SRQ and then destroy SRQ. All Recv buffers posted on SRQ that are not in *recv_evd* of associated EPs are under Consumer control after SRQ destruction.	*dat_srq_free*, *dat_srq_post_recv*, *dat_ep_create_with_srq*, *dat_ep_recv_query*, *dat_ep_set_watermark.*

dat_srq_free can be called on the SRQ in any state. It can be successful only if there are no EPs that use the SRQ.

EPs cannot dequeue any Recv buffers from SRQ that is in an error state.

An attempt by an EP to dequeue Recv buffer from SRQ will transition EP into error state and a generate connection broken event on the EP *connect_evd*. *dat_ep_create_with_srq* can use SRQ in an error state. All other EP operations can use SRQ in error state.

If SRQ is in an error state then all non allowed operations on SRQ shall synchronously fail.

When SRQ transitions into an error state an asynchronous event will be generated for it that will be delivered on IA *async_evd*.

6.5.2.1 MODEL IMPLICATION

Implementations may use registered memory for internal representation for DTOs (Work Requests). This memory will be typically allocated and registered by the Provider upon creation of a SRQ. If for some reason this memory becomes deregistered while the SRQ is operational, a subsequent attempt by the adapter to dequeue a Recv DTO from the SRQ will fail and result in the SRQ being transitioned to an error state.

6.5.3 SHARED RECEIVE QUEUE ATTRIBUTES

The list of Shared Receive Queue attributes is as follows:

Max_Recv_DTOs:	Maximum number of Receive DTOs for SRQ.
Max_Recv_IOV:	Maximum number of elements in IOV that the a posted Receive DTO for SRQ can have.
Low_Watermark	The low watermark for the number of Recv buffers on SRQ.

The Maximum Recv DTO is either the size of the shared receive queue request by Consumer on create or modify SRQ operations, or the actual size of SRQ.

The Recv buffer is outstanding and occupies an entry on SRQ until its DTO completion is dequeued from an associated EP *recv_evd*. The outstanding Receive buffers include the buffers on SRQ, the buffers that has been posted to SRQ that are at SRQ associated EPs, and the buffers posted to SRQ for which completions have been generated but not yet reaped by Consumer from *recv_evd*s of the EPs that use the SRQ.

Consumer requests the number of Receive DTOs on SRQ via *dat_srq_create* or *dat_srq_resize*. Provider must create or modify SRQ such that it has at least that many entries on SRQ. But it is allowed to have more than the number requested by the Consumer. Consumer can check the actual queue length of SRQ via *dat_srq_query* operation.

Consumer requests the maximum number of IOV elements in Receive DTOs on SRQ via *dat_srq_create*. Provider must create SRQ such that it supports at least that many elements in Recv IOV for the SRQ. But it is allowed to have more than the number requested by the Consumer. Consumer can check the actual maximum number of IOV elements for Receive of SRQ via *dat_srq_query* operation.

The low watermark attribute of SRQ allows Consumers to get an asynchronous event when the number of Recv buffers on SRQ falls below the Consumer-specified threshold. This allows Consumer to replenish the number of Recv. buffers on SRQ or take some other actions before SRQ runs out of buffers and connections that use it may break upon arrival of new messages.

An asynchronous event will be generated when the number of buffers on SRQ is below the low watermark for the first time. This may happen when low watermark is set, or when an associated EP takes a buffer from the SRQ. The event will be generated only once. In order for Provider to generate event again Consumer needs to arm the low watermark again via *dat_srq_set_lw.* Upon SRQ creation Low Watermark must be set to *DAT_SRQ_LW_DEFAULT* to avoid immediate generation of an synchronous event.

6.5.3.1 USAGE

Consumers who are not concern about Low Watermark semantics should set this attribute to DAT_SRQ_LW_DEFAULT. The default value guarantees that no asynchronous event will be generated for the SRQ regardless of how many buffers are on SRQ.

6.5.3.2 RATIONALE

6.5.3.3 MODEL IMPLICATION

There are two more attributes related to SRQ but they are attributes of an EP that is using SRQ. The first one is the hard limit high watermark for the number of buffers consumed by the EP from SRQ and not yet reaped by a Consumer. The other one is the soft high watermark for the number of buffers consumed by the EP from SRQ for which completions have not been generated yet. For more details of their use and rationale for them see (6.6.6 Endpoint Attributes on page 208).

6.5.4 DAT_SRQ_SET_LW

Synopsis:

```
DAT_RETURN
   dat_srq_set_lw (
   IN     DAT_SRQ_HANDLE             srq_handle,
   IN     DAT_COUNT                  low_watermark
   )
```

Parameters:

srq_handle:	Handle for an instance of a Shared Receive Queue.
low_watermark	The low watermark for the number of Recv buffers on SRQ.

Description: *dat_srq_set_lw* sets the low watermark value for SRQ and arms SRQ for generating an asynchronous event for low watermark. An asynchronous event will be generated when the number of buffers on SRQ is below the low watermark for the first time. This may happen during this call or when an associated EP takes a buffer from the SRQ.

The asynchronous event will be generated only once per setting of the low watermark. Once an event is generated no new asynchronous events for the number of buffers in SRQ below the specified value will be generated again until the SRQ is set for the Low Watermark again. If Consumer is interested in generating the event again Consumer should set the low watermark again.

dat_srq_set_lw is synchronous. Its thread safety is Provider dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *srq_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *low_watermark* is higher than the *max_recv_dtos*.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. Provider does not support SRQ Low Watermark.

6.5.4.1 USAGE

Upon getting the asynchronous event for srq low watermark Consumer can replenish Recv buffers on SRQ or take any other action that is appropriate.

6.5.4.2 RATIONALE

6.5.4.3 MODEL IMPLICATIONS

Regardless of whether or not an asynchronous event for the low watermark has been generated this operation will set the generation of an asynchronous event with the Consumer-provided low watermark value. If the new low watermark value is below the current number of free Receive DTOs posted to the SRQ then an asynchronous event will be generated immediately. Otherwise the old low watermark value is simply replaced with the new one.

6.5.5 DAT_SRQ_FREE

Synopsis:

```
DAT_RETURN
   dat_srq_free (
   IN     DAT_SRQ_HANDLE     srq_handle
   )
```

Parameters:

srq_handle:	Handle for an instance of SRQ to be destroyed.

Description: *dat_srq_free* destroys an instance of the SRQ. The SRQ cannot be destroyed if it is in use by an EP.

It is illegal to use the destroyed handle in any subsequent operation.

dat_srq_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *srq_handle* is invalid.
DAT_SRQ_IN_USE	Shared Receive Queue cannot be destroyed because it is in still associated with an EP instance.

6.5.5.1 USAGE

6.5.5.2 RATIONALE

6.5.5.3 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.5.6 DAT_SRQ_QUERY

Synopsis:

```
DAT_RETURN
   dat_srq_query (
   IN     DAT_SRQ_HANDLE          srq_handle,
   IN     DAT_SRQ_PARAM_MASK      srq_param_mask,
   OUT    DAT_SRQ_PARAM           *srq_param
   )
```

Parameters:

srq_handle:	Handle for an instance of the SRQ.
srq_param_mask:	Mask for SRQ parameters.
srq_param:	Pointer to a Consumer-allocated structure that the Provider fills with SRQ parameters.

Description: *dat_srq_query* provides SRQ parameters to the Consumer. The Consumer passes in a pointer to the Consumer-allocated structures for SRQ parameters that the Provider fills.

srq_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *srq_param_mask* requested parameters. The Provider can return values for any other parameters.

In addition to the elements in SRQ attribute, *dat_srq_query* provides additional information in the *srq_param* structure if Consumer requests it via *srq_param_mask* settings. The two that are related to entry counts on SRQ are: the number of Receive buffers (*available_dto_count*) available for EPs to dequeue and the number of occupied SRQ entries (*outstanding_dto_count*) not available for new Recv buffer postings.

dat_srq_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *srq_param_mask* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *srq_handle* is invalid.

6.5.6.1 USAGE

Provider may not be able to provide the number of outstanding Recv of SRQ or available Recvs of SRQ. If Provider does not support query for one of these values then the Provider attribute indicates this. Even when Provider supports the query for one or both of these values it may not be able to provide this value at this moment. In either case the return value for the attribute that cannot be provided will be DAT_VALUE_UNKNOWN.

Example: Consumer created SRQ with 10 entries and associated 1 EP with it. 3 Recv buffers have been posted to it. Query will report: *max_recv_dtos*=10, *available_dto_count*=3, *outstanding_dto_count*=3. After Send message arrival the query will report: *max_recv_dtos*=10, *available_dto_count*=2, *outstanding_dto_count*=3. After Consumer dequeues Recv completion the query will report: *max_recv_dtos*=10, *available_dto_count*=2, *outstanding_dto_count*=2. In general each EP associated with SRQ may have multiple buffers in progress of receiving messages as well completed Recv on EVDs. Watermark setting help to control how many Recv buffers posted to SRQ an Endpoint can own.

6.5.6.2 RATIONALE

6.5.6.3 MODEL IMPLICATIONS

If Provider cannot support query for the number of outstanding Recv of SRQ or available Recvs of SRQ the value returned for that attribute should be DAT_VALUE_UNKNOWN.

6.5.7 DAT_SRQ_RESIZE

Synopsis:

```
DAT_RETURN
   dat_srq_resize(
   IN     DAT_SRQ_HANDLE      srq_handle,
   IN     DAT_COUNT           srq_max_recv_dto
   )
```

Parameters:

srq_handle:	Handle for an instance of Shared Receive Queue.
srq_max_recv_dto:	New maximum number of Recv DTOs that Shared Receive Queue must hold.

Description: *dat_srq_resize* modifies the size of the queue of SRQ.

Resizing of Shared Receive Queue shall not cause any incoming messages on any of the EPs that use the SRQ or any SRQ buffers to be lost. If the number of outstanding Recv buffers on the SRQ is larger then the requested *srq_max_recv_dto*, the operation can return *DAT_INVALID_STATE* and does not change SRQ. This includes not just the buffers on the SRQ but all outstanding Receive buffers that had been posted to the SRQ and whose completions have not reaped yet. Thus, the outstanding buffers include the buffers on SRQ, the buffers posted to SRQ that are at SRQ associated EPs, and the buffers posted to SRQ for which completions have been generated but not yet reaped by Consumer from *recv_evd*s of the EPs that use the SRQ.

If the requested *srq_max_recv_dto* is below the SRQ *low_watermark* then the operation returns DAT_INVALID_STATE and does not change SRQ.

dat_srq_resize is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *srq_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *srq_max_recv_dto* is invalid.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_STATE	Invalid state. Either the number of entries on the SRQ exceeds the requested SRQ queue length or requested SRQ queue length is smaller than SRQ low watermark.

6.5.7.1 USAGE

6.5.7.2 RATIONALE

6.5.7.3 MODEL IMPLICATIONS

dat_srq_resize is required not to lose any buffers. Thus, it cannot shrink below the outstanding number of Recv buffers on SRQ. There is no requirement to shrink SRQ in order to return *DAT_SUCCESS*.

The quality of the implementation determines how closely to the Consumer-requested value Provider shrinks SRQ. For example, Provider can shrink SRQ to the Consumer requested value and if the requested value is smaller than outstanding buffers on SRQ then return *DAT_INVALID_STATE*. Or Provider can shrink to some value larger than the requested by the Consumer but below current SRQ size. Or Provider does not change the SRQ size and still returns *DAT_SUCCESS.*

6.5.8 DAT_SRQ_POST_RECV

Synopsis:

```
DAT_RETURN
   dat_srq_post_recv (
   IN     DAT_SRQ_HANDLE            srq_handle,
   IN     DAT_COUNT                 num_segments,
   IN     DAT_LMR_TRIPLET           *local_iov,
   IN     DAT_DTO_COOKIE            user_cookie
   )
```

Parameters:

srq_handle:	Handle for an instance of the Shared Receive Queue.
num_segments:	Number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a 0 size message.
local_iov:	I/O Vector that specifies the local buffer to be filled. Can be NULL for receiving a 0 size message.
user_cookie:	User-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be NULL.

Description: *dat_srq_post_recv* posts the receive buffer that can be used for the incoming message into the *local_iov* by any connected EP that uses SRQ.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the “front” segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all. The actual order of segment fillings is left to the implementation. The *local_iov* specification should adhere to the rules defined in Appendix A.4.

The user_cookie allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the EP that dequeues the posted buffer and the specified *completion_flags* value for Solicited Wait for the matching Send. If EP Recv Completion Flag is *DAT_COMPLETION_UNSIGNALLED_FLAG*, which is the default value for SRQ EP, then all posted Recvs will generate completions with Signal Notifications.

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the *dat_srq_post_recv* returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after *dat_srq_post_recv* returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The DAT_SUCCESS return of the *dat_srq_post_recv* is at least the equivalent of posting a Receive operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_srq_post_recv* to ensure that this operation is nonblocking.

The completion of the Receive posted to the SRQ is equivalent to what happened to the Receive posted to the Endpoint, for the Endpoint that dequeued the Receive buffer from the Shared Receive queue.

The posted Recv DTO will complete with signal, equivalent to the completion of Recv posted directly to the Endpoint that dequeued the Recv buffer from SRQ with *DAT_COMPLETION_UNSIGNALLED_FLAG* value not set for it.

The posted Recv DTOs will complete in the order of Send postings to the other endpoint of each connection whose local EP uses SRQ. There is no ordering among different connections regardless if they share SRQ and *recv_evd* or not.

Buffers posted to an SRQ will be allocated to a specific EP based upon arrival of actual traffic. In no case will more buffers be allocated to an EP than required for the messages sent by the remote peer.

When the EP transitions from the connected state, all buffers already allocated to it will be completed as flushed operations. The decision on whether to return these buffers to the SRQ is left to the Consumer.

If the reported *status* of the Completion DTO event corresponding to the posted RDMA Read DTO is not *DAT_DTO_SUCCESS*, the content of the *local_iov* is not defined and the *transfered_length* in the DTO Completion event is not defined.

The operation is valid for all states of the Shared Receive Queue.

dat_srq_post_recv is asynchronous, nonblocking, and its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
DAT_INVALID_HANDLE	Invalid DAT handle; *srq_handle* is invalid.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the SRQ.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local write privileges.

6.5.8.1 USAGE

For the best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the *DAT_OPTIMAL_ALIGNMENT.*

Since any of the Endpoints that use the SRQ can dequeue the posted buffer from SRQ, Consumers should post a buffer large enough to handle incoming messages on any of these Endpoint connections.

The Consumer should provision the SRQ with enough buffers to accommodate the maximum forseeable number of in-flight messages. This can typically be lower than the maximum per connection times the number of connections. However, reliable estimation of how much underprovisioning is safe is an application specific problem and left to the application.

When supporting multiple connections from a single SRQ, the Consumer should take extra steps to safeguard against a single remote client adversely impacting other clients through malicious or misbehaving code.

6.5.8.2 RATIONALE

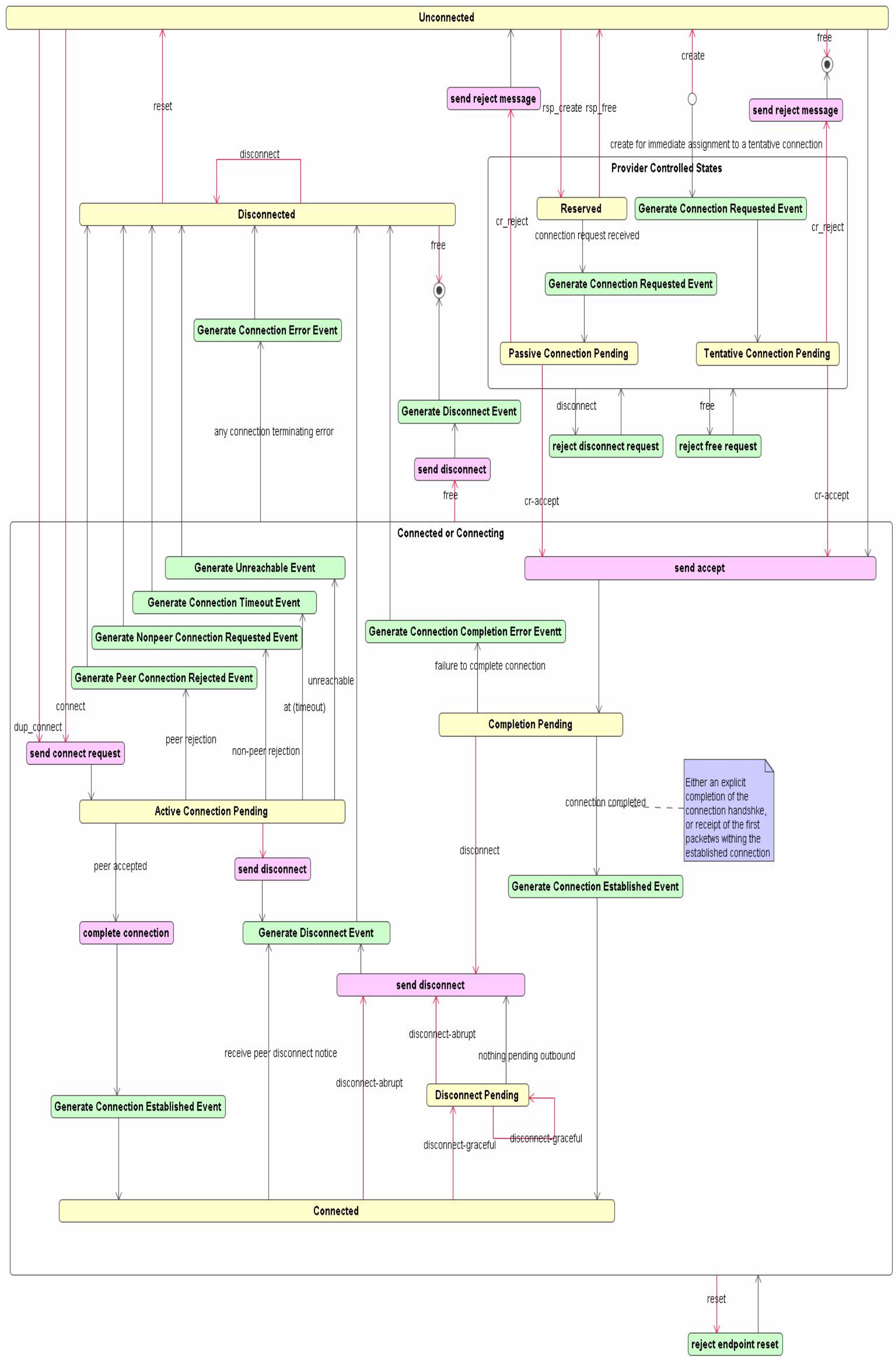
6.5.8.3 MODEL IMPLICATIONS

The buffer posted to SRQ does not have a DTO completion flag value. Posting Recv buffer to SRQ is semantically equivalent to posting to EP with *DAT_COMPLETION_UNSIGNALLED_FLAG* not set. The configuration of the Recv Completion flag of an Endpoint that dequeues the posted buffer defines how DTO completion is generated. If the Endpoint Recv Completion flag is *DAT_COMPLETION_SOLICITED_WAIT_FLAG* then matching Send DTO completion flag value for Solicited Wait determines if the completion will be Signalled or not. If the Endpoint Recv Completion flag is not *DAT_COMPLETION_SOLICITED_WAIT_FLAG* then the posted Recv completion will be generated with Signal. If the Endpoint Recv Completion flag is *DAT_COMPLETION_EVD_THRESHOLD_FLAG* then the posted Recv completion will be generated with Signal and *dat_evd_wait threshold* value controls if the waiter will be unblocked or not.

Only the Endpoint that is in Connected or Disconnect Pending states can dequeue buffers from SRQ. When an Endpoint is transitioned into Disconnected state then all the buffers that it dequeued from SRQ are queued on the Endpoint *recv_evd*. All the buffers that the Endpoint have not completed by the time of transition into Disconnected state, that have not completed message reception, will be flashed.

6.6 ENDPOINT

Following is the state transition diagram of the Endpoint Object.



6.6.1 ENDPOINT LIFECYCLE

This section discusses the lifecycle of DAT Endpoints, as illustrated in the Endpoint State Transition Diagram.

The diagram uses the following color conventions:

- Yellow represents Endpoint states observed by the DAT Consumer.
- Green represents Events related to the Connection establishment and tear down delivered to the DAT Consumer.
- Pink represents Provider-implied actions and potential Endpoint internal states not directly visible by DAT Consumers. They are defined on the diagram to clarify the state transitions for DAT Consumers and to provide guidance for DAT Providers.
- Red arrows represent transitions caused by direct Consumer actions.
- Gray arrows represent transitions caused indirectly, on behalf of the Consumer by the Provider.

The diagram uses the following Object conventions:

- States are shown with slightly rounded edges. A state is characterized by the need for an external act by a Consumer (local or remote), or the Provider acts like a timeout expiration, to cause a transition to another state.
    - Most states are shown in yellow.
    - States shown in pink represent states that are typically invisible to the DAT Consumer because they reflect transport-dependent implementation details that can have some time duration. For example, the *Completion Pending* state for InfiniBand can represent the time waiting for an RTU.
- Actions are shown with fully rounded edges:
    - Actions are green when the action involves Provider interaction with the DAT Consumer.
    - Actions are pink when the action involves Provider interactions with the native transport or the Provider of the remote peer.
- Transitions are represented by lines. The labels indicate Consumer DAT Call, an external event, such as a timeout, that triggers the transition from one state to another.

Endpoints can be created explicitly by the Consumer, or implicitly by the Provider. The Provider is strongly encouraged to create an EP that is ready to be connected (see 6.5.3.2 Rationale on page 107). The explicit creation path is described first.

The first use of an *Unconnected* Endpoint is on the active side of the active-passive model of the Connection establishment to initiate a Connection establishment:

- The Consumer requests the connection by issuing *dat_ep_connect*, *dat_ep_common_connect* or *dat_ep_dup_connect* on the *Unconnected* Endpoint.

- The Transport-specific Connection Request must be sent by the local Provider to the remote peer. This can be sent to the appropriate Connection Manager, but this is invisible to either Consumer.
- This results in transitioning the Endpoint into the *Active Connection Pending* state. The Endpoint transitions out of this state by:
  - Receiving an acceptance from the remote peer
  - Receiving an explicit rejection from the remote peer
  - Receiving a rejection locally due to an inability to reach the remote host or the remote host is not responding within the timeout
  - Receiving a rejection due to timeout
  - Receiving a rejection from another source (such as the local networking stack).

  Each of these receives generates an event on *connect_evd_handle* of the Endpoint. Although DAT does not define an explicit operation for canceling a Connection Request, the Consumer can call *dat_ep_disconnect* (abrupt or graceful results in the same abrupt disconnect) in any Endpoint state during connection setup. (The *Reserved, Passive Connection Pending* and *Tentative Connection Pending* states, as well as *Unconnected* state, are not states of connection setup; therefore, they do not support the *dat_ep_disconnect* operation.) This transitions the Endpoint into *Disconnected* state, flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise, and generates a *Disconnected* event on the Endpoint *connect_evd_handle.*
- When a peer rejection is received by the Provider, a Peer Connection Rejected Event must be generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise. Peer rejections occur when the remote Consumer rejects the connection (*dat_cr_reject*).
- When a nonpeer rejection is received by the Provider, a Nonpeer Connection Rejected Event must be generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise. Nobody listening on the Connection Qualifier, backlog of the Service Point on the Connection Qualifier is full are some of the examples of the nonpeer rejection. If remote Provider rejects the connection request, that will result in local nonpeer reject, it can provide additional reason for rejection or even hints on how to fix it that may be delivered in the reject private data. The format of this private data is transport-specific and is outside the scope of this API.

- When the Provider times out the Connection Request but the remote host is reachable, a Connection Timeout Event must be generated on the *connect_evd_handle* of the Endpoint before the Endpoint transitions state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise.
- When the Provider cannot reach the remote host (if the Provider can determine locally that *remote_ia_address* is invalid, *dat_ep_connect* fails synchronously with *DAT_INVALID_ADDRESS),* or if the remote host does not respond within the Consumer requested *Timeout*, an *Unreachable* Event must be generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise. Inablity to generate a Path to the remote host or the remote host not responding for security reason are examples of the unreachable situation.
- When the requesting Endpoint cannot be connected to the requested remote side before *Timeout* expiration, because of the Transport-specific remnants of the previous connection (IB TimeWait state) or previous connection setup attempt, the local Provider generates a *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* event on the Endpoint *connect_evd_handle.* The Endpoint transitions into a *Disconnected* state, flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise, and no Transport-specific Connection Request is generated for the remote side. The Provider should avoid this scenario by allocating an underlying Transport Endpoint that is ready to be connected.
- When a peer acceptance is received by the Provider, the response can require Transport-dependent actions to complete the Connection establishment (such as sending an RTU under InfiniBand). The Provider must generate a Connection Established Event on the *connect_evd_handle* of the Endpoint as the Endpoint transitions to the *Connected* State.

  If the local Consumer called *dat_ep_disconnect* prior to or during the time when the Provider is receiving or processing remote acceptance, the Provider can either hide the receive of acceptance from the Consumer (connection never established), or present to the Consumer that a connection was established and then torn down. The Consumer must be prepared for both situations: receiving only *Disconnected* event or receiving *Connection Established* event followed by a *Disconnect* one. In either case, the Endpoint ends up in a *Disconnected* state and all preposted DTOs are completed or flushed except recv buffers posted to SRQ associated with EP that remains on SRQ unless they have been dequeue by the EP prior to disconnect or an error.

The second use of an *Unconnected* Endpoint is to use it when accepting a Pending Connection Request. This follows the Public Service Point usage model where the Consumer manages its Unconnected Endpoints. It has the following steps:

- The accept method (*dat_cr_accept*) on the previously uncorrelated Pending Connection Request transitions the Endpoint directly to the Provider's internal but potentially Consumer-visible *Completion Pending* state for the Endpoint.
- In the Endpoint *Completion Pending* state, the Provider takes whatever Transport-specific actions are required to complete the Connection establishment, and then waits for the Transport-specific events to confirm the connection. For InfiniBand, this state is waiting for the active side Provider to send an RTU. Under other transports, this state can be empty or nonexistent.
- When Transport-required actions are completed successfully (connection completed), the Connection Established Event is generated on the *connect_evd_handle* of the Endpoint as the Endpoint transitions to the *Connected* state.
- Alternately, if any error occurs in the specific steps required to complete the connection (for example, active side timed out, Transport error, no IB RTU message), a *Connection Completion Error* Event must be generated on the *connect_evd_handle* of the Endpoint as the Endpoint transitions into a Disconnected state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise.

  If the requesting Endpoint cannot be connected to the requested remote side because of the Transport-specific remnants of the previous connection (IB TimeWait state) or previous connection setup attempt, the Provider generates a *DAT_CONNECTION_COMPLETION_ERROR* event on the Endpoint *connect_evd_handle.* The Endpoint transitions into *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise. Transport-specific nonpeer rejection is generated for the remote side.

The third use of an *Unconnected* Endpoint is to attach it to a Reserved Service Point.

- Create the Endpoint is in the *Unconnected* state (see the initial *create* transition).
- From this state, the Endpoint can be reserved for a Reserved Service Point by the *dat_rsp_create* method (rsp.create). This results in the Endpoint transitioning into the *Reserved* state.

- The *Reserved* state can be exited by releasing the Service Point (rsp.free), which returns the Endpoint to the *Unconnected* state.
- Alternately, the *Reserved* state can be transitioned by arrival of a Connection Request. When the Connection Request arrives, the Provider must generate a Connection Request Event on the Event Dispatcher of the Reserved Service Point before the Endpoint transitions to the *Passive Connection Pending* state.
- In the *Passive Connection Pending* state, the Endpoint waits for the Consumer to accept (*dat_cr_accept*) or reject (*dat_cr_reject*) the pending Connection Request. If rejected, the Provider must send whatever Transport-specific rejection message to the peer, and then transition the Endpoint to the *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise.
- The accept method (*dat_cr_accept*) on the correlated pending Connection Request (cr.accept) transitions the Endpoint to the Provider internal *Completion Pending* state. The transitions from the *Completion Pending* state were described previously.

  If the requesting Endpoint cannot be connected to the requested remote side because of the Transport-specific remnants of the previous connection (IB TimeWait state) or previous connection setup attempt, the Provider generates a *DAT_CONNECTION_COMPLETION_ERROR* event on the Endpoint *connect_evd_handle.* The Endpoint transitions into *Disconnected* state and flushes all preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise. Transport-specific nonpeer rejection is generated for the remote side.

Endpoints can also be created indirectly by the Provider in response to arrived Connection Requests for Public Service Points. The following steps follow this model:

- The Endpoint is created by the Provider as the result of the arrival of a Connection Request.
- This results in the immediate generation of a Connection Requested Event on the Event Dispatcher of the Public Service Point.
- The created Endpoint is provided to a Consumer in the *Tentative Connection Pending* state as a parameter of a pending Connection Request that is given to the Consumer as part of a Connection Requested Event.
- Typically, the Consumer needs to modify the Endpoint before accepting it to assign it to the Consumer-chosen Protection Zone, to assign Consumer-chosen/created Event Dispatchers (*recv_evd_handle, request_evd_handle, connect_evd_handle*), to enable RDMA operations, and/or to modify any other Endpoint parameters and attributes.

- When the Consumer accepts the Connection Request (*dat_cr_accept*), the Endpoint is transitioned to the *Completion Pending* state, as previously described.
- Alternately, when the Consumer rejects the Connection Request (*dat_cr_reject*), the Provider must send a Transport-specific rejection message, and then the Endpoint is destroyed.

If connection cannot be established for whatever reason, the Endpoint transitions into a *Disconnected* state and flushes preposted DTOs if EP is not associated with SRQ and leaves preposted Recv buffers on SRQ otherwise are flushed. The only exception to this rule is Passive side rejection of the Connection Request. This is where the Consumer-created Endpoint transitions from *Passive Connection Pending* back into an *Unconnected* state, and the Provider-created Endpoint in *Tentative Connection pending* state is returned to Provider control and is destroyed as far as the Consumer is concerned. The Provider is responsible for any required Transport-specific action and messages.

If The Consumer called *dat_ep_disconnect* while in *Active Connection Pending* or *Completion Pending* Endpoint states, the Endpoint is transitioned into a *Disconnected* state. The Provider is responsible for any Transport-specific actions and messages. If the Consumer does not want the connection when the Endpoint is in *Passive Connection Pending* or in *Tentative Connection Pending* states, the Consumer should call *dat_cr_reject*. That transitions the Endpoint into an *Unconnected* state for *Passive Connection Pending*, or returns the Endpoint back to the Provider. For the Endpoint in the *Reserved* state, the Consumer shall call *dat_rsp_free* if the Consumer does not want to wait for connection establishment. *dat_ep_disconnect* is not supported in *Unconnected, Reserved, Passive Connection Pending*, or *Tentative Connection Pending* states.

If the Consumer called *dat_ep_free* while in *Active Connection Pending* or *Completion Pending* Endpoint states, the Endpoint is destroyed. Semantically, this is equivalent to first calling *dat_ep_disconnect* and then *dat_ep_free*. The Provider is responsible for any Transport-specific actions and messages that need to be generated for the remote side. If the Endpoint is in *Passive Connection Pending* or *Tentative Connection Pending* states, the Consumer should call *dat_cr_reject* if the Consumer does not want the connection. For *Reserved* state, the Consumer should first destroy the RSP that transitions the Endpoint into an *Unconnected* state where it can be destroyed. *dat_ep_free* is not supported in *Reserved, Passive Connection Pending*, or *Tentative Connection Pending* states where the Endpoint is under Provider control.

From the Endpoint *Connected* state, the Provider responds to a Consumer's request to *graceful* disconnect (*dat_ep_disconnect(graceful)*) as follows:

- The Endpoint is transitioned into *Disconnect Pending* state. The Endpoint remains in this state while the Provider is trying to complete all pending Requests and RMR Binds. Connection being broken, the local or remote Consumer calling *dat_ep_ disconnect(abrupt)* or *dat_ep_free* transitions the Endpoint out of that state.
- If all pending Requests and RMR Binds complete, the Transport-specific disconnect message is sent to the peer or the designated Connection Manager.
- The Disconnected Event is generated on the *connect_evd_handle* of the Endpoint.
- The Endpoint is transitioned into the *Disconnected* state.

From the Endpoint *Connected* or *Disconnect Pending* state, the Provider responds to a Consumer's request to *abrupt* disconnect (*dat_ep_ disconnect(abrupt)*) as follows:

- The Transport-specific disconnect message is sent to the peer or the designated Connection Manager.
- The Disconnected Event is generated on the *connect_evd_handle* of the Endpoint.
- The Endpoint is transitioned into the *Disconnected* state.

There is an inherent race condition between *dat_ep_disconnect* and the connection being broken. The Provider shall not generate both *Connection Broken* and *Disconnected* events for the same connection tear down. The Provider must take whatever Transport-specific action is required. Regardless of the cause of connection tear down, the Endpoint transitions into a *Disconnected* state. The Consumer shall be ready to receive either *Connection Broken* or *Disconnected* events.

From the Endpoint *Connected* state, the Provider responds to the arrival of a peer request to disconnect (receive peer disconnect request), as follows:

- The Disconnected Event is generated on the *connect_evd_handle* of the Endpoint if *Connection Broken* or *Disconnected* events were not already generated.
- The Endpoint is transitioned to the *Disconnected* state.

Upon entering the *Disconnected* state, the Provider is responsible for flushing all outstanding DTOs and RMRs except recv buffers posted to SRQ associated with EP that remains on SRQ unless they have been dequeue by the EP prior to disconnect or an error and completing in-progress DTOs and RMRs, preserving completion ordering. The Consumer can post any DTO and RMR to the Endpoint in the *Disconnected* state, but these postings are flushed immediately except posting Recv DTOs to EP that is associated with SRQ that is not allowed. The Consumer can use this feature by posting a DTO or an RMR operation that results in the completion entry for each of the *recv_evd_ handle, request_evd_handle,* and *rmr_evd_handle.* These completions can serve as markers, so that when the Consumer dequeues them from the EVD, there is no more completion on the EVD related to the Endpoint.

From the Endpoint *Disconnected* state, the Provider responds to a Consumer *dat_ep_reset* by transitioning the Endpoint into an *Unconnected* state. The *dat_ep_reset* is a synchronous operation and does not cause any event generation. The return of a *dat_ep_reset* operation might cause the loss of any DTO/RMR completions related to the Endpoint that were not dequeued by the Consumer. The Provider can hide any Transport-specific remnants of the previous connection or connection establishment attempt in this transition.

The Endpoint can be destroyed by the Consumer in any Endpoint state, except *Reserved, Passive Connection Pending,* and *Tentative Connection Pending.* When the Endpoint is destroyed, any DTO/RMR completions not dequeued by the Consumer might be lost. This includes completions for all outstanding and in-progress DTOs/RMRs. The Consumer must be ready for all completions that were not dequeued yet either still being on the Endpoint *recv_evd_handle, request_evd_handle, rmr_evd_handle* or not being there.

If the Endpoint is in *Reserved* state, the Consumer shall first destroy the associated Reserved Service Point that transitions the Endpoint into an *Unconnected* state where the Endpoint can be destroyed. If the Endpoint is in *Passive Connection Pending* state, the Consumer shall first reject the associated Connection Request that transitions the Endpoint into an *Unconnected* state where the Endpoint can be destroyed. If the Endpoint is in a *Tentative Connection Pending* state, the Consumer shall reject the associated Connection Request that transitions the Endpoint back to Provider control, and the Endpoint is destroyed as far as the Consumer is concerned.

When an Endpoint is in a *Connected* state or in the process of establishing a Connection, a transport level error might occur. This might happen because of the violation to the reliability model. The error results in a Connection Broken Event being generated on the *connect_evd_handle* of the Endpoint by the Provider and the Endpoint transitioning into the *Disconnected* state. All the posted DTO and RMR Bind operations to the Endpoint Request Work Queue are automatically flushed by the Provider. If the EP is not associated with SRQ then posted Recv DTO are also flushed, and leaves preposted Recv buffers on SRQ otherwise. From the *Disconnected* state, the Endpoint can be transitioned into the *Unconnected* state via *dat_ep_reset,* or destroyed via *dat_ep_free.*

Note that when the violation was a result of a local operation, the Connection *Disconnected* Event is in addition to the Error Completion event. This allows the DAT Consumer to remain ignorant of which errors cause connection terminations for which types of Endpoints.

The only allowed transitions from the *Disconnected* state is to free/destroy the Endpoint or transition the Endpoint into the *Unconnected* state via *dat_ep_reset*.

6.6.1.1 ADVICE TO IB IMPLEMENTORS

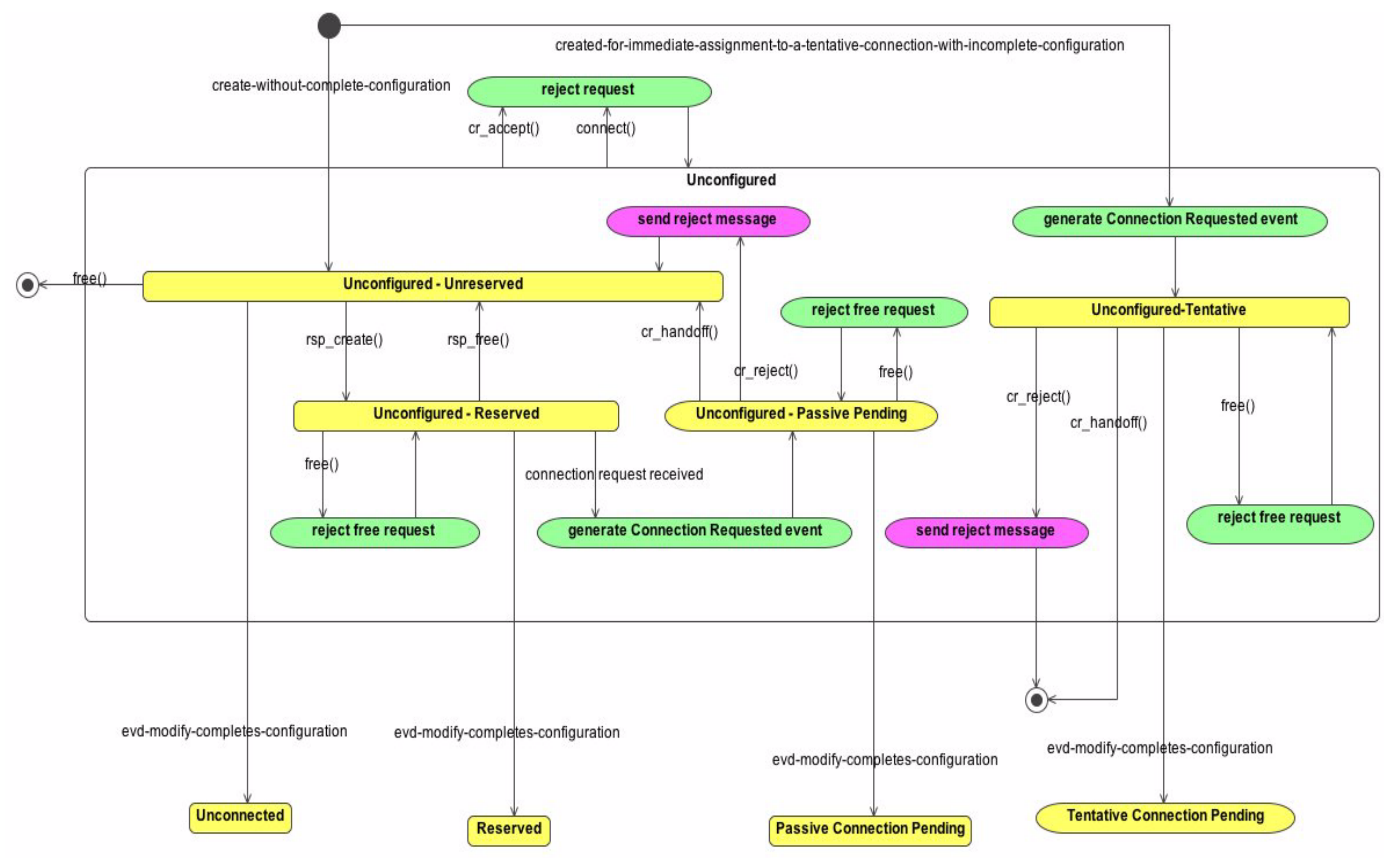
For IB Providers, the following algorithm can be employed to for Active-side Providers:

- If the Provider can locally determine that the *remote_ia_address* is invalid, or that the *remote_ia_address* cannot be converted to a Transport-specific address, generate *DAT_INVALID_ADDRESS* return synchronously.
- If *remote_ia_address* cannot be converted to the IB path, generate an Unreachable event. This is done if the address cannot be resolved remotely (say on a switch) by SA or IPoverIB ARP. Nothing is generated for the remote side.
- Convert local Endpoint IA Address and *comm* into remote Connection Qualifier using IBTA RDMA IP CM Service Annex specification. If remote Connection Qualifier can not be deduced from the provide information, generate an Unreachable event. Nothing is generated for the remote side.
- Encode connection addressing information using IBTA RDMA IP CM Service Annex specification. If an error is discovered during encoding either return *DAT_INVALID_ADDRESS* synchronously, or generate an Unreachable event. Nothing is generated for the remote side.
- If the remote Consumer accepts the connection before *Timeout* expires, and if the remote Endpoint RDMA Read credits match the local Endpoint, generate a Connection Established event with Passive-side Consumer passed-in Private Data and generate an Accept message for the Passive side without any Private Data.
- If the remote Consumer rejects connection, generate a Peer Rejected event.
- If the remote Provider/CM rejected but not with a redirect reason, generate a Nonpeer Reject event.
- If the remote side Provider/CM rejected with a redirect reason and the Consumer *Timeout* allows time for another remote host specified in a redirection reason, do so. If not, generate a Nonpeer Reject event.
- If the Consumer *Timeout* expires, generate a Timeout event if an MRA or a reject with redirection reason was received. Also generate a Reject with Reject reason 4 (timeout) for the remote side.
- If the Consumer *Timeout* expires, generate an Unreachable event if an MRA or a reject with redirection reason was not received.

- If CM timeout expires and remote side have not responded, generate an Unreachable event.
- The IB CM rejects with Reject codes: 12-17, 24-26, 32. This corresponds to a Redirect reject. The Provider is responsible for fixing the Path appropriately to the IB Reject reason and trying to connect to the specified redirection host if the Consumer *Timeout* permits.
- The remote DAPL Provider rejects the connection it may specify reject reason. For example, RDMA IP CM Service Provider may indicate badly formed private data. This may indicate a problem with local DAPL Provider, local RDMA IP CM Service Provider or Consumer specified arguments.

6.6.2 DEFERRED CONFIGURATION ENDPOINT

The Unconfigured state is entered when an Endpoint is created with an



incomplete configuration. A configuration is deferred if PZ or any of the EVD references supplied are NULL. This can be done in either of the ways that a fully configuration-deferred (unconfigured) Endpoint could have been created:

- By a Consumer directly creating the Endpoint. If not fully specified this results in the Unconfigured-Unreserved sub-state.
- By the Provider creating an Endpoint for a tentative connection (and delivering it with a Connection Request). This results in the Unconfigured-Tentative sub-state.

While EP is configuration-deferred any attempt to initiate a connection (*dat_ep_connect*) or accept a connection (*dat_cr_accept*) with the Endpoint will be rejected.

An Unconfigured Endpoint may be freed.

An Unconfigured Endpoint may be referenced in a newly created RSP (*dat_rsp_create*). This transitions the Endpoint from the Unconfigured-Unreserved sub-state to the Unconfigured-Reserved sub-state.

As with any Endpoint claimed by an RSP, an Unconfigured-Reserved Endpoint will reject an attempt to delete it (*dat_ep_free*).

While in the Unconfigured-Reserved substate, a Connection Request for the RSP can be received. This results in generating the Connection Request Event and transitioning the Endpoint to the Unconfigured-Passive Pending sub-state.

In the Unconfigured-Passive Pending sub-state the Connection Request can be rejected. This results in freeing the RSP, just as would have happened with a fully configured Endpoint. Note that the Connection Request cannot be accepted while the Endpoint is deferred configuration.

If *dat_cr_handoff* is issued on the Endpoint in Unconfigured-Passive Pending sub-state or Unconfigured-Tentative sub-state then Endpoint transitions into Unconfigured state.

A *dat_ep_modify* that completes the EVD's configuration (i.e, there are no remaining NULL references to PZ or EVDs) will transition out of the Unconfigured state. The resulting state depends on the current sub-state of the Endpoint:

- From Unconfigured-Unreserved to Unconnected
- From Unconfigured-Reserved to Reserved
- From Unconfigured-Passive Pending to Passive Connection Pending
- From Unconfigured-Tentative to Tentative Connection Pending

In all cases the result of fully configuring the Endpoint is to be in the same state that it would have been had it been fully configured when it was created.

There is no transition from Configured Endpoint to Unconfigured one.

6.6.3 CONNECTION ESTABLISHMENT MODELS

DAT defines the following methods by which a connection can be established for the passive side:

- Using a Public Service Point with Consumer-allocated Endpoints.
- Using a Public Service Point with Provider-allocated Endpoints.
- Using a Reserved Service Point.
- Using a Common Service Point

This section describes when and why each model is used.

Regardless of which model is used when the Consumer accepts a Connection Request, it is notified by the Provider that the connection is established by receiving *DAT_CONNECTION_EVENT_ESTABLISHED* on the *connect_evd_handle* of the Endpoint on which the connection is accepted.

6.6.3.1 USING A PUBLIC SERVICE POINT WITH CONSUMER-ALLOCATED ENDPOINTS

Public Service Points are used when the passive-side Consumer intends to accept connections from a wide audience of possible clients. The invitation to connect remains in place until explicitly removed by destroying the Public Service Point.

Consumer-allocated Endpoints should be used except when the Provider attributes specify that the Provider must create the Endpoint, which might be the Transport requirement. It must be used if the Provider indicates that it never creates Endpoints.

Consumer-allocated Endpoints can be created and fully configured *before* the Connection Request is received. This is especially optimal when the service has been preconfigured to support a specific peak number of Connections. If *n* are to be accepted at peak, *n* preconfigured Endpoints are placed in a pool. When one is available, the Connection Request is accepted with the next free *Unconnected* Endpoint. When none are available, the Connection Request is rejected by the Consumer.

The Consumer should check Provider attributes or Provider documentation before creating an Endpoint pool. There is no reason to create a pool if the Provider insists on allocating all Endpoints.

The Consumer can choose to create the Endpoints in response to Connection Requests. One advantage of this approach is that it can be implemented without advance checking of Endpoint parameters. When a Connection Request is received, the Consumer checks to see whether an Endpoint was supplied. If none was, it creates it with the desired characteristics; otherwise, it modifies it to have the desired characteristics.

6.6.3.2 USING A PUBLIC SERVICE POINT WITH PROVIDER-ALLOCATED ENDPOINTS

Public Service Points can also be used with Provider-allocated Endpoints. There are two potential reasons for using Provider-allocated Endpoints:

- The Provider insists on it. This is the case whenever RDMA services are defined over an existing transport layer, such as with iWARP. A non-RDMA Connection is established automatically before the RDMA layers are invoked. That connection must then be modified to enable RDMA.
- The attributes for Endpoints cannot be easily predicted in advance. If the Endpoints have to be modified, a single Provider pool is more efficient that multiple Consumer pools or creating Endpoints for each Connection Request.

When using a Provider-allocated Endpoint, the Consumer **must** modify it before accepting the Connection Request. As delivered by the Provider, the Endpoint does not have a valid Protection Zone or Event Dispatchers assigned.

After a Consumer accepts a Connection Request, it becomes the owner of the Endpoint regardless of whether the f connection setup is successful. When the connection is eventually torn down, the Endpoint transitions into a *Disconnected* state. However it is not destroyed until the Consumer does so explicitly (or closes the IA handle).

6.6.3.3 USING A RESERVED SERVICE POINT

A Reserved Service Point allows the Consumer to supply a single preconfigured Endpoint for accepting a single connection. This is typically used to create auxiliary connections in an already established session. Protocols such as FTP, RTSP, and DAFS can negotiate additional connections using a primary connection established with a Public Service Point.

Reserved Service Points can also be used to establish peer-to-peer connections when the active/passive roles are not appropriate to the relationship between the two parties.

Typically, an Endpoint is fully configured before the Reserved Service Point is created. It could then just be accepted without modifications. If rejected, the Endpoint should either be destroyed or returned to whatever pool the Consumer allocated it from.

6.6.3.4 USING A COMMON SERVICE POINT

This model is analogous to the PSP model with the Consumer create Endpoints. The main difference that Common Service Points have all the parameters of the platform socket. This simplifes an integration of the RDMA connection model with the networking one. This allows ULP to use the existing model for ULP service advertisement, IP ports, and protocol simulation support.

6.6.3.5 MIXING CONNECTION MODELS

It is recommended not to mix the connection models on two sides of the connection. So for common connection model use CSP, *dat_ep_common_connect* together with setting up connecting Endpoints *comm*, and local IA_Address, and for others use PSP and RSP and *dat_ep_connect* with inheriting addressing information from the IA.

6.6.4 DAT_EP_CREATE

Synopsis:

```
DAT_RETURN
   dat_ep_create (
   IN     DAT_IA_HANDLE           ia_handle,
   IN     DAT_PZ_HANDLE           pz_handle,
   IN     DAT_EVD_HANDLE          recv_evd_handle,
   IN     DAT_EVD_HANDLE          request_evd_handle,
   IN     DAT_EVD_HANDLE          connect_evd_handle,
   IN     DAT_EP_ATTR             *ep_attributes,
   OUT    DAT_EP_HANDLE           *ep_handle
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA to which the created Endpoint belongs.
pz_handle:	Handle for an instance of the Protection Zone.
recv_evd_handle:	Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in events for completions of receives.
request_evd_handle:	Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in events for completions of requests.
connect_evd_handle:	Handle for the Event Dispatcher where Connection events are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in *connection* events for now.
ep_attributes:	Pointer to a structure that contains Consumer-requested Endpoint attributes. Can be NULL.
ep_handle:	Handle for the created instance of an Endpoint.

Description: *dat_ep_create* creates an instance of an Endpoint that is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the value of DAT_RETURN is not *DAT_SUCCESS*.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint.

recv_evd_handle and *request_evd_handle* are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

The *ep_attributes* parameter specifies the initial attributes of the created Endpoint. If the Consumer specifies NULL, the Provider fills it with its default Endpoint attributes. The Consumer might not be able to do any posts to the Endpoint or use the Endpoint in connection establishment until certain Endpoint attributes are set. Maximum Message Size and Maximum Recv DTOs are examples of such attributes.

For *max_recv_dtos, max_request_dtos, max_recv_iov,* and *max_request_iov* the created Endpoint will have at least the Consumer requested values but may have larger values. Consumer can query the created Endpoint to find out the actual values for these attributes. Created Endpoint has the exact Consumer requested values for *max_message_size, max_rdma_size, max_rdma_read_in,* and *max_rdma_read_out.* For all other attributes the created Endpoint has the exact values requested by Consumer. If Provider cannot satisfy the Consumer requested attribute values the operation fails.

dat_ep_create is synchronous and thread safe.

Returns

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle.
DAT_INVALID_PARAMETER	Invalid parameter; One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid. For example, requested maximum RDMA Read IOV exceeds IA capabilities.

DAT_MODEL_NOT_SUPPORTED	The requested Provider Model was not supported.

6.6.4.1 USAGE

The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in *DAT_EP_STATE_UNCONNECTED*. Consumers can do the following:

1) Request a connection on the Endpoint through *dat_ep_connect* or *dat_ep_dup_connect* for the active side of the connection model.
2) Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the connection model.
3) Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it.

The Consumer cannot specify a *request_evd_handle (recv_evd_handle)* with *Request Completion Flags (Recv Completion Flags)* that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle (recv_evd_handle)* is used for an EVD that is fed by any event stream other than DTO or RMR completion event streams, only *DAT_COMPLETION_THRESHOLD* is valid for *Request/Recv Completion Flags* for the Endpoint completion streams that use that EVD. If *request_evd_handle (recv_evd_handle)* is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is *DAT_COMPLETION_UNSIGNALLED_FLAG*, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flags attribute is *DAT_COMPLETION_SOLICITED_WAIT*, the Recv Completion Flags for all Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types.

If EP is created with NULL attributes, Provider can fill them with its own default values. The Consumer should not rely on the Provider-filled attribute defaults, especially for portable applications. The Consumer cannot do any operations on the created Endpoint except for *dat_ep_query*, *dat_ep_get_status*, *dat_ep_modify*, and *dat_ep_free*, depending on the values that the Provider picks.

The Provider is encouraged to pick up reasonable defaults because unreasonable values might restrict Consumers to the *dat_ep_query*, *dat_* *ep_get_status*, *dat_ep_modify*, and *dat_ep_free* operations. The Consumer should check what values the Provider picked up for the attributes. It is especially important to make sure that the number of posted operations is not too large to avoid EVD overflow. Depending on the values picked up by the Provider, the Consumer might not be able to do any RDMA operations; it might only be able to send or receive messages of very small sizes, or it might not be able to have more than one segment in a buffer. Before doing any operations, except the ones listed above, the Consumer can configure the Endpoint using *dat_ep_modify* to the attributes they want.

One reason the Consumer might still want to create an Endpoint with Null attributes is for the Passive side of the connection establishment, where the Consumer sets up Endpoint attributes based on the connection request of the remote side.

Consumers might want to create Endpoints with NULL attributes if Endpoint properties are negotiated as part the Consumer connection establishment protocol.

Consumers that create Endpoints with Provider default attributes should always verify that the Provider default attributes meet their application's requirements with regard to the number of request/receive DTOs that can be posted, maximum message sizes, maximum request/receive IOV sizes, and maximum RDMA sizes.

6.6.4.2 RATIONALE

Note to Provider: The Provider is strongly encouraged to create an EP that is ready to be connected. This means that any effects of previous connections or connection establishment attempts on the underlying Transport-specific Endpoint to which the DAT Endpoint is mapped to should be hidden from the Consumer. There are multiple ways to do that. The methods described below are examples of it:

- The Provider does not create an underlying Transport Endpoint until the Consumer is connecting the Endpoint or accepting a connection request on it. This allows the Provider to accumulate Consumer requests for attribute settings even for attributes that the underlying transport does not allow to change after the Transport Endpoint is created. On the negative side, handling of *dat_ep_post_recv* becomes much harder.
- The Provider creates the underlying Transport Endpoint or chooses one from a pool of Provider-controlled Transport Endpoints when the Consumer creates the Endpoint. The Provider chooses the Transport Endpoint that is free from any underlying internal attributes that might prevent the Endpoint from being connected. For IB and IP, that means that the Endpoint is not in the TimeWait state. Handling of *dat_ep_post_recv* becomes a simple map to the Transport post. Changing of some of the Endpoint attributes

becomes hard and might potentially require mapping the Endpoint to another underlying Transport Endpoint that might not be feasible for all transports.

- The Provider allocates a Transport-specific Endpoint without worrying about impact on it from previous connections or connection establishment attempts. Hide the Transport-specific TimeWait state or CM timeout of the underlying transport Endpoint within *dat_ep_connect, dat_ep_dup_connect,* or *dat_cr_accept.* On the Active side of the connection establishment, if the remnants of a previous connection for Transport-specific Endpoint can be hidden within the *Timeout* parameter, do so. If not, generating *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* is an option. For the Passive side, generating a *DAT_CONNECTION_COMPLETION_ERROR* event locally, while sending a non-peer-reject message to the active side, is a way of handling it.

Any transitions of an Endpoint into an *Unconnected* state can be handled similarly. One transition from a *Disconnected* to an *Unconnected* state is a special case.

For *dat_ep_reset*, *the* Provider can hide any remnants of the previous connection or failed connection establishment in the operation itself. Because the operation is synchronous, the Provider can block in it until the TimeWait state effect of the previous connection or connection setup is expired, or until the Connection Manager timeout of an unsuccessful connection establishment attempt is expired. Alternatively, the Provider can create a new Endpoint for the Consumer that uses the same handle.

6.6.4.3 MODEL IMPLICATIONS

DAT Providers are required not to change any Consumer-specified Endpoint attributes during connection establishment. If the Consumer does not specify an attribute, the Provider can set it to its own default. Some EP attributes, like outstanding RDMA Read incoming or outgoing, if not set up by the Consumer, can be changed by Providers to establish connection. It is recommended that the Provider pick the default for outstanding RDMA Read attributes as 0 if the Consumer has not specified them. This ensures that connection establishment does not fail due to insufficient outstanding RDMA Read resources, which is a requirement for the Provider.

Provider is not required to check for a mismatch between the maximum RDMA Read IOV and maximum RDMA Read outgoing attributes, but is allowed to do so. In the later case it is allowed to return *DAT_INVALID_PARAMETER* when a mismatch is detected. Provider must allocate resources to satisfy the combination of these two EP attributes for local RDMA Read DTOs.

6.6.5 DAT_EP_CREATE_WITH_SRQ

Synopsis:

```
DAT_RETURN
   dat_ep_create_with_srq (
   IN     DAT_IA_HANDLE           ia_handle,
   IN     DAT_PZ_HANDLE           pz_handle,
   IN     DAT_EVD_HANDLE          recv_evd_handle,
   IN     DAT_EVD_HANDLE          request_evd_handle,
   IN     DAT_EVD_HANDLE          connect_evd_handle,
   IN     DAT_SRQ_HANDLE          srq_handle,
   IN const DAT_EP_ATTR           *ep_attributes,
   OUT    DAT_EP_HANDLE           *ep_handle
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA to which the created Endpoint belongs.
pz_handle:	Handle for an instance of the Protection Zone.
recv_evd_handle:	Handle for the Event Dispatcher where events for completions of incoming (receive) DTOs are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in events for completions of receives.
request_evd_handle:	Handle for the Event Dispatcher where events for completions of outgoing (Send, RDMA Write, RDMA Read, and RMR Bind) DTOs are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in events for completions of requests.
connect_evd_handle:	Handle for the Event Dispatcher where Connection events are reported. *DAT_HANDLE_NULL* specifies that the Consumer is not interested in *connection* events for now.
srq_handle:	Handle for an instance of the Shared Receive Queue.
ep_attributes:	Pointer to a structure that contains Consumer-requested Endpoint attributes. Cannot be NULL.
ep_handle:	Handle for the created instance of an Endpoint.

Description: *dat_ep_create_with_srq* creates an instance of an Endpoint that is using SRQ for Recv buffers is provided to the Consumer as *ep_handle*. The value of *ep_handle* is not defined if the value of DAT_RETURN is not *DAT_SUCCESS*.

The Endpoint is created in the Unconnected state.

Protection Zone *pz_handle* allows Consumers to control what local memory the Endpoint can access for DTOs except Recv and what memory remote RDMA operations can access over the connection of a created Endpoint. Only memory referred to by LMRs and RMRs that match the Endpoint Protection Zone can be accessed by the Endpoint. The Recv DTO buffers PZ must match the SRQ PZ. The SRQ PZ may or may not be the same as the EP PZ. Check Provider attribute for the support of different PZs between SRQ and its EPs.

recv_evd_handle and *request_evd_handle* are Event Dispatcher instances where the Consumer collects completion notifications of DTOs. Completions of Receive DTOs are reported in *recv_evd_handle* Event Dispatcher, and completions of Send, RDMA Read, and RDMA Write DTOs are reported in *request_evd_handle* Event Dispatcher. All completion notifications of RMR bindings are reported to a Consumer in *request_evd_handle* Event Dispatcher.

All Connection events for the connected Endpoint are reported to the Consumer through *connect_evd_handle* Event Dispatcher.

Shared Receive Queue *srq_handle* specifies from where the EP will dequeue Recv DTO buffers.

The created EP can be reset. The relationship between SRQ and EP is not effected by *dat_ep_reset.*

SRQ cannot be disassociated or replaced from created EP. The only way to disassociate SRQ from EP is to destroy EP.

Receive buffers cannot be posted to the created Endpoint. Receive buffers must be posted to the SRQ to be used for the created Endpoint.

The *ep_attributes* parameter specifies the initial attributes of the created Endpoint. Consumer cannot specify NULL for *ep_attributes* but can specify values only for the parameters needed and default for the rest.

For *max_request_dtos* and *max_request_iov* the created Endpoint will have at least the Consumer requested values but may have larger values. Consumer can query the created Endpoint to find out the actual values for these attributes. Created Endpoint has the exact Consumer requested values for *max_recv_dtos, max_message_size, max_rdma_size, max_rdma_read_in,* and *max_rdma_read_out.* For all other attributes, except *max_recv_iov* that is ignored, the created Endpoint has the exact values requested by Consumer. If Provider cannot satisfy the Consumer requested attribute values the operation fails.

dat_ep_create_with_srq is synchronous and thread safe.

Returns

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle.
DAT_INVALID_PARAMETER	Invalid parameter; One of the requested EP parameters or attributes was invalid or a combination of attributes or parameters is invalid. For example, *pz_handle* specified does not match the one for SRQ or requested maximum RDMA Read IOV exceeds IA capabilities.
DAT_MODEL_NOT_SUPPORTED	The requested Provider Model was not supported.

6.6.5.1 USAGE

The Consumer creates an Endpoint prior to the establishment of a connection. The created Endpoint is in *DAT_EP_STATE_ UNCONNECTED*. Consumers perform one of the following actions:

1) Request a connection on the Endpoint through *dat_ep_connect* or *dat_ep_dup_connect* for the active side of the connection model.
2) Associate the Endpoint with the Pending Connection Request that does not have an associated local Endpoint for accepting the Pending Connection Request for the passive/server side of the connection model.
3) Create a Reserved Service Point with the Endpoint for the passive/server side of the connection model. Upon arrival of a Connection Request on the Service Point, the Consumer accepts the Pending Connection Request that has the Endpoint associated with it.

The Consumer cannot specify a *request_evd_handle (recv_evd_handle)* with *Request Completion Flags (Recv Completion Flags)* that do not match the other Endpoint Completion Flags for the DTO/RMR completion streams that use the same EVD. If *request_evd_handle (recv_evd_ handle)* is used for request (recv) completions of an Endpoint whose associated Request (Recv) Completion Flag attribute is *DAT_ COMPLETION_UNSIGNALLED_FLAG*, the Request Completion Flags and Recv Completion Flags for all Endpoint completion streams that use the EVD must specify the same Completion Flag values. Recall that by definition completions of all Recv DTO posted to SRQ complete with Signal. Analogously, if *recv_evd_handle* is used for recv completions of an Endpoint whose associated Recv Completion Flag attribute is *DAT_ COMPLETION_SOLICITED_WAIT*, the Recv Completion Flags for all

Endpoint Recv completion streams that use the same EVD must specify the same Recv Completion Flags attribute value and the EVD cannot be used for any other event stream types. If *recv_evd_handle* is used for Recv completions of an Endpoint that uses SRQ and whose Recv Completion Flag attribute is *DAT_COMPLETION_EVD_THRESHOLD* then all Endpoint DTO completion streams (request and/or recv completion streams) that use that *recv_evd_handle* must specify *DAT_COMPLETION_EVD_THRESHOLD.* Other event stream types can also use the same EVD.

Consumers may want to use *DAT_COMPLETION_UNSIGNALLED_FLAG* for Request and/or Recv completions when they control locally via posted DTO/RMR completion flag (not needed for Recv posted to SRQ) whether posted DTO/RMR completes with Signal or not. Consumers may want to use *DAT_COMPLETION_SOLICITED_WAIT* for Recv Completion Flags attribute when the remote sender side controls whether posted Recvs complete with Signal or not. uDAPL Consumers may want to use *DAT_COMPLETION_EVD_THRESHOLD* for Request and/or Recv Completion Flags attributes when they control EVD waiter unblocking via *threshold* parameter of the *dat_evd_wait*.

Some Providers may restrict whether multiple EPs that share a SRQ can have different Protection Zones (see *srq_ep_pz_difference_support* Provider attribute).

Consumers may want to have a different PZ between EP and SRQ. This allows incoming RDMA operations to be specific to this EP PZ and not the same for all EPs that share SRQ. This is critical for servers that support multiple independent clients.

6.6.5.2 RATIONALE

Note to Provider: The Provider is strongly encouraged to create an EP that is ready to be connected. This means that any effects of previous connections or connection establishment attempts on the underlying Transport-specific Endpoint to which the DAT Endpoint is mapped should be hidden from the Consumer. There are multiple ways to do that. Several methods for hiding these effects described below.

- The Provider does not create an underlying Transport Endpoint until the Consumer is connecting the Endpoint or accepting a connection request on it. This allows the Provider to accumulate Consumer requests for attribute settings, including attributes that the underlying transport does not allow to change after the Transport Endpoint is created.
- The Provider creates the underlying Transport Endpoint or chooses one from a pool of Provider-controlled Transport Endpoints when the Consumer creates the Endpoint. The Provider chooses the Transport Endpoint that is free from any underlying internal attributes that might prevent the Endpoint from being connected. For IB and IP, that means that the Endpoint is not in the TimeWait state. Changing of

some of the Endpoint attributes becomes difficult and might potentially require mapping the Endpoint to another underlying Transport Endpoint. This might not be feasible for all transports.

- The Provider allocates a Transport-specific Endpoint without considering about impact on it from previous connections or connection establishment attempts. Hide the Transport-specific TimeWait state or CM timeout of the underlying transport Endpoint within *dat_ep_connect, dat_ep_dup_connect,* or *dat_cr_accept.* On the Active side of the connection establishment, if the remnants of a previous connection for Transport-specific Endpoint can be hidden within the *Timeout* parameter, do so. If not, generating *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* is an option. For the Passive side, one option is enerating a *DAT_CONNECTION_COMPLETION_ERROR* event locally, while sending a non-peer-reject message to the active side.

Any transitions of an Endpoint into an *Unconnected* state can be handled similarly. One transition from a *Disconnected* to an *Unconnected* state is a special case.

For *dat_ep_reset*, *the* Provider can hide any remnants of the previous connection or failed connection establishment in the operation itself. Because the operation is synchronous, the Provider can block in it until the TimeWait state effect of the previous connection or connection setup is expired, or until the Connection Manager timeout of an unsuccessful connection establishment attempt is expired. Alternatively, the Provider can create a new Endpoint for the Consumer that uses the same handle.

6.6.5.3 MODEL IMPLICATIONS

DAT Providers are required not to change any Consumer-specified Endpoint attributes during connection establishment. If the Consumer does not specify an attribute, the Provider can set it to its own default. Some EP attributes, such as outstanding RDMA Read incoming or outgoing, if they are not set up by the Consumer can be changed by Providers to establish connection. It is recommended that the Provider pick the default for outstanding RDMA Read attributes as 0 if the Consumer has not specified them. This ensures that connection establishment does not fail due to insufficient outstanding RDMA Read resources, which is a requirement for the Provider.

Provider is not required to check for a mismatch between the maximum RDMA Read IOV and maximum RDMA Read outgoing attributes, but is allowed to do so. In the later case it is allowed to return *DAT_INVALID_PARAMETER* when a mismatch is detected. Provider must allocate resources to satisfy the combination of these two EP attributes for local RDMA Read DTOs.

6.6.6 ENDPOINT ATTRIBUTES

The list of Endpoint attributes is as follows:

Service Type:	For uDAPL-2.0, the only allowed type is Reliable Connection (for requirements, see the reliability model discussion on page 41. In the future, other types can be defined.
Max_Message_Size:	Requested maximum message transfer size for the Connection on the Endpoint. The MaxMessageSize specifies the maximum amount of payload data that can be transferred in a single DTO send/receive message in either direction of the Connection on the Endpoint.
Max_RDMA_Size:	Requested maximum RDMA transfer size for the Connection on the Endpoint. The Max_ RDMA_Size specifies the maximum amount of payload data that can be transferred in a single RDMA DTO initiated by the Endpoint in either direction of the Connection on the Endpoint.
QoS:	Quality of Service of the Connection on the Endpoint.
Recv Completion Flags:	Indicator of support for Completion Notification of posted receive operations. *DAT_COMPLETION_SOLICITED_WAIT_ FLAG* indicates that Notifications of the posted Receive DTOs are controlled by *DAT_ COMPLETION_SOLICITED_WAIT_FLAG* value of the matching Send. *DAT_ COMPLETION_UNSIGNALLED_FLAG* indicates that Notifications of the posted Recvs are explicitly controlled by the Consumer via posted Recvs Notification Suppression flag value. The default value for EPs that use SRQ is *DAT_COMPLETION_ UNSIGNALLED_FLAG* which means that all Recv buffers posted to SRQ will complete with Signal Notification. *DAT_COMPLETION_ EVD_THRESHOLD_FLAG* indicates that all Recv completions are generated with Notifications and Consumer controls unblocking of the EVD waiters via *threshold* of the associated EVD. This attribute is local and has no effect on the remote side of the connection or on connection establishment.

Request Completion Flags:	Indicator of support for Completion Notifications of posted outgoing operations. *DAT_COMPLETION_UNSIGNALLED_FLAG* indicates that Notification of the posted Send, RDMA Read, RDMA Write, and RMR Bind are explicitly controlled by Consumer via posted operation Notification Suppression flag value. *DAT_COMPLETION_EVD_THRESHOLD_FLAG* indicates that all posted Send, RDMA Read, RDMA Write, and RMR Bind operation completions are generated with Notifications and Consumer controls unblocking of the EVD waiters via *threshold* of the associated EVD. This attribute is local and has no effect on the remote side of the connection or on connection establishment.
Max_Recv_DTOs:	Maximum number of outstanding Consumer-submitted Receive DTOs that a Consumer expects at any one time at the Endpoint. If SRQ is associated with the EP then this attribute specifies the hard high watermark limit for the number of Recv buffers consumed by the EP from SRQ. When EP tries to exceed this limit the connection will be broken. If Provider does not support hard limit high watermark then the value of *DAT_HW_DEFAULT* must be specified, otherwise, *dat_ep_create_with_srq* will fail with *DAT_MODEL_NOT_SUPPORTED.*
Max_Request_DTOs:	Maximum number of outstanding Consumer-submitted Send, RDMA Read, RDMA Write DTOs, and RMR Binds combined that the Consumer expects at any one time at the Endpoint.
Max_Recv_IOV:	Maximum number of elements in IOV that the Consumer specifies for a posted Receive DTO for the Endpoint. If SRQ is associated with EP then this value is ignored for create and modify operations. For query this value will return 0 if SRQ is associated with EP.
Max_Request_IOV:	Maximum number of elements in IOV that the Consumer specifies for a posted Send DTO, or RMR Bind for the Endpoint.
Max_RDMA_Read_in	Maximum number of outstanding RDMA Reads that have the Endpoint as the target.
Max_RDMA_Read_out	Maximum number of outstanding RDMA Reads that have the Endpoint as the originator.

soft_high_watermark	The soft high watermark is the number of buffers consumed by the EP. When EP exceeds this number an event will be generated on IA *async_evd*. If the Provider does not support soft limit high watermark then the value of *DAT_HW_DEFAULT* must be specified, otherwise, *dat_ep_create_with_srq* will fail with *DAT_MODEL_NOT_SUPPORTED.*
Max_RDMA_Read_IOV:	Maximum number of elements in IOV that the Consumer specifies for a posted RDMA Read DTO for the Endpoint.
Max_RDMA_Write_IOV:	Maximum number of elements in IOV that the Consumer specifies for a posted RDMA Write DTO for the Endpoint.
Num transport attributes:	Number of transport-specific Endpoint attributes.
Transport-specific attributes:	Array of transport-specific Endpoint attributes. Each entry has the format of *DAT_NAMED_ATTR*, which is a structure with two elements. The first element is the name of the attribute, and the second is the value of the attribute as a string.
Num provider attributes:	Number of provider-specific Endpoint attributes.
Provider-specific attributes:	Array of provider-specific Endpoint attributes. Each entry has the format of *DAT_NAMED_ATTR*, which is a structure with two elements. The first element is the name of the attribute, and the second is the value of the attribute as a string.

Both soft and hard High Watermarks are attributes of an Endpoint and allow Consumer to control the behavior of the individual connection. When multiple connections take buffers from the same SRQ if one or several connections take all the buffers from SRQ, a connection that uses SRQ will be broken when a message arrives and there is no Recv buffer for it based on the definition of reliable connection.

The High Watermark allows Consumer to monitor and control the behavior of connections. When an Endpoint takes more Recv buffers from SRQ or EP RQ then is expected as specified by Soft High Watermark, Consumer will be notified by an event on IA asynchronous EVD. If an Endpoint takes more Recv buffers from SRQ or EP RQ than it is expected as specified by Hard High Watermark, EP connection will be broken and Consumer will be notified by an event on EP *connect_evd*. It is expected that Soft High

Watermark will be smaller than Hard High Watermark to give Consumer a chance to take action before the connection is broken.

If Consumer is not concern about Soft, Hard or both High Watermarks, the value of *DAT_WATERMARK_INFINITE* can be specified. This guarantees that no event will be generated. But the connection will be broken if a message arrives and no Recv buffer is available based on the definition of reliable connection.

6.6.6.1 USAGE

6.6.6.2 RATIONALE

When underlying RDMA transport and implementation supports multiple messages in progress over the same connection, all recv buffers for that connection can be consumed even without a single completion being generated for them. This applies to EP whether or not it is associated with SRQ.

For an Endpoint associated with SRQ *srq_soft_high_watermark* attribute allows Consumers to find out when the Endpoint has more buffers consumed without generating completions for them than the requested value for *srq_soft_high_watermark*. For transports that support multiple outstanding Sends in progress, such as iWARP, this information is critical to detect rogue connections that consume too many buffers, thus, depriving other connections sharing the SRQ of receive buffers. This lack of buffers may lead to connection break up. Consumer may not be able to remedy the situation since new Receive buffers cannot be posted since there is no room on SRQ. The buffers consumed by EPs may still be outstanding so there is no room on SRQ for new posting. However, completion for consumed SRQ receive buffers cannot be generated since the earlier posted Sends have not been received by an Endpoint that consumed too many buffers. The only real remedy Consumer has for this situation is to break the rogue connection. Notice that more than one connection may be in this situation.

For the EP that do not use SRQ the impact of too many arriving messages is not as great since the damage in the worst case will break only the rogue connection and will not impact other connections.

For an Endpoint associated with SRQ *Max_Recv_DTOs* allows Consumer to avoid control the above described situation by breaking a rogue connection that consumed more than *Max_Recv_DTOs* from SRQ.

6.6.6.3 MODEL IMPLICATION

Be aware that definitions of High Watermarks have some flexibility. It is clear that the high watermark checks are done when Recv buffer is taken from SRQ. However, it is left open when recv buffer is no longer counted against EP buffers. It can be counted when completion is generated for Recv buffer, when completion is queued on *recv_evd*, when completion is removed from *recv_evd*, or some time in between them.

Remote Consumer should not send more messages than it is suppose to. The number of posted Sends can be controlled via ULP flow control. The remote Consumer shall expect the same behavior regardless if local side uses SRQ or not. Thus, when more messages are sent than expected, remote Consumer shall be ready for the connection to be broken. But remote Consumer shall not rely on the connection to be broken when they exceed the limit.

For local Consumer it means that the connection will definitely break, but it may not break if the number of incoming messages is below the high hard watermark even with many completions on the Recv EVD.

6.6.7 ENDPOINT STATES

The list of Endpoint States and allowed EP operations is as follows:

Endpoint State	Description	Allowed Endpoint Operations
Unconnected	Endpoint is ready to be used for connection setup or Reserved Service Point.	dat_ep_connect, dat_ep_common_connect, dat_ep_dup_connect, dat_ep_free, dat_ep_reset, (and dat_cr_accept)
Unconfigured	Endpoint is created in configuration deferred state and is not ready to be used. It misses one EVDs or PZ.	dat_ep_free
Active Connection Pending	Endpoint is in use for Active side of connection establishment and a Connection Request was issued on it. No action is required by the Consumer to get the Endpoint out of this state.	dat_ep_disconnect, dat_ep_free
Reserved	Endpoint is associated with Reserved Service Point. No action is required by the Consumer to get the Endpoint out of this state.	
Unconfigured Reserved	Deferred configuration Endpoint is associated with Reserved Service Point. Endpoint is not ready to be used. It misses one EVD or PZ. Consumer can modify the Endpoint so it is configured or the Endpoint can transition into Passive Connection Pending upon Connection Request arrival.	
Passive Connection Pending	The Connection Request was received on the Endpoint-associated Reserved Service Point.	
Unconfigured Passive Connection Pending	The Connection request was received on the Unconfigured Reserved Endpoint. Deferred configuration Endpoint is not ready to be used. It misses one EVD or PZ. Consumer can modify the Endpoint so it is configured and ready to accept the Connection Request.	(dat_cr_reject, dat_cr_handoff)

Unconfigured Tentative Connection Pending	Provider-allocated Endpoint is associated with a received Connection Request on the Passive side of Connection Establishment. Endpoint is deferred configuration and is not ready to be used. It misses one EVD or PZ. Consumer can modify the Endpoint so it is configured and ready to accept the Connection Request.	(dat_cr_reject, dat_cr_handoff)
Tentative Connection Pending	Provider-allocated Endpoint is associated with a received Connection Request on the Passive side of Connection Establishment, that has been configured by the Consumer via *dat_ep_modify*.	dat_ep_modify, (dat_cr_reject, dat_cr_handoff, dat_cr_accept)
Completion Pending	Transport -dependent state on the Passive side of the connection establishment when the Consumer accepts a Connection Request on the Endpoint and the Provider is completing Transport-specific steps of Connection establishment. No action is required from the Consumer to get the Endpoint out of this state.	dat_ep_disconnect, dat_ep_free
Connected	Endpoint is Connected to a remote Endpoint and data can be transferred between them.	dat_ep_disconnect, dat_ep_free, dat_ep_post_send, dat_ep_post_rdma_read, dat_ep_post_rdma_write (and dat_rmr_bind)
Disconnect Pending	Endpoint was gracefully disconnected by the Consumer and is completing outstanding and in-progress posted DTOs and RMRs. No action is required by the Consumer to get the Endpoint out of this state.	dat_ep_disconnect, dat_ep_free
Disconnected	Endpoint is not associated with a remote Endpoint.	dat_ep_disconnect, dat_ep_reset, dat_ep_free, dat_ep_post_send, dat_ep_post_rdma_read, dat_ep_post_rdma_write (and dat_rmr_bind)

dat_ep_query, *dat_ep_get_status*, *dat_ep_modify*, and *dat_ep_post_recv* can be called on the Endpoint in any state.

6.6.7.1 USAGE

6.6.7.2 RATIONALE

Note to Provider: Upon connection establishment attempt failure for any reason, the Endpoint transitions into the *Disconnected* state and all preposted Recvs are flushed.

For IB, it is the semantic defined by the IBTA spec, as the *Disconnected* state is mapped to the Error state in IB. For iWARP, the Endpoint remains in the Unoperational state, but the Provider can transition the Endpoint itself into the iWARP QP Error state, which causes any preposted DTOs and RMRs to be flushed. For VI, the Endpoint transitions back into Unconnected state, but the Provider can call VipDisconnect, which causes preposted Recvs to be flushed. The *Disconnected* state is mapped to the Idle state of VI.

To support posting in a *Disconnected* state, the following strategy can be employed:

For IB, the post semantics supports posting in the Error state; these postings are flushed. For iWARP, the Provider can transition QP into Unconnected state, then post the requested DTO or RMR, and then transition the QP back into Error state. That causes the posted DTO or RMR to be flushed. If underlying QP is already in Unconnected state then post DTO and transition it into Error state. That will cause DTOs, RMRs to be flushed. Transiting QP back into Unconnected state completes the transformation and returns QP into starting state. For VI, Recv can be posted, followed by VipDisconnect. For any request posting, the Provider needs to generate flushed completions themselves without involving any posts.

The *dat_ep_reset* is mapped directly to IB and iWARP qp_modify that transition QP into Idle/Reset/Initialized state. For VI, the Provider internally changes its perception of the same Idle state. After reset, the Recv posting shall remain posted and other postings return with the immediate error of Invalid state.

6.6.7.3 MODEL IMPLICATIONS

When setting the Error sub-type for Invalid State, the state supplied is the current state of the Endpoint. The rationale is that the error happened because the application was confused which state it was in.

6.6.8 DAT_EP_FREE

Synopsis:

```
DAT_RETURN
   dat_ep_free (
   IN     DAT_EP_HANDLE     ep_handle
   )
```

Parameters:

ep_handle: Handle for an instance of the Endpoint.

Description: *dat_ep_free* destroys an instance of the Endpoint.

The Endpoint can be destroyed in any Endpoint state except *Reserved, Passive Connection Pending,* and *Tentative Connection Pending.* The destruction of the Endpoint can also cause the destruction of DTOs and RMRs posted to the Endpoint and not dequeued yet. This includes completions for all outstanding and in-progress DTOs/RMRs. The Consumer must be ready for all completions that are not dequeued yet either still being on the Endpoint *recv_evd_handle* and *request_evd_handle* or not being there.

The destruction of the Endpoint during connection setup aborts connection establishment.

If the Endpoint is in the *Reserved* state, the Consumer shall first destroy the associated Reserved Service Point that transitions the Endpoint into the *Unconnected* state where the Endpoint can be destroyed. If the Endpoint is in the *Passive Connection Pending* state, the Consumer shall first reject the associated Connection Request that transitions the Endpoint into the *Unconnected* state where the Endpoint can be destroyed. If the Endpoint is in the *Tentative Connection Pending* state, the Consumer shall reject the associated Connection Request that transitions the Endpoint back to Provider control, and the Endpoint is destroyed as far as the Consumer is concerned.

The freeing of an Endpoint also destroys an Event Stream for each of the associated Event Dispatchers.

The operation is synchronous and successful return of the operation indicated that the Endpoint is in *Unconnected* state. No connection event will be generated locally for the Endpoint on behalf of this operation.

It is illegal to use the destroyed handle in any subsequent operation.

dat_ep_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. The Endpoint is in *DAT_EP_STATE_RESERVED, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING*, or *DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING*.

6.6.8.1 USAGE

6.6.8.2 RATIONALE

6.6.8.3 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.6.9 DAT_EP_GET_STATUS

Synopsis:

```
DAT_RETURN
   dat_ep_get_status (
   IN     DAT_EP_HANDLE      ep_handle,
   OUT    DAT_EP_STATE       *ep_state,
   OUT    DAT_BOOLEAN        *recv_idle,
   OUT    DAT_BOOLEAN        *request_idle
   )
```

Parameters:

ep_handle:	Handle for an instance of the Endpoint.
ep_state:	Current state of the Endpoint.
recv_idle:	Status of the incoming DTOs on the Endpoint.
request_idle:	Status of the outgoing DTOs and RMR Bind operations on the Endpoint.

Description:

dat_ep_get_status provides the Consumer a quick snapshot of the Endpoint: The snapshot consists of the Endpoint state and whether there are outstanding or in-progress, incoming or outgoing DTOs. Incoming DTOs consist of Receives. Outgoing DTOs consist of the Requests, Send, RDMA Read, RDMA Write, and RMR Bind.

ep_state returns the value of the current state of the Endpoint *ep_handle*. State value is one of the following (as defined in Appendix A.4): DAT_EP_STATE_UNCONNECTED, DAT_EP_STATE_RESERVED, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, DAT_EP_STATE_ACTIVE_CONNECTION_PENDING, DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING, DAT_EP_STATE_CONNECTED, or DAT_EP_STATE_DISCONNECT_PENDING, DAT_EP_STATE_DISCONNECTED.

recv_idle value of DAT_TRUE specifies that there are no outstanding or in-progress Receive DTOs at the Endpoint, and DAT_FALSE otherwise. If SRQ is associated with the EP then the return value of *recv_idle* is undefined.

request_idle value of DAT_TRUE specifies that there are no outstanding or in-progress Send, RDMA Read, and RDMA Write DTOs, and RMR Binds at the Endpoint, and DAT_FALSE otherwise.

This call provides a snapshot of the Endpoint status only. No heroic synchronization with DTO queuing or processing is implied.

dat_ep_get_status is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.

6.6.9.1 USAGE

6.6.9.2 RATIONALE

6.6.9.3 MODEL IMPLICATIONS

6.6.10 DAT_EP_QUERY

Synopsis:

```
DAT_RETURN
   dat_ep_query (
   IN     DAT_EP_HANDLE          ep_handle,
   IN     DAT_EP_PARAM_MASK      ep_param_mask,
   OUT    DAT_EP_PARAM           *ep_param
   )
```

Parameters:

ep_handle:	Handle for an instance of the Endpoint.
ep_param_mask:	Mask for Endpoint parameters.
ep_param:	Pointer to a Consumer-allocated structure that the Provider fills with Endpoint parameters.

Description: *dat_ep_query* provides the Consumer parameters, including attributes and status, of the Endpoint. Consumers pass in a pointer to Consumer-allocated structures for Endpoint parameters that the Provider fills.

ep_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *ep_param_mask* requested parameters. The Provider can return values for any other parameters.

Some of the parameters only have values for certain Endpoint states. Specifically, the values for *remote_ia_address* and *remote_port_qual* are valid only for Endpoints in the DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, DAT_EP_STATE_ACTIVE_CONNECTION_PENDING, DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING, DAT_EP_STATE_DISCONNECT_PENDING, DAT_EP_STATE_COMPLETION_PENDING, or DAT_EP_STATE_CONNECTED states. The value of *local_port_qual* is valid only for Endpoints in the DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, DAT_EP_STATE_ACTIVE_CONNECTION_PENDING, DAT_EP_STATE_DISCONNECT_PENDING, DAT_EP_STATE_COMPLETION_PENDING, or DAT_EP_STATE_CONNECTED states, and may be valid for DAT_EP_STATE_UNCONNECTED, DAT_EP_STATE_RESERVED, DAT_EP_STATE_

TENTATIVE_CONNECTION_PENDING, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING, and DAT_EP_STATE_UNCONNECTED states.

If SRQ is not associated with EP then returned *srq_handle* is *DAT_HANDLE_NULL* and *srq_soft_hw* is *DAT_HW_DEFAULT*, if requested. If SRQ is associated with EP then returned *srq_soft_hw*, *max_recv_dtos* are as assigned and *max_recv_iov* is 0, if requested. If hard high watermark and/or soft hard watermark is not supported by Provider then the returned value for them is *DAT_HW_DEFAULT.*

dat_ep_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *ep_param_mask* is invalid.

6.6.10.1 USAGE

6.6.10.2 RATIONALE

6.6.10.3 MODEL IMPLICATIONS

6.6.11 DAT_EP_RECV_QUERY

Synopsis:

```
DAT_RETURN
   dat_ep_recv_query (
   IN     DAT_EP_HANDLE           ep_handle,
   OUT    DAT_COUNT               *nbufs_allocated,
   OUT    DAT_COUNT               *bufs_alloc_span
   )
```

Parameters:

ep_handle:	Handle for an instance of the EP.
nbufs_allocated:	The number of buffers at the EP for which completions have not been generated yet.
bufs_alloc_span:	The span of buffers that EP needs to complete arriving messages.

Description: *dat_ep_recv_query* provides to the Consumer a snapshot for Recv buffers on EP. The values for *nbufs_allocated* and *bufs_alloc_span* are not defined if the value of DAT_RETURN is not *DAT_SUCCESS.*

Provider may choose not to support *nbufs_allocated*, *bufs_alloc_span* or both. Check Provider attribute for EP Recv info support for it. When Provider does not support either of these counts then the return value the operation can be *DAT_MODEL_NOT_SUPPORTED*.

If *nbufs_allocated* is not NULL, then the count *nbufs_allocated* will return a snapshot count of the number of buffers allocated to *ep_handle* but not yet completed.

Once a buffer has been allocated to an EP it will be completed to the EP *recv_evd* if the EVD has not overflown. When an EP does not use SRQ a buffer is allocated as soon as it is posted to the EP. For an EP that uses SRQ a buffer is allocated to the EP when the EP removes it from SRQ.

If *bufs_alloc_span* is not NULL, then the count *bufs_alloc_span* will return the span of buffers allocated to the *ep_handle*. The span is the number of additional successful Recv completions that EP can generate if all the messages it is currently receiving complete successfully.

If a message sequence number is assigned to all received messages, the buffer span is the difference between the latest message sequence number of an allocated buffer minus the latest message sequence number for which completion has been generated. This sequence number only counts Send messages of remote Endpoint of the connection.

The Message Sequence Number (MSN) represents the order that send messages were submitted by the remote Consumer. The ordering of sends is intrinsic to the definition of a reliable service. Therefore every send message does have an MSN whether or not the native transport has a field with that name.

For both *nbufs_allocated* and *bufs_alloc_span* the Provider may return the reserved value DAT_VALUE_UNKNOWN if it cannot obtain the requested count at a reasonable cost.

dat_ep_recv_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.6.11.1 USAGE

6.6.11.2 RATIONALE

6.6.11.3 MODEL IMPLICATIONS

If Provider cannot support query for *nbufs_allocated* or *bufs_alloc_span* then the value returned for that attribute must be DAT_VALUE_UNKNOWN.

Note that for iWarp there already is a valid Message Sequence Number in the message header.

An implementation that processes incoming packets out of order, and which allocates from SRQs on an arrival basis, can have gaps in the MSNs associated with buffers allocated to an Endpoint.

For example: suppose Endpoint X has received buffer fragments for MSNs 19, 22 and 23. With arrival ordering the EP would have allocated three buffers from the SRQ for messages 19, 22 and 23. The number allocated would be 3, but the span would be 5.

The extra two represents the buffers that will have to be allocated for messages 20 and 21. They have not been allocated yet, but messages 22 and 23 will not be delivered until after messages 20 and 21 have not only had their buffers allocated but have also completed.

An implementation may choose to allocate 20 and 21 as soon as any higher buffer is allocated. If you presume that this is a valid connection this makes sense, because obviously 20 and 21 are in flight.

However, it creates a greater vulnerability to Denial Of Service attacks. There are also other implementation tradeoffs, which is why the Consumer should accept that different RNICs for iWARP will employ different strategies on when to do these allocations.

Each implementation will have some method of tracking the receive buffers already associated with an EP, and knowing which buffer matches which incoming message. However, those methods may vary.

In particular, there are valid implementations such as linked lists, where a count of the outstanding buffers is not instantly available. Such implementations would have to scan the allocated list to determine both the number of buffers and their span.

If such a scan is necessary, it is important that it be only a single scan. That is, the set of buffers that was counted must be the same set of buffers for which the span is reported.

The implementation should not scan twice, first to count the buffers and then again to determine their span. Not only it is inefficient, but it may produce inconsistent results if buffers were completed or arrived between the two scans.

Other implementations may simply maintain counts of these values in should be updated and referenced in atomically.

In either case the implementation must never report *n* buffers in a span that is less than *n*.

6.6.12 DAT_EP_MODIFY

Synopsis:

```
DAT_RETURN
   dat_ep_modify (
   IN     DAT_EP_HANDLE            ep_handle,
   IN     DAT_EP_PARAM_MASK        ep_param_mask,
   IN     DAT_EP_PARAM             *ep_param
   )
```

Parameters:

ep_handle: Handle for an instance of the Endpoint.

ep_param_mask: Mask for Endpoint parameters.

ep_param: Pointer to the Consumer-allocated structure that contains Consumer-requested Endpoint parameters.

Description: *dat_ep_modify* provides the Consumer a way to change parameters of an Endpoint.

ep_param_mask allows Consumers to specify which parameters to modify. Providers modify values for *ep_param_mask* requested parameters only.

Not all the parameters of the Endpoint can be modified. Some can be modified only when the Endpoint is in a specific state. Table 5 specifies which parameters can be changed and when.

Table 5 Modifiable Endpoint Parameters

Parameter/Attribute	States when modify allowed	Description
Interface Adapter	None	Endpoint belongs to an open instance of IA and that association cannot be changed.
Endpoint State	None	State of Endpoint cannot be changed by a *dat_ep_modify* operation.
Communicator	Unconnected, Unconfigured	The Communicator can be assigned only once. It can only be changed if previous value of pointer to Communicator is *NULL*.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Local IA Address	Unconnected, Unconfigured	Local IA can be modified for a Consumer controlled Endpoint prior to its use for the connection establishment. Platform rules are applicable to local IA Addresses. For example, use of the “privileged” ports.
Local Port Qualifier	None	Local port qualifier cannot be changed by a *dat_ep_modify* operation. It can be changed indirectly by modifying Local IA Address.
Remote IA Address	None	Remote IA Address cannot be changed by a *dat_ep_modify* operation.
Remote Port Qualifier	None	Remote port qualifier cannot be changed by a *dat_ep_modify* operation.
Protection Zone	Quiescent state, Unconnected, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Protection Zone can be changed only when the Endpoint is in quiescent state. The Endpoint states that are quiescent are *DAT_EP_STATE_UNCONNECTED* and *DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING*. Consumers should be aware that any Receive DTOs currently posted to the Endpoint that do not match the new Protection Zone fail with a *DAT_PROTECTION_VIOLATION* return.
In DTO Event Dispatcher	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Event Dispatcher for incoming DTOs (Receive) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.
Out DTO Event Dispatcher	Unconnected, Reserved, Passive Connection Pending, and Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Event Dispatcher for outgoing DTOs (Send, RDMA Read, and RDMA Write) can be changed only prior to a request for a connection for an Active side or prior to accepting a Connection Request for a Passive side.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Connection Event Dispatcher	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Event Dispatcher for the Endpoint Connection events can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Shared Receive Queue	None	SRQ cannot be changed.
Service Type	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Service Type can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Maximum Message Size	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum Message Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Maximum RDMA Size	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum RDMA Size can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Quality of Service	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	QoS can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Recv Completion Flags	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Recv Completion Flags specifies what DTO flags the Endpoint should support for Receive DTO operations. The value can be DAT_COMPLETION_NOTIFICATION_SUPPRESS_FLAG, DAT_COMPLETION_SOLICITED_WAIT_FLAG, or DAT_COMPLETION_EVD_THRESHOLD_FLAG. Recv posting does not support DAT_COMPLETION_SUPPRESS_FLAG or DAT_COMPLETION_BARRIER_FENCE_FLAG *dat_completion_flags* values that are only applicable to Request postings. Recv Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side, but before posting of any Recvs.
Request Completion Flags	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Request Completion Flags specifies what DTO flags the Endpoint should support for Send, RDMA Read, RDMA Write, and RMR Bind operations. The value can be: DAT_COMPLETION_UNSIGNALLED_FLAG or DAT_COMPLETION_EVD_THRESHOLD_FLAG. Request postings always support DAT_COMPLETION_SUPPRESS_FLAG, DAT_COMPLETION_SOLICITED_WAIT_FLAG, or DAT_COMPLETION_BARRIER_FENCE_FLAG *completion_flags* values. Request Completion Flags can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Maximum Recv DTO	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum Recv DTO specifies the maximum number of outstanding Consumer-submitted Receive DTOs that a Consumer expects at any time at the Endpoint. Maximum Recv DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side. If SRQ is associated with EP then Maximum Recv DTO represents Hard High Watermark and it cannot be modified by *dat_ep_modify*. Consumers should use *dat_ep_set_watermark* operation instead. An attempt to modify this attribute for EP with SRQ will result in *DAT_INVALID_PARAMETER.*
Maximum Request DTO	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum Request DTO specifies the maximum number of outstanding Consumer-submitted send and RDMA DTOs and RMR Binds that a Consumer expects at any time at the Endpoint. Maximum Out DTO can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Maximum Recv IOV	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum Recv IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Receive DTO for the Endpoint. Maximum Recv IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Maximum Request IOV	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum Request IOV specifies the maximum number of elements in IOV that a Consumer specifies for posting a Send, RDMA Read, or RDMA Write DTO for the Endpoint. Maximum Request IOV can be changed only prior to a request for a connection for an Active side or accepting a Connection Request for a Passive side.
Maximum outstanding RDMA Read as target	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum number of outstanding RDMA Reads for which the Endpoint is the target.
Maximum outstanding RDMA Read as originator	Unconnected, Reserved, Passive Connection Pending, Tentative Connection Pending, Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Maximum number of outstanding RDMA Reads for which the Endpoint is the originator.
Soft High Watermark	All	If SRQ is not associated with EP then Soft Hard Watermark cannot be changed. Soft High Watermark cannot be modified by *dat_ep_modify.* Consumers should use *dat_ep_set_watermark* operation instead. An attempt to modify this attribute will result in *DAT_INVALID_PARAMETER.*
Num transport-specific attributes	Quiescent state (unconnected), Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Number of transport-specific attributes to be modified.
Transport-specific endpoint attributes	Quiescent state (unconnected), Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Transport-specific attributes can be modified only in the transport-defined Endpoint state. The only guaranteed safe state in which to modify transport-specific Endpoint attributes is the quiescent state *DAT_EP_STATE_UNCONNECTED*.

Table 5 Modifiable Endpoint Parameters (Continued)

Parameter/Attribute	States when modify allowed	Description
Num provider-specific attributes	Quiescent state (unconnected), Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Number of Provider-specific attributes to be modified.
Provider-specific endpoint attributes	Quiescent state (unconnected), Unconfigured, Unconfigured Reserved, Unconfigured Passive Connection Pending, and Unconfigured Tentative Connection Pending	Provider-specific attributes can be modified only in the Provider-defined Endpoint state. The only guaranteed safe state in which to modify Provider-specific Endpoint attributes is the quiescent state *DAT_EP_STATE_UNCONNECTED*.

Endpoints without SRQ *max_recv_dtos, max_request_dtos, max_recv_iov,* and *max_request_iov* will have at least the Consumer requested values but may have larger values. Endpoints with SRQ *max_request_dtos* and *max_request_iov* will have at least the Consumer requested values but may have larger values. Consumer can query the Endpoint to find out the actual values for these attributes. The Endpoint has the exact Consumer requested values for *max_message_size, max_rdma_size, max_rdma_read_in,* and *max_rdma_read_out.* For all other attributes the created Endpoint has the exact values requested by Consumer. If Provider cannot satisfy the Consumer requested attribute values the operation fails.

dat_ep_modify is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter; *ep_param_mask* is invalid, or one of the requested Endpoint parameters or attributes was invalid, not supported, or cannot be modified.
DAT_INVALID_STATE	Parameter in an invalid state. The Endpoint was not in the state that allows one of the parameters or attributes to be modified.

6.6.12.1 USAGE

Note to Provider: Upon connection establishment attempt failure for any reason, the Endpoint transitions into the *Disconnected* state and all preposted Recvs are flushed with one exception: Recv DTOs posted to SRQ associated with EP but not yet allocated to an EP remain unaffected.

For IB, it is the semantic defined by the IBTA spec, as the *Disconnected* state is mapped to the Error state in IB. For iWARP, the Endpoint remains in the Unoperational state, but the Provider can transition the Endpoint itself into the iWARP QP Error state. This causes any preposted DTOs and RMRs to be flushed, except Recv DTOs posted to SRQ associated with EP but not yet allocated to an EP, which remain unaffected. If underlying QP is already in Unconnected state then post DTO and transition QP into the Error state. This causes DTOs and RMRs to be flushed. Transiting QP back into Unconnected state completes the transformation and returns QP into starting state. If underlying QP is in Error state then posting to it causes DTOs and RMRs to be flushed automatically. For VI, the Endpoint transitions back into Unconnected state, but the Provider can call VipDisconnect, which causes preposted Recvs to be flushed. The *Disconnected* state is mapped to the Unconnected state of VI.

To support posting in a *Disconnected* state, the following strategy can be employed:

For IB, the post semantics supports posting in the Error state; these postings are flushed except Recv DTOs posted to SRQ associated with EP but not yet allocated to an EP which remain unaffected. For iWARP, the Provider can transition QP into Unconnected state, then post the requested DTO or RMR, and then transition the QP back into Error state. If underlying QP is in Error state then posting to it causes DTOs and RMRs to be flushed automatically. That causes the posted DTO or RMR to be flushed. For VI, Recv can be posted, followed by VipDisconnect. For any request posting, the Provider needs to generate flushed completions themselves without involving any posts.

The *dat_ep_reset* is mapped directly to IB and iWARP qp_modify that transition QP into Idle/Reset/Initialized state. For VI, the Provider internally changes its perception of the same Unconnected state. After reset, the Recv posting shall remain posted and other postings return with the immediate error of Invalid state. Note that for SRQ Recv buffers are not effected by *dat_ep_reset* and they remain on SRQ.

For the common connection model Consumer should define Communicator and local IA_Address for the Common Service Point and to an Endpoint prior to requsting its connection. This follows the platform convension for the socket IP Addresses, domains, types and protocols.

6.6.12.2 RATIONALE

6.6.12.3 MODEL IMPLICATIONS

It is up to the Provider to ensure that Consumer specified *comm* and IA Address are valid and that IA can support them. For example, Provider can restrict what IP Addresses can be used for IA but belonging to an IA range defined by an administrator.

6.6.13 DAT_EP_SET_WATERMARK

Synopsis:

```
DAT_RETURN
   dat_ep_set_watermark (
   IN     DAT_EP_HANDLE           ep_handle,
   IN     DAT_COUNT               soft_high_watermark,
   IN     DAT_COUNT               hard_high_watermark

   )
```

Parameters:

ep_handle:	Handle for an instance of an Endpoint.
soft_high_ watermark	The soft high watermark for the number of Recv buffers consumed by the Endpoint.
hard_high_ watermark	The hard high watermark for the number of Recv buffers consumed by the Endpoint.

Description:

dat_ep_set_watermark sets the soft and hard high watermark values for EP and arms EP for generating asynchronous events for high watermarks. An asynchronous event will be generated for IA *async_evd* when the number of Recv buffers at EP is above the soft high watermark for the first time. A connection broken event will be generated for EP *connect_evd* when this may happen during the call or when EP takes a buffer from the SRQ or EP RQ. The soft and hard high watermark asynchronous event generation and setting are independent from each other.

The asynchronous event for soft high watermark will be generated only once per setting. Once an event is generated no new asynchronous events for the soft high watermark will generated again until the EP is set for the soft high watermark again. If Consumer wants to generate the event again Consumer should set the soft high watermark again.

If Consumer is not interested in soft or hard high watermark the value of *DAT_WATERMARK_INFINITE* can be specified for that case which is the default value. It specifies that no asynchronous event will be generated for high watermark EP attribute for which this value is set. It does not prevent generation of connection broken events for EP when no Recv buffer is available for a message arrived on the EP connection.

The operation is supported for all states of Endpoint.

dat_ep_set_watermark is synchronous. Its thread safety is Provider dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. Provider does not support EP Soft or Hard High Watermarks.

6.6.13.1 USAGE

For hard high watermark the Provider is ready to generate a connection broken event as soon as the connection is established.

If the asynchronous event for soft or hard high watermark has not been generated yet then this call modifies the values for these attributes. The Provider remains “armed” for a generation of these asynchronous event.

6.6.13.2 RATIONALE

6.6.13.3 MODEL IMPLICATIONS

Regardless of whether or not an asynchronous event for the soft and hard high watermark has been generated this operation will set the generation of an asynchronous event with the Consumer-provided high watermark values. If the new high watermark values are below the current number of Receive DTOs at EP then an asynchronous event will be generated immediately. Otherwise the old soft or hard or both high watermark values are simply replaced with the new ones.

6.6.14 DAT_EP_CONNECT

Synopsis:

```
DAT_RETURN
   dat_ep_connect (
   IN     DAT_EP_HANDLE          ep_handle,
   IN     DAT_IA_ADDRESS_PTR     remote_ia_address,
   IN     DAT_CONN_QUAL          remote_conn_qual,
   IN     DAT_TIMEOUT            timeout,
   IN     DAT_COUNT              private_data_size,
   IN const DAT_PVOID            private_data,
```

```
IN    DAT_QOS              qos,
IN    DAT_CONNECT_FLAGS    connect_flags
)
```

Parameters:

ep_handle: Handle for an instance of an Endpoint.

remote_ia_address: The Address of the remote IA to which an Endpoint is requesting a connection.

remote_conn_qual: Connection Qualifier of the remote IA from which an Endpoint requests a connection.

timeout: Duration of time, in microseconds, that a Consumer waits for Connection establishment. The value of *DAT_TIMEOUT_INFINITE* represents no timeout, indefinite wait. Values must be positive.

private_data_size: Size of the *private_data.* Must be nonnegative.

private_data: Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be NULL.

qos: Requested quality of service of the connection.

connect_flags: Flags for the requested connection. The default value is *DAT_CONNECT_DEFAULT_FLAG*, which is 0. See Table 6 for flag definitions.

Table 6 Connection Request Flag Definitions

Features	**Definition/Bit**	**Value**	**Description**
MultiPathing	DAT_MULTIPATH_FLAG least significant	0	Consumer does not request multipathing.
		1	Consumer requests multipathing.
		2	Consumer requires multipathing.

Description: *dat_ep_connect* requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the Connection establishment model. The remote Endpoint is identified by Remote IA and Remote Connection Qualifier.

As part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the *DAT_CONNECTION_REQUEST_EVENT.*

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into *DAT_EP_STATE_ACTIVE_CONNECTION_PENDING*.

Consumers can request a specific value of *qos*. The Provider specifies which quality of service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos,* the operation fails and *DAT_MODEL_NOT_SUPPORTED* is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into the *DAT_EP_STATE_DISCONNECTED* state, the connection is not established, and the event returned on the *connect_evd_handle* is *DAT_CONNECTION_EVENT_NON_PEER_REJECTED*. The same *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* event is returned if the connection cannot be established for all reasons of not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into *DAT_EP_STATE_DISCONNECTED* state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a *DAT_EP_STATE_CONNECTED* state.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_PEER_REJECTED* event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer requested *Timeout*, a *DAT_CONNECTION*_EVENT_UNREACHABLE event is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a *DAT_EP_STATE_DISCONNECTED* state.

If the Provider can locally determine that the *remote_ia_address* is invalid, or that the *remote_ia_address* cannot be converted to a Transport-specific address, the operation can fail synchronously with a *DAT_INVALID_ADDRESS* return.

The local Endpoint is automatically transitioned into a *DAT_EP_STATE_CONNECTED* state when a Connection Request accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state and the local Consumer through a *DAT_CONNECTION_EVENT_TIMED_OUT* event on the *connect_evd_handle* of the local Endpoint. The *timeout* of 0 is invalid since connection cannot be established instantaneously. If *timeout=0* is specified the *DAT_INVALID_PARAMETER* is returned.

If the local Endpoint does not have a Protection Zone defined or one of its EVDs is not defined then the operation fails with *DAT_INVALID_STATE* return.

connect_flags allows Consumer to specify multipathing information for the connection. Consumer can request no multipathing, which is the default value. It can require multipathing, which means that connection should not be established if only a single path is available. Or multipathing can be requested, which means that multipathed connection can be established even if only a single path is available now.

dat_ep_connect is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_ADDRESS	Invalid address.
DAT_INVALID_HANDLE	Invalid DAT handle; Invalid Endpoint handle.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in *DAT_EP_STATE_UNCONNECTED* state.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. For example, the requested *qos* was not supported by the local Provider.

6.6.14.1 USAGE

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, Connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The Provider is allowed to change these default values during connection setup.

6.6.14.2 RATIONALE

uDAPL does not provide an API parameter for a Consumer to bind the local Endpoint to a specific Port Qualifier. VI do not provide this capability and uDAPL Consumers do not have such functionality as a requirement.

6.6.14.3 MODEL IMPLICATIONS

Note to Provider: If the Provider can locally determine that the *remote_ia_address* is invalid, or that the *remote_ia_address* cannot be converted to a Transport-specific address, *DAT_INVALID_ADDRESS* should be returned.

Note to Provider: The *DAT_CONNECTION_EVENT_UNREACHABLE* event is returned asynchronously by the Provider if it does not have RDMA Transport connectivity to the remote host specified by *remote_ia_address* but cannot determine that locally. Inability to convert the *remote_ia_address* into a Transport-specific address should also result in the same event return. For example, if remote IB SA indicates that there are no paths between the local IA and the remote IA, this scenario is mapped to this error. The remote side not responding to the request within the Consumer-specified *Timeout* is also mapped to the same event. In contrast, if the remote side, but not the remote Consumer, is responding, either a *reject* or a Message Receipt Acknowledgement is mapped to *DAT_CONNECTION_EVENT_NON_PEER_REJECT.*

The Provider is not allowed to fail connection establishment because of insufficient resources to support the Provider-chosen outstanding RDMA Read default attributes for the Endpoint.

DAT Providers are required not to change any Consumer-specified Endpoint attributes. If the Consumer does not specify outstanding RDMA Read incoming or outgoing attributes, Providers can change them. It is recommended that the Provider set the outstanding RDMA Read attributes to 0 if the Consumer has not specified them, to ensure that a connection establishment does fail due to insufficient local or remote resources to satisfy local or remote Provider-chosen values for the outstanding RDMA Read incoming and outgoing for the Endpoint.

If Consumer specified more private data than local Provider supports the operations fails synchronously with DAT_INVALID_PARAMETER. If local Provider support the amount of private data but remote Provider cannot the remote Provider will pass the truncated private data to the Consumer and set the *truncate_flag* in the Connection Request Arrival event.

For the IB transport, Provider shall zero out transport specific private data fields beyond the Consumer provided private data. This ensures that remote Provider can detect the extra private data beyond what it can support.

For iWARP Providers that support IETF MPA both the size of the private data and the private data shall be mapped into MPA Request frame.

If multipathing was requested and the connection was established in the degraded mode, the HA event stream will deliver an event when more than one path becomes available under the connection.

6.6.15 DAT_EP_COMMON_CONNECT

Synopsis:

```
DAT_RETURN
   dat_ep_common_connect (
   IN     DAT_EP_HANDLE              ep_handle,
   IN     DAT_IA_ADDRESS_PTR         remote_ia_address,
   IN     DAT_TIMEOUT                timeout,
   IN     DAT_COUNT                  private_data_size,
   IN const DAT_PVOID                private_data
   )
```

Parameters:

ep_handle:	Handle for an instance of an Endpoint.
remote_ia_address:	IA_address of the remote endpoint of the requested connection.
timeout:	Duration of time, in microseconds, that a Consumer waits for Connection establishment. The value of *DAT_TIMEOUT_INFINITE* represents no timeout, indefinite wait. Values must be positive.
private_data_size:	Size of the *private_data*. Must be nonnegative.
private_data:	Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If *private_data_size* is zero, then *private_data* can be NULL.

Description: *dat_ep_common_connect* requests that a connection be established between the local Endpoint and a remote Endpoint specified by the *remote_ia_address*. This operation is used by the active/client side Consumer of the Connection establishment model.

EP must be properly configured for this operation. The EP Communicator must be specified. As part of the successful completion of this operation, the local Endpoint is bound to a local IA Address if it had these assigned before.

The local IP Address, port and protocol are passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the *DAT_CONNECTION_REQUEST_EVENT*.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol.

Upon successful completion of this operation, the local Endpoint is transferred into *DAT_EP_STATE_ACTIVE_CONNECTION_PENDING*.

The *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* event is returned if the connection cannot be established for all reasons of not establishing the connection, except timeout, remote host not reachable, and remote peer reject. In this case, the local Endpoint is automatically transitioned into *DAT_EP_STATE_DISCONNECTED* state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a *DAT_EP_STATE_CONNECTED* state.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_PEER_REJECTED* event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer requested *Timeout*, a *DAT_CONNECTION*_EVENT_UNREACHABLE event is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a *DAT_EP_STATE_DISCONNECTED* state.

The local Endpoint is automatically transitioned into a *DAT_EP_STATE_CONNECTED* state when a Connection Request accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state and the local Consumer through a *DAT_CONNECTION_EVENT_TIMED_OUT* event on the *connect_evd_handle* of the local Endpoint. The *timeout* of 0 is invalid since connection cannot be established instantaneously.

If the local Endpoint does not have a Protection Zone defined or one of its EVDs is not defined then the operation fails with *DAT_INVALID_STATE* return.

dat_ep_common_connect is synchronous. Its thread safety is Provider-dependent. The UpCall safety of the operation is not guaranteed.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	Invalid DAT handle; Invalid Endpoint handle.
DAT_INVALID_STATE	Parameter in an invalid state. For example, endpoint was not in *DAT_EP_STATE_UNCONNECTED* state, or the EP Communicator is not defined.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.6.15.1 USAGE

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, Connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any value for default. The

Provider is allowed to change these default values during connection setup.

6.6.15.2 RATIONALE

The common model allows Consumer to use a well-known socket connection model regardless of the underlying RDMA transport. Thus, IP port can be used instead of transport-dependent Connection Qualifier and connection is bounded to a specific protocol of a protocol family.

6.6.15.3 MODEL IMPLICATIONS

Note to Provider: The common model allows iWARP Providers to create sockets ahead of time if Consumer sets *comm* and IA Address of the EP prior to connection establishment. Otherwise, an iWARP Provider will create a socket, bind it and then connect. An IA defaults should be used for unspecified local EP *ia_address.* The RDMA connection request must be mapped into MPA request frame over the socket connection.

For an IB Provider the RDMA IP CM Annex specification shall be used to support common model. So, the connection request should be sent to the Connection Qualifier defined by *comm* and *remote_ia_address* as specified by the IBTA RDMA IP CM Annex.

Provider can assign the default IA Address of the IA and some port to the EP if Consumer does not assigned its local IA Address. Or Provider may require that the local IA Addres is assigned by the Consumer.

It is up to the Provider to ensure that Consumer specified *comm* and IA Address are valid and that IA can support them. For example, Provider can restrict what IP Addresses can be used for IA but belonging to an IA range defined by an administrator. This is typically done in *dat_ep_modify.*

Note to Provider: The remote side not responding to the request within the Consumer-specified *Timeout* is mapped to the *DAT_CONNECTION_EVENT_TIMEOUT_EXPIRED*. In contrast, if the remote side, but not the remote Consumer, is responding, either a *reject* or a Message Receipt Acknowledgement is mapped to *DAT_CONNECTION_EVENT_NON_PEER_REJECT.*

The Provider is not allowed to fail connection establishment because of insufficient resources to support the Provider-chosen outstanding RDMA Read default attributes for the Endpoint.

DAT Providers are required not to change any Consumer-specified Endpoint attributes. If the Consumer does not specify outstanding RDMA Read incoming or outgoing attributes, Providers can change them. It is recommended that the Provider set the outstanding RDMA Read attributes to 0 if the Consumer has not specified them, to ensure that a connection establishment does fail due to insufficient local or remote resources to satisfy local or remote Provider-chosen values for the outstanding RDMA Read incoming and outgoing for the Endpoint.

If Consumer specified more private data than local Provider supports the operations fails synchronously with DAT_INVALID_PARAMETER. If local Provider support the amount of private data but remote Provider cannot the remote Provider will pass the truncated private data to the Consumer and set the *truncate_flag* in the Connection Request Arrival event.

For the IB transport, Provider shall zero out transport specific private data fields beyond the Consumer provided private data. This ensures that remote Provider can detect the extra private data beyond what it can support.

For iWARP/TCP Providers that support IETF MPA both the size of the private data and the private data shall be mapped into MPA Request frame.

Neither the Provider nor RNIC can change the routing of the TCP connection for socket connection provided by Consumer.

For iWARP/SCTP Providers the size of the private data and the private data shall be mapped into a DDP Session Initiate message.

We should also note that the iWARP Provider should release the socket back to the system promptly after the connection is broken. It may do so earlier if the socket resource is not tied to the TCP connection itself.

When a socket is converted for use by an EP the existing TCP options should be preserved, except when they are contrary to normal RDMA operations. For example, iWARP implementations should disable Nagle. But options such as KEEPALIVE should be used as is whenever reasonably possible.

6.6.16 DAT_EP_DUP_CONNECT

Synopsis:

```
DAT_RETURN
   dat_ep_dup_connect (
   IN     DAT_EP_HANDLE           ep_handle,
   IN     DAT_EP_HANDLE           dup_ep_handle,
   IN     DAT_TIMEOUT             timeout,
   IN     DAT_COUNT               private_data_size,
   IN const DAT_PVOID             private_data,
   IN     DAT_QOS                 qos
   )
```

Parameters:

ep_handle: Handle for an instance of an Endpoint.

dup_ep_handle: Connected local Endpoint that specifies a requested connection remote end.

timeout:	Duration of time, in microseconds, that Consumers wait for Connection establishment. The value of DAT_TIMEOUT_INFINITE represents no timeout, indefinite wait. Values must be positive.
private_data_size:	Size of *private_data*. Must be nonnegative.
private_data:	Pointer to the private data that should be provided to the remote Consumer as part of the Connection Request. If private_data_size is zero, then private_data can be NULL.
qos:	Requested Quality of Service of the connection.

Description: *dat_ep_dup_connect* requests that a connection be established between the local Endpoint and a remote Endpoint. This operation is used by the active/client side Consumer of the connection model. The remote Endpoint is identified by the *dup_ep_handle*. The remote end of the requested connection shall be the same as the remote end of the *dup_ep_handle*. This is equivalent to requesting a connection to the same remote IA, Connection Qualifier, and *connect_flags* as used for establishing the connection on duplicated Endpoints and following the same redirections. *dup_ep_handle* and *ep_handle* should be of the same DAT_Provider.

As part of the successful completion of this operation, the local Endpoint is bound to a Port Qualifier of the local IA. The Port Qualifier is passed to the remote side of the requested connection and is available to the remote Consumer in the Connection Request of the *DAT_CONNECTION_REQUEST_EVENT*.

The Consumer-provided *private_data* is passed to the remote side and is provided to the remote Consumer in the Connection Request. Consumers can encapsulate any local Endpoint attributes that remote Consumers need to know as part of an upper-level protocol. Providers can also provide a Provider on the remote side any local Endpoint attributes and Transport-specific information needed for Connection establishment by the Transport.

Upon successful completion of this operation, the local Endpoint is transferred into *DAT_EP_STATE_ACTIVE_CONNECTION_PENDING*.

Consumers can request a specific value of *qos*. The Provider specifies which Quality of Service it supports in documentation and in the Provider attributes. If the local Provider or Transport does not support the requested *qos*, the operation fails and *DAT_MODEL_NOT_SUPPORTED* is returned synchronously. If the remote Provider does not support the requested *qos*, the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state, the connection is not established, and the event returned on the *connect_evd_handle* is *DAT_CONNECTION_EVENT_NON_PEER_REJECTED*. The same *DAT_CONNECTION_EVENT_NON_PEER_REJECTED* event is returned if connection cannot be established for all reasons for not establishing the connection, except timeout, remote host not reachable, and remote peer reject. For example, remote Consumer is not listening on the requested Connection Qualifier, Backlog of the requested Service Point is full, and Transport errors. In this case, the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state.

The acceptance of the requested connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint.

The rejection of the connection by the remote Consumer is reported to the local Consumer through a *DAT_CONNECTION_EVENT_PEER_REJECTED* event on the *connect_evd_handle* of the local Endpoint and the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state.

When the Provider cannot reach the remote host or the remote host does not respond within the Consumer-requested *Timeout*, a *DAT_CONNECTION_EVENT_UNREACHABLE* is generated on the *connect_evd_handle* of the Endpoint. The Endpoint transitions into a *DAT_EP_STATE_DISCONNECTED* state.

The local Endpoint is automatically transitioned into a *DAT_EP_STATE_CONNECTED* state when a Connection Request is accepted by the remote Consumer and the Provider completes the Transport-specific Connection establishment. The local Consumer is notified of the established connection through a *DAT_CONNECTION_EVENT_ESTABLISHED* event on the *connect_evd_handle* of the local Endpoint.

When the *timeout* expired prior to completion of the Connection establishment, the local Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state and the local Consumer through a *DAT_CONNECTION_EVENT_TIMED_OUT* event on the *connect_evd_handle* of the local Endpoint. The *timeout* of 0 is invalid since connection cannot be established instantaneously. If *timeout=0* is specified the *DAT_INVALID_PARAMETER* is returned.

dat_ep_dup_connect is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* or *dup_ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state.

DAT_MODEL_NOT_SUPPORTED	The requested Model is not supported by the Provider. For example, requested *qos* was not supported by the local Provider.

6.6.16.1 USAGE

It is up to the Consumer to negotiate outstanding RDMA Read incoming and outgoing with a remote peer. The outstanding RDMA Read outgoing attribute should be smaller than the remote Endpoint outstanding RDMA Read incoming attribute. If this is not the case, connection establishment might fail.

DAT API does not define a protocol on how remote peers exchange Endpoint attributes. The exchange of outstanding RDMA Read incoming and outgoing attributes of EPs is left to the Consumer ULP. The Consumer can use Private Data for it.

If the Consumer does not care about posting RDMA Read operations or remote RDMA Read operations on the connection, it can set the two outstanding RDMA Read attribute values to 0.

If the Consumer does not set the two outstanding RDMA Read attributes of the Endpoint, the Provider is free to pick up any values as a default. The Provider is allowed to change these default values during connection setup.

6.6.16.2 RATIONALE

6.6.16.3 MODEL IMPLICATIONS

The Provider is not allowed to fail connection establishment because of insufficient resources to support the Provider-chosen outstanding RDMA Read default attributes for the Endpoint.

DAT Providers are required not to change any Consumer-specified Endpoint attributes. If the Consumer does not specify outstanding RDMA Read incoming or outgoing attributes, Providers can change them. It is recommended that the Provider set these outstanding RDMA Read attributes to 0 if the Consumer has not specified them, to ensure that the connection establishment does not fail due to insufficient local or remote resources to satisfy the local or remote Provider-chosen values for the outstanding RDMA Read incoming and outgoing for the Endpoint.

If Consumer specified more private data than local Provider supports the operations fails synchronously with DAT_INVALID_PARAMETER. If local Provider support the amount of private data but remote Provider cannot the remote Provider will pass the truncated private data to the Consumer and set the *truncate_flag* in the Connection Request Arrival event.

For the IB transport, Provider shall zero out transport specific private data fields beyond the Consumer provided private data. This ensures that remote Provider can detect the extra private data beyond what it can support.

For iWARP Providers that support IETF MPA both the size of the private data and the private data shall be mapped into MPA Request frame.

6.6.17 DAT_EP_DISCONNECT

Synopsis:

```
DAT_RETURN
    dat_ep_disconnect (
    IN     DAT_EP_HANDLE      ep_handle,
    IN     DAT_CLOSE_FLAGS    disconnect_flags
    )
```

Parameters:

ep_handle: Handle for an instance of Endpoint.

disconnect_flags: Flags for disconnect. Default value of *DAT_CLOSE_DEFAULT = DAT_CLOSE_ABRUPT _FLAG* represents abrupt disconnect. See Table 7 for flag definitions.

Description:

dat_ep_disconnect requests a termination of a connection or connection establishment. This operation is used by the active/client or a passive/server side Consumer of the connection model.

disconnect_flags allows Consumers to specify whether they want graceful or abrupt disconnect. Upon disconnect, all outstanding and in-progress DTOs and RMR Binds must be completed.

Table 7 EP Disconnect Flag Definitions

Features	Definition
Abrupt close	DAT_CLOSE_ABRUPT_FLAG
Graceful close	DAT_CLOSE_GRACEFUL_FLAG

For abrupt disconnect, all outstanding DTOs and RMR Binds are completed unsuccessfully, and in-progress DTOs and RMR Binds can be completed successfully or unsuccessfully. If an in-progress DTO is completed unsuccessfully, all follow on in-progress DTOs in the same direction also must be completed unsuccessfully. This order is presented to the Consumer through a DTO completion Event Stream of the *recv_evd_handle and request_evd_handle* of the Endpoint.

For graceful disconnect, all outstanding and in-progress request DTOs and RMR Binds must try to be completed successfully first, before disconnect proceeds. During that time, the local Endpoint is in a *DAT_EP_DISCONNECT_PENDING* state.

The Consumer can call abrupt *dat_ep_disconnect* when the local Endpoint is in the *DAT_EP_DISCONNECT_PENDING* state. This causes the Endpoint to transition into *DAT_EP_STATE_DISCONNECTED* without waiting for outstanding and in-progress request DTOs and RMR Binds to successfully complete. The graceful *dat_ep_disconnect* call when the local Endpoint is in the *DAT_EP_DISCONNECT_PENDING* state has no effect.

If the Endpoint is not in *DAT_EP_STATE_CONNECTED*, the semantic of the operation is the same for *graceful* or *abrupt disconnect_flags* value.

No new Send, RDMA Read, and RDMA Write DTOs, or RMR Binds can be posted to the Endpoint when the local Endpoint is in the *DAT_EP_DISCONNECT_PENDING* state.

The successful completion of the disconnect is reported to the Consumer through a *DAT_CONNECTION_EVENT_DISCONNECTED* event on *connect_evd_handle* of the Endpoint. The Endpoint is automatically transitioned into a *DAT_EP_STATE_DISCONNECTED* state upon successful asynchronous completion. If the same EVD is used for *connect_evd_handle* and any *recv_evd_handle* and *request_evd_handle*, all successful Completion events of in-progress DTOs shall precede the Disconnect Completion event.

Disconnecting a *Disconnected* Endpoint is no-op. Disconnecting an Endpoint in *DAT_EP_STATE_UNCONNECTED, DAT_EP_STATE_RESERVED, DAT_EP_STATE_PASSIVE_CONNECTION_PENDING,* and *DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING* is disallowed.

Both abrupt and graceful disconnect of the Endpoint during connection establishment, *DAT_EP_STATE_ACTIVE_CONNECTION_PENDING,* and *DAT_EP_STATE_COMPLETION_PENDING* “aborts” the connection establishment and transitions the local Endpoint into *DAT_EP_STATE_DISCONNECTED.* That causes preposted Recv DTOs to be flushed to *recv_evd_handle* except recv buffers posted to SRQ associated with EP that remains on SRQ unless they have been dequeue by the EP prior to the disconnect.

dat_ep_disconnect is asynchronous. The operation return does not indicate that the Endpoint is disconnected. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.

DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter; *disconnect_flags* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint is not in the valid state for disconnect.

6.6.17.1 USAGE

6.6.17.2 RATIONALE

6.6.17.3 MODEL IMPLICATIONS

The behavior of posting a DTO to an Endpoint that is being disconnected (*dat_ep_disconnect* is in progress and was not yet returned) is not defined. Implementation does not require any locks to ensure that other threads cannot post DTOs while a Disconnect is in progress.

After Disconnect is returned, the Endpoint is in *DAT_EP_STATE_ DISCONNECTED* and posting of Send, Recv, RDMA Read, RDMA Write DTOs, or RMR Bind to the Endpoint results in the “flushing” of the posted DTO or RMR to *recv_evd_handle or request_evd_handle*, regardless of which thread is posting it except recv buffers posted to SRQ associated with EP that remains on SRQ unless they have been dequeue by the EP prior to disconnect or an error.

Note to Provider: For IB Providers, here is a way to support *dat_ep_ disconnect* (and *dat_ep_free)* in various Endpoint underlying QP/CM states. Notice that this way is safe; it does not rely on any protocol beyond the one defined in chapter 12 of the IBTA spec. It does rely on local CM transitioning QP into an Error state at any time. This is consistent with what the remote CM sees if the local QP is destroyed at any time that is allowed by the IBTA spec.

Active side:

- CM: REQ Send state, QP is in Init (EP is in Active Connection Pending State).
  - When *dat_ep_disconnect* is issued, CM transitions the local QP into an Error state (EP in a Disconnected state). Now wait for CM timeout. If REP is received before timeout expired, send REJ with code 4 (timeout), done.
  - If REJ is received, done.
  - If MRA received expired, send REJ with code 4 (timeout); done.
- CM: REP Wait state. The same actions as above.
- CM: REP Rcvd state, QP is in RTR (EP is still in Active Connection Pending State).

    - Consumer called *dat_ep_disconnect* when the Provider was generating Connection Established event and has not sent Accept to the remote side yet. Either way of handling this race is fine. The Provider must either generate REJ with code 4 (timeout) or REP followed by a DREQ message. In the second case, two local events, Connection Established followed by Disconnect, are generated. In the first case, only a Disconnect event is generated.
- CM: MRA(REP) Sent: QP is in RTR (EP is in Active Connection Pending State).
    - Send REJ, done.
- CM: Established, QP in RTS, EP connected.
    - Normal disconnect sequence.

Passive side:

- CM: REQ Rcvd, QP in Init, EP in Unconnected.
    - Disconnect is rejected. Consumer shall call *dat_cr_reject* instead of *dat_ep_disconnect*.
- CM: MRA Sent, QP is in Init, EP is not involved yet.
    - Send REJ with reason code 1.
- CM: REP Sent, QP RTR, EP in Pending state.
    - Transition QP into error state. Wait for CM Timeout to expire. There are multiple race conditions here:
        - If Recv message arrives on connection before QP is in Error state, go through a Connection Established state followed by Send DREQ.
        - If QP is moved to an Error state before Recv message, the Recv fails. DREQ can be sent. CM is in TimeWait state, QP is in error state, done.
        - If REJ is received, done.
        - If RTU is received, there is the race analogous to the Recv message above. The same steps apply.
        - If MRA recv, ignore the MRA, wait for the original timeout period, and then send an REJ with reason code 4 if the Provider doesn't receive an RTU or a message before that timeout period is up.
        - When timeout expired before an RTU or Recv message, send REJ with reason code 4.
        - If an RTU or Recv message arrived, *Disconnect* is done already, done.

- CM is in TimeWait state. QP is in Error state, EP is in Unconnected state.
    - Disconnected or Broken event was generated already. Ignore *dat_ep_disconnect*, done.

6.6.18 DAT_EP_RESET

Synopsis:

```
DAT_RETURN
   dat_ep_reset (
   IN     DAT_EP_HANDLE      ep_handle
   )
```

Parameters:

ep_handle:	Handle for an instance of Endpoint.

Description:

dat_ep_reset transitions the local Endpoint from a *Disconnected* to an *Unconnected* state.

The operation might cause the loss of any completions of previously posted DTOs and RMRs that were not dequeued yet.

dat_ep_reset is valid for both *Disconnected* and *Unconnected* states. For *Unconnected* state, the operation is no-op because the Endpoint is already in an *Unconnected* state. For an *Unconnected* state, the preposted Recvs are not affected by the call.

dat_ep_reset is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint is not in the valid state for reset.

6.6.18.1 USAGE

If the Consumer wants to ensure that all Completions are dequeued, the Consumer can post DTO or RMR operations as a “marker” that are flushed to *recv_evd_handle* or *request_evd_handle.* Now, when the Consumer dequeues the completion of the “marker” from the EVD, it is guaranteed that all previously posted DTO and RMR completions for the Endpoint were dequeued for that EVD. Now, it is safe to reset the Endpoint without losing any completions.

6.6.18.2 RATIONALE

6.6.18.3 MODEL IMPLICATIONS

6.6.19 DATA TRANSFER OPERATIONS

The following operations are defined below to do data transfer over the connection that is represented locally by an Endpoint in the connected state: *send, receive, rdma_read and rdma_write.* All these operations are asynchronous and the completion of the data transfer it return asynchronously via DTO completion event on the Send or Recv EVD of the Endpoint. The DTO completion event provides: Endpoint to which DTO was posted, *user_cookie* Consumer provided at the post of DTO, *status* of the data transfer, *transfered_length* for successful transfer for receive and RDMA Read operations, and the type of the completed operation.

6.6.19.1 USAGE

DAPL-2.0 had added a new field for DTO completion type to *dat_dto_completion_event_data*. As long as an application used the names of the fields and not a position in the data structure a recompile will maintain the application code compatibility. For applications that use the names of the fields and used *dat_event_data* for memory allocation rather than *dat_dto_completion_event_data* also maintain the backwards binary compatibility.

6.6.20 DAT_EP_POST_SEND

Synopsis:

```
DAT_RETURN
   dat_ep_post_send (
   IN     DAT_EP_HANDLE            ep_handle,
   IN     DAT_COUNT                num_segments,
   IN     DAT_LMR_TRIPLET          *local_iov,
   IN     DAT_DTO_COOKIE           user_cookie,
   IN     DAT_COMPLETION_FLAGS     completion_flags
   )
```

Parameters:

ep_handle: Handle for an instance of the Endpoint.

num_segments: Number of *lmr_triplets* in *local_iov*. Can be 0 for 0 size message.

local_iov: I/O Vector that specifies the local buffer that contains data to be transferred. Can be NULL for 0 size message.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the send. Can be NULL.

completion_flags: Flags for posted Send. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0x00 (see Appendix A.4). See Table 8 for flag definitions.

Table 8 Send DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Completion Suppression		0x00	Generate Completion.	
	DAT_COMPLETION_ SUPPRESS_FLAG	0x01	Suppress successful Completion.	
Solicited Wait		0x00	No request for notification completion for matching receive on the other side of the connection.	
	DAT_COMPLETION_ SOLICITED_WAIT_FLAG	0x02	Request for notification completion for matching receive on the other side of the connection.	
Notification of Completion		0x00	Notification completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_COMPLETION_ UNSIGNALLED_FLAG	0x04	Non-notification completion.	
Barrier Fence		0x00	No request for RDMA Read Barrier Fence.	.
	DAT_COMPLETION_ BARRIER_FENCE_FLAG	0x08	Request for RDMA Read Barrier Fence.	

Description: *dat_ep_post_send* requests a transfer of all the data from the *local_iov* over the connection of the *ep_handle* Endpoint to the remote side.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred. The actual order of transfer of the data from the segments is left to the implementation. The *local_iov* specification should adhere to the rules defined in Appendix A.4.

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* back after the *dat_ep_post_send* returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov*, but not the memory it specifies after *dat_ep_post_send* returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The DAT_SUCCESS return of the *dat_ep_post_send* is at least the equivalent of posting a Send operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_send* to ensure that this operation is nonblocking.

The completion of the posted Send is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION _ UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_ INVALID_PARAMETER* is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Send.

The operation is valid for the Endpoint in the *DAT_EP_STATE_ CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in the *DAT_EP_STATE_ DISCONNECTED* state, the posted Send is immediately flushed to *request_evd_handle.*

If the reported *status* of the Completion DTO event corresponding to the posted Send DTO is not *DAT_DTO_SUCCESS*, the *transfered_length* in the DTO Completion event is not defined.

dat_ep_post_send is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.

DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_STATE_CONNECTED* or *DAT_EP_STATE_DISCONNECTED* state.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.

6.6.20.1 USAGE

For best Send operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the *DAT_OPTIMAL_ALIGNMENT.*

6.6.20.2 RATIONALE

6.6.20.3 MODEL IMPLICATIONS

6.6.21 DAT_EP_POST_SEND_WITH_INVALIDATE

Synopsis:

```
DAT_RETURN
   dat_ep_post_send_with_invalidate (
   IN     DAT_EP_HANDLE             ep_handle,
   IN     DAT_COUNT                 num_segments,
   IN     DAT_LMR_TRIPLET           *local_iov,
   IN     DAT_DTO_COOKIE            user_cookie,
   IN     DAT_COMPLETION_FLAGS      completion_flags,
   IN     DAT_BOOLEAN               invalidate_flag,
   IN     DAT_RMR_CONTEXT           rmr_context
   )
```

Parameters:

ep_handle:	Handle for an instance of the Endpoint.
num_segments:	Number of *lmr_triplets* in *local_iov*. Can be 0 for 0 size message.
local_iov:	I/O Vector that specifies the local buffer that contains data to be transferred. Can be NULL for 0 size message.

user_cookie:	User-provided cookie that is returned to the Consumer at the completion of the send. Can be NULL.
completion_flags:	Flags for posted Send. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0x00 (see Appendix A.4). See Table 8 for flag definitions.
invalidate_flag:	A binary indicator that indicated whether remote invalidation of *rmr_context* was requested. The value of DAT_TRUE indicate that RMR_context invalidation is requested and the value of DAT_FALSE indicates no remote invlidation.
rmr_context:	Remote Memory Context to be invalidated at remote side of the connection.

Table 9 Send with Invalidate DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Completion Suppression		0x00	Generate Completion.	
	DAT_COMPLETION_SUPPRESS_FLAG	0x01	Suppress successful Completion.	
Solicited Wait		0x00	No request for notification completion for matching receive on the other side of the connection.	
	DAT_COMPLETION_SOLICITED_WAIT_FLAG	0x02	Request for notification completion for matching receive on the other side of the connection.	
Notification of Completion		0x00	Notification completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_COMPLETION_UNSIGNALLED_FLAG	0x04	Non-notification completion.	
Barrier Fence		0x00	No request for RDMA Read Barrier Fence.	.
	DAT_COMPLETION_BARRIER_FENCE_FLAG	0x08	Request for RDMA Read Barrier Fence.	

Description: *dat_ep_post_send_with_invalidate* requests a transfer of all the data from the *local_iov* over the connection of the *ep_handle* Endpoint to the remote side and invalidates the Remote Memory Region context.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred. The actual order of transfer of the data from the segments is left to the implementation. The *local_iov* specification should adhere to the rules defined in <u>Appendix A.4</u>.

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* back after the *dat_ep_post_send* returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov*, but not the memory it specifies after *dat_ep_post_send* returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The DAT_SUCCESS return of the *dat_ep_post_send* is at least the equivalent of posting a Send operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_send* to ensure that this operation is nonblocking.

The completion of the posted Send is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION_UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_INVALID_PARAMETER* is returned.

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Send.

The operation is valid for the Endpoint in the *DAT_EP_STATE_CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in the *DAT_EP_STATE_DISCONNECTED* state, the posted Send is immediately flushed to *request_evd_handle.*

The *invalidate_flag* indicate whether the requested *rmr_context* requested for invalidation. The value of *DAT_TRUE* specify that invalidation is requested, and the value of *DAT_FALSE* specify that invalidation is not requested. If invalidation is not requested the value of *rmr_context* is undefined.

If the reported *status* of the Completion DTO event corresponding to the posted Send DTO is not *DAT_DTO_SUCCESS*, the *transfered_length* in the DTO Completion event is not defined.

dat_ep_post_send is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv*. The operation is UpCall safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_STATE_CONNECTED* or *DAT_EP_STATE_DISCONNECTED* state.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local read privileges.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.

6.6.21.0.1 USAGE

For best Send operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the *DAT_OPTIMAL_ALIGNMENT.*

6.6.21.0.2 RATIONALE

6.6.21.0.3 MODEL IMPLICATIONS

6.6.22 DAT_EP_POST_RECV

Synopsis:

```
DAT_RETURN
   dat_ep_post_recv (
   IN     DAT_EP_HANDLE            ep_handle,
   IN     DAT_COUNT                num_segments,
```

```
IN    DAT_LMR_TRIPLET          *local_iov,
IN    DAT_DTO_COOKIE           user_cookie,
IN    DAT_COMPLETION_FLAGS     completion_flags
)
```

Parameters:

ep_handle: Handle for an instance of the Endpoint.

num_segments: Number of *lmr_triplets* in *local_iov*. Can be 0 for receiving a 0 size message.

local_iov: I/O Vector that specifies the local buffer to be filled. Can be NULL for receiving a 0 size message.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the Receive DTO. Can be NULL.

completion_flags: Flags for posted Receive. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0x00 (see Appendix A.4). See Table 10 for flag definitions.

.

Table 10 Receive DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Notification of Completion		0x00	Notification completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_ COMPLETION_ UNSIGNALLED_ FLAG	0x04	Non-notification completion.	

Description: *dat_ep_post_recv* requests the receive of the data over the connection of the *ep_handle* Endpoint of the incoming message into the *local_iov*.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the “front” segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all. The actual order of segment fillings is left to the implementation. The *local_iov* specification should adhere to the rules defined in Appendix A.4.

The user_cookie allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted Receive.

The completion of the posted Receive is reported to the Consumer asynchronously through a DTO Completion event based on the configuration of the connection for Solicited Wait and the specified *completion_flags* value for the matching Send. The value of *DAT_ COMPLETION _UNSIGNALLED_FLAG* is only valid if the Endpoint Recv Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_INVALID_PARAMETER* is returned.

The asynchronous successful completion of the posted Receive will report which if it matched one of the remote Send operations. The size of the transfered data is reported in the *transfered_length* of the DAT_DTO_ COMPLETION_EVENT_DATA.

The asynchronous successful completion of the posted Receive also indicate whether remote side invalidated an *rmr_context,* and if yes, which *rmr_context* has been invalidated. If *rmr_context* has been invalidated the field *operation* in DAT_DTO_COMPLETION_EVENT_DATA returned is *DAT_RECEIVE_WITH_INVALIDATE*. In this case the value of *rmr_ context* in DAT_DTO_COMPLETION_EVENT_DATA indicates which RMR context (of LMR or of RMR) was invalidated. If *operation* field in DAT_DTO_COMPLETION_EVENT_DATA is not *DAT_RECEIVE_ WITH_INVALIDATE* (*DAT_RECEIVE*) then the value of the field *rmr_ context* in DAT_DTO_COMPLETION_EVENT_DATA is undefined.

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specified back after the *dat_ep_post_recv* returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumer full control of the *local_iov* content after *dat_ep_post_ recv* returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The DAT_SUCCESS return of the *dat_ep_post_recv* is at least the equivalent of posting a Receive operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_recv* to ensure that this operation is nonblocking.

If the size of an incoming message is larger than the size of the *local_iov*, the reported *status* of the posted Receive DTO in the corresponding Completion DTO event is *DAT_DTO_LENGTH_ERROR.* If the reported *status* of the Completion DTO event corresponding to the posted Receive DTO is not *DAT_DTO_SUCCESS*, the content of the *local_iov* is not defined, the and the *transfered_length* in the DTO Completion event is not defined. If the reported *status* of the Completion DTO event corresponding to the posted Receive DTO is not *DAT_DTO_SUCCESS*, the *operation, rmr_context,* are not defined.

The operation is valid for all states of the Endpoint. The actual data transfer does not take place until the Endpoint is in the *DAT_EP_STATE_CONNECTED* state. The operation on the Endpoint in *DAT_EP_STATE_DISCONNECTED* is allowed. If the operation returns successfully, the posted Recv is immediately flushed to *recv_evd_handle.*

If SRQ is associated with EP the operation is illegal and will return *DAT_INVALID_STATE.*

dat_ep_post_recv is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_send, dat_ep_post_rdma_read, dat_ep_post_drma_write,* and *dat_rmr_bind*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter. For example, one of the IOV segments pointed to a memory outside its LMR.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between an LMR of one of the *local_iov* segments and the local Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. One of the LMRs used in *local_iov* was either invalid or did not have the local write privileges.

6.6.22.1 USAGE

For the best Recv operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the *DAT_OPTIMAL_ALIGNMENT.*

6.6.22.2 RATIONALE

For the Recv with Invalidate case the returned invalidated *rmr_context* can be used to verified that the correct RMR or *rmr_context* of the correct LMR has been invalidated. In a typical scenario Recv data will include the ULP operation info that has been completion by remote side for which local side provided *rmr_context* that has been invalidated.

6.6.22.3 MODEL IMPLICATIONS

The invalidation of the *rmr_context* of LMR has no effect on the *lmr_context* of the LMR. The invalidation of the *rmr_context* of an RMR transitions the RMR into unbound state analogous to the state RMR will transition upon local *dat_rmr_bind* with *length* of zero.

6.6.23 DAT_EP_POST_RDMA_READ

Synopsis:

```
DAT_RETURN
    dat_ep_post_rdma_read (
    IN     DAT_EP_HANDLE            ep_handle,
    IN     DAT_COUNT                num_segments,
    IN     DAT_LMR_TRIPLET          *local_iov,
    IN     DAT_DTO_COOKIE           user_cookie,
    IN     DAT_RMR_TRIPLET          *remote_buffer,
    IN     DAT_COMPLETION_FLAGS     completion_flags
    )
```

Parameters:

ep_handle: Handle for an instance of the Endpoint.

num_segments: Number of *lmr_triplets* in *local_iov*.

local_iov: I/O Vector specifying the local data buffer to fill.

user_cookie: User-provided cookie that is returned to the Consumer at the completion of the RDMA Read. Can be NULL.

remote_buffer: A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.

completion_flags: Flags for posted RDMA Read. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0x00 (see Appendix A.4). See Table 11 for flag definitions.

Table 11 RDMA Read DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Completion Suppression		0x00	Generate Completion.	
	DAT_COMPLETION_SUPPRESS_FLAG	0x01	Suppress successful Completion.	
Notification of Completion		0x00	Notification Completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_COMPLETION_UNSIGNALLED_FLAG	0x04	Non-notification Completion.	

Table 11 RDMA Read DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Barrier Fence		0x00	No request for RDMA Read Barrier Fence.	
	DAT_COMPLETION_ BARRIER_FENCE_FLAG	0x08	Request for RDMA Read Barrier Fence.	

Description: *dat_ep_post_rdma_read* requests the transfer of all the data specified by the *remote_buffer* over the connection of the *ep_handle* Endpoint into the *local_iov*.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are filled in the I/O Vector order until the whole message is received. This ensures that all the “front” segments of the *local_iov* I/O Vector are completely filled, only one segment is partially filled, if needed, and all segments that follow it are not filled at all. The actual order of segment fillings is left to the implementation. The *local_iov* and *remote_buffer* specifications should adhere to the rules defined in Appendix A.4.

The requested length of the data transfer is specified by the local buffer length. That is the sum of the *segment_lengths* of *local_iov.*

The user_cookie allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Read.

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the *dat_ep_post_rdma_read* returns should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* content after *dat_ep_post_rdma_read* is returned. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The completion of the posted RDMA Read is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION _ UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_ INVALID_PARAMETER* is returned.

The DAT_SUCCESS return of the *dat_ep_post_rdma_read* is at least the equivalent of posting an RDMA Read operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_rdma_read* to ensure that this operation is nonblocking.

The operation is valid for the Endpoint in the *DAT_EP_STATE_CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in the *DAT_EP_STATE_DISCONNECTED* state, the posted RDMA Read is immediately flushed to *request_evd_handle.*

If EP *max_rdma_read_out* is zero then Consumer posting of an RDMA Read will succeed but will cause RDMA Read to complete in error with *DAT_DTO_ERROR_LOCAL_PROTECTION*.

If the reported *status* of the Completion DTO event corresponding to the posted RDMA Read DTO is not *DAT_DTO_SUCCESS*, the content of the *local_iov* is not defined and the *transfered_length* in the DTO Completion event is not defined.

dat_ep_post_rdma_read is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter; For example, one of the IOV segments pointed to a memory outside its LMR, or the number of IOVs specified exceeds EP capacity.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_STATE_CONNECTED* or *DAT_EP_STATE_DISCONNECTED* state.
DAT_LENGTH_ERROR	The size of the receiving buffer is too small for sending buffer data. The size of the local buffer is too small for the data of the remote buffer.

DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* is invalid or does not have the local write privileges, or *rmr_context* does not have the remote read privileges.

6.6.23.1 USAGE

For the best RDMA Read operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to the *DAT_OPTIMAL_ALIGNMENT.*

If connection was established without outstanding RDMA Read attributes matching on Endpoints on both sides (outstanding RDMA Read outgoing on one end is larger than the outstanding RDMA Read incoming on the other end), connection is broken when the number of incoming RDMA Read exceeds the outstanding RDMA Read incoming attribute of the Endpoint. The Consumer can use its own flow control to ensure that it does not post more RDMA Reads then the remote EP outstanding RDMA Read incoming attribute is. Thus, they do not rely on the underlying Transport enforcing it.

For some RDMA Transports and Providers a local RDMA Read buffer memory require both RDMA Read and Write memory privileges. The Provider attribute r*dma_write_for_rdma_read_req* indicate if this is the case. Failure to set up local buffer memory privileges for these Providers will result in asynchronous DTO completion error and connection being broken.

DAT does not guarantee any ordering between multiple RDMA DTO even over the same connection to the same remote memory.

6.6.23.2 RATIONALE

The pipeline of RDMA DTOs over a single connection can proceed simultaneously. Thus, if they access the same remote memory the result of the remote buffer is indeterminate. Consumer can control RDMA Read ordering with respect to other RDMA Reads or Writes or Sends via *DAT_COMPLETION_BARRIER_FENCE_FLAG*.

6.6.23.3 MODEL IMPLICATIONS

The number of posted RDMA Reads on Send WQ can exceed *max_rdma_read_out* attribute of the EP. DAT Provider ensures that the number of outstanding RDMA Reads on the remote endpoint of the connection does not exceed the EP attribute. Consumer should rely on its own RDMA Read flow control to ensure that the number of RDMA Reads for which completions have not been generated does not exceed the EP *max_rdma_read_out* attribute value.

While Provider does guarantee flow control for RDMA Read DTOs (the maximum number of RDMA Reads reaching the remote host simultaneously over a single connection), Consumer should avoid posting more than *max_rdma_read_out* RDMA Reads to the connection. Since all DTOs posted to the Send WQ of the EP are processed in order, inability to process an RDMA Read that exceeds *max_rdma_read_out* will stall processing of all other DTOs of the Send WQ of the EP. This is irrespective of the Consumer specification of *DAT_COMPLETION_BARRIER_FENCE_FLAG* value that is Consumer requested stalling of the Send WQ processing.

The error behavior for the case when remote buffer is too small for requested transfered data may be transport specific. The remote buffer size is defined the size of the RMR and not necessarily the *segment_length* of the *DAT_RMR_TRIPLET* specified locally.

The error can be provided synchronously or asynchronously. If the error is return synchronously then *DAT_LENGTH_ERROR* is returned. A synchronously returned error has no effect on the state of the Endpoint to which operation was posted nor any other posted operations. A behavior of the connection as well as the type of the asynchronous error return when an error is return asynchronously is defined by the underlying RDMA transport. For example, a connection may be broken as the result of the asynchronous error. An asynchronous error may be return locally, remotely or both.

6.6.24 DAT_EP_POST_RDMA_READ_TO_RMR

Synopsis:

```
DAT_RETURN
   dat_ep_post_rdma_read_to_rmr (
   IN     DAT_EP_HANDLE            ep_handle,
   IN const DAT_RMR_TRIPLET        *local_iov,
   IN     DAT_DTO_COOKIE           user_cookie,
   IN     DAT_RMR_TRIPLET          *remote_buffer,
   IN     DAT_COMPLETION_FLAGS     completion_flags
   )
```

Parameters:

ep_handle:	Handle for an instance of the Endpoint.
local_iov:	A pointer to an RMR Triplet that specifies the local data buffer to fill.
user_cookie:	User-provided cookie that is returned to the Consumer at the completion of the RDMA Read. Can be NULL.
remote_buffer:	A pointer to an RMR Triplet that specifies the remote buffer from which the data is read.
completion_flags:	Flags for posted RDMA Read. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0x00 (see Appendix A.4). See Table 11 for flag definitions.

Description: *dat_ep_post_rdma_read_to_rmr* requests the transfer of all the data specified by the *remote_buffer* over the connection of the *ep_handle* Endpoint into the *local_iov* specified by the RMR segments.

The requested length of the data transfer is specified by the local buffer length. That is the *segment_length* of *local_iov.*

The *user_cookie* allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Read.

A Consumer shall not modify the *local_iov* content inclusing RMR and its underlying LMR until the DTO is completed. When a Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined.

The completion of the posted RDMA Read is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION_UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_INVALID_PARAMETER* is returned.

The DAT_SUCCESS return of the *dat_ep_post_rdma_read_to_rmr* is at least the equivalent of posting an RDMA Read operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_rdma_read_to_rmr* to ensure that this operation is nonblocking.

The operation is valid for the Endpoint in the *DAT_EP_STATE_CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in the *DAT_EP_STATE_DISCONNECTED* state, the posted RDMA Read is immediately flushed to *request_evd_handle.*

If EP *max_rdma_read_out* is zero then Consumer posting of an RDMA Read will succeed but will cause RDMA Read to complete in error with *DAT_DTO_ERROR_LOCAL_PROTECTION*.

If the reported *status* of the Completion DTO event corresponding to the posted RDMA Read DTO is not *DAT_DTO_SUCCESS*, the content of the *local_iov* is not defined and the *transfered_length* in the DTO Completion event is not defined.

dat_ep_post_rdma_read_to_rmr is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter; For example, a local buffer includes memory outside its RMR.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_STATE_CONNECTED* or *DAT_EP_STATE_DISCONNECTED* state.
DAT_LENGTH_ERROR	The size of the remote buffer is too large for requested data size.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between either an RMR of one of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. Either one of the RMRs used in *local_iov* is invalid or does not have the local write privileges, or *rmr_context* of *remote_buffer* does not have the remote read privileges.

6.6.24.1 USAGE

For the best RDMA Read operation performance, the Consumer should align the *local_iov* buffer to the *Optimal Buffer Alignment* attribute of the

Provider. For portable applications, the Consumer should align the *local_iov* buffer to the *DAT_OPTIMAL_ALIGNMENT.*

If connection was established without outstanding RDMA Read attributes matching on Endpoints on both sides (outstanding RDMA Read outgoing on one end is larger than the outstanding RDMA Read incoming on the other end), connection is broken when the number of incoming RDMA Read exceeds the outstanding RDMA Read incoming attribute of the Endpoint. The Consumer can use its own flow control to ensure that it does not post more RDMA Reads then the remote EP outstanding RDMA Read incoming attribute is. Thus, they do not rely on the underlying Transport enforcing it.

For some RDMA Transports and Providers a local RDMA Read buffer memory require both RDMA Read and RDMA Write memory privileges. The Provider attribute *rdma_write_for_rdma_read_req* indicate if this is the case. Failure to set up local buffer memory privileges for these Providers will result in asynchronous DTO completion error and connection being broken.

DAT does not guarantee any ordering between multiple RDMA DTO even over the same connection to the same remote memory.

6.6.24.2 RATIONALE

The pipeline of RDMA DTOs over a single connection can proceed simultaneously. Thus, if they access the same remote memory the result of the remote buffer is indeterminate. Consumer can control RDMA Read ordering with respect to other RDMA Reads or Writes or Sends via *DAT_COMPLETION_BARRIER_FENCE_FLAG.*

This capability is needed for applications that wish to guarantee that the RMR Context for an LMR Context is not exposed to the network. An RMR Context can be invalidated at a lower cost, and is therefore preferable.

6.6.24.3 MODEL IMPLICATIONS

The number of posted RDMA Reads on Send WQ can exceed *max_rdma_read_out* attribute of the EP. DAT Provider ensures that the number of outstanding RDMA Reads on the remote endpoint of the connection does not exceed the EP attribute. Consumer should rely on its own RDMA Read flow control to ensure that the number of RDMA Reads for which completions have not been generated does not exceed the EP *max_rdma_read_out* attribute value.

While Provider does guarantee flow control for RDMA Read DTOs (the maximum number of RDMA Reads reaching the remote host simultaneously over a single connection), Consumer should avoid posting more than *max_rdma_read_out* RDMA Reads to the connection. Since all DTOs posted to the Send WQ of the EP are processed in order, inability to process an RDMA Read that exceeds *max_rdma_read_out* will stall processing of all other DTOs of the Send WQ of the EP. This is irrespective of the Consumer specification of *DAT_COMPLETION_*

BARRIER_FENCE_FLAG value that is Consumer requested stalling of the Send WQ processing.

The error behavior for the case when a remote buffer is too small for requested transfered data may be transport specific. The remote buffer size is defined the size of the RMR and not necessarily the *segment_length* of the *DAT_RMR_TRIPLET* specified locally.

The error can be provided synchronously or asynchronously. If the error is return synchronously then *DAT_LENGTH_ERROR* is returned. A synchronously returned error has no effect on the state of the Endpoint to which operation was posted nor any other posted operations. A behavior of the connection as well as the type of the asynchronous error return when an error is return asynchronously is defined by the underlying RDMA transport. For example, a connection may be broken as the result of the asynchronous error. An asynchronous error may be return locally, remotely or both.

6.6.25 DAT_EP_POST_RDMA_WRITE

Synopsis:

```
DAT_RETURN
   dat_ep_post_rdma_write (
   IN     DAT_EP_HANDLE            ep_handle,
   IN     DAT_COUNT                num_segments,
   IN     DAT_LMR_TRIPLET          *local_iov,
   IN     DAT_DTO_COOKIE           user_cookie,
   IN     DAT_RMR_TRIPLET          *remote_buffer,
   IN     DAT_COMPLETION_FLAGS     completion_flags
   )
```

Parameters:

ep_handle:	Handle for an instance of the Endpoint.
num_segments:	Number of *lmr_triplets* in *local_iov*.
local_iov:	I/O Vector specifying the local buffer from which the data is transferred.
user_cookie:	User-provided cookie that is returned to a Consumer at the completion of the RDMA Write.
remote_buffer:	A pointer to an RMR triplet that specifies the remote buffer to which the data shall be written.
completion_flags:	Flags for posted RDMA Write. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0 (see Appendix A.4). See Table 12 for flag definitions.

Table 12 RDMA Write DTO Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Completion Suppression		0x00	Generate Completion.	
	DAT_COMPLETION_ SUPPRESS_FLAG	0x01	Suppress successful Completion.	
Notification of Completion		0x00	Notification Completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_COMPLETION_ UNSIGNALLED_ FLAG	0x04	Non-notification Completion.	
Barrier Fence		0x00	No request for RDMA Read Barrier Fence.	
	DAT_COMPLETION_ BARRIER_FENCE_ FLAG	0x08	Request for RDMA Read Barrier Fence.	

Description: *dat_ep_post_rdma_write* requests a transfer of all the data from the *local_iov* over the connection of the *ep_handle* Endpoint into the *remote_buffer*.

num_segments specifies the number of segments in the *local_iov*. The *local_iov* segments are traversed in the I/O Vector order until all the data is transferred. The actual order of transfer of the data from the segments is left to the implementation. The *local_iov* and the *remote_buffer* specifications should adhere to the rules defined in Appendix A.4.

The requested length of the data transfer is specified by the local buffer length. That is the sum of the *segment_lengths* of *local_iov.*

A Consumer shall not modify the *local_iov* or its content until the DTO is completed. When Consumer does not adhere to this rule, the behavior of the Provider and the underlying Transport is not defined. Providers that allow Consumers to get ownership of the *local_iov* but not the memory it specifies back after the *dat_ep_post_rdma_write* returns, should document this behavior and also specify its support in Provider attributes. This behavior allows Consumers full control of the *local_iov* after *dat_ep_post_rdma_write* returns. Because this behavior is not guaranteed by all Providers, portable Consumers shall not rely on this behavior. Consumers shall not rely on the Provider copying *local_iov* information.

The DAT_SUCCESS return of the *dat_ep_post_rdma_write* is at least the equivalent of posting an RDMA Write operation directly by native Transport. Providers shall avoid resource allocation as part of *dat_ep_post_rdma_write* to ensure that this operation is nonblocking.

The completion of the posted RDMA Write is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION _ UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG*. Otherwise, *DAT_ INVALID_PARAMETER* is returned.

The user_cookie allows Consumers to have unique identifiers for each DTO. These identifiers are completely under user control and are opaque to the Provider. There is no requirement on the Consumer that the value *user_cookie* should be unique for each DTO. The *user_cookie* is returned to the Consumer in the Completion event for the posted RDMA Write.

The operation is valid for the Endpoint in the *DAT_EP_STATE_ CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in the *DAT_EP_STATE_ DISCONNECTED* state, the posted RDMA Write is immediately flushed to *request_evd_handle.*

If the reported *status* of the Completion DTO event corresponding to the posted RDMA Write DTO is not *DAT_DTO_SUCCESS*, the *transfered_ length* in the DTO Completion event is not defined.

dat_ep_post_rdma_write is asynchronous and nonblocking. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter; For example, one of the IOV segments pointed to a memory outside its LMR, or the number of IOVs specified exceeds EP capacity.
DAT_INVALID_HANDLE	Invalid DAT handle; *ep_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_ STATE_CONNECTED* or *DAT_EP_ STATE_DISCONNECTED* state.
DAT_LENGTH_ERROR	The size of the receiving buffer was too small for sending buffer data. The size of the remote buffer was too small for the data of the local buffer.

DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access. Protection Zone mismatch between either an LMR of one of the *local_iov* segments and the local Endpoint or the *rmr_context* and the remote Endpoint.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access. Either one of the LMRs used in *local_iov* was invalid or did not have the local read privileges, or *rmr_context* did not have the remote write privileges.

6.6.25.1 USAGE

For the best RDMA Write operation performance, the Consumer should align each buffer segment of *local_iov* to the *Optimal Buffer Alignment* attribute of the Provider. For portable applications, the Consumer should align each buffer segment of *local_iov* to *DAT_OPTIMAL_ALIGNMENT.*

DAT does not guarantee any ordering between multiple RDMA DTO even over the same connection to the same remote memory.

The pipeline of RDMA DTOs over a single connection can proceed simultaneously. Thus, if they access the same remote memory the result of the remote buffer is indeterminate. The result of multiple RDMA Writes accessing the same buffer simultaneously can range from data in the buffer from any one of those RDMA Write operations, to data in the buffer being a mixture from multiple RDMA Writes. Consumer can control RDMA Read ordering with respect to other RDMA Writes via *DAT_ COMPLETION_BARRIER_FENCE_FLAG*.

If Consumer desires a deterministic result they should use ULP protocol to ensure that only one RDMA Write operation accesses remote buffer at a time. For example, they can use 0-size RDMA Read between a pair of RDMA Writes that access the same remote location. See 6.8.2.1 Usage on page 299 for details and more advice.

6.6.25.2 RATIONALE

Each instance of multiple RDMA Writes accessing the same remote location generates a return code the same as if it were a single RDMA Write accessing that memory location. Another words, no error will be generated because multiple RDMA Writes access the same memory location.

6.6.25.3 MODEL IMPLICATIONS

The error behavior for the case when remote buffer is too small for transfered data may be transport specific. The remote buffer size is defined the size of the RMR and not necessarily the *segment_length* of the *DAT_RMR_TRIPLET* specified locally.

The error can be provided synchronously or asynchronously. If the error is return synchronously then *DAT_LENGTH_ERROR* is returned. A synchronously returned error has no effect on the state of the Endpoint to which operation was posted nor any other posted operations. A behavior of the connection as well as the type of the asynchronous error return when an error is return asynchronously is defined by the underlying RDMA transport. For example, a connection may be broken as the result of the asynchronous error. An asynchronous error may be return locally, remotely or both.

6.7 MEMORY MANAGEMENT

6.7.1 PROTECTION ZONE

6.7.1.1 DAT_PZ_CREATE

Synopsis:

```
DAT_RETURN
   dat_pz_create (
   IN     DAT_IA_HANDLE     ia_handle,
   OUT    DAT_PZ_HANDLE     *pz_handle
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA.
pz_handle:	Handle for the created instance of Protection Zone.

Description:

dat_pz_create creates an instance of the Protection Zone. The Protection Zone provides Consumers a mechanism for association Endpoints with LMRs and RMRs to provide protection for local and remote memory accesses by DTOs.

dat_pz_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	Invalid DAT handle; *ia_handle* is invalid.

6.7.1.1.1 Usage

6.7.1.1.2 Rationale

6.7.1.1.3 Model Implications

6.7.1.2 DAT_PZ_FREE

Synopsis:

```
DAT_RETURN
   dat_pz_free (
   IN     DAT_PZ_HANDLE     pz_handle
   )
```

Parameters:

pz_handle: Handle for an instance of Protection Zone to be destroyed.

Description:

dat_pz_free destroys an instance of the Protection Zone. The Protection Zone cannot be destroyed if it is in use by an Endpoint, LMR, or RMR.

It is illegal to use the destroyed handle in any subsequent operation.

dat_pz_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_STATE	Parameter in an invalid state. The Protection Zone was in use by Endpoint, LMR, or RMR instances.
DAT_INVALID_HANDLE	Invalid DAT handle; *pz_handle* is invalid.

6.7.1.2.1 Usage

6.7.1.2.2 Rationale

6.7.1.2.3 Model Implications

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.7.1.3 DAT_PZ_QUERY

Synopsis:

```
DAT_RETURN
   dat_pz_query (
   IN     DAT_PZ_HANDLE          pz_handle,
   IN     DAT_PZ_PARAM_MASK      pz_param_mask,
   OUT    DAT_PZ_PARAM           *pz_param
```

```
)
```

Parameters:

pz_handle:	Handle for the created instance of the Protection Zone.
pz_param_mask:	Mask for Protection Zone parameters.
pz_param:	Pointer to a Consumer-allocated structure that the Provider fills with Protection Zone parameters.

Description: *dat_pz_query* provides the Consumer parameters of the Protection Zone. The Consumer passes in a pointer to the Consumer-allocated structures for Protection Zone parameters that the Provider fills.

pz_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *pz_param_mask* requested parameters. The Provider can return values for any other parameters.

dat_pz_query is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *pz_param_mask* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *pz_handle* is invalid.

6.7.1.3.1 Usage

6.7.1.3.2 Rationale

6.7.1.3.3 Model Implications

6.7.2 Local Memory Region

6.7.2.1 DAT_LMR_Create

Synopsis:

```
DAT_RETURN
   dat_lmr_create (
   IN     DAT_IA_HANDLE             ia_handle,
   IN     DAT_MEM_TYPE              mem_type,
   IN     DAT_REGION_DESCRIPTION    region_description,
   IN     DAT_VLEN                  length,
   IN     DAT_PZ_HANDLE             pz_handle,
   IN     DAT_MEM_PRIV_FLAGS        mem_privileges,
   IN     DAT_VA_TYPE               va_type,
   OUT    DAT_LMR_HANDLE            *lmr_handle,
   OUT    DAT_LMR_CONTEXT           *lmr_context,
```

```
        OUT    DAT_RMR_CONTEXT         *rmr_context,
        OUT    DAT_VLEN                *registered_size,
        OUT    DAT_VADDR               *registered_address
        )
```

Parameters:

Table 13 LMR Memory Type Specification Definitions

Memory Type	Description	Region description	Length
DAT_MEM_TYPE_ VIRTUAL	Consumer virtual memory.	A pointer to a contiguous user virtual range.	Length of the Memory Region
DAT_MEM_TYPE_LMR	LMR	An LMR_handle	Length parameter is ignored
DAT_MEM_TYPE_ SHARED_VIRTUAL	Shared memory region. All DAT Consumers of the same uDAPL Provider specify the same Consumer cookie to indicate who is sharing the shared memory region. This supports a peer-to-peer model of shared memory. All DAT Consumers of the shared memory must allocate the memory region as shared memory using Platform-specific primitives.	A structure with 2 elements, where the first one is a pointer to a contiguous user virtual range, and the second one is of type DAT_LMR_ COOKIE is a unique identifier of the shared memory region.	Length of the Memory Region

ia_handle: Handle for an open instance of the IA.

mem_type: Type of memory to be registered. See Table 13 for memory type specifications.

region_description: Pointer to type-specific data describing the memory in the region to be registered. The type is derived from the *mem_type* parameter.

length: Length parameter accompanying the *region_ description*.

pz_handle: Handle for an instance of the Protection Zone.

mem_privileges: Consumer-requested memory access privileges for the registered local memory region. The Default value is DAT_MEM_PRIV_NONE_FLAG. The constant value DAT_MEM_PRIV_ALL_FLAG, which specifies both Read and Write privileges, is also defined. See Table 14 for memory privilege definitions.

Table 14 LMR Memory Privilege Definitions

Privileges	Definition/Bit	Value	Description
Local Read		0x00	No local read access requested.
	DAT_MEM_PRIV_LOCAL_READ_FLAG	0x01	Local read access requested.
Local Write		0x00	No local write access requested.
	DAT_MEM_PRIV_LOCAL_WRITE_FLAG	0x10	Local write access requested.
Remote Read		0x00	No remote read access requested.
	DAT_MEM_PRIV_REMOTE_READ_FLAG	0x02	Remote read access requested.
Remote Write		0x00	No remote write access requested.
	DAT_MEM_PRIV_REMOTE_WRITE_FLAG	0x20	Remote write access requested.

lmr_handle:	Handle for the created instance of the LMR.
lmr_context:	Context for the created instance of the LMR to use for DTO local buffers.
rmr_context:	Remote Memory region Context for the created instance of the LMR suitable to be shared with a remote peer.
registered_size:	Actual memory size registered by the Provider.
registered_address:	Actual base address of the memory registered by the Provider.

Description: *dat_lmr_create* registers a memory region with an IA. The specified buffer must have been previously allocated by the uDAPL Consumer on the platform. The Provider must do memory pinning if needed, which includes whatever OS-dependent steps are required to ensure that the memory is available on demand for the Interface Adapter. uDAPL does not require that the memory never be swapped out; just that neither the hardware nor the Consumer ever has to deal with it not being there. The created *lmr_context* can be used for local buffers of DTOs and for binding RMRs, and *lmr_handle* can be used for creating other LMRs. For uDAPL the scope of the *lmr_context* is the address space of the DAT Consumer.

The return values of *registered_size* and *registered_address* indicate to the Consumer how much the contiguous region of Consumer virtual memory was registered by the Provider and where the region starts in the Consumer virtual address.

The *mem_type* parameter indicates to the Provider the kind of memory to be registered, and can take on any of the values defined in Table 13.*pz_* *handle* allows Consumers to restrict local accesses to the registered LMR by DTOs.

DAT_LMR_COOKIE is a pointer to a unique identifier of the shared memory region of the *DAT_MEM_TYPE_SHARED_VIRTUAL* DAT memory type*.* The identifier is an array of 40 bytes allocated by the Consumer. The Provider must check the entire 40 bytes and shall not interpret it as a NULL-terminated string.

The return value of *rmr_context* can be transferred by the local Consumer to a Consumer on a remote host to be used for an RDMA DTO.

If *mem_privileges* does not specify remote Read and Write privileges, *rmr_context* is not generated and NULL is returned. No remote privileges are given for Memory Region unless explicitly asked for by the Consumer.

Consumer can specify what type of virtual addressing to be used for the create LMR. The 0-bazed VA assigns the VA of 0 to the beginning of the registered memory region.

dat_lmr_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_INVALID_HANDLE	Invalid DAT handle.
DAT_INVALID_STATE	Parameter in an invalid state. For example, shared virtual buffer was not created shared by the platform.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider. For example, requested Memory Type was not supported by the Provider.

6.7.2.1.1 USAGE

Consumers can create an LMR over the existing LMR memory with different Protection Zones and privileges using previously created IA translation table entries.

The Consumer shall use *rmr_context* with caution. Once advertised to a remote peer, the *rmr_context* of the LMR cannot be invalidated. The only way to invalidate it is to destroy the LMR *(dat_lmr_free*).

DAT-2.0 had modified the format of *dat_lmr_triplet.* As long as an application used the names of the field and not a position in the data structures a recompile will maintain the application code compatibility.

6.7.2.1.2 RATIONALE

6.7.2.1.3 MODEL IMPLICATIONS

Upon creation of the LMR, the Provider adds the registered region to the IA translation table.

For some transports, like IB, the actual registered memory protection enforced for local and remote accesses for the Memory Region can be different. But the registered memory region requested by the Consumer-enforced protection bounds for the remote access are always within the enforced protection bounds of the actual memory registered for local access. Because the operation only returns a single set of actual registered region boundary, the Provider should return the actual memory boundary registered for remote access. The remote Consumer cannot access the local Consumer memory using *rmr_context* outside the returned *registered_size* and *registered_address*. The Consumer should not bind RMR for the LMR for a memory outside the region defined by returned *registered_size* and *registered_address.*

Note to Provider: Some systems allow applications to map a large memory space (36 or more bits) dynamically into a standard virtual address memory space (such as 32 bits). The user application has the ability to move one or more windows (typically page-sized) anywhere over a larger set of pages that are assigned to the process.

This creates a three-tier addressing structure: the process-specific 32-bit address, the process-specific larger address, and the physical address. DAT only recognizes the first and the last.

However, a DAT Provider can fully support these memory architectures by following a simple rule—always honor the semantics of the user memory map at the time the LMR is registered. Suppose that the application has a single 256-MB window that it is allowed to move to any of three different extended memory banks. The application could register each of the different regions with different LMRs as follows:

```
mapSet(windowPtr,extAddressA);
dat_lmr_create(ia,DAT_MEM_TYPE_VIRTUAL,windowPtr,
    windowSize,pz,privFlags,&lmrhA,&lmrcA,&rmrcA
    &regSize,&regAddr);
mapSet(windowPtr,extAddressB);
dat_lmr_create(ia,DAT_MEM_TYPE_VIRTUAL,windowPtr,
    windowSize,pz,privFlags,&lmrhB,&lmrcB,&rmrcB
    &regSize,&regAddr);
mapSet(windowPtr,extAddressC);
dat_lmr_create(ia,DAT_MEM_TYPE_VIRTUAL,windowPtr,
    windowSize,pz,privFlags,&lmrhC,&lmrcC,&rmrcC
```

```
&regSize,&regAddr);
```

From this point, *LMR B* refers to the memory that was selected after the second mapSet call, whether or not that was how the user's memory was currently mapped.

Even while this memory is not mapped in the user's window, it can be referenced in receive operations, in send operations, or via RMR Contexts for remote accesses. Neither the DAT Provider nor the remote peer care what the application currently maps.

An address specified in an LMR triplet or an RMR triplet is always interpreted in the context of the virtual memory map in operation at the time of the registration.

When the local application receives a completion of a receive that posted a buffer using *LMR B*, it presumably wants to reset its window to that memory. However, it could send the data back out in another request without doing so. The only requirement to ever restore the original mapping is created by the local application's need to access that memory on its own.

6.7.2.2 DAT_LMR_FREE

Synopsis:

```
DAT_RETURN
   dat_lmr_free (
   IN     DAT_LMR_HANDLE     lmr_handle
   )
```

Parameters:

lmr_handle: Handle for an instance of LMR to be destroyed.

Description: *dat_lmr_free* destroys an instance of the LMR. The LMR cannot be destroyed if it is in use by an RMR. The operation does not deallocate the memory region or unpin memory on a host.

It is illegal to use the destroyed handle in any subsequent operation. Any DTO operation that uses the destroyed LMR after the *dat_lmr_free* is completed shall fail and report a protection violation. The use of *rmr_context* of the destroyed LMR by a remote peer for an RDMA DTO results in an error and broken connection on which it was used. Any remote RDMA operation that uses the destroyed LMR *rmr_context*, whose Transport-specific request arrived to the local host after the *dat_lmr_free* has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed LMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the *dat_lmr_free* returns, might or might not complete successfully. If it fails, *DAT_DTO_ERR_*

REMOTE_ACCESS is reported in *DAT_DTO_COMPLETION_STATUS* for the remote RDMA DTO and the connection is broken.

dat_lmr_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *lmr_handle* is invalid.
DAT_INVALID_STATE	Parameter in an invalid state; LMR is in use by an RMR instance.

6.7.2.2.1 USAGE

6.7.2.2.2 RATIONALE

6.7.2.2.3 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.7.2.3 DAT_LMR_QUERY

Synopsis:

```
DAT_RETURN
   dat_lmr_query (
   IN     DAT_LMR_HANDLE          lmr_handle,
   IN     DAT_LMR_PARAM_MASK      lmr_param_mask,
   OUT    DAT_LMR_PARAM           *lmr_param
   )
```

Parameters:

lmr_handle:	Handle for an instance of the LMR.
lmr_param_mask:	Mask for LMR parameters.
lmr_param:	Pointer to a Consumer-allocated structure that the Provider fills with LMR parameters.

Description:

dat_lmr_query provides the Consumer LMR parameters. The Consumer passes in a pointer to the Consumer-allocated structures for LMR parameters that the Provider fills.

lmr_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *lmr_param_mask* requested parameters. The Provider can return values for any other parameters.

dat_lmr_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *lmr_param_ mask* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *lmr_handle* is invalid.

6.7.2.3.1 USAGE

6.7.2.3.2 RATIONALE

6.7.2.3.3 MODEL IMPLICATIONS

6.7.3 REMOTE MEMORY REGION

Consumers that would like to use RMR specific to one connection at a time should use *dat_rmr_create_for_ep*, while RMR whose *rmr_context* can be shared between multiple connections should use *dat_rmr_create.*

6.7.3.1 DAT_RMR_CREATE

Synopsis:

```
DAT_RETURN
   dat_rmr_create(
   IN     DAT_PZ_HANDLE              pz_handle,
   OUT    DAT_RMR_HANDLE             *rmr_handle
   )
```

Parameters:

pz_handle: Handle for an instance of the Protection Zone.

rmr_handle: Handle for the created instance of an RMR.

Description:

dat_rmr_create creates an RMR for the specified Protection Zone. This operation is relatively heavy. The created RMR can be bound to a memory region within the LMR through a lightweight *dat_rmr_bind* operation that generates *rmr_context*.

If the operation fails (does not return DAT_SUCCESS), the return values of *rmr_handle* are undefined and Consumers should not use them.

pz_handle provide Consumers a way to restrict access to an RMR by authorized connection only.

dat_rmr_create is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; pz_handle is invalid.

6.7.3.1.1 USAGE

6.7.3.1.2 RATIONALE

6.7.3.1.3 MODEL IMPLICATIONS

6.7.3.2 DAT_RMR_CREATE_FOR_EP

Synopsis:

```
DAT_RETURN
   dat_rmr_create_for_ep (
   IN     DAT_PZ_HANDLE          pz_handle,
   OUT    DAT_RMR_HANDLE         *rmr_handle
   )
```

Parameters:

pz_handle:	Handle for an instance of the Protection Zone.
rmr_handle:	Handle for the created instance of an RMR.

Description:

dat_rmr_create_for_ep creates an RMR that is specific to a single connection at a time.

This operation is relatively heavy. The created RMR can be bound to a memory region within the LMR through a lightweight *dat_rmr_bind* operation for *EPs* that use the *pz_handle* that generates *rmr_context*.

If the operation fails (does not return DAT_SUCCESS), the return values of *rmr_handle* are undefined and Consumers should not use it.

pz_handle provide Consumers a way to restrict access to an RMR by authorized connections only.

dat_rmr_create_for_ep is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_HANDLE	Invalid DAT handle; *pz_handle* is invalid.

6.7.3.2.1 USAGE

6.7.3.2.2 RATIONALE

6.7.3.2.3 MODEL IMPLICATIONS

Consumers can develop ULP which can adopt to whether Provider supports EP scoped or PZ scoped for RMR protection but not both. But since this requires logic for creating sufficient number of PZs and registering memory specific for them it was decided that there is no benefit for allowing Consumers to specify that they do not care which protection they get for RMR.

6.7.3.3 DAT_RMR_FREE

Synopsis:

```
DAT_RETURN
   dat_rmr_free (
   IN     DAT_RMR_HANDLE     rmr_handle
   )
```

Parameters:

rmr_handle: Handle for an instance of the RMR to be destroyed.

Description:

dat_rmr_free destroys an instance of the RMR.

It is illegal to use the destroyed handle in any subsequent operation. Any remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host after the *dat_rmr_free* has completed, fails and reports a protection violation. Remote RDMA operation that uses the destroyed RMR *rmr_context*, whose Transport-specific request arrived to the local host prior to the *dat_rmr_free* return, might or might not complete successfully. If it fails, *DAT_DTO_ERR_REMOTE_ACCESS* is reported in *DAT_DTO_COMPLETION_STATUS* for the remote RDMA DTO and the connection is broken.

dat_rmr_free is allowed on either bound or unbound RMR. If RMR is bound, *dat_rmr_free* unbinds (free HCA TPT and other resources and whatever else binds with length of 0 should do), and then free RMR.

dat_rmr_free is synchronous and non-thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle; *rmr_handle* is invalid.

6.7.3.3.1 USAGE

6.7.3.3.2 RATIONALE

6.7.3.3.3 MODEL IMPLICATIONS

If Provider detects the use of deleted object handle it should return *DAT_INVALID_HANDLE*. Provider should avoid assigning the used handle as long as possible. Once reassigned the handle is no longer belongs to a destroyed object.

6.7.3.4 DAT_RMR_QUERY

Synopsis:

```
DAT_RETURN
   dat_rmr_query (
   IN     DAT_RMR_HANDLE          rmr_handle,
   IN     DAT_RMR_PARAM_MASK      rmr_param_mask,
   OUT    DAT_RMR_PARAM           *rmr_param
   )
```

Parameters:

rmr_handle:	Handle for an instance of the RMR.
rmr_param_mask:	Mask for RMR parameters.
rmr_param:	Pointer to a Consumer-allocated structure that the Provider fills with RMR parameters.

Description:

dat_rmr_query provides RMR parameters to the Consumer. The Consumer passes in a pointer to the Consumer-allocated structures for RMR parameters that the Provider fills.

rmr_param_mask allows Consumers to specify which parameters to query. The Provider returns values for *rmr_param_mask* requested parameters. The Provider can return values for any other parameters.

Not all parameters can have a value at all times. For example, *mem_privileges*, *rmr_context*, and *lmr_context*, *virtual_address*, *segment_length* of l*mr_triplet* are not defined for an unbounded RMR.

dat_rmr_query is synchronous. Its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter; *rmr_param_mask* is invalid.
DAT_INVALID_HANDLE	Invalid DAT handle; *rmr_handle* is invalid.

6.7.3.4.1 Usage

6.7.3.4.2 Rationale

6.7.3.4.3 Model Implications

There is no modify operation for the RMR. The *dat_rmr_bind* serves this purpose.

6.7.3.5 DAT_RMR_BIND

Synopsis:

```
DAT_RETURN
   dat_rmr_bind(
   IN     DAT_RMR_HANDLE            rmr_handle,
   IN     DAT_LMR_HANDLE            lmr_handle,
   IN     DAT_LMR_TRIPLET           *lmr_triplet,
   IN     DAT_MEM_PRIV_FLAGS        mem_privileges,
   IN     DAT_VA_TYPE               va_type,
   IN     DAT_EP_HANDLE             ep_handle,
   IN     DAT_RMR_COOKIE            user_cookie,
   IN     DAT_COMPLETION_FLAGS      completion_flags,
   OUT    DAT_RMR_CONTEXT           *rmr_context
   )
```

Parameters:

rmr_handle:	Handle for an RMR instance.
lmr_handle:	Handle for an LMR instance that contains the memory for the bind.
lmr_triplet:	A pointer to an *lmr_triplet* that defines the memory region of the LMR.
mem_privileges:	Consumer-requested memory access privileges for the registered remote memory region. The Default value is DAT_MEM_PRIV_NONE_FLAG. The constant value DAT_MEM_PRIV_ALL_FLAG, which specifies both local and remote Read and Write privileges, is also defined. See Table 15 for remote memory privilege definitions.
va_type:	Consumer requested Virtual Addressing for LMR. The value DAT_VA_TYPE_VA requests a process virtual address, while the value of DAT_VA_TYPE_ZB requests virtual addressing that assigns virtual address of 0 to the start of LMR.

Table 15 RMR Remote Memory Privilege Definitions

Privileges	Definition/Bit	Value	Description
Remote Read		0x00	No remote read access requested.
	DAT_MEM_PRIV_REMOTE_READ_FLAG	0x02	Remote read access requested.
Remote Write		0x00	No remote write access requested.
	DAT_MEM_PRIV_REMOTE_WRITE_FLAG	0x20	Remote write access requested.

ep_handle: Endpoint to which *dat_rmr_bind* is posted.

user_cookie: User-provided cookie that is returned to a Consumer at the completion of the *dat_rmr_bind*. Can be NULL.

completion_flags: Flags for RMR Bind. The default *DAT_COMPLETION_DEFAULT_FLAG* is 0 (see Appendix A.4). See Table 16 for flag definitions.

Table 16 RMR Bind Flag Definitions

Features	Definition/Bit	Value	Description	Caveat
Completion Suppression		0x00	Generate Completion.	
	DAT_COMPLETION_SUPPRESS_FLAG	0x01	Suppress successful Completion.	
Notification of Completion		0x00	Notification Completion.	Local Endpoint must be configured for Notification Suppression.
	DAT_COMPLETION_UNSIGNALLED_FLAG	0x04	Non-notification Completion.	
Barrier Fence		0x00	No request for Barrier Fence.	
	DAT_COMPLETION_BARRIER_FENCE_FLAG	0x08	Request for Barrier Fence.	

rmr_context: New *rmr_context* for the bound RMR suitable to be shared with a remote host.

Description: *dat_rmr_bind* binds the RMR to the specified memory region within an LMR and provides the new *rmr_context* value. The *dat_rmr_bind* operation is a lightweight asynchronous operation that generates a new *rmr_context*. The Consumer is notified of the completion of this operation through a *rmr_bind* Completion event on the *request_evd_handle* of the specified Endpoint *ep_handle*.

The return value of *rmr_context* can be transferred by local Consumer to a Consumer on a remote host to be used for an RDMA DTO. The use of *rmr_context* by a remote host for an RDMA DTO prior to the completion of the *dat_rmr_bind* can result in an error and a broken connection. The local Consumer can ensure that the remote Consumer does not have *rmr_context* before *dat_rmr_bind* is completed. One way is to "wait" for the completion *dat_rmr_bind* on the *rmr_bind* Event Dispatcher of the specified Endpoint *ep_handle.* Another way is to send *rmr_context* in a Send DTO over the connection of the Endpoint *ep_handle.* The barrier-fencing behavior of the *dat_rmr_bind* with respect to Send and RDMA DTOs ensures that a Send DTO does not start until *dat_rmr_bind* completed.

For the EP scoped RMR protection the RDMA operation that can use *rmr_context* must be over the connection which is used for *dat_rmr_bind*. That is the RDMA must use the remote EP of the connection of local *ep_handle.*

dat_rmr_bind automatically fences all Send, RDMA Read, and RDMA Write DTOs and *dat_rmr_bind* operations submitted on the Endpoint *ep_handle* after the *dat_rmr_bind.* Therefore, none of these operations starts until *dat_rmr_bind* is completed.

If the RMR Bind fails after *dat_rmr_bind* returns, connection of *ep_handle* is broken. The Endpoint transitions into a *DAT_EP_STATE_DISCONNECTED* state and the *DAT_CONNECTION_EVENT_BROKEN* event is delivered to the *connect_evd_handle* of the Endpoint.

dat_rmr_bind employs fencing to ensure that operations sending the RMR Context on the same Endpoint as the bind specified cannot result in an error from the peer side using the delivered RMR Context too soon. One method, used by InfiniBand, is to ensure that none of these operations start on the Endpoint until after the bind is completed. Other transports can employ different methods to achieve the same goal.

Any RDMA DTO that uses the previous value of *rmr_context* after the *dat_rmr_bind* is completed fail and report a protection violation.

By default, *dat_rmr_bind* generates notification completions.

mem_privileges allows Consumers to restrict the type of remote accesses to the registered RMR by RDMA DTOs. Providers whose underlying Transports require that privileges of the requested RMR and the associated LMR match, that is

- Set RMR's *DAT_MEM_PRIV_REMOTE_READ_FLAG* requires that LMR's *DAT_MEM_PRIV_LOCAL_READ_FLAG* is also set,
- Set RMR's *DAT_MEM_PRIV_REMOTE_WRITE_FLAG* requires that LMR's *DAT_MEM_PRIV_LOCAL_WRITE_FLAG* is also set,

or the operation fails will return DAT_PRIVILEGES_VIOLATION if the *mem_privileges* is not set according to this rule.

Consumer can specify what type of virtual addressing to be used for the create LMR. The 0-bazed VA assigns the VA of 0 to the beginning of the registered memory region.

In the *lmr_triplet,* the value of *length* of zero means that the Consumer does not want to associate an RMR with any memory region within the LMR and the return value of *rmr_context* for that case is undefined.

The completion of the posted RMR Bind is reported to the Consumer asynchronously through a DTO Completion event based on the specified *completion_flags* value. The value of *DAT_COMPLETION _ UNSIGNALLED_FLAG* is only valid if the Endpoint Request Completion Flags *DAT_COMPLETION_UNSIGNALLED_FLAG.* Otherwise, *DAT_ INVALID_PARAMETER* is returned.

user_cookie allows Consumers to have unique identifiers for each *dat_ rmr_bind*. These identifiers are completely under user control and are opaque to the Provider. The Consumer is not required to ensure the uniqueness of the *user_cookie* value*.* The *user_cookie* is returned to the Consumer in the *rmr_bind* Completion event for this operation.

The operation is valid for the Endpoint in the *DAT_EP_STATE_ CONNECTED* and *DAT_EP_STATE_DISCONNECTED* states. If the operation returns successfully for the Endpoint in *DAT_EP_STATE_ DISCONNECTED* state, the posted RMR Bind is immediately flushed to *request_evd_handle.*

dat_rmr_bind is asynchronous. Its thread safety is Provider-dependent. This routine is always thread safe with respect to *dat_ep_post_recv.*

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The operation failed due to resource limitations.
DAT_INVALID_PARAMETER	Invalid parameter. For example, the *target_address* or *segment_length* exceeded the limits of the existing LMR.
DAT_INVALID_HANDLE	Invalid DAT handle.

DAT_INVALID_STATE	Parameter in an invalid state. Endpoint was not in the *DAT_EP_STATE_CONNECTED* or *DAT_EP_STATE_DISCONNECTED* state.
DAT_MODEL_NOT_SUPPORTED	The requested Model was not supported by the Provider.
DAT_PRIVILEGES_VIOLATION	Privileges violation for local or remote memory access.
DAT_PROTECTION_VIOLATION	Protection violation for local or remote memory access.

6.7.3.5.1 Usage

In DAT-2.0 signature of *DAT_RMR_BIND* has been changed. An Consumer will need to add *dat_lmr_handle* compare to the previous versions of the spec.

DAT-2.0 had modified the format of *dat_lmr_triplet* and *dat_rmr_triplet.* As long as an application used the names of the field and not a position in the data structures a recompile will maintain the application code compatibility.

6.7.3.5.2 Rationale

6.7.3.5.3 Model Implications

The *rmr_context* is the OUT parameter only. For the Providers that need that value to support RMR_Bind, they can extract the current value from RMR_Handle themselves.

Consumers do not have control of assigning *rmr_context* values. The values are picked by the Provider/IA for the Consumers, which ensures that they are different from the previous one and ensures that the previous *rmr_context* is invalid.

Consumer should not post a second RMR bind on the same RMR until the first one completes. How Consumer knows that RMR bind has been completed is not specified. Consumer can reap the first RMR bind completion, reap completion of some other operation posted to the same EP Send Queue after the first RMR Bind post or any other method. It is not viewed as an inconvinience for Consumer since the reason RMR bind is done in the first place so that generated RMR_context can be shared with the remote side. So Consumer does not want to issue the second RMR Bind which will invalidate the first RMR bind generated RMR_context and cause RDMA operation from remote side that uses that RMR_context to fail and break the connection.

But this trivial Consumer restriction allows Provider a freedom of implementation with potential performance and/or robustness improvements. This is especially true for EVD resize for EVDs used for SQ and RQ of an EP.

6.7.4 NON-COHERENT MEMORY SUPPORT

The following two operations allow Consumer to synchronize local memory in order to support RDMA operations with non-coherent memory.

6.7.4.1 DAT_LMR_SYNC_RDMA_READ

Synopsis:

```
DAT_RETURN
   dat_lmr_sync_rdma_read (
   IN     DAT_IA_HANDLE            ia_handle,
   IN const     DAT_LMR_TRIPLET    *local_segments,
   IN     DAT_VLEN                 num_segments
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA.
local_segments:	Array of buffer segments.
num_segments	Number of segments in *local_segments* argument.

Description:

dat_lmr_sync_rdma_read makes memory changes visible to an incoming RDMA Read operation. This operation guarantees consistency by locally flushing the non-coherent cache prior to it being retrieved by remote peer RDMA read operation.

The *dat_lmr_sync_rdma_read* is needed only if Provider attribute specifies that this operation is needed prior to incoming RDMA Read operation. Consumer must call *dat_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. *Dat_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.

If Provider attribute specifies that this operation is required, attempts to read from a memory range that is not properly synchronized using *dat_lmr_sync_rdma_read* the result in returned contents that are undefined.

dat_lmr_sync_rdma_read is synchronous, and its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle.
DAT_INVALID_PARAMETER	Invalid parameter. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region, or LMR_handle was invalid.

6.7.4.1.1 USAGE

Determining when an RDMA Read will start and what memory range it will read is the Consumer's responsibility. One possibility is to have the Consumer that is modifying memory call *dat_lmr_sync_rdma_read* and then post a Send DTO message that identifies the range in the body of the Send. The Consumer wishing to do the RDMA Read can receive this message and thus know when it is safe to initiate the RDMA Read operation.

6.7.4.1.2 RATIONALE

This call ensures that the Provider receives a coherent view of the buffer contents upon a subsequent remote RDMA Read operation. After the call completes, the Consumer is assured that all platform-specific buffer and cache updates have been performed, and that the LMR range is consistent with the Provider hardware. Any subsequent write by the Consumer may void this consistency. The Provider is not required to detect such access.

The action performed on the cache before the RDMA Read depends on the cache type.

- I/O noncoherent cache will be invalidated.
- CPU noncoherent cache will be flushed.

6.7.4.1.3 MODEL IMPLICATIONS

6.7.4.2 DAT_LMR_SYNC_RDMA_WRITE

Synopsis:

```
DAT_RETURN
   dat_lmr_sync_rdma_write (
   IN     DAT_IA_HANDLE              ia_handle,
   IN const     DAT_LMR_TRIPLET      *local_segments,
   IN     DAT_VLEN                   num_segments
   )
```

Parameters:

ia_handle:	Handle for an open instance of the IA.

local_segments:	Array of buffer segments.
num_segments	Number of segments in *local_segments* argument.

Description:

dat_lmr_sync_rdma_write makes the effects of an incoming RDMA Write operation visible to Consumer. This operation guarantees consistency by locally invalidating the non-coherent cache whose buffer has been populated by remote peer RDMA write operation.

The *dat_lmr_sync_rdma_write* is needed if and only if Provider attribute specifies that this operation is needed after an incoming RDMA Write operation. Consumer must call *dat_lmr_sync_rdma_write* before reading data from a memory range in this region that was the target of an incoming RDMA Write operation. *dat_lmr_sync_rdma_write* must be called after the RDMA Write operation completes, and the memory range that was modified by the RDMA Write must be supplied by the caller in the *local_segments* array. After this call returns, the Consumer may safely see the modified contents of the memory range. It is permissible to batch synchronizations of multiple RDMA Write 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.

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

If Provider attribute specifies that this operation is needed and the Consumer attempts to read from a memory range in an LMR without properly synchronizing using *dat_lmr_sync_rdma_write*, the returned contents are undefined. If the Consumer attempts to write to a memory range without properly synchronizing, the contents of the memory range become undefined.

dat_lmr_sync_rdma_write is synchronous, and its thread safety is Provider-dependent.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_HANDLE	Invalid DAT handle.
DAT_INVALID_PARAMETER	Invalid parameter. For example, the address range for a local segment fell outside the boundaries of the corresponding Local Memory Region, or LMR_handle was invalid.

6.7.4.2.1 Usage

Determining when an RDMA Write completes and determining which memory range was modified is the Consumer's responsibility. One possibility is for the RDMA Write initiator to post a Send DTO message after each RDMA Write that identifies the range in the body of the Send. The Consumer at the target of the RDMA Write can receive the message and thus know when and how to call *dat_lmr_sync_rdma_write*.

6.7.4.2.2 Rationale

This call ensures that the Provider receives a coherent view of the buffer contents after a subsequent remote RDMA Write operation. After the call completes, the Consumer is assured that all platform-specific buffer and cache updates have been performed, and that the LMR range is consistent with the Provider hardware. Any subsequent read by the Consumer may void this consistency. The Provider is not required to detect such access.

The action performed on the cache after the RDMA Write depends on the cache type.

- I/O noncoherent cache will be flashed the I/O cache.
- CPU noncoherent cache will be invalidated the CPU cache.

6.7.4.2.3 Model Implications

6.8 Completions

6.8.1 Completion Events and Posting Interactions

Completion of posted Send, RDMA Read, RDMA Write, Recv, or RMR Bind operations is returned to the Consumer via DTO or RMR Bind Completion events, respectively. The Consumer can get this event using *dat_evd_dequeue* or *dat_evd_wait.*

Until Completion is reaped by the Consumer, the Request or Recv is still outstanding and occupies an entry on the Request or Recv queue of the Endpoint where it was posted. When the Consumer reaps a completion event, the Request or Recv is no longer outstanding and its entry on the Request or Recv queue of the Endpoint becomes available for another posting. The Successful Completion of a posted DTO or RMR Bind to the Request queue of an Endpoint, which had *DAT_COMPLETION_SUPPRESSION_FLAG* set for *completion_flags,* has no completion event generated for it. Hence, it remains outstanding and occupies an entry on the Request queue of the Endpoint until the completion of a DTO or RMR Bind posted after it to the same Request queue has been reaped.

Unsuccessful DTOs always generate completions. If posted DTO had *DAT_COMPLETION_SUPPRESSION_FLAG* set then unsuccessful completion is generated with Notification. If posted DTO had *DAT_COMPLETION_SUPPRESSION_FLAG* not set then Notification for unsuccessful completion is controlled by setting of the *DAT_*

COMPLETION_UNSIGNALLED_FLAG and for Recv by *DAT_ COMPLETION_SOLICITED_WAIT_FLAG* in addition to *DAT_ COMPLETION_UNSIGNALLED_FLAG*.

The Consumer can dequeue an event from EVD at any time except when there is already a waiter on the EVD. The Consumer controls when a waiter is unblocked by the timeout, to control maximum duration of time for the blocking of a dat_evd_wait and either, but not both, of the following:

1) Threshold of the EVD waiter.

2) Notification Flags of posted DTO or RMR Bind. The notification is controlled locally via *DAT_COMPLETION_UNSIGNALLED_FLAG* of *completion_flags* of a posted DTO or RMR Bind, or remotely for local Recv completion via *DAT_COMPLETION_SOLICITED_WAIT_FLAG* of *completion_flags* of matching Send, but not both at the same time.

When the Consumer controls waiter unblocking via *threshold*, the following table specifies the behavior:

Case	**Completion Suppression of locally posted operation**	**Notification Suppression of locally posted operation**	**Solicited Wait of remote Send**	**Result**
Send, RDMA Read, RDMA Write, RMR Bind	0 or 1	1 - suppress	N/A	Post error - invalid flag value.
Recv	1 - suppress	0 or 1	N/A	Post error - invalid flag value.
Recv	0 - no suppress	1 suppress	N/A	Post error - invalid flag value.
Recv - Successful or Unsuccessful Completion	0 - no suppress	0 - no suppress	0 or 1	Completion IS generated WITH notification. Waiter unblocked only when Threshold is reached.
Send, RDMA Read, RDMA Write, RMR Bind - Successful completion	1 - suppress	0 - no suppress	N/A	Completion NOT generated.

Case	Completion Suppression of locally posted operation	Notification Suppression of locally posted operation	Solicited Wait of remote Send	Result
Send, RDMA Read, RDMA Write, RMR Bind - Unsuccessful completion	0 - no suppress	0 - no suppress	N/A	Completion IS generated WITH notification. Waiter unblocked only when Threshold is reached

If the EP Request Completion Flag is set for Notification Suppression support, the following table specifies the behavior. When a Notification event is generated, the waiter is unblocked.

Case	Completion Suppression of locally posted operation	Notification Suppression of locally posted operation	Solicited Wait of remote Send	Result
Send, RDMA Read, RDMA Write, RMR Bind - Successful completion	1 - suppress	0 or 1	N/A	Completion NOT generated.
Send, RDMA Read, RDMA Write, RMR Bind - Unsuccessful completion	1 - suppress	0 - no suppress	N/A	Completion IS generated WITH notification.
Send, RDMA Read, RDMA Write, RMR Bind - Unsuccessful completion	1 - suppress	1 - suppress	N/A	Completion IS generated WITHOUT notification.

Case	Completion Suppression of locally posted operation	Notification Suppression of locally posted operation	Solicited Wait of remote Send	Result
Send, RDMA Read, RDMA Write, RMR Bind - Unsuccessful completion	0 - no suppress	0 - no suppress	N/A	Completion IS generated WITH notification.
Send, RDMA Read, RDMA Write, RMR Bind - Successful or Unsuccessful completion	0 - no suppress	1 - suppress	N/A	Completion IS generated WITHOUT notification.

If the EP Recv Completion Flag is set for Notification Suppression support, the following table specifies the behavior. When a Notification event is generated, the waiter is unblocked.

Case	Completion Suppression of locally posted operation	Notification Suppression of locally posted operation	Solicited Wait of remote Send	Result
Recv	1 - suppress	0 or 1	N/A	Post error - invalid flag value.
Recv - Successful or Unsuccessful completion	0 - no suppress	1 - suppress	0 or 1	Completion IS generated WITHOUT notification.
Recv - Successful or Unsuccessful completion	0 - no suppress	0 - no suppress	0 or 1	Completion IS generated WITH notification.

If the EP Recv Completion Flag is set for Solicited Wait support, the following table specifies the behavior. When a Notification event is generated, the waiter is unblocked.

Case	Completion Suppression of locally posted operation	Notification Suppression of locally posted operation	Solicited Wait of remote Send	Result
Recv	1 - suppress	0 or 1	N/A	Post error - invalid flag value.
Recv	0 or 1	1 - suppress	N/A	Post error - invalid flag value.
Recv - Successful or Unsuccessful completion	0 - no suppress	0 - no suppress	1- solicited wait	Completion IS generated WITH notification.
Recv - Successful or Unsuccessful completion	0 - no suppress	0 - no suppress	0 - no solicited wait	Completion IS generated WITHOUT notification.

6.8.2 COMPLETION STATUS

Any data transfer operation (that is, Send, Receive, RDMA Read, or RDMA Write) or RMR operation (RMR Bind) returns its completion status asynchronously via an event enqueued on an EVD.

If the completion status is anything other than *DAT_DTO_SUCCESS* for DTOs and *DAT_RMR_BIND_SUCCESS* for RMR operations, the connection to whose local Endpoint it is posted is broken.

The following table enumerates all the allowed values for *DAT_DTO_COMPLETION_STATUS*.

For each value, a description of what that value means and whether a given DTO can return that value for its completion status is shown.

DAT_DTO_COMPLETION_STATUS value	Description	Applicable Operations
DAT_DTO_SUCCESS	The DTO completed successfully.	Send, Receive, RDMA Read, RDMA Write, Send with Invalidate, RDMA Read to RMR

DAT_DTO_COMPLETION_STATUS value	Description	Applicable Operations
DAT_DTO_ERR_LOCAL_LENGTH	The length of the incoming DTO was larger than the *max_message_size* attribute of the Endpoint.	Receive
	The total length of the receive buffer associated with a Receive DTO was too small to hold all the incoming data from a Send DTO.	Receive
	The length of the outgoing DTO was larger than the *max_message_size* attribute of the Endpoint.	Send, Send with Invalidate
	The length of the outgoing DTO was larger than the *max_rdma_size* attribute of the Endpoint.	RDMA Read, RDMA Write, RDMA Read to RMR
DAT_DTO_ERR_LOCAL_EP	An internal local Endpoint consistency error was detected while processing a DTO.	Send, Receive, RDMA Read, RDMA Write, Send with Invalidate, RDMA Read to RMR
DAT_DTO_ERR_LOCAL_PROTECTION	One of the segments in the *local_iov* of 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 *virtual_address* and *segment_length* in the *dat_lmr_triplet* 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, Receive, RDMA Read, RDMA Write, Send with Invalidate, RDMA Read to RMR

DAT_DTO_COMPLETION_STATUS value	Description	Applicable Operations
DAT_DTO_ERR_FLUSHED	The Endpoint entered the DAT_EP_STATE_DISCONNECTED state before processing of the DTO could begin.	Send, Receive, RDMA Read, RDMA Write, Send with Invalidate, RDMA Read to RMR
DAT_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, Send with Invalidate, RDMA Read to RMR
DAT_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 *dat_rmr_context* and the Endpoint that is responding to the RDMA DTO operation, an attempt being made to do an RDMA Read or Write using an *dat_rmr_context* 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, RDMA Read to RMR, Send with Invalidate
DAT_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 a DAT_DTO_ERR_LOCAL_EP error to be returned.	Send, RDMA Read, RDMA Write, Send with Invalidate, RDMA Read to RMR

DAT_DTO_COMPLETION_STATUS value	Description	Applicable Operations
DAT_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, Send with Invalidate, RDMA Read to RMR
DAT_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, Send with Invalidate, RDMA Read to RMR
DAT_DTO_ERR_PARTIAL_PACKET	The data delivered by the Receive DTO was truncated. The contents of the receiver's buffer are unspecified.	Receive

As guaranteed by requirements (see Section g on page 41, Section h on page 41, Section i on page 41), the *DAT_SUCCESS* only guarantees that the local buffer can reused. The local buffer may be reused earlier, dependent on the value of the Provider attribute *iov_ownership_on_return.* For RDMA Read it means that the requested data is in the local buffer. For RMR Bind it means that RMR is bound to the requested memory. For Send and RDMA Write it does NOT mean that data have reached the remote peer. For Recv it means that data have been received into the local Recv buffer.

Consumer should not rely on the asynchronous return values defined in the table above. Depending on the underlying RDMA Transport and the Provider implementation these values may be returned. Consumer can rely on these values defined for RDMA Read, Recv, and RMR Bind being returned. For Send and RDMA Write Consumer can check the Provider attribute *dat_async_return_guarantee* to determine if Provider guarantees the defined return values for Send and RDMA Write.

The following table enumerates all the allowed values for *DAT_RMR_BIND_COMPLETION_STATUS*. For each value, a description of what that value means and whether an RMR Bind operation can return that value.

DAT_DTO_COMPLETION_STATUS value	Description
DAT_DTO_SUCCESS	The DTO completed successfully.
DAT_DTO_ERR_LOCAL_EP	An internal local Endpoint consistency error was detected while processing a DTO.
DAT_DTO_ERR_FLUSHED	The Endpoint entered the DAT_EP_STATE_DISCONNECTED state before processing of the DTO could begin.
DAT_RMR_OPERATION_FAILED	An RMR operation failed due to a protection violation. Possible causes for this error include the LMR specified in the call was invalid, the range specified by the *virtual_address* and *segment_length* in the *dat_lmr_triplet* in the call was outside the bounds of the LMR, the Protection Zones associated with the LMR, RMR, and Endpoint to which the RMR Bind 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.

6.8.2.1 USAGE

DAT only provides minimal guarantees as to what successful completion of a Send or RDMA Write operation means to the Consumer:

- The Consumer's local buffer for the request is no longer required by the Provider. It may be released, re-used or altered.
- Barring a transport or remote host error the message/data will be delivered to the remote peer at some time in the future or it may be delivered already. The delivery will follow all the normal ordering rules.

There are several things that are NOT guaranteed by DAT universally, but which are guaranteed on some RDMA transports (such as InfiniBand).

- The payload is not guaranteed to already have been delivered to remote peer memory.
- The target memory access requested is not guaranteed to have been validated. An invalid matching Recv for Send or remote RDMA Write request may not have been detected remotely until after the Send or RDMA Write operation has been completed successfully

locally. The Send matching Recv is only guaranteed to be validated before the destination's Recv completion. The RDMA Write target buffer is guaranteed to be validated prior to completion of remote Recv matching Send that was posted after the RDMA Write.

Provider may provide such guarantee, for example, because the underlying RDMA Transport is InfiniBand that provides such a guarantee. Consumer should check the Provider attribute for *dto_async_return_guarantee* to see if the Provider makes the above additional guarantee.

Below are examples on how Consumer can design ULP to be transport independent.

The first example is for Send. The sender must have a method to ensure that a send is not transmitted until after a prior send has been received and completed at the destination.

The second example is for RDMA Write. One method is based on a Send after RDMA Write that is part of the a round-trip ping to the peer ULP after successful Recv of the Send. The other method is using an RDMA Read that uses RDMA Write RMR after RDMA Write to take advantage of RDMA Transport round-trip for RDMA Read. The RDMA Read can be of size 0.

A peer ULP ping is accomplished by the peers exchanging send messages. The Data Sink cannot receive the completion of the ping message until after all prior RDMA Writes have been properly placed. Therefore when the Data Source receives the ping reply message it knows that all of the data sent has been received by the Data Sink.

An RDMA Read can be used to order writes even without requiring interaction with the peer for each ordering guarantee.

The RDMA Read will not be replied by the remote side of the connection until all prior RDMA Writes on the same connection have been placed. So placing an RDMA Read between RDMA Writes can guarantee that the second write will update target memory only after the first RDMA write's updates have completed. The second RDMA Write must set *DAT_COMPLETION_BARRIER_FENCE_FLAG* to ensure that RDMA Write will not start until the previous RDMA Read has completed.

Consider the following sequence:

```
RDMA Write (rmr)
RDMA Read (rmr)
RDMA Write (barrier_fence)
RDMA Read
```

By the time first RDMA Read is completed the RDMA Write data has been placed into the target buffer. But the remote peer may not be aware of it. In order to notify the peer the Send can be used after the first RDMA Read. Then successful completion of the Send matching Recv guarantee that

RDMA Write data have been delivered into target buffer. The second RDMA Read is need for the second RDMA Write. The second RDMA Write can use the same RMR as the first RDMA Write.

Using an RDMA Read ping, however, cannot guarantee that the peer Data Sink will see both updates. The RDMA Read Reply can be generated before the Data Sink ULP has noted the first updated memory. This is true even if the Data Source posted a Send message between the RDMA Writes.

Consider the following sequence:

```
RDMA Write
Send
RDMA Read
RDMA Write
Send
```

By the time the Data Sink reaps the completion matching the first Send it knows that the first RDMA Write was fully placed. But the Data Sink does not know that it is processing the completion before any of second RDMA Write has been placed.

This is not merely a race condition. Portions of second RDMA Write may have been placed before first Send was completed.

6.8.2.2 Completion Status Transport Mappings

Note to Provider: Following are mappings of the Completion Status values to the Transport-defined entities.

For the InfiniBand transport, the following table maps the values in the *DAT_DTO_COMPLETION_STATUS* and *DAT_RMR_BIND_COMPLETION_STATUS* enumeration to their corresponding "Completion Return Status" values, as specified in Volume 1, Chapter 11 of the InfiniBand specification.

DAT_DTO_COMPLETION_STATUS and DAT_RMR_BIND_COMPLETION_STATUS value	IB "Completion Return Status" Name
DAT_DTO_SUCCESS	Success
DAT_DTO_ERR_LOCAL_LENGTH	Local Length Error
DAT_DTO_ERR_LOCAL_EP	Local QP Operation Error
DAT_DTO_ERR_LOCAL_PROTECTION	Local Protection Error
DAT_DTO_ERR_FLUSHED	Work Request Flushed Error
DAT_DTO_ERR_BAD_RESPONSE	Bad Response Error
DAT_DTO_ERR_REMOTE_ACCESS	Remote Access Error

DAT_DTO_COMPLETION_STATUS and DAT_RMR_BIND_COMPLETION_STATUS value	IB “Completion Return Status” Name
DAT_DTO_ERR_REMOTE_RESPONDER	Remote Operation Error, or Remore Invalid Request Error
DAT_DTO_ERR_TRANSPORT	Transport Retry Counter Exceeded
DAT_DTO_ERR_RECEIVER_NOT_READY	RNR Retry Counter Exceeded, or RNR NAK
DAT_DTO_ERR_PARTIAL_PACKET	(Not applicable to the IB transport.)
DAT_RMR_OPERATION_FAILED	Memory Window Bind Error

For the VIA transport, the following table maps the values in the *DAT_DTO_COMPLETION_STATUS* and *DAT_RMR_BIND_COMPLETION_STATUS* enumeration to their corresponding bits in the Descriptor Control Segment “Status” field, as documented in the Appendix of the VIA specification.

DAT_DTO_COMPLETION_STATUS and DAT_RMR_BIND_COMPLETION_STATUS value	VIA “Status Bit” Name
DAT_DTO_SUCCESS	Done
DAT_DTO_ERR_LOCAL_LENGTH	Local Length Error
DAT_DTO_ERR_LOCAL_EP	Local Format Error
DAT_DTO_ERR_LOCAL_PROTECTION	Local Protection Error
DAT_DTO_ERR_FLUSHED	Descriptor Flushed
DAT_DTO_ERR_BAD_RESPONSE	(Not applicable to the VIA transport.)
DAT_DTO_ERR_REMOTE_ACCESS	RDMA Protection Error
DAT_DTO_ERR_REMOTE_RESPONDER	(Not applicable to the VIA transport.)
DAT_DTO_ERR_TRANSPORT	Transport Error
DAT_DTO_ERR_RECEIVER_NOT_READY	(Not applicable to the VIA transport.)
DAT_DTO_ERR_PARTIAL_PACKET	Partial Packet Error
DAT_RMR_OPERATION_FAILED	(There is no operation corresponding to an RMR Bind 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.)

6.9 Operating System Specific Notes

This section addresses portions of the specification that are operating system specific. Note that this section should not be taken as advice to Providers for other operating systems to solve the problem in similar ways. Specifically, providers on operating systems other than Windows® or Unix® should feel free NOT to make distinctions between execution contexts in general and signal/event-handling execution contexts in particular.

6.9.1 Unix® Operating System Specific Notes

Under Unix®, if a signal is handled by a thread while that thread is blocked in *dat_evd_wait* or *dat_cno_wait*, that DAT call returns with an error of *DAT_INTERRUPTED_CALL*. Note that applications should not rely on this return code always occurring; if the signal arrives during the prologue of the subroutine before the thread actually blocks, the fact that a signal has arrived might be lost.

Consumers can unblock a waiting thread by posting *dat_evd_post_se* to the EVD. This platform-independent method ensures that there is a notification event on the EVD queue. If there is a race between *dat_evd_wait* and *dat_evd_post_se,* since there is a notification event on the EVD, the waiter will either be unblocked or it would not block at all. This is subject to *threshold* value specified. In the worst case, Consumer may need to post *threshold* number of SEs. The EVD must be configured to support SE event stream. For a platform independent way of unblocking *dat_cno_wait see* *Section 6.3.2.4.1, “Usage,” on page 122**.*

Signal handlers are considered a nonstandard execution context within the framework of the uDAPL specification. The Consumer may invoke *dat_ia_close* to recover DAT resources as described in section 5.1 Local Resource Model 11) on page 34. Provider implementations are not expected to be fully reentrant but are required to correctly implement the *dat_ia_close* return codes described in section 6.2.1.2 DAT_IA_Close on page 88. The quality of the Provider's implementation will determine the precise set of scenarios where the provider will be able to successfully close the IA. Semantics of all other uDAPL calls within the scope of a UNIX® signal handler is undefined.

6.9.2 Windows Operating System Specific Notes

APC's (asynchronous procedure calls) and SEH (structured exception handlers) are considered nonstandard execution contexts within the framework of the uDAPL specification. Calling *dat_evd_post_se* and *dat_ia_close* are explicitly supported within both APCs and SEH's. Provider implementations are not expected to be fully reentrant but are required to correctly implement the *dat_ia_close* return codes described in section 6.2.1.2 DAT_IA_Close on page 88. The quality of the Provider's implementation will determine the precise set of scenarios where the

Provider will be able to successfully close the IA. Semantics of all other uDAPL calls within the scope of an Windows® APC or SEH is undefined.

CHAPTER 7: ERROR HANDLING

The *DAT_RETURN* is a 3-tuple. The major status codes (*DAT_RETURN_TYPE*) occupy the upper 16 bits of the 32-bit field. The detailed status code (*DAT_RETURN_SUBTYPE*) occupies the lower 16 bits. The upper 2 bits of the major status code specify the class of the return. DAT defines only three classes: *DAT_CLASS_SUCCESS, DAT_CLASS_ERROR*, and *DAT_CLASS_WARNING*.

The major status code for both *DAT_CLASS_SUCCESS* and *DAT_CLASS_WARNING* is always success. The warning can provide additional information, for example, some resource is exhausted.

DAT supports a single error return definition per process (per registry) and does not provide support for error code definitions per Provider.

If an error occurs, the Provider can return any applicable error. There is no guarantee on which error among several possible errors is returned.

For the definitions of the Major and Minor status codes, see error header file (Section A.5, "Generic Status Codes," on page 395).

The *dat_strerror* operation defined below converts *DAT_RETURN* into human readable strings.

7.1 DAT_STRERROR

Synopsis:

```
DAT_RETURN
   dat_strerror(
   IN     DAT_RETURN              return,
   OUT    const char              **major_message,
   OUT    const char              **minor_message,
   )
```

Parameters:

return:	DAT function return value.
major_message:	A pointer to a character string for the return major status code.
minor_message:	A pointer to a character string for the return detailed status code.

Description: The *dat_strerror* function converts a DAT return code into human readable strings. The *major_message* is *a* string-converted *DAT_TYPE_STATUS,* while *minor_message* is a string-converted *DAT_SUBTYPE_STATUS.* If the return of this function is not *DAT_SUCCESS*, the values of *major_message* and *minor_message* are not defined.

If an undefined *DAT_RETURN* value was passed as the *return* parameter, the operation fails with *DAT_INVALID_PARAMETER* returned. The operation succeeds when *DAT_SUCCESS* is passed in as the *return* parameter.

Note for Provider: Implementation can allocate two static tables with a string message for each major and minor return value, respectively; it can use *dat_return* as an index into them.

Note to Provider: This API must be implemented in the same library as the Registry APIs. It is shared between all Providers on the platform.

Note for Consumer: The string major and minor messages for each *return* value is implementation-dependent and Consumers should not rely on it to be the same for each Provider.

This operation is nonblocking, synchronous, and thread-safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter. The *return* value is invalid.

7.1.1 USAGE

7.1.2 RATIONALE

Splitting, the message between major and minor parts allows greater flexibility for implementation for memory allocation. The Provider can create two separate tables, one for type and one for subtype. This puts the memory requirements for the type and subtype strings as linear in the number of types and subtypes, rather than quadratic for all possible combinations of types and subtypes.

One or both string arguments can be NULL.

7.1.3 MODEL IMPLICATIONS

CHAPTER 8: uDAPL PROVIDER MANAGEMENT

This chapter defines Interface Adapter enumeration and DAT Provider management, as well as provides an example of Static Registry for RedHat RPM and Windows.

This chapter also defines the DAT registry responsibilities and requirements. The registry maps Interface Adapter names to Provider libraries. It allows DAT Providers to register their libraries dynamically. It also allows Consumers to open Interface Adapters by name and to enumerate available Providers along with their library attributes. The dynamic registry is instantiated at most once per address space, no matter how many different Providers are in use.

8.1 OVERVIEW

8.1.1 INTERFACE ADAPTER

An Interface Adapter is an identified interface to an external network available to a Consumer. It is identified by a printable text name that is unique to the local host. As with all host-wide resources, the choice of Interface Adapter names (*ia_name*) is under the control of the system administrator.

Typically, an Interface Adapter is a physical port or set of ports on an HCA or NIC. However, there is considerable flexibility in defining the mapping. Multiple physical ports could be declared as a single Interface Adapter. For example, this allows a Provider to provide automatic path migration to a Consumer without Consumer involvement. Multiple IAs could be declared for the same port for a variety of reasons. For example, a different IA can be defined for each P-Key or VLAN.

8.1.2 PROVIDER MULTIPLE LIBRARIES

Each Provider can have multiple versions of itself installed on a system. Only a single version of a Provider can be loaded within an address space for any given IA name. The system administrator specifies a library for each IA name that is opened by *dat_ia_open*. Consumers can use *dat_registry_list_providers* to examine the IA names and the attributes of the available Providers, if they want to determine whether an appropriate Provider is available before they call *dat_ia_open.* If they need a library with different attributes, they can request the system administrator to provide the version with their needed attributes in the static registry. Alternatively, they can install the version themselves in the static registry if they have the appropriate privileges.

At most one library with a given name may be loaded in a single address space. However, different libraries for the same Interface Adapter can be loaded concurrently in different address spaces. For example, one application may require thread safety, while another does not. It is the Provider's responsibility to coordinate the resource usage among their open library instances.

In almost all cases, the Provider libraries instantiated in each Consumer address space communicate with a single system-wide Provider Driver that is the actual owner of the hardware resource. The Provider Driver is documented here solely to clarify that Provider-specific code can interact with the static registry and transport-specific naming and routing services prior to the Provider library being loaded.

The division of responsibilities between the Provider library and any system-wide supporting software is totally Provider-dependent. A Provider may have each library interact directly with the hardware whenever its design allows.

8.1.3 PROVIDER POLYMORPHISM

A major feature of the DAT API is that a Consumer can use a DAT handle (type *DAT_HANDLE)* without having to know which Provider issued it. With the exception of *dat_ia_open* and *dat_ia_close,* the methods defined by this specification are not found in a symbol map. They are actually macros that invoke the correct Provider-specific routine through a Provider-supplied table. The *dat_ia_open* and *dat_ia_close* symbols are defined in the registration code, not in the Provider library. *dat_ia_open* and *dat_ia_close* call the Provider-specific open and close routines specified in the *dat_provider* structure as part of their implementation.

DAT requires that every *dat_handle* created by a Provider has a *dat_provider* pointer as its first field. This mechanism allows DAT to provide multi-Provider polymorphism.

DAT Consumers must not attempt to access the *dat_provider* structure or to bypass any DAT-supplied Provider routing. DAT Providers may assume that their methods will be invoked only with their objects. They are allowed to be more cautious in accepting parameters, but that is a per-Provider implementation decision.

The size of any DAT object is known only by the Provider that created it and are opaque to a Consumer. DAT objects may be destroyed only by that Provider. The Consumer only has *dat_handle* to refer to Provider DAT objects.

Once an Interface Adapter is open, the generated *dat_handle* for the Provider allows the Consumer to interact directly with the Provider library. The Dynamic Registry does not place itself in the performance path on a per-call basis.

8.1.4 Registry Implementation

The DAT Collaborative provides the sole implementation of the Registry for use with all Providers per platform. It can be downloaded from http://www.datcollaborative.org/registry.html.

The Provider MUST NOT install the Registry directly. Instead, the Provider should point the system administrator to the Web site above as the source for the latest version of the Registry for the platform. Provider Installation should report back if the existing platform Registry version, if it exists, works for the Installed Provider. If not, the Provider shall report what minimal version of Registry it requires. If the Provider has a later version of the Registry as part of its package, it can notify installer about it along with the guidelines about how to install it safely. These guidelines must be consistent with the Registry installation guidelines at www.datcollaborative.org/registry.html for the platform.

Providers can include a copy of the latest Registry implementation for the operating systems they support within their packaging for the convenience of system administrators, display the version included in their package versus the version currently installed, recommend a minimum version that should be installed, or provide separate installation instructions for updating the Registry implementation. But the administrator has absolute control over which version is installed, and must not be left without a Registry installed when any given Provider is uninstalled.

8.2 Registry APIs

Each DAT Provider must inform the DAT registration library of its identity and provide pointers to the functions that implement its methods. The DAT Registry supports at least five services: *dat_ia_open, dat_ia_close, dat_registry_add_provider, dat_registry_remove_provider,* and *dat_ registry_list_providers.*

DAT registration library should support registration of multiple Providers. As a rule of thumb it should support a double digit number of registered Providers.

dat_ia_open and *dat_ia_close* perform the uDAPL/kDAPL-specific open and close, as well as DAT registry-specific semantics. For example, these registry semantics can do reference counting. The Provider cannot be added to the registry multiple times, and it cannot be removed from the registry while it is in use (even during *dat_ia_close*).

8.2.1 DAT_PROVIDER STRUCTURE

The *dat_provider* structure has the following fields:

device_name	The name of the Interface Adapter that the Provider wants to provide. This typically follows Host OS conventions for device names. This is the same as the DAT_PROVIDER_INFO member *ia_name*, and the *dat_ia_open* argument *ia_name_ptr*.
extension	A void pointer that the Provider can use for its own data or method extensions. This is particularly useful when the same routines are registered as multiple Providers.
ia_open_func	The routine that is invoked by the registration code of the *dat_ia_open()*, specifying as a parameter the *ia_name*. A Consumer specifies the *ia_name*. The open call is directed to the correct Provider library if it can be found. Otherwise the registry returns a *DAT_INVALID_PARAMETER* error that is reported to the Consumer as *DAT_NAME_NOT_FOUND*. Because this function is used by user and kernel Consumers, it is specified in the kDAPL and uDAPL API documents.
ia_close_func	The routine that is invoked by the registration code of the *dat_ia_close()*. It ensures that any registry-specific functionality, as well as the Provider functionality, is performed.
other functions	All other function pointers for each uDAPL/kDAPL defined functions. These exist only as function pointers in the Provider table and not as symbols in kernel space.

All *dat_handle*s created by the Provider must point to a structure that begins with a *dat_provider* pointer. All other fields are at the discretion of each Provider's implementation.

8.2.2 CONSUMER EXPOSED APIS

8.2.2.1 DAT_REGISTRY_LIST_PROVIDERS

Synopsis

```
typedef struct dat_provider_info {
   char ia_name[DAT_NAME_MAX_LENGTH];
   DAT_UINT32        dapl_version_major;
   DAT_UINT32        dapl_version_minor;
   DAT_BOOLEAN       is_thread_safe;
} DAT_PROVIDER_INFO;
```

DAT_NAME_MAX_LENGTH includes the NULL character for string termination.

```
DAT_RETURN
   dat_registry_list_providers (
```

```
        IN    DAT_COUNT          max_to_return,
        OUT   DAT_COUNT          *number_entries,
        OUT   DAT_PROVIDER_INFO  *(dat_provider_list[])
    )
```

Parameters:

max_to_return: Maximum number of entries that can be returned to the Consumer in the *dat_provider_list*.

number_entries: The actual number of entries returned to the Consumer in the *dat_provider_list* if successful or the number of Providers available.

dat_provider_list: Points to an array of *DAT_PROVIDER_INFO* pointers supplied by the Consumer. Each Provider's information will be copied to the destination specified.

Description: The Consumer obtains a list of available Providers from the Static Registry. The information provided is the Interface Adapter name, the uDAPL/kDAPL API version supported, and whether the provided version is thread-safe. The Consumer can examine the attributes to determine which (if any) Interface Adapters it wants to open. This operation has no effect on the Registry itself.

The Registry can open an IA using a Provider whose *dapl_version_minor* is larger than the one the Consumer requests if no Provider entry matches exactly. Therefore, Consumers should expect that an IA can be opened successfully as long as at least one Provider entry returned by *dat_registry_list_providers* matches the *ia_name*, *dapl_version_major*, and *is_thread_safe* fields exactly, and has a *dapl_version_minor* that is equal to or greater than the version requested.

If the operation is successful the returned code is *DAT_SUCCESS*, then *number_entries* indicates the number of entries filled by the registry in *dat_provider_list*.

If the operation is not successful, then *number_entries* returns the number of entries in the registry. Consumers can use this return to allocate *dat_provider_list* large enough for the registry entries. This number is just a snapshot at the time of the call and may be changed by the time of the next call. If the operation is not successful, then the content of *dat_provider_list* is not defined.

If *dat_provider_list* is too small, including pointing to NULL for the registry entries, then the operation fails with the return *DAT_INVALID_PARAMETER*.

dat_registry_list_providers is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter. For example, *dat_provider_list* is too small or *dat_provider_list* NULL.
DAT_INTERNAL_ERROR	Internal error. The DAT static registry is missing.

8.2.2.2 DAT_REGISTRY_PROVIDERS_RELATED

Synopsis

```
DAT_RETURN
   dat_registry_providers_related (
         IN     const DAT_NAME_PTR      ia1_name_ptr,
         IN     const DAT_NAME_PTR      ia2_name_ptr,
         OUT    DAT_HA_RELATIONSHIP     *relationship)
   )
```

Parameters:

ia1_name_ptr:	Name of one IA.
ia2_name_ptr:	Name of another IA.
relationship:	Indicator of relationship between 2 IAs. DAT_HA_FALSE indicates that the two IAs are not related, DAT_HA_TRUE indicate that the two IAs are related, DAT_HA_CONFLICTING indicates that 2 IAs do not agree on the relationship, and DAT_HA_UNKNOWN indicating that Registry can not find the answer, for example, one of the Providers do not provide the underpinning to support this operation.

Description:

The *dat_registry_providers_related* let Consumer know whether two IAs share HW resources.

dat_registry_providers_related is synchronous and thread safe.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter. One of the IAs is unknown.
DAT_INTERNAL_ERROR	Internal error. The DAT static registry is missing.

8.2.2.2.1 Usage

The operations allows Consumer to detect whether two IAs share unrelying resources. This allows Consumer to build their own High Availability over available IAs.

It is recommended that Consumers do not use their own HA when IA_names are not related.

8.2.2.2.2 Rationale

8.2.2.2.3 Model Implications

DAT Registry ask each Provider for each IA if they are related. If they both report the same thing the TRUE or FALSE answer is returned. If one of them does not support this operation UNKNOWN is return. If Provider disagree on the relationship CONFLICTING is returned to Consumer. This may happen because the later Provider is aware of the relationship but the earlier one is not.

DAT Registry expects each Provider to support internal DAT_PROVIDER_HA_RELATED call.

8.2.3 Consumer Nonexposed APIs

8.2.3.1 DAT_Registry_Add_Provider

Synopsis

```
DAT_RETURN
   dat_registry_add_provider (
         IN    const DAT_PROVIDER      *provider,
         IN    const DAT_PROVIDER_INFO *provider_info
   )
```

Parameters:

provider:	Self-description of a Provider.
provider_info	Attributes of the Provider.

Description:

The Provider declares itself with the Dynamic Registry. Each registration provides an Interface Adapter to DAT. Each Provider must have a unique name.

The same IA Name cannot be added multiple times. An attempt to register the same IA Name again results in an error with the return code *DAT_PROVIDER_ALREADY_REGISTERED.*

The contents of *provider_info* must be the same as those the Consumer uses in the call to *dat_ia_openv* directly, or the ones provided indirectly defined by the header files the Consumer compiled with.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INSUFFICIENT_RESOURCES	The maximum number of Providers was already registered.
DAT_INVALID_PARAMETER	Invalid parameter.
DAT_PROVIDER_ALREADY_REGISTERED	Invalid or nonunique name.

8.2.3.2 DAT_REGISTRY_REMOVE_PROVIDER

Synopsis

```
DAT_RETURN
   dat_registry_remove_provider  (
         IN    const DAT_PROVIDER*provider
   )
```

Parameters:

provider: Self-description of a Provider.

Description:

The Provider unregisters itself from the Dynamic Registry. It is the Provider’s responsibility to complete its sessions. Removal of the registration only prevents new sessions.

The Provider cannot be removed while it is in use. An attempt to remove the Provider while it is in use results in an error with the return code *DAT_PROVIDER_IN_USE*.

Returns:

DAT_SUCCESS	The operation was successful.
DAT_INVALID_PARAMETER	Invalid parameter. The Provider was not found.
DAT_PROVIDER_IN_USE	The Provider was in use.

8.2.4 PROVIDER-SUPPLIED APIS

The following APIs must be implemented by a Provider that supports being loaded on demand (see Section 8.4.2, “Load on Demand,” on page 319).

The entry points for these APIs must be exported as public symbols so the Registry can locate and call them after loading the Provider library but before the Provider calls *dat_registry_add_provider* to register its routing table.

A Provider Library SHOULD NOT export any other symbols. Multiple Provider Libraries will be loaded into the same address space, possibly even multiple versions of the same Provider. Exporting any other symbols that are not within a fully qualified namespace is likely to result in symbol collisions.

8.2.4.1 DAT_PROVIDER_INIT

Synopsis

```
void
   dat_provider_init (
         IN    const DAT_PROVIDER_INFO *provider_info,
         IN    const char *            instance_data
   )
```

Parameters:

provider_info:	The information that was provided by the Consumer to locate the Provider in the Static Registry.
instance_data:	The instance data string obtained from the entry found in the Static Registry for the Provider.

Description:

A constructor the Registry calls on a Provider before the first call to *dat_ia_open* for a given IA name when the Provider is auto-loaded. An application that explicitly loads a Provider on its own can choose to use *dat_provider_init* just as the Registry would have done for an auto-loaded Provider.

The Provider's implementation of this method must call *dat_registry_add_provider*, using the IA name in the provider_info.ia_name field, to register itself with the Dynamic Registry. The implementation must not register other IA names at this time. Otherwise, the Provider is free to perform any initialization it finds useful within this method.

This method is called before the first call to *dat_ia_open* for a given IA name after one of the following has occurred:

- The Provider library was loaded into memory.
- The Registry called *dat_provider_fini* for that IA name.
- The Provider called *dat_registry_remove_provider* for that IA name (but it is still the Provider indicated in the Static Registry).

If this method fails, it should ensure that it does not leave its entry in the Dynamic Registry.

Returns:

None.

8.2.4.2 DAT_PROVIDER_FINI

Synopsis

```
void
   dat_provider_fini (
         IN    const DAT_PROVIDER_INFO *provider_info
   )
```

Parameters:

provider_info:	The information that was provided when *dat_provider_init* was called.

Description: A destructor the Registry calls on a Provider before it disassociates the Provider from a given IA name.

The Provider can use this method to undo any initialization it performed when *dat_provider_init* was called for the same IA name. The Provider's implementation of this method should call *dat_registry_remove_provider* to unregister its IA Name. If it does not, the Registry might remove the entry itself.

This method can be called for a given IA name at any time after all open instances of that IA are closed, and is certainly called before the Registry unloads the Provider library. However, it is not called more than once without an intervening call to *dat_provider_init* for that IA name.

Returns: None.

8.3 DAT.H API VERSION AND THREAD SAFETY AUTO SUPPORT

8.3.1 COMPILE TIME API VERSION SUPPORT

The dat.h include file shall include #defines for DAT_VERSION_MAJOR and DAT_VERSION_MINOR. These are constants of the same semantics as the *dat_version_major* and *dat_minor_version* fields in the *DAT_PROVIDER_INFO* struct.

8.3.2 THREAD SAFETY SUPPORT

To suppress the need for threadsafe code, the Consumer should add the following before including dat.h:

- #define DAT_THREADSAFE DAT_FALSE

The Consumer can set *DAT_THREADSAFE* in source files, before including any DAT header files, or set it as part of the project's build environment.

8.3.3 Version Support for IA open

A Registry method is defined to accept versioned open calls: *dat_ia_openv*.

dat.h includes the following macro:

```
#define dat_ia_open(name,qlen,async_evd,iah) \
    dat_ia_openv((name),(qlen),(asynch_evd),(iah),\
        DAT_VERSION_MAJOR,DAT_VERSION_MINOR,DAT_THREADSAFE)
```

Consumers still invoke *dat_ia_open* as currently defined. The macro automatically adds the correct version numbers and thread safety flag.

The registry library provides a *dat_ia_open* function as follows:

```
#undef dat_ia_open
DAT_RETURN dat_ia_open (
    IN const DAT_NAME_PTR ia_name_ptr,
    IN DAT_COUNT async_evd_min_qlen,
    INOUT DAT_EVD_HANDLE *async_evd_handle,
    OUT DAT_IA_HANDLE *ia_handle)
{
    return dat_ia_openv(ia_name_ptr,async_evd_min_qlen,
                        async_evd_handle,ia_handle,1,0,
                        DAT_TRUE);
}
```

So if the Consumer directly calls *dat_ia_open*, it means that it has the original “dat.h” file without a *dat_ia_open* #define. Hence, they are using a thread-safe version 1.0.

8.4 Provider Registry Guidelines

8.4.1 Provider Installation Advice

Typically, installation of a DAT Provider requires installation of multiple files and potentially an update of the Static Registry.

The files to be installed can include:

- Dynamically loadable kDAPL and/or uDAPL Provider libraries
- Supporting dynamically loadable libraries
- Statically linkable kDAPL and/or uDAPL Provider libraries
- Supporting statically linked libraries
- The Host OS specific implementation of the Registry
- Supporting executables (a Communications Manager, or other supporting daemons, for example)

- Kernel device drivers

These files should be installed in accordance with normal Host Operating System conventions for these types of files. Existing facilities to track inter-component dependencies, such as Redhat Package Manager (RPM) and Windows Registry, should be used when they are part of the existing infrastructure.

The Host OS-specific implementation of the registry might need to be installed or updated. For each Host OS, there is a designated name and location.

The installation procedure must allow the system configuration to specify the IA Name and location for files to be installed. A Provider should provide defaults for these values.

The installation procedure should be fully logged according to existing Host OS conventions. An uninstall procedure should be provided, if this is not already part of the existing conventions.

The Provider should be aware that any system administrator is allowed to edit the Static Registry at any time. The Provider *must not* assume that its Static Registry entries are unchanged since the Provider's install script was executed.

Editing of the Static Registry can be done at installation time or when the device is loaded at run-time. The latter approach is best suited to support plug-and-play. Editing of the Static Registry should follow normal procedures for the host OS, but should be user-friendly. The install script should edit the Registry with permission of the user, not ask the user to edit the Static Registry.

Each entry in the Static Registry identifies an Interface Adapter name, the library that must be loaded, and instance data. The instance data serves many purposes: identifying the driver level resources to be used, specifying options such as IB Partition Keys or VLANs, and other load-time options.

Except as directed by the installer, the install script should not remove prior versions of the Provider library. Nor should those entries be removed from the Static Registry, although installation can certainly designate a new default version.

The installation procedure should include an option to install the new version *without* designating it as the new default.

It is important to remember that the Registry operates within a user process. Device drivers and supporting daemons more typically have a lifespan that exceeds any single user process. On some systems, drivers can be loaded while *any* user process requires them. On others, they are loaded at system boot time (or at first demand) and only unloaded by explicit user action or a reboot.

A typical plug-and-play scenario would load the installed device driver, as selected by existing plug-and-play procedures. It would then determine what Interface Adapter names it will use, and register them with the Static Registry.

Plug-and-play installations should include some provision to remove entries after the driver is unloaded. This must include when the drivers are unloaded by rebooting the system.

8.4.2 Load on Demand

For uDAPL Consumers, the registry must support loading the Provider library on demand. To do so the registry makes use of the static registry (see “Static Registry” on page 320) to locate the Provider library that must be loaded.

Normally, the library loaded will be for the default version of the Provider. However, there may be OS-specific methods defined to allow an alternative version of the Provider to be loaded for debugging and testing purposes. The override mechanism must be available on a scope less than the entire system, such as the current shell, working directory, or user account. When the host OS supports environment variables, these should be at least one method available to specify the override.

A Provider library loaded by other OS-specific methods must still register with the Dynamic Registry but should not register with the Static Registry. The registry must support all Provider libraries, not just those it loaded directly.

8.4.3 Dynamic Provider Registration

Providers register themselves dynamically using the *dat_registry_add_provider* method, and may dynamically deregister themselves using the *dat_registry_remove_provider* method.

Providers must register themselves with the IA name assigned to them by the system administrator.

Provider libraries are responsible for ensuring the loading of any other system components on which they rely, or at least for validating their presence. For example, a Provider library frequently will be dependent on a kernel mode driver to support critical OS-dependent operations, such as pinning memory and registering interrupt handlers.

Provider libraries must dynamically register themselves with the registry, even if they were loaded by the registry. Only the Provider instance itself can properly initialize the DAT_PROVIDER structure identifying its methods and know when it has completed its internal initialization.

Providers are responsible for ensuring that their behavior is consistent with the data maintained by the IA selection service.

Providers that are loaded by the Registry must register themselves for a particular IA name when the registry calls *dat_provider_init*, passing them the IA name to register with. Providers that are explicitly bound (statically or dynamically linked) to the Consumer must register themselves for all IA names they will support before the Consumer makes its first call to *dat_ia_open*.

The dynamic registry offers the following services to DAT Consumers:

1) *dat_ia_open*: The Consumer specifies the IA name and Library qualifiers. The open call is directed to the correct Provider library if it can be found. Otherwise, the registry returns a "not found" error on its own.
2) *dat_ia_close*: In addition to passing the close call through to the Provider library, the dynamic registry may track the total number of concurrent opens for each Provider library.

8.4.4 STATIC REGISTRY

The static registry is a persistent data resource maintained by system administration when IA names are declared. The format is host OS-specific. No run-time library is associated with it, beyond any already defined by the host OS. Standard editing and access procedures for the host OS must be used to maintain and reference the static registry.

RATIONALE: Use of an existing resource takes advantage of authentication, backup, and auditing solutions already in place. Scattering system configuration data makes reliable system administration, particularly of multiple hosts, more difficult.

Entries are created when IA names are created. Normally entries would be created during software installation, but this could also be done as the result of a plug-and-play driver installation or system administration edits of the configuration.

RATIONALE: Plug-and-play drivers are fully supported with this approach. However, these drivers are dynamically declared once on a system-wide basis. From the perspective of the single address space that the dynamic registry operates in, they are just static registry entries, whether they have existed for seconds or months.

The static registry offers the following service to DAT Consumers:

- *dat_registry_list_providers*: Allows a DAT Consumer to obtain a snapshot of currently registered IA names and their attributes. This includes those statically registered but not loaded.

8.4.4.1 STATIC REGISTRY ENTRY CONTENTS

Each static registry entry specifies the following:

- The IA name, as assigned by the system administrator or generated by the Provider consistent with all constraints imposed by the system administrator.

- The API version of the library.
- Whether the library is thread-safe.
- Any required instance data, such as the device identifier and partition key. The same HCA or NIC driver may be referenced by multiple Interface Adapter entries with just the instance data varying.
- The path name for the Provider library image to be loaded.
- The version of the Provider library. This should include a string unique to the company that provides the Provider, as well as version numbers.
- The Platform-specific information. This field is a string completely under Platform control. This field is not passed to the Provider.

One entry per unique combination of IA name, API version, and thread-safe attribute must be marked as the default entry. Other entries can be selected via OS-specific override methods.

8.4.4.2 STATIC REGISTRY EDITING

The static registry must be in a format that is already supported by the host OS. No special editing tools may be required. Solutions known to be compatible with this requirement include the file system itself, text configuration files, preference files, and system registries.

RATIONALE: If the host OS editing procedures support temporary edits that are not persistent (for example, only until the next system restart), then these edits should be used for plug-and-play originate entries.

By default, the static registry must be maintained on a system-wide basis. However, local means for specifying default Registry entries must be available. The methods and scope of local substitution are OS-specific. Acceptable scopes include the user account, a specific process launch, and the current working directory.

RATIONALE: Use of existing OS-specific mechanisms inherit the maintenance, auditing, backup, and validation features from the OS solution. No duplication of effort is required. Some form of local override is required to enable testing of alternative configurations.

Providers, when adding a new entry to the dat.conf file, must fill the last field for the Platform-specific information with an empty string. If Providers edit the dat.conf file, it must adhere to the Platform rules, including rules for this field.

Advice to Implementors: Platform can have a simple rule of not touching the field by Providers. If an entry is deleted, the Platform can specify that all the fields of the entry are deleted, including a Platform-specific field.

Rationale: Platform can use this field to maintain information needed for Platform-specific administrative tools, including value-added platform management.

8.4.5 UNIX AND WINDOWS STATIC REGISTRIES

For UNIX and Windows systems, the Static Registry is a text file named dat.conf. The intention is for dat.conf to be located with other .conf files. For the following systems, the location of dat.conf is as specified:

Operating System	**Location of dat.conf**
Red Hat Linux	/etc
Windows	%SYSTEMDRIVE%/DAT

All characters after the "#" on any line will be ignored. Lines on which there are no characters other than white space and comments are considered blank lines and are ignored. Each field can contain white space if the field is quoted with double quotes. Within fields quoted with double quotes, the following are valid escape sequences:

Sequence	**Effect**
\\	backslash, \
\"	quote, "

For all API versions, each nonblank line contains the following white space-separated fields in order:

- The IA Name.
- The API version of the library: ["k"|"u"]major.minor. Examples: "k1.0", "u1.0", and "u1.1".
- Whether the library is thread-safe: ["threadsafe"|"nonthreadsafe"]
- Whether this is the default section: ["default"|"nondefault"]
- The path name for the library image to be loaded.
- The version of the Provider: id.major.minor. Where "id" is a string unique to the Provider company and "major" and "minor" are both integers in decimal format. For example, "xyz.5.13".
- A string with the instance data, which will be passed to the loaded library as its run-time arguments.
- The Platform-specific information. This field is a string completely under Platform control. This field is not passed to the Provider.

The format of the remaining fields on the line is dependent on the API version of the library specified in that line. Future API versions might add additional fields here.

The *dat.conf* file can be updated using normal file access procedures. Providers can alter only lines they created themselves. The site administrator can impose security restrictions on the file. Installers can be forced to create the image they would have installed and ask the user to complete the installation.

The following is an example of a *dat.conf* file:

```
# dat.conf sample
# for device hca1 - two IAs for different P-Keys
hca00 k1.0 nonthreadsafe default
/usr/local/foobarinc/kdapl.so xyz.2.3 "/dev/hca0 0" ""
hca00 k1.0 nonthreadsafe nondefault
/usr/local/foobarinc/kdapl_old.so xyz.2.0 "/dev/hca0 0" ""
hca00 u1.0 nonthreadsafe default
/usr/local/foobarinc/udapl.so xyz.2.3 "/dev/hca0 0" ""
hca00 u1.0 nonthreadsafe nondefault
/usr/local/foobarinc/udapl_old.so xyz.2.0 "/dev/hca0 0" ""
hca01 k1.0 nonthreadsafe default
/usr/local/foobarinc/kdapl.so xyz.2.3 "/dev/hca0 1" ""
hca01 k1.0 nonthreadsafe nondefault
/usr/local/foobarinc/kdapl_old.so xyz.2.0 "/dev/hca0 1" ""
hca01 u1.0 nonthreadsafe default
/usr/local/foobarinc/udapl.so xyz.2.3 "/dev/hca0 1" ""
hca01 u1.0 nonthreadsafe nondefault
/usr/local/foobarinc/udapl_old.so xyz.2.0 "/dev/hca0 1" ""
# for device hca1 - three different P-Keys
hca10 k1.0 nonthreadsafe default
/usr/local/foobarinc/kdapl.so xyz.2.3 "/dev/hca1 0" ""
hca10 k1.0 nonthreadsafe nondefault
/usr/local/foobarinc/kdapl_old.so xyz.2.0 "/dev/hca1 0" ""
hca10 u1.0 nonthreadsafe default
/usr/local/foobarinc/udapl.so xyz.2.3 "/dev/hca1 0" ""
hca10 u1.0 nonthreadsafe nondefault
/usr/local/foobarinc/udapl_old.so xyz.2.0 "/dev/hca1 0" ""
hca11 k1.0 nonthreadsafe default
/usr/local/foobarinc/kdapl.so xyz.2.3 "/dev/hca1 1" ""
hca11 k1.0 nonthreadsafe nondefault
/usr/local/foobarinc/kdapl_old.so xyz.2.0 "/dev/hca1 1" ""
hca11 u1.0 nonthreadsafe default
/usr/local/foobarinc/udapl.so xyz.2.3 "/dev/hca1 1" ""
hca11 u1.0 nonthreadsafe nondefault
/usr/local/foobarinc/udapl_old.so xyz.2.0 "/dev/hca1 1" ""
hca12 k1.0 nonthreadsafe default
/usr/local/foobarinc/kdapl.so xyz.2.3 "/dev/hca1 2" ""
hca12 k1.0 nonthreadsafe nondefault
/usr/local/foobarinc/kdapl_old.so xyz.2.0 "/dev/hca1 2" ""
```

```
hca12 u1.0 nonthreadsafe default
/usr/local/foobarinc/udapl.so xyz.2.3 "/dev/hca1 2" ""
hca12 u1.0 nonthreadsafe nondefault
/usr/local/foobarinc/udapl_old.so xyz.2.0 "/dev/hca1 2" ""
```

If the *DAT_OVERRIDE* environment variable is specified, it is taken as the name of a file containing local override information. Each nonblank line of this file contains the following white space-separated fields in order, formats matching the values in the Static Registry file:

- The IA Name
- The API version of the library
- Whether the library is thread-safe
- The version of the Provider to use instead of the entry marked “default” in the Static Registry

At most, one entry per unique combination of IA name, API version, and thread-safe attribute can be specified in this file.

The following is an example of a local override file that indicates the older version of the Provider specified in the Static Registry that should be used for device hca0:

```
hca00 u1.0 nonthreadsafe xyz.2.0 "" ""
hca01 u1.0 nonthreadsafe xyz.2.0 "" ""
```

8.4.6 OTHER STATIC REGISTRY FORMATS

The format of the registry for other platforms will be specified by the DAT Collaborative as reference implementations for those platforms are developed.

8.4.7 REDHAT RPM INSTALLATION ADVICE

Per Section 8.4.1, “Provider Installation Advice,” on page 317, the following requirements are placed on any DAT Provider packaged in RPM (RedHat Package Manager) format.

8.4.7.1 GENERAL INSTALLATION

- The files automatically installed by the RPM should be in a Provider-specific location (for example, /opt/<provider name>/...) rather than a system default location (for example, /usr/lib/...).
- The files installed by the Provider should adhere to the current version of the Filesystem Hierarchy Standard available at http://www.pathname.com/fhs/.

8.4.7.2 EDITING DAT.CONF FILE

- The Provider post-install script SHOULD edit the /etc/dat.conf file to include references to all Interface Adapters known to be supported by the Provider.

- The Provider pre-uninstall script SHOULD edit the /etc/dat.conf file to remove references to all Interface Adapters supported by the Provider.

8.4.7.3 INTERACTION WITH SYSTEM REGISTRY

As per Section 8.1.4, it is not the Provider's responsibility to implement the Static Registry; the Static Registry implementation for RedHat RPMs will be provided by the DAT Collaborative. (See www.datcollaborative.org/Registry.html.) That RPM has the name “dat-registry” followed by the usual RPM version number; that version number is in the form <API major version>.<API minor version>.<registry implementation version>. The Provider should follow the following installation guidelines for interacting with this Registry:

- The Provider can ship a RedHat RPM provided by the DAT Collaborative in its install package. However, the Provider can install that RPM only if no DAT-Registry RPM is installed on the system or the system administrator has in some way explicitly authorized that installation.
- The package containing the Provider should include a dependency on the DAT-Registry package. It can optionally include a dependency on a specific version of the DAT-Registry package. If that version of the DAT-Registry package is not and cannot be installed, this dependency causes failure of the installation of the Provider library and signals to the system administrator that the Provider library must be upgraded.

8.4.7.4 SETTING THE DEFAULT PROVIDER

As per Section 8.4.1, the Provider must include an option by which it can be installed without setting itself as the default Provider. The mechanism used for this purpose is up to the Provider.

8.4.7.5 INSTALLATION OF MULTIPLE VERSIONS OF THE PROVIDER

Section 8.4.1 requires that installation of a Provider should not remove a prior version of that Provider library unless so directed by the installer. For RPM, this distinction is indicated by the installer based on whether they use “rpm—install” or “rpm—upgrade” (indicating that previous versions should be removed) in installing the Provider. However, to allow this distinction, the Provider is required to avoid overwriting old versions of itself with files from the newest install. In other words, the version number of the Provider should be part of the pathway in which Provider files are installed.

CHAPTER 9: DAT NAME SERVICE

DAT mandates that the host platform have a name service that translates from host names to IPv4 or IPv6 addresses, and vice versa.

DAT defines IA addresses as IPv4 or IPv6, and defines host names as being DNS-compatible.

All name service APIs supported for the platform's IP name service must be supported. Simply using the platform's existing IP name service is a way to accomplish this.

> Rationale: There are a wide variety of name service APIs: *gethostbyaddr, gethostbyname, getipnodebyaddr, getipnodebyaadr,* and *getaddrinfo*. Each API is used by existing code and for a variety of reasons. Supporting only a subset of standard APIs still forces some existing code to be rewritten and/or the software development staff to learn new APIs.

The site administrator may configure the DAT name service to simply pass resolution operations through to the platform default IP name service.

> Rationale: Data can be exported from or imported to the existing IP name service, which already supports the translations that DAT requires. There is no need to create a new service.

A site administrator may choose to install a name service, which would dynamically choose between multiple underlying name services.

> Rationale: A dynamic name service would allow the site administrator to partition naming data into transport-specific domains, each with data only relevant to that transport.

All naming data returned by any API must be consistent, hence, the same, regardless of which local name service that a Consumer is using.

> Rationale: Installation of an override name service must not limit the application developers' choice of name service routines. Whether the application developers want to use *gethostbyaddr*, *getaddrinfo,* or another routine, their choice should not be limited by DAT.

Maintenance of the local IP/DNS naming data is the responsibility of the site administrator. Implementation of any dynamic switching DAT name service is the responsibility of the site administrator. Neither of these tasks are the responsibility of the DAT Provider or the Consumer.

9.1 ADVICE TO CONSUMERS

DAT requires that an IA Address have a unique meaning within the scope of a given host. That is, the meaning of the IA Address is not dependent on the IA.

This requirement enables the host OS to offer a local routing service: translating any given IA Address to an ordered list of paths to that destination. (Of course, in an IP network, the "path" only specifies the first step on that path, what interface is used to send the packet out on, and the next IP address).

In either case, the starting point of the path is the IA Name of the interface that should be opened.

Following are two example algorithms that a Consumer can use to determine which IA to open without relying on a host routing service or partially relying on a routing service.

9.1.1 FIND IA FOR A LOCAL IA ADDRESS

On most IP oriented systems, there is already a system authenticated enumeration of IP interfaces. This enumeration is already fully intergrated into IP routing and access permissions. Typically this list is available through the *ifconfig* and/or *netstat* programs, as well as run-time interfaces.

The existing IP Interface enumeration already typically provides a mechanism for assigning alias IP Address, limiting an interface to a specific VLAN, quality of service control and access control. There is no need for the DAT Naming Service to duplicate these controls, and it would be counter-productive for it to do so. Using the existing pre-RDMA administrative controls avoids making new requirements on systems and avoids duplicate administration.

For each platform there should be a mapping to identify an Interface Adapter given the following:

- The IP Interface
- The RDMA Service to be used (if more than one RDMA Provider can actually use a given IP interface, which will be the exception).
- Any RDMA specific options, such as selection of the LLP to be used when multiple options exist. For example an RDMA Service could support pre-IETF MPA, IETF MPA and SCTP all over the same set of IP Interfaces.

9.1.2 FIND IA FOR A LOCAL IA ADDRESS

For Consumers that know which local IA address they want to use to reach a remote node, here is the algorithm to follow:

1) The DAT Consumer calls *dat_registry_list_providers*.

2) The DAT Consumer repeats the following steps until the IA with the requested IA Address is found:

   a) The DAT Consumer opens the next IA from the list.

      i) If the list is exhausted and the IA with the requested IA Address is not found, check the platform configuration or contact your system administrator.

   b) The DAT Consumer calls *dat_ia_query* to extract IA_Address of the opened IA.

   c) If IA Address matches, done.

   d) Otherwise, close the IA and repeat the steps of 2.

It is assumed that the Consumer obtained a local IA Address it needs to use. For example:

1) The DAT Consumer uses platform Name Service to obtain the IA Address of the remote server based upon its host name.

2) The DAT Consumer uses platform Reachability Service to obtain the local IA Address to be used to reach that remote IA Address.

9.1.3 FIND IA TO REACH REMOTE IA ADDRESS

For Consumers that know which remote IA Address they want to reach but do not know which local IA Address to use, the following algorithm can be used:

1) The DAT Consumer calls *dat_registry_list_providers*.

2) The DAT Consumer repeats the following steps until the IA that can reach the remote IA Address is found:

   a) The DAT Consumer opens the next IA from the list.

      ii) If the list is exhausted and the IA with the requested IA Address is not found, check the platform configuration or contact your system administrator.

   b) The DAT Consumer creates an Endpoint on the opened IA.

   c) The DAT Consumer invokes *dat_ep_connect* to initiate the connection to the requested remote IA Address.

   d) If connection establishment succeeds or fails NOT with either synchronous *DAT_INVALID_ADDRESS*, or asynchronously with *DAT_CONNECTION_EVENT_UNREACHABLE*, this IA can be used to reach the requested remote IA Address.

   e) Otherwise, close the IA and repeat the steps of 2.

It is assumed that the Consumer obtained a remote IA Address it needs to use. For example:

1) The DAT Consumer uses platform Name Service to obtain the IA Address of the remote server based upon its host name.

APPENDIX A: UDAPL-2.0 HEADERS

This chapter defines uDAPL-2.0 header files. uDAPL Consumers need include only the udat.h header file, which automatically includes all other header files.

A.1 UDAT.H

```
/*
 * Copyright (c) 2002-2006, Network Appliance, Inc. All
rights reserved.
 *
 * This Software is licensed under all of the following li-
censes:
 *
 * 1) under the terms of the "Common Public License 1.0".
The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "BSD License". The license is
also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License
(GPL) Version 2".
 *  The license is also available from the Open Source Ini-
tiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above li-
censes.
 *
 * Redistribution and use in source and binary forms, with
or without
 * modification, are permitted provided that the following
conditions are
 * met:
 *
```

```
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 */

/**********************************************************************
 *
 * HEADER: udat.h
 *
 * PURPOSE: defines the user DAT API
 *
 * Description: Header file for "uDAPL: User Direct Access Programming
 *        Library, Version: 2.0"
 *
 * Mapping rules:
 *       All global symbols are prepended with DAT_ or dat_
 *      All DAT objects have an 'api' tag which, such as 'ep' or 'lmr'
 *      The method table is in the provider definition structure.
 *
 ********************************************************************/

#ifndef _UDAT_H_
#define _UDAT_H_
```

```
#include <dat/udat_config.h>

#include <dat/dat_platform_specific.h>

typedef enum dat_mem_type
{
        /* Shared between udat and kdat */
    DAT_MEM_TYPE_VIRTUAL    = 0x00,
    DAT_MEM_TYPE_LMR        = 0x01,
   /* udat specific */
    DAT_MEM_TYPE_SHARED_VIRTUAL   = 0x02
} DAT_MEM_TYPE;

/* dat handle types */
typedef enum dat_handle_type
{
    DAT_HANDLE_TYPE_CR,
    DAT_HANDLE_TYPE_EP,
    DAT_HANDLE_TYPE_EVD,
    DAT_HANDLE_TYPE_IA,
    DAT_HANDLE_TYPE_LMR,
    DAT_HANDLE_TYPE_PSP,
    DAT_HANDLE_TYPE_PZ,
    DAT_HANDLE_TYPE_RMR,
    DAT_HANDLE_TYPE_RSP,
    DAT_HANDLE_TYPE_CNO,
    DAT_HANDLE_TYPE_SRQ,
   DAT_HANDLE_TYPE_CSP
#ifdef DAT_EXTENSIONS
    ,DAT_HANDLE_TYPE_EXTENSION_BASE
#endif
    } DAT_HANDLE_TYPE;

   /* EVD state consists of three orthogonal substates.
   One for enabled/disabled,
   one for waitable/unwaitable,
   and one for configuration.
```

```
   Within eachsubstates the values are mutually exclusive.
*/

typedef enum dat_evd_state
        {
        DAT_EVD_STATE_ENABLED           =0x01,
        DAT_EVD_STATE_DISABLED          =0x02,
   DAT_EVD_STATE_WAITABLE               =0x04,
   DAT_EVD_STATE_UNWAITABLE             =0x08,
   DAT_EVD_STATE_CONFIG_NOTIFY          =0x10,
   DAT_EVD_STATE_CONFIG_SOLICITED       =0x20,
   DAT_EVD_STATE_CONFIG_THRESHOLD       =0x30
} DAT_EVD_STATE;

typedef enum dat_evd_param_mask
{
    DAT_EVD_FIELD_IA_HANDLE            = 0x01,
    DAT_EVD_FIELD_EVD_QLEN              = 0x02,
    DAT_EVD_FIELD_EVD_STATE            = 0x04,
    DAT_EVD_FIELD_CNO                  = 0x08,
    DAT_EVD_FIELD_EVD_FLAGS            = 0x10,

    DAT_EVD_FIELD_ALL                  = 0x1F
   } DAT_EVD_PARAM_MASK;

typedef DAT_UINT64 DAT_PROVIDER_ATTR_MASK;

#include <dat/dat.h>

typedef DAT_HANDLE      DAT_CNO_HANDLE;

struct dat_evd_param
    {
    DAT_IA_HANDLE    ia_handle;
    DAT_COUNT        evd_qlen;
    DAT_EVD_STATE    evd_state;
    DAT_CNO_HANDLE   cno_handle;
    DAT_EVD_FLAGS    evd_flags;
    };
```

```
#define DAT_LMR_COOKIE_SIZE 40 /* size of DAT_LMR_COOKIE in bytes */
typedef char (* DAT_LMR_COOKIE)[DAT_LMR_COOKIE_SIZE];

/* Format for OS wait proxy agent function */

typedef void (*DAT_AGENT_FUNC)
        (
        DAT_PVOID,          /* instance data   */
        DAT_EVD_HANDLE      /* Event Dispatcher*/
        );

/* Definition */

typedef struct dat_os_wait_proxy_agent
        {
        DAT_PVOID    instance_data;
        DAT_AGENT_FUNC proxy_agent_func;
        } DAT_OS_WAIT_PROXY_AGENT;

/* Define NULL Proxy agent */

#define DAT_OS_WAIT_PROXY_AGENT_NULL \
        (DAT_OS_WAIT_PROXY_AGENT) {\
        (DAT_PVOID) NULL,\
        (DAT_AGENT_FUNC) NULL}

/* Flags */

/* The value specified by the uDAPL Consumer for dat_ia_open to indicate
 * that no async EVD should be created for the opening instance of an IA.
 * The same IA has been open before that has the only async EVD to
 * handle async errors for all open instances of the IA.
 */

#define DAT_EVD_ASYNC_EXISTS (DAT_EVD_HANDLE) 0x1
```

```
/*
 * The value returned by the dat_ia_query for the case when there is no
 * async EVD for the IA instance. The Consumer specified the value of
 * DAT_EVD_ASYNC_EXISTS for the async_evd_handle for dat_ia_open.
 */

#define DAT_EVD_OUT_OF_SCOPE (DAT_EVD_HANDLE) 0x2

/*
 * Memory types
 *
 * Specifying memory type for LMR create. A Consumer must use a single
 * value when registering memory. The union of any of these
 * flags is used in the Provider parameters to indicate what memory
 * type Provider supports for LMR memory creation.
 */

/* For udapl only */

typedef struct dat_shared_memory
    {
    DAT_PVOID                    virtual_address;
    DAT_LMR_COOKIE              shared_memory_id;
    } DAT_SHARED_MEMORY;

typedef union dat_region_description
    {
    DAT_PVOID                    for_va;
    DAT_LMR_HANDLE         for_lmr_handle;
    DAT_SHARED_MEMORY      for_shared_memory;      /* For udapl only */
   } DAT_REGION_DESCRIPTION;

/* LMR Arguments */
```

```
struct dat_lmr_param
    {
    DAT_IA_HANDLE           ia_handle;
    DAT_MEM_TYPE            mem_type;
    DAT_REGION_DESCRIPTION  region_desc;
    DAT_VLEN                length;
    DAT_PZ_HANDLE           pz_handle;
    DAT_MEM_PRIV_FLAGS      mem_priv;
    DAT_VA_TYPE             va_type;
    DAT_LMR_CONTEXT         lmr_context;
    DAT_RMR_CONTEXT         rmr_context;
    DAT_VLEN                registered_size;
    DAT_VADDR               registered_address;
    };

/* LMR Arguments Mask */

enum dat_lmr_param_mask
    {
    DAT_LMR_FIELD_IA_HANDLE                 = 0x001,
    DAT_LMR_FIELD_MEM_TYPE                  = 0x002,
    DAT_LMR_FIELD_REGION_DESC               = 0x004,
    DAT_LMR_FIELD_LENGTH                    = 0x008,
    DAT_LMR_FIELD_PZ_HANDLE                 = 0x010,
    DAT_LMR_FIELD_MEM_PRIV                  = 0x020,
    DAT_LMR_FIELD_VA_TYPE                   = 0x040,
    DAT_LMR_FIELD_LMR_CONTEXT               = 0x080,
    DAT_LMR_FIELD_RMR_CONTEXT               = 0x100,
    DAT_LMR_FIELD_REGISTERED_SIZE           =0x200,
    DAT_LMR_FIELD_REGISTERED_ADDRESS        = 0x400,
    DAT_LMR_FIELD_ALL                       = 0x7FF
    };

typedef enum dat_proxy_type
   {
   DAT_PROXY_TYPE_NONE              = 0x0,
   DAT_PROXY_TYPE_AGENT             = 0x1,
   DAT_PROXY_TYPE_FD                = 0x2
   } DAT_PROXY_TYPE;
```

```
typedef struct dat_cno_param
        {
   DAT_IA_HANDLE                        ia_handle;
   DAT_PROXY_TYPE                       proxy_type;
   union {
         DAT_OS_WAIT_PROXY_AGENT        agent;
         DAT_FD                         fd;
         DAT_PVOID                      none;
   } proxy;
} DAT_CNO_PARAM;

typedef enum dat_cno_param_mask
        {
        DAT_CNO_FIELD_IA_HANDLE  = 0x1,
        DAT_CNO_FIELD_PROXY_TYPE = 0x2,
        DAT_CNO_FIELD_PROXY      = 0x3,
        DAT_CNO_FIELD_ALL        = 0x4
        } DAT_CNO_PARAM_MASK;

struct dat_ia_attr
   {
   char                  adapter_name[DAT_NAME_MAX_LENGTH];
    char             vendor_name[DAT_NAME_MAX_LENGTH];
    DAT_UINT32             hardware_version_major;
    DAT_UINT32             hardware_version_minor;
    DAT_UINT32             firmware_version_major;
    DAT_UINT32             firmware_version_minor;
    DAT_IA_ADDRESS_PTR     ia_address_ptr;
    DAT_COUNT        max_eps;
    DAT_COUNT        max_dto_per_ep;
    DAT_COUNT        max_rdma_read_per_ep_in;
    DAT_COUNT        max_rdma_read_per_ep_out;
    DAT_COUNT        max_evds;
    DAT_COUNT        max_evd_qlen;
    DAT_COUNT        max_iov_segments_per_dto;
    DAT_COUNT        max_lmrs;
    DAT_SEG_LENGTH   max_lmr_block_size;
    DAT_VADDR        max_lmr_virtual_address;
```

```
    DAT_COUNT          max_pzs;
    DAT_SEG_LENGTH     max_message_size;
    DAT_SEG_LENGTH     max_rdma_size;
    DAT_COUNT          max_rmrs;
    DAT_VADDR          max_rmr_target_address;
    DAT_COUNT          max_srqs;
    DAT_COUNT          max_ep_per_srq;
    DAT_COUNT          max_recv_per_srq;
    DAT_COUNT          max_iov_segments_per_rdma_read;
    DAT_COUNT          max_iov_segments_per_rdma_write;
    DAT_COUNT          max_rdma_read_in;
    DAT_COUNT          max_rdma_read_out;
    DAT_BOOLEAN        max_rdma_read_per_ep_in_guaranteed;
    DAT_BOOLEAN        max_rdma_read_per_ep_out_guaranteed;
    DAT_BOOLEAN        zb_supported;
#ifdef DAT_EXTENSIONS
    DAT_EXTENSION      extension_supported;
    DAT_COUNT          extension_version;
#endif /* DAT_EXTENSIONS */
    DAT_COUNT          num_transport_attr;
    DAT_NAMED_ATTR     *transport_attr;
    DAT_COUNT          num_vendor_attr;
    DAT_NAMED_ATTR     *vendor_attr;
};

#ifdef DAT_EXTENSIONS
#define DAT_IA_FIELD_IA_EXTENSION UINT64_C(0x100000000)
#define DAT_IA_FIELD_IA_EXTENSION_VERSION UINT64_
C(0x200000000)
#endif /* DAT_EXTENSIONS */

#define DAT_IA_FIELD_IA_NUM_TRANSPORT_ATTR UINT64_
C(0x400000000)
#define DAT_IA_FIELD_IA_TRANSPORT_ATTR UINT64_
C(0x800000000)
#define DAT_IA_FIELD_IA_NUM_VENDOR_ATTR UINT64_
C(0x1000000000)
#define DAT_IA_FIELD_IA_VENDOR_ATTR UINT64_C(0x2000000000)
#define DAT_IA_FIELD_ALL UINT64_C(0x3FFFFFFFF)
```

```
/* General Provider attributes. udat specific. */
typedef enum dat_pz_support
    {
    DAT_PZ_UNIQUE,
    DAT_PZ_SHAREABLE
    } DAT_PZ_SUPPORT;

#include <dat/udat_vendor_specific.h>

/* Provider should support merging of all event stream types.
Provider
 * attribute specify support for merging different event
stream types.
 * It is a 2D binary matrix where each row and column repre-
sents an event
 * stream type. Each binary entry is 1 if the event streams
of its raw
 * and column can fed the same EVD, and 0 otherwise. The order
of event
 * streams in row and column is the same as in the definition
of
 * DAT_EVD_FLAGS: index 0 - Software Event, 1- Connection
Request,
 * 2 - DTO Completion, 3 - Connection event, 4 - RMR Bind
Completion,
 * 5 - Asynchronous event.
 * By definition each diagonal entry is 1.
 * Consumer allocates an array for it and passes it IN as a
pointer
 * for the array that Provider fills. Provider must fill the
array
 * that Consumer passes.
 */

struct dat_provider_attr
    {
    char            provider_name[DAT_NAME_MAX_LENGTH];
    DAT_UINT32                     provider_version_major;
    DAT_UINT32                     provider_version_minor;
    DAT_UINT32                         dapl_version_major;
    DAT_UINT32                         dapl_version_minor;
```

```
    DAT_MEM_TYPE             lmr_mem_types_supported;
    DAT_IOV_OWNERSHIP        iov_ownership_on_return;
    DAT_QOS                               dat_qos_supported;
    DAT_COMPLETION_FLAGS     completion_flags_supported;
    DAT_BOOLEAN                           is_thread_safe;
    DAT_COUNT                             max_private_data_
size;
    DAT_BOOLEAN                           supports_multi-
path;
    DAT_EP_CREATOR_FOR_PSP ep_creator;
    DAT_PZ_SUPPORT                   pz_support;
    DAT_UINT32                            optimal_buffer_
alignment;
    const DAT_BOOLEAN        evd_stream_merging_sup-
ported[6][6];
    DAT_BOOLEAN                           srq_supported;
    DAT_COUNT                srq_watermarks_supported;
    DAT_BOOLEAN              srq_ep_pz_difference_supported;
    DAT_COUNT                srq_info_supported;
    DAT_COUNT                ep_recv_info_supported;
  DAT_BOOLEAN                lmr_sync_req;
  DAT_BOOLEAN                dto_async_return_guaranteed;
  DAT_BOOLEAN                rdma_write_for_rdma_read_req;
  DAT_BOOLEAN                rdma_read_lmr_rmr_context_expo-
sure;
  DAT_RMR_SCOPE              rmr_scope_supported;
  DAT_BOOLEAN                is_signal_safe;
  DAT_BOOLEAN                ha_supported;
  DAT_HA_LB                  ha_loadbalancing;

  DAT_COUNT                  num_provider_specific_attr;
   DAT_NAMED_ATTR *                   provider_specific_attr;
    };

#define DAT_PROVIDER_FIELD_PROVIDER_NAME UINT64_
C(0x00000001)
#define DAT_PROVIDER_FIELD_PROVIDER_VERSION_MAJOR UINT64_
C(0x00000002)
#define DAT_PROVIDER_FIELD_PROVIDER_VERSION_MINOR UINT64_
C(0x00000004)
```

```
#define DAT_PROVIDER_FIELD_DAPL_VERSION_MAJOR UINT64_C(0x00000008)
#define DAT_PROVIDER_FIELD_DAPL_VERSION_MINOR UINT64_C(0x00000010)
#define DAT_PROVIDER_FIELD_LMR_MEM_TYPE_SUPPORTED UINT64_C(0x00000020)
#define DAT_PROVIDER_FIELD_IOV_OWNERSHIP UINT64_C(0x00000040)
#define DAT_PROVIDER_FIELD_DAT_QOS_SUPPORTED UINT64_C(0x00000080)
#define DAT_PROVIDER_FIELD_COMPLETION_FLAGS_SUPPORTED UINT64_C(0x00000100)
#define DAT_PROVIDER_FIELD_IS_THREAD_SAFE UINT64_C(0x00000200)
#define DAT_PROVIDER_FIELD_MAX_PRIVATE_DATA_SIZE UINT64_C(0x00000400)
#define DAT_PROVIDER_FIELD_SUPPORTS_MULTIPATH UINT64_C(0x00000800)
#define DAT_PROVIDER_FIELD_EP_CREATOR UINT64_C(0x00001000)
#define DAT_PROVIDER_FIELD_PZ_SUPPORT UINT64_C(0x00002000)
#define DAT_PROVIDER_FIELD_OPTIMAL_BUFFER_ALIGNMENT UINT64_C(0x00004000)
#define DAT_PROVIDER_FIELD_EVD_STREAM_MERGING_SUPPORTED UINT64_C(0x00008000)
#define DAT_PROVIDER_FIELD_SRQ_SUPPORTED UINT64_C(0x00010000)
#define DAT_PROVIDER_FIELD_SRQ_WATERMARKS_SUPPORTED UINT64_C(0x00020000)
#define DAT_PROVIDER_FIELD_SRQ_EP_PZ_DIFFERENCE_SUPPORTED UINT64_C(0x00040000)
#define DAT_PROVIDER_FIELD_SRQ_INFO_SUPPORTED UINT64_C(0x00080000)
#define DAT_PROVIDER_FIELD_EP_RECV_INFO_SUPPORTED UINT64_C(0x00100000)
#define DAT_PROVIDER_FIELD_LMR_SYNC_REQ UINT64_C(0x00200000)
#define DAT_PROVIDER_FIELD_DTO_ASYNC_RETURN_GUARANTEED UINT64_C(0x00400000)
#define DAT_PROVIDER_FIELD_RDMA_WRITE_FOR_RDMA_READ_REQ UINT64_C(0x00800000)
#define DAT_PROVIDER_FIELD_RDMA_READ_LMR_RMR_CONTEXT_EXPOSURE UINT64_C(0x01000000)
#define DAT_PROVIDER_FIELD_RMR_SCOPE_SUPPORTED UINT64_C(0x02000000)
```

```
#define DAT_PROVIDER_FIELD_IS_SIGNAL_SAFE UINT64_C(0x04000000)
#define DAT_PROVIDER_FIELD_HA_SUPPORTED UINT64_C(0x08000000)
#define DAT_PROVIDER_FIELD_HA_LB UINT64_C(0x10000000)
#define DAT_PROVIDER_FIELD_NUM_PROVIDER_SPECIFIC_ATTR UINT64_C(0x20000000)
#define DAT_PROVIDER_FIELD_PROVIDER_SPECIFIC_ATTR UINT64_C(0x40000000)

#define DAT_PROVIDER_FIELD_ALL UINT64_C(0x7FFFFFF)
#define DAT_PROVIDER_FIELD_NONE UINT64_C(0x0)

/**********************************************************************/
/*
 * User DAT function call definitions,
 */

extern DAT_RETURN dat_lmr_create (
        IN      DAT_IA_HANDLE,    /* ia_handle */
        IN      DAT_MEM_TYPE,     /* mem_type */
        IN      DAT_REGION_DESCRIPTION,/* region_description */
        IN      DAT_VLEN,         /* length */
        IN      DAT_PZ_HANDLE,    /* pz_handle */
        IN      DAT_MEM_PRIV_FLAGS,          /* privileges */
        IN      DAT_VA_TYPE,                 /* va_type */
                OUT     DAT_LMR_HANDLE *, /* lmr_handle */
        OUT     DAT_LMR_CONTEXT *,/* lmr_context */
        OUT     DAT_RMR_CONTEXT *,/* rmr_context */
        OUT     DAT_VLEN *,       /* registered_length */
        OUT     DAT_VADDR *);     /* registered_address */

extern DAT_RETURN dat_lmr_query (
   IN      DAT_LMR_HANDLE,/* lmr_handle           */
   IN      DAT_LMR_PARAM_MASK,/* lmr_param_mask       */
   OUT     DAT_LMR_PARAM *);/* lmr_param            */

/* Event Functions */
```

```
extern DAT_RETURN dat_evd_create (
        IN      DAT_IA_HANDLE,          /* ia_handle */
        IN      DAT_COUNT,              /* evd_min_qlen */
        IN      DAT_CNO_HANDLE,         /* cno_handle */
        IN      DAT_EVD_FLAGS,          /* evd_flags */
        OUT     DAT_EVD_HANDLE *);      /* evd_handle */

extern DAT_RETURN dat_evd_modify_cno (
        IN      DAT_EVD_HANDLE,         /* evd_handle          */
        IN      DAT_CNO_HANDLE);        /* cno_handle          */

extern DAT_RETURN dat_cno_create (
        IN DAT_IA_HANDLE,               /* ia_handle           */
        IN DAT_OS_WAIT_PROXY_AGENT,/* agent                    */
        OUT DAT_CNO_HANDLE *);          /* cno_handle */

extern DAT_RETURN dat_cno_modify_agent (
        IN DAT_CNO_HANDLE,              /* cno_handle          */
        IN DAT_OS_WAIT_PROXY_AGENT);/* agent                   */

extern DAT_RETURN dat_cno_query (
        IN      DAT_CNO_HANDLE,         /* cno_handle */
        IN      DAT_CNO_PARAM_MASK, /* cno_param_mask */
        OUT     DAT_CNO_PARAM *);       /* cno_param */

extern DAT_RETURN dat_cno_free (
        IN DAT_CNO_HANDLE);             /* cno_handle */

extern DAT_RETURN dat_cno_wait (
        IN DAT_CNO_HANDLE,              /* cno_handle          */
        IN DAT_TIMEOUT,                 /* timeout             */
        OUT DAT_EVD_HANDLE *);          /* evd_handle */

extern DAT_RETURN dat_evd_enable (
        IN      DAT_EVD_HANDLE);        /* evd_handle */

extern DAT_RETURN dat_evd_wait (
        IN DAT_EVD_HANDLE,              /* evd_handle          */
```

```
        IN DAT_TIMEOUT,              /* timeout            */
        IN DAT_COUNT,                /* threshold          */
        OUT DAT_EVENT *,             /* event              */
        OUT DAT_COUNT *);            /* N more events      */

extern DAT_RETURN dat_evd_disable (
        IN      DAT_EVD_HANDLE);         /* evd_handle */

extern DAT_RETURN dat_evd_set_unwaitable (
        IN      DAT_EVD_HANDLE);         /* evd_handle */

extern DAT_RETURN dat_evd_clear_unwaitable (
      IN     DAT_EVD_HANDLE);       /* evd_handle          */

extern DAT_RETURN dat_cno_fd_create (
   IN     DAT_IA_HANDLE,          /* ia_handle       */
   OUT    DAT_FD *,               /* file descriptor */
   OUT    DAT_CNO_HANDLE * );     /* cno_handle      */
extern DAT_RETURN dat_cno_trigger (
   IN     DAT_CNO_HANDLE,               /* cno_handle */
   OUT    DAT_EVD_HANDLE * );           /* evd_handle */

#endif /* _UDAT_H_ */
```

A.2 UDAT_CONFIG.H

```
/ * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
* This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
```

```
 *
 * 3) under the terms of the "GNU General Public License (GPL)
Version 2".
 *  The license is also available from the Open Source Ini-
tiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with
or without
 * modification, are permitted provided that the following
conditions are
 * met:
 *
 * Redistributions of source code must retain both the above
copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the
above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names
of other DAT
 * Collaborative contributors may be used to endorse or pro-
mote
 * products derived from this software without specific prior
written
 * permission.
 *
 */

/*************************************************************
*****
 *
 * HEADER: udat_config.h
 *
 * PURPOSE: provides uDAPL configuration information.
 *
```

```
 * Description: Header file for "uDAPL: User Direct Access Programming
 *        Library, Version: 2.0"
 *
**********************************************************************/

#ifndef _UDAT_CONFIG_H_
#define _UDAT_CONFIG_H_

#define DAT_VERSION_MAJOR 1
#define DAT_VERSION_MINOR 2

/*
 * The official header files will default DAT_THREADSAFE to DAT_TRUE. If
 * your project does not wish to use this default, you must ensure that
 * DAT_THREADSAFE will be set to DAT_FALSE. This may be done by an
 * explicit #define in a common project header file that is included
 * before any DAT header files, or through command line di-rectives to the
 * compiler (presumably controlled by the make environment).
 */

/*
 * A site, project or platform may consider setting an al-ternate default
 * via their make rules, but are discouraged from doing so by editing
 * the official header files.
 */

/*
 * The Reference Implementation is not Thread Safe. The Ref-erence
 * Implementation has chosen to go with the first method and define it
 * explicitly in the header file.
 */
```

```
#define DAT_THREADSAFE DAT_FALSE

#ifndef DAT_THREADSAFE
#define DAT_THREADSAFE DAT_TRUE
#endif /* DAT_THREADSAFE */

#endif /* _UDAT_CONFIG_H_ */
```

A.3 DAT_PLATFORM_SPECIFIC.H

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
 * This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the “Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the “BSD License”. The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the “GNU General Public License (GPL) Version 2”.
 *    The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
```

```
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 */

/************************************************************
 *
 * HEADER: dat_platform_specific.h
 *
 * PURPOSE: defines Platform-specific types.
 *
 * Description: Header file for "DAPL: Direct Access Programming
 *        Library, Version: 2.0"
 *
 * Mapping rules:
 *
 ************************************************************/

#ifndef _DAT_PLATFORM_SPECIFIC_H_
#define _DAT_PLATFORM_SPECIFIC_H_
```

```
/* OS, processor, compiler type definitions. Add OSes as
needed. */

/*
 * This captures the alignment for the bus transfer from the
HCA/IB chip
 * to the main memory.
 */
#ifndef DAT_OPTIMAL_ALIGNMENT
#define DAT_OPTIMAL_ALIGNMENT   256           /* Performance
optimal alignment */
#endif /* DAT_OPTIMAL_ALIGNMENT */

/* Assume all OSes use sockaddr, for address family: IPv4 ==
AF_INET,
 * IPv6 == AF_INET6. Use of "namelen" field indicated.
 *
 * The Interface Adapter Address names an Interface Adapter
local or
 * remote, that is used for connection management and Name
 * Service. The format of the dat_ia_address_ptr follows the
normal
 * socket programming practice of struct sockaddr *. DAT sup-
ports both
 * IPv4 and IPv6 address families. Allocation and initial-
ization of
 * DAT IA address structures must follow normal Sockets pro-
gramming
 * procedures. The underlying type of the DAT IA address is
the native
 * struct sockaddr for each target operating system. In all
cases,
 * storage appropriate for the address family in use by the
target
 * Provider must be allocated. For instance, when IPv6 ad-
dressing is
 * in use, this should be allocated as struct sockaddr_net6.
The
 * sockaddr sa_family and, if present, sa_len fields must be
 * initialized appropriately, as well as the address infor-
mation.
 * When passed across the DAPL API this storage is cast to the
```

```
 * DAT_IA_ADDRESS_PTR type. It is the responsibility of the
callee to
 * verify that the sockaddr contains valid data for the re-
quested
 * operation. It is always the responsibility of the caller
to manage
 * the storage.
 *
 * uDAPL code example for Linux (kdapl would be similar):
 * #include <stdio.h>
 * #include <sys/socket.h>
 * #include <netinet/in.h>
 * #include <dat/udat.h>
 *
 * struct sockaddr_in6 addr;
 * DAT_IA_ADDRESS_PTR ia_addr;
 *
*  // Note: linux pton requires explicit encoding of IPv4 in
IPv6
 *
 * addr.sin6_family = AF_INET6;
 * if (inet_pton(AF_INET6, "0:0:0:0:0:FFFF:192.168.0.1",
 *              &addr.sin6_addr) <= 0)
 *   return(-1); // Bad address or no address family support
*
*  // initialize other necessary fields such as port, flow,
etc.
 *
 * ia_addr = (DAT_IA_ADDRESS_PTR) &addr;
 * dat_ep_connect(ep_handle, ia_addr, conn_qual, timeout,
0, NULL,
 *               qos, DAT_CONNECT_DEFAULT_FLAG);
 *
 */

/* Solaris begins */

#if defined (sun) || defined(__sun) || defined(_sun_) || de-
fined (__solaris__)
#include <sys/types.h>
#include <inttypes.h>/* needed for UINT64_C() macro */
```

```
typedef uint32_t                DAT_UINT32;     /* Unsigned host order, 32 bits */
typedef uint64_t                DAT_UINT64;     /* unsigned host order, 64 bits */
typedef unsigned long longDAT_UVERYLONG; /* unsigned longest native to compiler */

typedef void *                  DAT_PVOID;
typedef int                     DAT_COUNT;

#include <sys/socket.h>
#include <netinet/in.h>
typedef struct sockaddr        DAT_SOCK_ADDR; /* Socket address header native to OS */
typedef struct sockaddr_in6    DAT_SOCK_ADDR6; /* Socket address header native to OS */

#define DAT_AF_INET             AF_INET
#define DAT_AF_INET6            AF_INET6

typedef DAT_UINT64   DAT_PADDR;

/* Solaris ends */


/* Linux begins */

#elif defined(__linux__) /* Linux */
#if defined(__KERNEL__)
#include <linux/types.h>
#else
#include <sys/types.h>
#endif /* defined(__KERNEL__) */

typedef u_int32_t               DAT_UINT32;     /* unsigned host order, 32 bits */
typedef u_int64_t               DAT_UINT64;     /* unsigned host order, 64 bits */
typedef unsigned long longDAT_UVERYLONG; /* unsigned longest native to compiler */
```

```
typedef void *        DAT_PVOID;
typedef int           DAT_COUNT;
typedef DAT_UINT64    DAT_PADDR;
#ifndef UINT64_C
#define UINT64_C(c)c ## ULL
#endif /* UINT64_C */

#if defined(__KERNEL__)
#include <linux/socket.h>
#include <linux/in.h>
#include <linux/in6.h>
#else
#include <sys/socket.h>
#endif /* defined(__KERNEL__) */

typedef struct dat_comm {
   int         domain;
   int         type;
   int         protocol;
} DAT_COMM;

typedef int DAT_FD; /* DAT file descriptor */

typedef struct sockaddr        DAT_SOCK_ADDR; /* Socket ad-
dress header native to OS */
typedef struct sockaddr_in6    DAT_SOCK_ADDR6; /* Socket ad-
dress header native to OS */

#define DAT_AF_INET              AF_INET
#define DAT_AF_INET6             AF_INET6
/* Linux ends */


/* Win32 begins */
#elif defined(_MSC_VER) || defined(_WIN32) /* NT. MSC com-
piler, Win32 platform */

typedef unsigned __int32       DAT_UINT32;     /* Unsigned
host order, 32 bits */
```

```
typedef unsigned __int64        DAT_UINT64;     /* unsigned host order, 64 bits */
typedef unsigned longDAT_UVERYLONG; /* unsigned longest native to compiler */

typedef void*        DAT_PVOID;
typedef long         DAT_COUNT;

typedef struct sockaddr     DAT_SOCK_ADDR; /* Socket address header native to OS */
typedef struct sockaddr_in6 DAT_SOCK_ADDR6; /* Socket address header native to OS */

#ifndef UINT64_C
#define UINT64_C(c) c ## i64
#endif /* UINT64_C */

#define DAT_AF_INET  AF_INET
#define DAT_AF_INET6 AF_INET6

#if defined(__KDAPL__)
/* must have the DDK for this definition */
typedef PHYSICAL_ADDRESS DAT_PADDR;
#endif /* __KDAPL__ */

/* Win32 ends */


#else
#error dat_platform_specific.h : OS type not defined
#endif

#ifndef IN
#define IN
#endif
#ifndef OUT
#define OUT
#endif
#ifndef INOUT
#define INOUT
```

```
#endif

#endif /* _DAT_PLATFORM_SPECIFIC_H_ */
```

A.4 DAT.H

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
 * This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
```

```
/**********************************************************************
 *
 * HEADER: dat.h
 *
 * PURPOSE: defines the common DAT API for uDAPL and kDAPL.
 *
 * Description: Header file for "DAPL: Direct Access Programming
 *       Library, Version: 2.0"
 *
 * Mapping rules:
 *      All global symbols are prepended with DAT_ or dat_
 *     All DAT objects have an 'api' tag which, such as 'EP' or 'LMR'
 *     The method table is in the provider definition structure.
 *
 *
 *********************************************************************/

#ifndef _DAT_H_
#define _DAT_H_

#include <dat/dat_error.h>
```

```
/* Generic DAT types */

typedef char * DAT_NAME_PTR;       /* Format for ia_name and
attributes */
#define DAT_NAME_MAX_LENGTH     256

/*
 * Used for provider, vendor, transport, hardware-specific
attributes
 * definitions.
 */

typedef struct dat_named_attr
    {
    const char *         name;    /* Name of attribute */
    const char *         value; /* Value of attribute */
    } DAT_NAMED_ATTR;

typedef enum dat_boolean
    {
    DAT_FALSE  = 0,
    DAT_TRUE   = 1
    } DAT_BOOLEAN;

#ifdef DAT_EXTENSIONS
#define DAT_IB_EXTENSION 1
#define DAT_IW_EXTENSION 2
#endif /* DAT_EXTENSIONS */

typedef DAT_UINT32 DAT_HA_LB;
#define DAT_HA_LB_NONE (DAT_HA_LB)0
#define DAT_HA_LB_INTERCOMM (DAT_HA_LB)1
#define DAT_HA_LB_INTRACOMM (DAT_HA_LB)2

typedef union dat_context
    {
    DAT_PVOID           as_ptr;
    DAT_UINT64       as_64;
```

```
        DAT_UVERYLONG    as_index;
    } DAT_CONTEXT;

typedef DAT_CONTEXT     DAT_DTO_COOKIE;
typedef DAT_CONTEXT     DAT_RMR_COOKIE;

typedef enum dat_completion_flags
    {
        /* Completes with notifica-
tion                                    */
    DAT_COMPLETION_DEFAULT_FLAG                 = 0x00,

        /* Completions suppressed if suc-
cessful                           */
    DAT_COMPLETION_SUPPRESS_FLAG                = 0x01,

    /* Sender controlled notification for recv completion
*/
    DAT_COMPLETION_SOLICITED_WAIT_FLAG          = 0x02,

        /* Completions with unsignaled notifica-
tions                           */
    DAT_COMPLETION_UNSIGNALLED_FLAG             = 0x04,

   /* Do not start processing until all previous RDMA reads
complete. */
   DAT_COMPLETION_BARRIER_FENCE_FLAG               = 0x08,
   /* Only valid for uDAPL as EP attribute for Recv Comple-
tion flags.
    * Waiter unblocking is controlled by the Threshold value
of dat_evd_wait.
    * UNSIGNALLED for RECV is not allowed when EP has this
attribute. */
   DAT_COMPLETION_EVD_THRESHOLD_FLAG               = 0x10,
   /* Only valid for kDAPL
    * Do not start processing LMR invalidate until all
    * previously posted DTOs to the EP Request Queue
    * have been completed.
    * The value for LMR Invalidate Fence does not
    * conflict with uDAPL so it can be extended
    * to uDAPL usage later.
```

```
    */
   DAT_COMPLETION_LMR_INVALIDATE_FENCE_FLAG        = 0x20
} DAT_COMPLETION_FLAGS;

typedef DAT_UINT32   DAT_TIMEOUT;          /* microseconds
*/

/* timeout = infinity */
#define DAT_TIMEOUT_INFINITE    ((DAT_TIMEOUT) ~0)

/* dat handles */
typedef DAT_PVOID       DAT_HANDLE;
typedef DAT_HANDLE      DAT_CR_HANDLE;
typedef DAT_HANDLE      DAT_EP_HANDLE;
typedef DAT_HANDLE      DAT_EVD_HANDLE;
typedef DAT_HANDLE      DAT_IA_HANDLE;
typedef DAT_HANDLE      DAT_LMR_HANDLE;
typedef DAT_HANDLE      DAT_PSP_HANDLE;
typedef DAT_HANDLE      DAT_PZ_HANDLE;
typedef DAT_HANDLE      DAT_RMR_HANDLE;
typedef DAT_HANDLE      DAT_RSP_HANDLE;
typedef DAT_HANDLE      DAT_SRQ_HANDLE;
typedef DAT_HANDLE      DAT_CSP_HANDLE;

typedef enum dat_dtos
    {
    DAT_DTO_SEND,
    DAT_DTO_RDMA_WRITE,
    DAT_DTO_RDMA_READ,
    DAT_DTO_RECEIVE,
    DAT_DTO_RECEIVE_WITH_INVALIDATE,
    DAT_DTO_LMR_FMR, /* kdat specific */
    DAT_DTO_LMR_INVALIDATE /* kdat specific */
#ifdef DAT_EXTENSIONS
   ,DAT_DTO_EXTENSION_BASE /* To be used by DAT extensions
         as a starting point of extension DTOs */
#endif /* DAT_EXTENSIONS */
} DAT_DTOS;
```

```
/* dat NULL handles */
#define DAT_HANDLE_NULL ((DAT_HANDLE)NULL)

typedef DAT_SOCK_ADDR*     DAT_IA_ADDRESS_PTR;
typedef DAT_UINT64      DAT_CONN_QUAL;
typedef DAT_UINT64      DAT_PORT_QUAL;

/* QOS definitions */
typedef enum dat_qos
    {
    DAT_QOS_BEST_EFFORT          = 0x00,
    DAT_QOS_HIGH_THROUGHPUT      = 0x01,
    DAT_QOS_LOW_LATENCY          = 0x02,
        /* not low latency, nor high throughput   */
    DAT_QOS_ECONOMY              = 0x04,
        /* both low latency and high throughput   */
    DAT_QOS_PREMIUM              = 0x08
    } DAT_QOS;

/*
 * FLAGS
 */
   /* for backward compatibility */
#define DAT_CONNECT_MULTIPATH_REQUESTED_FLAG DAT_CONNECT_
MULTIPATH_FLAG
typedef enum dat_connect_flags
    {
    DAT_CONNECT_DEFAULT_FLAG     = 0x00,
    DAT_CONNECT_MULTIPATH_REQUESTED_FLAG = 0x01,
    DAT_CONNECT_MULTIPATH_REQUIRED_FLAG = 0x02
    } DAT_CONNECT_FLAGS;

typedef enum dat_close_flags
    {
    DAT_CLOSE_ABRUPT_FLAG        = 0x00,
    DAT_CLOSE_GRACEFUL_FLAG      = 0x01
    } DAT_CLOSE_FLAGS;

#define DAT_CLOSE_DEFAULT DAT_CLOSE_ABRUPT_FLAG
```

```
typedef enum dat_evd_flags
    {
    DAT_EVD_SOFTWARE_FLAG          = 0x001,
    DAT_EVD_CR_FLAG                = 0x010,
    DAT_EVD_DTO_FLAG               = 0x020,
    DAT_EVD_CONNECTION_FLAG        = 0x040,
    DAT_EVD_RMR_BIND_FLAG          = 0x080,
    DAT_EVD_ASYNC_FLAG             = 0x100,
/* DAT events only, no software events */
    DAT_EVD_DEFAULT_FLAG           = 0x1F0
#ifdef DAT_EXTENSIONS
/* To be used by DAT extensions as a starting point for ex-
tended evd flags */
    ,DAT_EVD_EXTENSION_BASE        = 0x200
#endif /* DAT_EXTENSIONS */
    } DAT_EVD_FLAGS;

typedef enum dat_psp_flags
    {
   DAT_PSP_CONSUMER_FLAG = 0x00, /* Consumer creates an End-
point */
    DAT_PSP_PROVIDER_FLAG = 0x01 /* Provider creates an End-
point */

    } DAT_PSP_FLAGS;

/*
 * Memory Buffers
 *
 * Both LMR and RMR triplets specify 64-bit addresses in the
local host's byte
 * order, even when that exceeds the size of a DAT_PVOID for
the host
 * architecture.
 *
*/

/*
```

```
* Both LMR and RMR Triplets specify 64-bit addresses in the local host
* order, even when that exceeds the size of a void pointer for the host
* architecture. The DAT_VADDR type that represents addresses is in the
* native byte-order of the local host. Helper macros that allow Consumers
* to convert DAT_VADDR into various orders that might be useful for
* inclusion of RMR Triplets into a payload of a message follow.
*
* DAT defines the following macros to convert the fields on an RMR Triplet
* to defined byte orders to allow their export by the Consumer over wire
* protocols. DAT does not define how the two peers decide which byte should be
* used.
*
* DAT_LMRC_TO_LSB(lmrc) returns the supplied LMR Context in ls-byte
* order.
* DAT_LMRC_TO_MSB(lmrc) returns the supplied LMR Context in ms-byte
* order.
* DAT_RMRC_TO_LSB(rmrc) returns the supplied RMR Context in ls-byte
* order.
* DAT_RMRC_TO_MSB(rmrc) returns the supplied RMR Context in ms-byte
* order.
* DAT_VADDR_TO_LSB(vaddr) returns the supplied Virtual Address in ls-byte
* order.
* DAT_VADDR_TO_MSB(vaddr) returns the supplied Virtual Address in
* ms-byte order.
* DAT_VLEN_TO_LSB(vlen) returns the supplied length in ls-byte order.
* DAT_VLEN_TO_MSB(vlen) returns the supplied length in ms-byte order.
```

```
*
* Consumers are free to use 64-bit or 32-bit arithmetic for
local or remote
* memory address and length manipulation in their preferred
byte-order. Only the
* LMR and RMR Triplets passed to a Provider as part of a
Posted DTO are
* required to be in 64-bit address and local host order for-
mats. Providers shall
* convert RMR_Triplets to a Transport-required wire format.
*
* For the best performance, Consumers should align each
buffer segment to
* the boundary specified by the dat_optimal_alignment.
*/

typedef DAT_UINT32      DAT_LMR_CONTEXT;
typedef DAT_UINT32      DAT_RMR_CONTEXT;

typedef DAT_UINT64      DAT_VLEN;
typedef DAT_UINT64      DAT_VADDR;
typedef DAT_UINT32         DAT_SEG_LENGTH; /* The maximum
data segment length */

typedef struct dat_provider_attr  DAT_PROVIDER_ATTR;
typedef struct dat_evd_param      DAT_EVD_PARAM;
typedef struct dat_lmr_param      DAT_LMR_PARAM;
typedef enum   dat_lmr_param_mask      DAT_LMR_PARAM_MASK;

   /* It is legal for the Consumer to specify zero for
segment_length
   * of the dat_lmr_triplet. When 0 is specified for the
   * segment_length then the other two elements of the
   * dat_lmr_triplet are irrelevant and can be invalid.
   */

typedef struct dat_lmr_triplet
    {
    DAT_VADDR        virtual_address;/* 64-bit address */
    DAT_SEG_LENGTH   segment_length; /* 32-bit length */
    DAT_LMR_CONTEXT lmr_context; /* 32-bit lmr_context */
```

```
    } DAT_LMR_TRIPLET;

typedef struct dat_rmr_triplet
    {
    DAT_VADDR        virtual_address;/* 64-bit address */
    DAT_SEG_LENGTH   segment_length; /* 32-bit length */
    DAT_RMR_CONTEXT rmr_context; /* 32-bit rmr_context */
    } DAT_RMR_TRIPLET;

/* Memory privileges */

typedef enum dat_mem_priv_flags
    {
    DAT_MEM_PRIV_NONE_FLAG               = 0x00,
    DAT_MEM_PRIV_LOCAL_READ_FLAG         = 0x01,
    DAT_MEM_PRIV_REMOTE_READ_FLAG        = 0x02,
    DAT_MEM_PRIV_LOCAL_WRITE_FLAG        = 0x10,
    DAT_MEM_PRIV_REMOTE_WRITE_FLAG       = 0x20,
    DAT_MEM_PRIV_ALL_FLAG                = 0x33
#ifdef DAT_EXTENSIONS
   /* To be used by DAT extensions as a starting
   point of extension memory privileges */
   ,DAT_MEM_PRIV_EXTENSION_BASE          = 0x40
#endif /* DAT_EXTENSIONS */
} DAT_MEM_PRIV_FLAGS;

   /* For backward compatibility with DAT-1.0, memory priv-
ileges values */
   /* are supported */
#define DAT_MEM_PRIV_READ_FLAG   (DAT_MEM_PRIV_LOCAL_READ_
FLAG | DAT_MEM_PRIV_REMOTE_READ_FLAG)
#define DAT_MEM_PRIV_WRITE_FLAG (DAT_MEM_PRIV_LOCAL_WRITE_
FLAG | DAT_MEM_PRIV_REMOTE_WRITE_FLAG)

/* LMR VA types */
typedef enum dat_va_type
    {
    DAT_VA_TYPE_VA                       = 0x0,
    DAT_VA_TYPE_ZB                       = 0x1
    } DAT_VA_TYPE;
```

```
/* RMR Arguments & RMR Arguments Mask */
   /* DAPL 2.0 addition */
   /* Defines RMR protection scope */
typedef enum dat_rmr_scope
   {
   DAT_RMR_SCOPE_EP,  /* bound to at most one EP at a time.
*/
   DAT_RMR_SCOPE_PZ, /* bound to a Protection Zone */
   DAT_RMR_SCOPE_ANY /* Supports all types */
   } DAT_RMR_SCOPE;

typedef struct dat_rmr_param
    {
    DAT_IA_HANDLE       ia_handle;
    DAT_PZ_HANDLE       pz_handle;
    DAT_LMR_TRIPLET     lmr_triplet;
    DAT_MEM_PRIV_FLAGS mem_priv;
    DAT_RMR_CONTEXT     rmr_context;
    DAT_RMR_SCOPE          rmr_scope;
    DAT_VA_TYPE            va_type;
    } DAT_RMR_PARAM;

typedef enum dat_rmr_param_mask
    {
    DAT_RMR_FIELD_IA_HANDLE              = 0x01,
    DAT_RMR_FIELD_PZ_HANDLE              = 0x02,
    DAT_RMR_FIELD_LMR_TRIPLET            = 0x04,
    DAT_RMR_FIELD_MEM_PRIV               = 0x08,
    DAT_RMR_FIELD_RMR_CONTEXT            = 0x10,
    DAT_RMR_FIELD_RMR_SCOPE             = 0x20,
    DAT_RMR_FIELD_VA_TYPE               = 0x40,

    DAT_RMR_FIELD_ALL                    = 0x7F
    } DAT_RMR_PARAM_MASK;

/* Provider attributes */

typedef enum dat_iov_ownership
```

```
    {
        /* Not a modification by the Provider; the Consumer can use anytime.     */
    DAT_IOV_CONSUMER                          = 0x0,

        /* Provider does not modify returned IOV DTO on completion.    */
    DAT_IOV_PROVIDER_NOMOD                    = 0x1,

        /* Provider can modify IOV DTO on completion; can't trust it.  */
    DAT_IOV_PROVIDER_MOD                      = 0x2

    } DAT_IOV_OWNERSHIP;

typedef enum dat_ep_creator_for_psp
    {
    DAT_PSP_CREATES_EP_NEVER,     /* Provider never creates Endpoint. */
    DAT_PSP_CREATES_EP_IFASKED,   /* Provider creates Endpoint if asked. */
    DAT_PSP_CREATES_EP_ALWAYS     /* Provider always creates Endpoint. */
    } DAT_EP_CREATOR_FOR_PSP;

/* General Interface Adapter attributes. These apply to both udat and kdat. */

/* To support backwards compatibility for DAPL-1.0 */
#define max_rdma_read_per_ep max_rdma_read_per_ep_in
#define DAT_IA_FIELD_IA_MAX_DTO_PER_OP DAT_IA_FIELD_IA_MAX_DTO_PER_EP_IN

/* To support backwards compatibility for DAPL-1.0 & DAPL-1.1 */
#define max_mtu_size max_message_size

#ifdef DAT_EXTENSIONS
   /* DAPL 2.0 addition */
   /* Defines extensions */
typedef enum dat_extension
```

```
    {
    DAT_EXTENSION_IB,   /* IB extension. */
    DAT_EXTENSION_IW,   /* iWARP extension. */
    DAT_EXTENSION_NONE /* no extension supported. */
    } DAT_EXTENSION;
#endif /* DAT_EXTENSIONS */

typedef struct dat_ia_attr DAT_IA_ATTR;

/* To support backwards compatibility for DAPL-1.0 & DAPL-1.1 */
#define DAT_IA_FIELD_IA_MAX_MTU_SIZE DAT_IA_FIELD_IA_MAX_MESSAGE_SIZE

typedef DAT_UINT64 DAT_IA_ATTR_MASK;

#define DAT_IA_FIELD_IA_ADAPTER_NAME UINT64_C(0x000000001)
#define DAT_IA_FIELD_IA_VENDOR_NAME UINT64_C(0x000000002)
#define DAT_IA_FIELD_IA_HARDWARE_MAJOR_VERSION UINT64_C(0x000000004)
#define DAT_IA_FIELD_IA_HARDWARE_MINOR_VERSION UINT64_C(0x000000008)
#define DAT_IA_FIELD_IA_FIRMWARE_MAJOR_VERSION UINT64_C(0x000000010)
#define DAT_IA_FIELD_IA_FIRMWARE_MINOR_VERSION UINT64_C(0x000000020)
#define DAT_IA_FIELD_IA_ADDRESS_PTR UINT64_C(0x000000040)
#define DAT_IA_FIELD_IA_MAX_EPS UINT64_C(0x000000080)
#define DAT_IA_FIELD_IA_MAX_DTO_PER_EP UINT64_C(0x000000100)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_PER_EP_IN UINT64_C(0x000000200)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_PER_EP_OUT UINT64_C(0x000000400)
#define DAT_IA_FIELD_IA_MAX_EVDS UINT64_C(0x000000800)
#define DAT_IA_FIELD_IA_MAX_EVD_QLEN UINT64_C(0x000001000)
#define DAT_IA_FIELD_IA_MAX_IOV_SEGMENTS_PER_DTO UINT64_C(0x000002000)
#define DAT_IA_FIELD_IA_MAX_LMRS UINT64_C(0x000004000)
#define DAT_IA_FIELD_IA_MAX_LMR_BLOCK_SIZE UINT64_C(0x000008000)
```

```
#define DAT_IA_FIELD_IA_MAX_LMR_VIRTUAL_ADDRESS UINT64_C(0x000010000)
#define DAT_IA_FIELD_IA_MAX_PZS UINT64_C(0x000020000)
#define DAT_IA_FIELD_IA_MAX_MESSAGE_SIZE UINT64_C(0x000040000)
#define DAT_IA_FIELD_IA_MAX_RDMA_SIZE UINT64_C(0x000080000)
#define DAT_IA_FIELD_IA_MAX_RMRS UINT64_C(0x000100000)
#define DAT_IA_FIELD_IA_MAX_RMR_TARGET_ADDRESS UINT64_C(0x000200000)
#define DAT_IA_FIELD_IA_MAX_SRQS UINT64_C(0x000400000)
#define DAT_IA_FIELD_IA_MAX_EP_PER_SRQ UINT64_C(0x000800000)
#define DAT_IA_FIELD_IA_MAX_RECV_PER_SRQ UINT64_C(0x001000000)
#define DAT_IA_FIELD_IA_MAX_IOV_SEGMENTS_PER_RDMA_READ UINT64_C(0x002000000)
#define DAT_IA_FIELD_IA_MAX_IOV_SEGMENTS_PER_RDMA_WRITE UINT64_C(0x004000000)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_IN UINT64_C(0x008000000)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_OUT UINT64_C(0x010000000)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_PER_EP_IN_GUARANTEED UINT64_C(0x020000000)
#define DAT_IA_FIELD_IA_MAX_RDMA_READ_PER_EP_OUT_GUARANTEED UINT64_C(0x040000000)
#define DAT_IA_FIELD_IA_ZB_SUPPORTED UINT64_C(0x080000000)

/* To support backwards compatibility for DAPL-1.0 & DAPL-1.1 */
#define DAT_IA_ALL DAT_IA_FIELD_ALL

#define DAT_IA_FIELD_NONE UINT64_C(0x0)

/* Endpoint attributes */

typedef enum dat_service_type
    {
    DAT_SERVICE_TYPE_RC          /* reliable connections */
    } DAT_SERVICE_TYPE;

typedef struct dat_ep_attr
```

```
{
    DAT_SERVICE_TYPE            service_type;
    DAT_SEG_LENGTH               max_message_size;
    DAT_SEG_LENGTH               max_rdma_size;
    DAT_QOS                     qos;
    DAT_COMPLETION_FLAGS        recv_completion_flags;
    DAT_COMPLETION_FLAGS        request_completion_flags;
    DAT_COUNT                   max_recv_dtos;
    DAT_COUNT                   max_request_dtos;
    DAT_COUNT                   max_recv_iov;
    DAT_COUNT                   max_request_iov;
    DAT_COUNT                   max_rdma_read_in;
    DAT_COUNT                   max_rdma_read_out;
    DAT_COUNT                   srq_soft_hw;
    DAT_COUNT                   max_rdma_read_iov;
    DAT_COUNT                   max_rdma_write_iov;

    DAT_COUNT                   ep_transport_specific_
count;
    DAT_NAMED_ATTR *            ep_transport_specific;
    DAT_COUNT                   ep_provider_specific_
count;
    DAT_NAMED_ATTR *           ep_provider_specific;
    } DAT_EP_ATTR;

/* Endpoint Parameters */

/* For backwards compatibility */
#define DAT_EP_STATE_ERROR DAT_EP_STATE_DISCONNECTED

typedef enum dat_ep_state
{
    DAT_EP_STATE_UNCONNECTED,  /* quiescent state */
    DAT_EP_STATE_UNCONFIGURED_UNCONNECTED,
    DAT_EP_STATE_RESERVED,
    DAT_EP_STATE_UNCONFIGURED_RESERVED,
    DAT_EP_STATE_PASSIVE_CONNECTION_PENDING,
    DAT_EP_STATE_UNCONFIGURED_PASSIVE,
    DAT_EP_STATE_ACTIVE_CONNECTION_PENDING,
```

```
        DAT_EP_STATE_TENTATIVE_CONNECTION_PENDING,
        DAT_EP_STATE_UNCONFIGURED_TENTATIVE,
        DAT_EP_STATE_CONNECTED,
        DAT_EP_STATE_DISCONNECT_PENDING,
        DAT_EP_STATE_DISCONNECTED,
        DAT_EP_STATE_COMPLETION_PENDING,
        DAT_EP_STATE_CONNECTED_SINGLE_PATH,
        DAT_EP_STATE_CONNECTED_MULTI_PATH
    } DAT_EP_STATE;

    typedef struct dat_ep_param
        {
        DAT_IA_HANDLE                ia_handle;
        DAT_EP_STATE                 ep_state;
        DAT_COMM                      comm;
        DAT_IA_ADDRESS_PTR           local_ia_address_ptr;
        DAT_PORT_QUAL                local_port_qual;
        DAT_IA_ADDRESS_PTR           remote_ia_address_ptr;
        DAT_PORT_QUAL                remote_port_qual;
        DAT_PZ_HANDLE                pz_handle;
        DAT_EVD_HANDLE               recv_evd_handle;
        DAT_EVD_HANDLE               request_evd_handle;
        DAT_EVD_HANDLE               connect_evd_handle;
        DAT_SRQ_HANDLE                srq_handle;
         DAT_EP_ATTR                  ep_attr;
        } DAT_EP_PARAM;

    typedef DAT_UINT64 DAT_EP_PARAM_MASK;

    #define DAT_EP_FIELD_IA_HANDLE UINT64_C(0x00000001)
    #define DAT_EP_FIELD_EP_STATE UINT64_C(0x00000002)
    #define DAT_EP_FIELD_COMM UINT64_C(0x00000004)
    #define DAT_EP_FIELD_LOCAL_IA_ADDRESS_PTR UINT64_
    C(0x00000008)
    #define DAT_EP_FIELD_LOCAL_PORT_QUAL UINT64_C(0x00000010)
    #define DAT_EP_FIELD_REMOTE_IA_ADDRESS_PTR UINT64_
    C(0x00000020)
    #define DAT_EP_FIELD_REMOTE_PORT_QUAL UINT64_C(0x00000040)
    #define DAT_EP_FIELD_PZ_HANDLE UINT64_C(0x00000080)
```

```
#define DAT_EP_FIELD_RECV_EVD_HANDLE UINT64_C(0x00000100)
#define DAT_EP_FIELD_REQUEST_EVD_HANDLE UINT64_
C(0x00000200)
#define DAT_EP_FIELD_CONNECT_EVD_HANDLE UINT64_
C(0x00000400)
#define DAT_EP_FIELD_SRQ_HANDLE UINT64_C(0x00000800)

    /* Remainder of values from EP_ATTR, 0x00001000 and up */

#define DAT_EP_FIELD_EP_ATTR_SERVICE_TYPE UINT64_
C(0x00001000)
#define DAT_EP_FIELD_EP_ATTR_MAX_MESSAGE_SIZE UINT64_
C(0x00002000)
#define DAT_EP_FIELD_EP_ATTR_MAX_RDMA_SIZE UINT64_
C(0x00004000)
#define DAT_EP_FIELD_EP_ATTR_QOS UINT64_C(0x00008000)

#define DAT_EP_FIELD_EP_ATTR_RECV_COMPLETION_FLAGS UINT64_
C(0x00010000)
#define DAT_EP_FIELD_EP_ATTR_REQUEST_COMPLETION_FLAGS
UINT64_C(0x00020000)
#define DAT_EP_FIELD_EP_ATTR_MAX_RECV_DTOS UINT64_
C(0x00040000)
#define DAT_EP_FIELD_EP_ATTR_MAX_REQUEST_DTOS UINT64_
C(0x00080000)

#define DAT_EP_FIELD_EP_ATTR_MAX_RECV_IOV UINT64_
C(0x00100000)
#define DAT_EP_FIELD_EP_ATTR_MAX_REQUEST_IOV UINT64_
C(0x00200000)

#define DAT_EP_FIELD_EP_ATTR_MAX_RDMA_READ_IN UINT64_
C(0x00400000)
#define DAT_EP_FIELD_EP_ATTR_MAX_RDMA_READ_OUT UINT64_
C(0x00800000)

#define  DAT_EP_FIELD_EP_ATTR_SRQ_SOFT_HWUINT64_
C(0x01000000)

#define  DAT_EP_FIELD_EP_ATTR_MAX_RDMA_READ_IOV UINT64_
C(0x02000000)
#define  DAT_EP_FIELD_EP_ATTR_MAX_RDMA_WRITE_IOV UINT64_
C(0x04000000)
```

```
#define  DAT_EP_FIELD_EP_ATTR_NUM_TRANSPORT_ATTR UINT64_
C(0x08000000)
#define  DAT_EP_FIELD_EP_ATTR_TRANSPORT_SPECIFIC_ATTR
UINT64_C(0x10000000)

#define  DAT_EP_FIELD_EP_ATTR_NUM_PROVIDER_ATTR UINT64_
C(0x20000000)
#define  DAT_EP_FIELD_EP_ATTR_PROVIDER_SPECIFIC_ATTR
UINT64_C(0x40000000)

#define DAT_EP_FIELD_EP_ATTR_ALL UINT64_C(0x7FFFF000)
#define DAT_EP_FIELD_ALL UINT64_C(0x7FFFFFFF)

#define DAT_WATERMARK_INFINITE    ((DAT_COUNT) ~0)

#define DAT_HW_DEFAULT DAT_WATERMARK_INFINITE

#define DAT_SRQ_LW_DEFAULT 0x0

typedef enum dat_srq_state
{
    DAT_SRQ_STATE_OPERATIONAL,
    DAT_SRQ_STATE_ERROR
} DAT_SRQ_STATE;

#define DAT_VALUE_UNKNOWN (((DAT_COUNT) ~0)-1)

typedef struct dat_srq_attr
{
        DAT_COUNT    max_recv_dtos;
        DAT_COUNT    max_recv_iov;
        DAT_COUNT    low_watermark;
} DAT_SRQ_ATTR;

typedef struct dat_srq_param
    {
    DAT_IA_HANDLE          ia_handle;
    DAT_SRQ_STATE          srq_state;
    DAT_PZ_HANDLE          pz_handle;
```

```
    DAT_COUNT               max_recv_dtos;
    DAT_COUNT               max_recv_iov;
    DAT_COUNT               low_watermark;
    DAT_COUNT               available_dto_count;
    DAT_COUNT               outstanding_dto_count;
    } DAT_SRQ_PARAM;

typedef enum dat_srq_param_mask
    {
    DAT_SRQ_FIELD_IA_HANDLE                     = 0x001,
    DAT_SRQ_FIELD_SRQ_STATE                     = 0x002,
    DAT_SRQ_FIELD_PZ_HANDLE                     = 0x004,
    DAT_SRQ_FIELD_MAX_RECV_DTO                  = 0x008,
    DAT_SRQ_FIELD_MAX_RECV_IOV                  = 0x010,
    DAT_SRQ_FIELD_LOW_WATERMARK                 = 0x020,
    DAT_SRQ_FIELD_AVAILABLE_DTO_COUNT           = 0x040,
    DAT_SRQ_FIELD_OUTSTANDING_DTO_COUNT         = 0x080,

    DAT_SRQ_FIELD_ALL                           = 0x0FF
    } DAT_SRQ_PARAM_MASK;

/* PZ Parameters */

typedef struct dat_pz_param
    {
    DAT_IA_HANDLE       ia_handle;
    } DAT_PZ_PARAM;

typedef enum dat_pz_param_mask
    {
    DAT_PZ_FIELD_IA_HANDLE      = 0x01,

    DAT_PZ_FIELD_ALL            = 0x01
    } DAT_PZ_PARAM_MASK;

/* PSP Parameters */

typedef struct dat_psp_param
    {
```

```
        DAT_IA_HANDLE                    ia_handle;
        DAT_CONN_QUAL                    conn_qual;
        DAT_EVD_HANDLE                   evd_handle;
        DAT_PSP_FLAGS                    psp_flags;
        } DAT_PSP_PARAM;

    typedef enum dat_psp_param_mask
        {
        DAT_PSP_FIELD_IA_HANDLE           = 0x01,
        DAT_PSP_FIELD_CONN_QUAL           = 0x02,
        DAT_PSP_FIELD_EVD_HANDLE          = 0x04,
        DAT_PSP_FIELD_PSP_FLAGS           = 0x08,

        DAT_PSP_FIELD_ALL                 = 0x0F
        } DAT_PSP_PARAM_MASK;

    /* RSP Parameters */

    typedef struct dat_rsp_param
        {
        DAT_IA_HANDLE    ia_handle;
        DAT_CONN_QUAL    conn_qual;
        DAT_EVD_HANDLE   evd_handle;
        DAT_EP_HANDLE    ep_handle;
        } DAT_RSP_PARAM;

    typedef enum dat_rsp_param_mask
        {
        DAT_RSP_FIELD_IA_HANDLE           = 0x01,
        DAT_RSP_FIELD_CONN_QUAL           = 0x02,
        DAT_RSP_FIELD_EVD_HANDLE          = 0x04,
        DAT_RSP_FIELD_EP_HANDLE           = 0x08,

        DAT_RSP_FIELD_ALL                 = 0x0F
        } DAT_RSP_PARAM_MASK;

    /* CSP Parameters */
    typedef struct dat_csp_param
        {
```

```
    DAT_IA_HANDLE                ia_handle;
    DAT_COMM                      *comm;
    DAT_IA_ADDRESS_PTR            address_ptr;
    DAT_EVD_HANDLE               evd_handle;
    } DAT_CSP_PARAM;

typedef enum dat_csp_param_mask
    {
    DAT_CSP_FIELD_IA_HANDLE       = 0x01,
    DAT_CSP_FIELD_COMM            = 0x02,
    DAT_CSP_FIELD_IA_ADDRESS      = 0x04,
    DAT_CSP_FIELD_EVD_HANDLE      = 0x08,

    DAT_CSP_FIELD_ALL             = 0x0F
    } DAT_CSP_PARAM_MASK;

/* Connection Request Parameters.
 *
 * The Connection Request does not provide Remote Endpoint attributes.
 * If a local Consumer needs this information, the remote Consumer should
 * encode it into Private Data.
 */

typedef struct dat_cr_param
    {
   /* Remote IA whose Endpoint requested the connection. */
    DAT_IA_ADDRESS_PTR           remote_ia_address_ptr;

        /* Port qualifier of the remote Endpoint of the requested connection.  */
    DAT_PORT_QUAL                remote_port_qual;

     /* Size of the Private Data. */
    DAT_COUNT                    private_data_size;
```

```
   /* Pointer to the Private Data passed by remote side in
the Connection
    * Request.
   */
    DAT_PVOID                    private_data;

        /* The local Endpoint provided by the Service Point
for the requested
        * connection. It is the only Endpoint that can accept
a Connection
          * Request on this Service Point. The value DAT_
HANDLE_NULL
    * represents that there is no associated local Endpoint
for the requested
   * connection.
   */
    DAT_EP_HANDLE                local_ep_handle;

    } DAT_CR_PARAM;

typedef enum dat_cr_param_mask
    {
    DAT_CR_FIELD_REMOTE_IA_ADDRESS_PTR = 0x01,
    DAT_CR_FIELD_REMOTE_PORT_QUAL       = 0x02,
    DAT_CR_FIELD_PRIVATE_DATA_SIZE      = 0x04,
    DAT_CR_FIELD_PRIVATE_DATA           = 0x08,
    DAT_CR_FIELD_LOCAL_EP_HANDLE        = 0x10,

    DAT_CR_FIELD_ALL                    = 0x1F
  } DAT_CR_PARAM_MASK;

/**************************
Events******************************/

/* Completion status flags */

    /* DTO completion status */

   /* For backwards compatibility */
#define DAT_DTO_LENGTH_ERROR DAT_DTO_ERR_LOCAL_LENGTH
```

```
#define DAT_DTO_FAILURE    DAT_DTO_ERR_FLUSHED

typedef enum dat_dto_completion_status
    {
    DAT_DTO_SUCCESS                = 0,
   DAT_DTO_ERR_FLUSHED                  = 1,
   DAT_DTO_ERR_LOCAL_LENGTH             = 2,
   DAT_DTO_ERR_LOCAL_EP                 = 3,
   DAT_DTO_ERR_LOCAL_PROTECTION        = 4,
   DAT_DTO_ERR_BAD_RESPONSE             = 5,
   DAT_DTO_ERR_REMOTE_ACCESS            = 6,
   DAT_DTO_ERR_REMOTE_RESPONDER         = 7,
   DAT_DTO_ERR_TRANSPORT                = 8,
   DAT_DTO_ERR_RECEIVER_NOT_READY       = 9,
   DAT_DTO_ERR_PARTIAL_PACKET           = 10,
   DAT_RMR_OPERATION_FAILED             = 11,
   DAT_DTO_ERR_LOCAL_MM_ERROR          = 12 /* kdat spe-
cific */
} DAT_DTO_COMPLETION_STATUS;

   /* RMR completion status */

   /* For backwards compatibility */
#define DAT_RMR_BIND_SUCCESS     DAT_DTO_SUCCESS
#define DAT_RMR_BIND_FAILURE     DAT_DTO_ERR_FLUSHED

/* RMR completion status */

#define DAT_RMR_BIND_COMPLETION_STATUS DAT_DTO_COMPLETION_
STATUS

/* Completion group structs (six total) */

    /* DTO completion event data */
/* transfered_length is not defined if status is not DAT_
SUCCESS */
/*invalidate_flag and rmr_context are not defined if status
is not DAT_SUCCESS */

typedef struct dat_dto_completion_event_data
```

```
    {
    DAT_EP_HANDLE                         ep_handle;
    DAT_DTO_COOKIE                        user_cookie;
    DAT_DTO_COMPLETION_STATUS            status;
    DAT_SEG_LENGTH                       transfered_length;
    DAT_DTOS                             operation;
   DAT_RMR_CONTEXT                       rmr_context;
  } DAT_DTO_COMPLETION_EVENT_DATA;

    /* RMR bind completion event data */
typedef struct dat_rmr_bind_completion_event_data
    {
    DAT_RMR_HANDLE                        rmr_handle;
    DAT_RMR_COOKIE                        user_cookie;
    DAT_RMR_BIND_COMPLETION_STATUS       status;
  } DAT_RMR_BIND_COMPLETION_EVENT_DATA;

typedef union dat_sp_handle
{
        DAT_RSP_HANDLE rsp_handle;
        DAT_PSP_HANDLE psp_handle;
         DAT_CSP_HANDLE csp_handle;
} DAT_SP_HANDLE;

    /* Connection Request Arrival event data */
typedef struct dat_cr_arrival_event_data
    {
   /* Handle to the Service Point that received the Connec-
tion Request from
    * the remote side. If the Service Point was Reserved, sp_
handle is
    * DAT_HANDLE_NULL because the reserved Service Point is
    * automatically destroyed upon generating this event. Can
be PSP, CSP, or RSP.                                   */
    DAT_SP_HANDLE                 sp_handle;

       /* Address of the IA on which the Connection Request
arrived.                   */
    DAT_IA_ADDRESS_PTR            local_ia_address_ptr;
```

```
      /* Connection Qualifier of the IA on which the Service Point received a
       * Connection Request.                                      */
    DAT_CONN_QUAL               conn_qual;

      /* The Connection Request instance created by a Provider for the arrived
       * Connection Request. Consumers can find out private_data passed by a remote
       * Consumer from cr_handle. It is up to a Consumer to dat_cr_accept or
       * dat_cr_reject of the Connection Request.                 */
    DAT_CR_HANDLE               cr_handle;

    /* The binary indicator whether the arrived privata data was trancated or not.
     * The default value of 0 means not truncation of received private data. */
    DAT_BOOLEAN      truncate_flag;

  } DAT_CR_ARRIVAL_EVENT_DATA;

/* Connection event data */
typedef struct dat_connection_event_data
    {
    DAT_EP_HANDLE    ep_handle;
    DAT_COUNT                   private_data_size;
    DAT_PVOID                   private_data;
} DAT_CONNECTION_EVENT_DATA;

/* Async Error event data */
/* For unaffiliated asynchronous event dat_handle is ia_handle. For Endpoint affiliated asynchronous event dat_handle is ep_handle. For EVD affiliated asynchronous event dat_handle is evd_handle. For SRQ affiliated asynchronous event dat_handle is srq_handle. For Memory affiliated asynchronous event
dat_handle is either lmr_handle, rmr_handle or pz_handle. */
typedef struct dat_asynch_error_event_data
    {
```

```
    DAT_HANDLE dat_handle; /* either IA, EP, EVD, SRQ, LMR,
RMR, SP, or PZ handle */
    DAT_COUNT  reason;     /* object specific */
} DAT_ASYNCH_ERROR_EVENT_DATA;

/* The reason is object type specific and its values are de-
fined below. */
typedef enum ia_async_error_reason
    {
   DAT_IA_CATASTROPHIC_ERROR,
   DAT_IA_OTHER_ERROR
} DAT_IA_ASYNC_ERROR_REASON;
typedef enum ep_async_error_reason
    {
   DAT_EP_TRANSFER_TO_ERROR,
   DAT_EP_OTHER_ERROR,
   DAT_SRQ_SOFT_HIGH_WATERMARK_EVENT
} DAT_EP_ASYNC_ERROR_REASON;
typedef enum evd_async_error_reason
    {
   DAT_EVD_OVERFLOW_ERROR,
   DAT_EVD_OTHER_ERROR
} DAT_EVD_ASYNC_ERROR_REASON;
typedef enum srq_async_error_reason
    {
   DAT_SRQ_TRANSFER_TO_ERROR,
   DAT_SRQ_OTHER_ERROR,
   DAT_SRQ_LOW_WATERMARK_EVENT
} DAT_SRQ_ASYNC_ERROR_REASON;
typedef enum lmr_async_error_reason
    {
   DAT_LMR_OTHER_ERROR
} DAT_LMR_ASYNC_ERROR_REASON;
typedef enum rmr_async_error_reason
    {
   DAT_RMR_OTHER_ERROR
} DAT_RMR_ASYNC_ERROR_REASON;
typedef enum pz_async_error_reason
    {
```

```
   DAT_PZ_OTHER_ERROR
} DAT_PZ_ASYNC_ERROR_REASON;

/* Software event data */
typedef struct dat_software_event_data
    {
    DAT_PVOID   pointer;
  } DAT_SOFTWARE_EVENT_DATA;

typedef enum dat_event_number
    {
   DAT_DTO_COMPLETION_EVENT                          = 0x00001,

    DAT_RMR_BIND_COMPLETION_EVENT                          =
0x01001,

   DAT_CONNECTION_REQUEST_EVENT                      = 0x02001,

    DAT_CONNECTION_EVENT_ESTABLISHED                       =
0x04001,
    DAT_CONNECTION_EVENT_PEER_REJECTED                     =
0x04002,
    DAT_CONNECTION_EVENT_NON_PEER_REJECTED                 =
0x04003,
    DAT_CONNECTION_EVENT_ACCEPT_COMPLETION_ERROR           =
0x04004,
    DAT_CONNECTION_EVENT_DISCONNECTED                      =
0x04005,
   DAT_CONNECTION_EVENT_BROKEN                       = 0x04006,
    DAT_CONNECTION_EVENT_TIMED_OUT                         =
0x04007,
    DAT_CONNECTION_EVENT_UNREACHABLE = 0x04008,

   DAT_ASYNC_ERROR_EVD_OVERFLOW                      = 0x08001,
    DAT_ASYNC_ERROR_IA_CATASTROPHIC                        =
0x08002,
   DAT_ASYNC_ERROR_EP_BROKEN                         = 0x08003,
   DAT_ASYNC_ERROR_TIMED_OUT                         = 0x08004,
    DAT_ASYNC_ERROR_PROVIDER_INTERNAL_ERROR                =
0x08005,
```

```
    DAT_HA_DOWN_TO_1                                = 0x08101,
    DAT_HA_UP_TO_MULTI_PATH                         = 0x08102,

   DAT_SOFTWARE_EVENT                               = 0x10001
#ifdef DAT_EXTENSIONS
   ,DAT_EXTENSION_EVENT                             = 0x20000,
   DAT_IB_EXTENSION_RANGE_BASE                      = 0x40000,
   DAT_IW_EXTENSION_RANGE_BASE                      = 0x80000
#endif /* DAT_EXTENSIONS */
} DAT_EVENT_NUMBER;

/* Union for event Data */

typedef union dat_event_data
    {
   DAT_DTO_COMPLETION_EVENT_DATA            dto_completion_
event_data;
    DAT_RMR_BIND_COMPLETION_EVENT_DATA          rmr_
completion_event_data;
    DAT_CR_ARRIVAL_EVENT_DATA    cr_arrival_event_data;
    DAT_CONNECTION_EVENT_DATA    connect_event_data;
    DAT_ASYNCH_ERROR_EVENT_DATA  asynch_error_event_data;
    DAT_SOFTWARE_EVENT_DATA      software_event_data;
} DAT_EVENT_DATA;

/* Event struct that holds all event information */

typedef struct dat_event
    {
    DAT_EVENT_NUMBER event_number;
    DAT_EVD_HANDLE      evd_handle;
    DAT_EVENT_DATA      event_data;
#ifdef DAT_EXTENSIONS
    DAT_UINT64             event_extension_data[8];
#endif /* DAT_EXTENSIONS */
  } DAT_EVENT;

/* Provider/registration info */
```

```
typedef struct dat_provider_info
{
   char              ia_name[DAT_NAME_MAX_LENGTH];
   DAT_UINT32                dapl_version_major;
   DAT_UINT32                dapl_version_minor;
   DAT_BOOLEAN       is_thread_safe;
   } DAT_PROVIDER_INFO;

/**********************************************************
*******
 * FUNCTION PROTOTYPES

**********************************************************
*****/
/*
 * IA functions
 *
 * Note that there are actual 'dat_ia_open' and 'dat_ia_
close'
 * functions, it is not just a re-directing #define. That is
 * because the functions may have to ensure that the provider
 * library is loaded before it can call it, and may choose to
 * unload the library after the last close.
 */
extern DAT_RETURN dat_ia_openv (
   IN       const DAT_NAME_PTR,/* provider             */
   IN       DAT_COUNT,/* asynch_evd_min_qlen  */
   INOUT    DAT_EVD_HANDLE *,/* asynch_evd_handle    */
   OUT      DAT_IA_HANDLE *,/* ia_handle            */
   IN       DAT_UINT32,/* dat major version number */
   IN       DAT_UINT32,/* dat minor version number */
   IN       DAT_BOOLEAN);/* dat thread safety */

#define dat_ia_open(name, qlen, async_evd, ia) \
   dat_ia_openv((name), (qlen), (async_evd), (ia), \
         DAT_VERSION_MAJOR, DAT_VERSION_MINOR, \
         DAT_THREADSAFE)

extern DAT_RETURN dat_ia_query (
   IN       DAT_IA_HANDLE,/* ia_handle            */
```

```
    OUT     DAT_EVD_HANDLE *,/* async_evd_handle     */
    IN      DAT_IA_ATTR_MASK,/* ia_attr_mask         */
    OUT     DAT_IA_ATTR *,/* ia_attr             */
    IN      DAT_PROVIDER_ATTR_MASK,/* provider_attr_mask   */
    OUT     DAT_PROVIDER_ATTR * );/* provider_attr       */

extern DAT_RETURN dat_ia_close (
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
        IN      DAT_CLOSE_FLAGS );/* close_flags      */

/* helper functions */

extern DAT_RETURN dat_set_consumer_context (
        IN      DAT_HANDLE,             /* dat_handle
*/
        IN      DAT_CONTEXT);/* context              */

extern DAT_RETURN dat_get_consumer_context (
        IN      DAT_HANDLE,             /* dat_handle
*/
        OUT     DAT_CONTEXT * );/* context              */

extern DAT_RETURN dat_get_handle_type (
        IN      DAT_HANDLE,       /* dat_handle */
        OUT     DAT_HANDLE_TYPE * );/* handle_type */

/* CR functions */

extern DAT_RETURN dat_cr_query (
        IN      DAT_CR_HANDLE,          /* cr_handle
*/
        IN      DAT_CR_PARAM_MASK,/* cr_param_mask        */
        OUT     DAT_CR_PARAM * ); /* cr_param             */

extern DAT_RETURN dat_cr_accept (
        IN      DAT_CR_HANDLE,          /* cr_handle
*/
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
```

```
        IN      DAT_COUNT,              /* private_data_
size    */
        IN      const DAT_PVOID );/* private_data        */

extern DAT_RETURN dat_cr_reject (
        IN      DAT_CR_HANDLE,          /* cr_handle */
        IN      DAT_COUNT, /* private_data_size    */
        IN      const DAT_PVOID );/* private_data        */

   /* For DAT-1.1 and above, this function is defined for
both uDAPL and
    * kDAPL. For DAT-1.0, it is only defined for uDAPL.
    */
extern DAT_RETURN dat_cr_handoff(
        IN DAT_CR_HANDLE,            /* cr_handle            */
        IN DAT_CONN_QUAL);           /* handoff              */

/* EVD functions */

extern DAT_RETURN dat_evd_resize (
        IN      DAT_EVD_HANDLE,/* evd_handle              */
        IN      DAT_COUNT );            /* evd_min_qlen
*/

extern DAT_RETURN dat_evd_post_se (
        INDAT_EVD_HANDLE,                   /* evd_handle
*/
        INconst DAT_EVENT * );              /* event
*/

extern DAT_RETURN dat_evd_dequeue (
        IN      DAT_EVD_HANDLE,         /* evd_handle
*/
        OUT     DAT_EVENT * );          /* event
*/

extern DAT_RETURN dat_evd_query (
   IN       DAT_EVD_HANDLE,/* evd_handle              */
   IN       DAT_EVD_PARAM_MASK,/* evd_param_mask        */
   OUT      DAT_EVD_PARAM * );/* evd_param            */
```

```
extern DAT_RETURN dat_evd_free (
        IN      DAT_EVD_HANDLE );/* evd_handle           */

/* EP functions */

extern DAT_RETURN dat_ep_create (
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
        IN      DAT_PZ_HANDLE,          /* pz_handle
*/
        IN      DAT_EVD_HANDLE,         /* recv_completion_
evd_handle */
        IN      DAT_EVD_HANDLE,         /* request_
completion_evd_handle */
        IN      DAT_EVD_HANDLE,         /* connect_evd_
handle   */
         IN      const DAT_EP_ATTR *,    /* ep_at-
tributes        */
        OUT     DAT_EP_HANDLE * );/* ep_handle           */

extern DAT_RETURN dat_ep_query (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_EP_PARAM_MASK,/* ep_param_mask       */
        OUT     DAT_EP_PARAM * ); /* ep_param            */

extern DAT_RETURN dat_ep_modify (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_EP_PARAM_MASK,      /* ep_param_mask
*/
        IN      const DAT_EP_PARAM * );/* ep_param
*/

extern DAT_RETURN dat_ep_connect (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_IA_ADDRESS_PTR,    /*remote_ia_address
*/
        IN      DAT_CONN_QUAL,          /*remote_conn_qual
*/
```

```
        IN      DAT_TIMEOUT,            /* tim-
eout                 */
        IN      DAT_COUNT,              /* private_data_
size    */
        IN      const DAT_PVOID,        /* private_data
*/
        IN      DAT_QOS,                /* quality_of_ser-
vice   */
        IN      DAT_CONNECT_FLAGS ); /* connect_flags
*/

extern DAT_RETURN dat_ep_dup_connect (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_EP_HANDLE,          /* ep_dup_handle
*/
        IN      DAT_TIMEOUT,            /* tim-
eout                 */
        IN      DAT_COUNT,              /* private_data_
size    */
        IN      const DAT_PVOID,/* private_data         */
        IN      DAT_QOS);               /* quality_of_ser-
vice   */

extern DAT_RETURN dat_ep_common_connect (
        IN      DAT_EP_HANDLE,          /* ep_handle */
        IN     DAT_IA_ADDRESS_PTR, /* remote_ia_address */
        IN     DAT_TIMEOUT,       /* timeout */
        IN      DAT_COUNT,        /* private_data_size */
        IN      const DAT_PVOID); /* private_data */

extern DAT_RETURN dat_ep_disconnect (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_CLOSE_FLAGS ); /* close_flags         */

extern DAT_RETURN dat_ep_post_send (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_COUNT,              /* num_seg-
ments          */
```

```
        IN      DAT_LMR_TRIPLET *,      /* local_iov
*/
        IN      DAT_DTO_COOKIE,         /* user_cookie
*/
        IN      DAT_COMPLETION_FLAGS );/* completion_flags
*/

extern DAT_RETURN dat_ep_post_send_with_invalidate (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_COUNT,              /* num_seg-
ments           */
        IN      DAT_LMR_TRIPLET *,      /* local_iov
*/
        IN      DAT_DTO_COOKIE,         /* user_cookie
*/
        IN      DAT_COMPLETION_FLAGS, /* completion_flags */
         IN     DAT_BOOLEAN,      /* invalidate flag */
         IN     DAT_RMR_CONTEXT ); /* RMR to invalidate */

extern DAT_RETURN dat_ep_post_recv (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_COUNT,              /* num_seg-
ments           */
        IN      DAT_LMR_TRIPLET *,      /* local_iov
*/
        IN      DAT_DTO_COOKIE,         /* user_cookie
*/
        IN      DAT_COMPLETION_FLAGS );/* completion_flags
*/

extern DAT_RETURN dat_ep_post_rdma_read (
        IN      DAT_EP_HANDLE,          /* ep_handle
*/
        IN      DAT_COUNT,              /* num_seg-
ments           */
        IN      DAT_LMR_TRIPLET *,      /* local_iov
*/
        IN      DAT_DTO_COOKIE,         /* user_cookie
*/
        IN      const DAT_RMR_TRIPLET *,        /* remote_
iov             */
```

```
        IN      DAT_COMPLETION_FLAGS );/* completion_flags
*/

extern DAT_RETURN dat_ep_post_rdma_read_to_rmr (
        IN      DAT_EP_HANDLE,            /* ep_handle
*/
        IN      const DAT_RMR_TRIPLET *, /* local_iov
*/
        IN      DAT_DTO_COOKIE,           /* user_cookie
*/
        IN      const DAT_RMR_TRIPLET *, /* remote_iov
*/
        IN      DAT_COMPLETION_FLAGS );/* completion_flags
*/

extern DAT_RETURN dat_ep_post_rdma_write (
        IN      DAT_EP_HANDLE,            /* ep_handle
*/
        IN      DAT_COUNT,                /* num_seg-
ments         */
        IN      DAT_LMR_TRIPLET *,        /* local_iov
*/
        IN      DAT_DTO_COOKIE,           /* user_cookie
*/
        IN      const DAT_RMR_TRIPLET *,          /* remote_
iov           */
        IN      DAT_COMPLETION_FLAGS );/* completion_flags
*/

extern DAT_RETURN dat_ep_get_status (
        IN      DAT_EP_HANDLE,            /* ep_handle
*/
        OUT     DAT_EP_STATE *,           /* ep_state
*/
        OUT     DAT_BOOLEAN *,            /* recv_idle
*/
        OUT     DAT_BOOLEAN * );/* request_idle         */

extern DAT_RETURN dat_ep_free (
       IN      DAT_EP_HANDLE);        /* ep_handle          */

extern DAT_RETURN dat_ep_reset (
       IN      DAT_EP_HANDLE);        /* ep_handle          */
```

```
extern DAT_RETURN dat_ep_create_with_srq(
        IN      DAT_IA_HANDLE,          /* ia_handle            */
        IN      DAT_PZ_HANDLE,          /* pz_handle            */
        IN      DAT_EVD_HANDLE,         /* recv_evd_handle      */
        IN      DAT_EVD_HANDLE,         /* request_evd_handle
*/
        IN      DAT_EVD_HANDLE,         /* connect_evd_handle
*/
        IN      DAT_SRQ_HANDLE,         /* srq_handle           */
          IN const DAT_EP_ATTR *,   /* ep_attributes        */
        OUT     DAT_EP_HANDLE *);       /* ep_handle            */

extern DAT_RETURN dat_ep_recv_query(
        IN      DAT_EP_HANDLE,          /* ep_handle            */
        OUT     DAT_COUNT *,            /* nbufs_allocated      */
        OUT     DAT_COUNT *);           /* bufs_alloc_span      */

extern DAT_RETURN dat_ep_set_watermark(
        IN      DAT_EP_HANDLE,          /* ep_handle            */
        IN      DAT_COUNT,              /* soft_high_watermark  */
        IN      DAT_COUNT);             /* hard_high_watermark  */

/* LMR functions */

extern DAT_RETURN dat_lmr_free (
          IN      DAT_LMR_HANDLE);/* lmr_handle             */

/* Non-coherent memory functions */

extern DAT_RETURN dat_lmr_sync_rdma_read(
    IN      DAT_IA_HANDLE,          /* ia_handle            */
    IN      const DAT_LMR_TRIPLET *, /* local_segments
*/
    IN      DAT_VLEN);              /* num_segments         */

extern DAT_RETURN dat_lmr_sync_rdma_write(
    IN      DAT_IA_HANDLE,          /* ia_handle            */
    IN      const DAT_LMR_TRIPLET *, /* local_segments */
    IN      DAT_VLEN);              /* num_segments         */
```

```
/* RMR functions */

extern DAT_RETURN dat_rmr_create (
        IN      DAT_PZ_HANDLE,          /* pz_handle */
        OUT     DAT_RMR_HANDLE *);/* rmr_handle          */

extern DAT_RETURN dat_rmr_create_for_ep (
        IN      DAT_PZ_HANDLE,          /* pz_handle
*/
        OUT     DAT_RMR_HANDLE *);      /* rmr_handle
*/

extern DAT_RETURN dat_rmr_query (
        IN      DAT_RMR_HANDLE,         /* rmr_handle */
        IN      DAT_RMR_PARAM_MASK,/* rmr_param_mask
*/
        OUT     DAT_RMR_PARAM *);       /* rmr_param */

extern DAT_RETURN dat_rmr_bind (
        IN      DAT_RMR_HANDLE,         /* rmr_handle */
        IN      DAT_LMR_HANDLE,         /* lmr_handle */
        IN      const DAT_LMR_TRIPLET *, /* lmr_triplet */
        IN      DAT_MEM_PRIV_FLAGS,     /* mem_priv */
        IN      DAT_VA_TYPE,             /* va_type */
        IN      DAT_EP_HANDLE,          /* ep_handle */
        IN      DAT_RMR_COOKIE,         /* user_cookie */
        IN      DAT_COMPLETION_FLAGS,/* completion_flags */
        OUT     DAT_RMR_CONTEXT * );    /* context */

extern DAT_RETURN dat_rmr_free (
        IN      DAT_RMR_HANDLE);/* rmr_handle          */

/* PSP functions */

extern DAT_RETURN dat_psp_create (
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
        IN      DAT_CONN_QUAL,          /* conn_qual
*/
```

```
            IN      DAT_EVD_HANDLE,         /* evd_handle
*/
            IN      DAT_PSP_FLAGS,          /* psp_flags
*/
            OUT     DAT_PSP_HANDLE * );/* psp_handle          */

extern DAT_RETURN dat_psp_create_any (
            IN      DAT_IA_HANDLE,          /* ia_handle
*/
            OUT DAT_CONN_QUAL *,          /* conn_qual
*/
            IN      DAT_EVD_HANDLE,         /* evd_handle
*/
            IN      DAT_PSP_FLAGS,          /* psp_flags
*/
            OUT     DAT_PSP_HANDLE * );/* psp_handle          */

extern DAT_RETURN dat_psp_query (
            IN      DAT_PSP_HANDLE,  /* psp_handle            */
            IN      DAT_PSP_PARAM_MASK,/* psp_param_mask      */
            OUT     DAT_PSP_PARAM * ); /* psp_param           */

extern DAT_RETURN dat_psp_free (
          IN      DAT_PSP_HANDLE );      /* psp_handle         */

/* RSP functions */

extern DAT_RETURN dat_rsp_create (
            IN      DAT_IA_HANDLE,          /* ia_handle
*/
            IN      DAT_CONN_QUAL,          /* conn_qual
*/
            IN      DAT_EP_HANDLE,          /* ep_handle
*/
            IN      DAT_EVD_HANDLE,         /* evd_handle
*/
            OUT     DAT_RSP_HANDLE * );/* rsp_handle          */

extern DAT_RETURN dat_rsp_query (
            IN      DAT_RSP_HANDLE,  /* rsp_handle            */
            IN      DAT_RSP_PARAM_MASK,/* rsp_param_mask      */
```

```
    OUT     DAT_RSP_PARAM * );    /* rsp_param           */

extern DAT_RETURN dat_rsp_free (
        IN      DAT_RSP_HANDLE );      /* rsp_handle          */

/* CSP functions */

extern DAT_RETURN dat_csp_create (
        IN      DAT_IA_HANDLE,           /* ia_handle */
        IN      DAT_COMM *,           /* communicator */
        IN      DAT_IA_ADDRESS_PTR,     /* address */
        IN      DAT_EVD_HANDLE,          /* evd_handle */
        OUT     DAT_CSP_HANDLE * ); /* csp_handle */

extern DAT_RETURN dat_csp_query (
        IN      DAT_CSP_HANDLE,   /* csp_handle           */
        IN      DAT_CSP_PARAM_MASK,/* csp_param_mask
*/
    OUT     DAT_CSP_PARAM * );    /* csp_param           */

extern DAT_RETURN dat_csp_free (
        IN      DAT_CSP_HANDLE );      /* csp_handle          */

/* PZ functions */

extern DAT_RETURN dat_pz_create (
        IN      DAT_IA_HANDLE,           /* ia_handle
*/
        OUT     DAT_PZ_HANDLE * );/* pz_handle           */

extern DAT_RETURN dat_pz_query (
        IN      DAT_PZ_HANDLE,           /* pz_handle
*/
        IN      DAT_PZ_PARAM_MASK,/* pz_param_mask       */
       OUT     DAT_PZ_PARAM *);       /* pz_param           */

extern DAT_RETURN dat_pz_free (
       IN      DAT_PZ_HANDLE );       /* pz_handle           */

/* SRQ functions */
```

```
extern DAT_RETURN dat_srq_create(
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
        IN      DAT_PZ_HANDLE,          /* pz_handle
*/
        IN      DAT_SRQ_ATTR *,         /* srq_attr
*/
        OUT     DAT_SRQ_HANDLE *); /* srq_handle         */

extern DAT_RETURN dat_srq_free(
   IN      DAT_SRQ_HANDLE);        /* srq_handle         */

extern DAT_RETURN dat_srq_post_recv(
   IN      DAT_SRQ_HANDLE,         /* srq_handle         */
   IN      DAT_COUNT,              /* num_segments       */
   IN      DAT_LMR_TRIPLET *,      /* local_iov          */
   IN      DAT_DTO_COOKIE);        /* user_cookie        */

extern DAT_RETURN dat_srq_query(
   IN      DAT_SRQ_HANDLE,         /* srq_handle         */
   IN      DAT_SRQ_PARAM_MASK,     /* srq_param_mask     */
   OUT     DAT_SRQ_PARAM *);       /* srq_param          */

extern DAT_RETURN dat_srq_resize(
   IN      DAT_SRQ_HANDLE,         /* srq_handle         */
   IN      DAT_COUNT);             /* srq_max_recv_dto
*/

extern DAT_RETURN dat_srq_set_lw(
   IN      DAT_SRQ_HANDLE,         /* srq_handle         */
   IN      DAT_COUNT);             /* low_watermark      */

#ifdef DAT_EXTENSIONS
typedef int    DAT_EXTENDED_OP;
extern DAT_RETURN dat_extension_op(
   IN    DAT_HANDLE,        /* handle */
   IN    DAT_EXTENDED_OP,   /* operation */
   IN    ... );             /* args */
#endif
```

```
/*
* DAT registry functions.
*
* Note the dat_ia_open and dat_ia_close functions are linked
to
* registration code which "redirects" to the appropriate
provider.
*/

extern DAT_RETURN dat_registry_list_providers(
   IN    DAT_COUNT,          /* max_to_return */
   OUT DAT_COUNT *,          /* entries_returned */
   OUT DAT_PROVIDER_INFO *(dat_provider_list[])); /* dat_
provider_list */

/*
* DAT error functions.
*/
extern DAT_RETURN dat_strerror (
   IN      DAT_RETURN,       /* dat function return */
   OUT const char ** ,       /* major message string */
   OUT const char ** );      /* minor message string */

#endif /* _DAT_H_ */
```

A.5 GENERIC STATUS CODES

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All
rights reserved.
 *
 * This Software is licensed under one of the following li-
censes:
 *
 * 1) under the terms of the “Common Public License 1.0".
The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
```

```
 * 2) under the terms of the "BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 */

/**********************************************************************
 *
 * HEADER: dat_error.h
```

```
 *
 * PURPOSE: DAT return codes
 *
 * Description: Header file for "DAPL: Direct Access Pro-
gramming
 *       Library, Version: 2.0"
 *
* Mapping rules: Error types are compound types, as mapped
out below.
*
 *********************************************************/

#ifndef _DAT_ERROR_H_
#define _DAT_ERROR_H_

/*
 *
 * All return codes are actually a 3-way tuple:
 *
 * type: DAT_RETURN_CLASS DAT_RETURN_TYPE DAT_RETURN_SUBTYPE
 * bits: 31-30       29-16             15-0
 *
 * +-------------------------------------------------------
---------------+
 * |3130 | 2928272625242322212019181 7 |
1615141312111009080706054003020100|
 * |CLAS |    DAT_TYPE_STATUS |     SUBTYPE_STATUS |
 * +-------------------------------------------------------
---------------+
 */
/*
 * Class Bits
 */

#define DAT_CLASS_ERROR    0x80000000
#define DAT_CLASS_WARNING  0x40000000
#define DAT_CLASS_SUCCESS  0x00000000

/*
 * DAT Error bits
```

```
 * /
#define DAT_TYPE_MASK 0x3fff0000 /* mask for DAT_TYPE_STATUS bits */
#define DAT_SUBTYPE_MASK 0x0000FFFF/* mask for DAT_SUBTYPE_STATUS bits */

/*
 * Determining the success of an operation is best done with a macro;
 * each of these returns a boolean value.
 */

#define DAT_IS_WARNING(status)    ((DAT_UINT32)(status) & DAT_CLASS_WARNING)

#define DAT_GET_TYPE(status)    ((DAT_UINT32)(status) & DAT_TYPE_MASK)
#define DAT_GET_SUBTYPE(status)    ((DAT_UINT32)(status) & DAT_SUBTYPE_MASK)

/*
 * DAT return types. The ERROR bit is enabled for these definitions
 */
typedef enum dat_return_type
    {
    /* The operation was successful.                          */
    DAT_SUCCESS                    = 0x00000000,
   /* The operation was aborted because IA was closed or EVD was
     * destroyed.                                              */
    DAT_ABORT                      = 0x00010000,

  /* The specified Connection Qualifier was in use. */
    DAT_CONN_QUAL_IN_USE           = 0x00020000,

    /* The operation failed due to resource limitations.      */
    DAT_INSUFFICIENT_RESOURCES     = 0x00030000,
```

```
    /* Provider internal error. This error can be returned
by any operation
    * when the Provider has detected an internal error. This
error does not
     * mask any error caused by the Con-
sumer.                                                */
    DAT_INTERNAL_ERROR            = 0x00040000,

    /* One of the DAT handles was in-
valid.                                               */
    DAT_INVALID_HANDLE            = 0x00050000,

    /* One of the parameters was in-
valid.                                                */
    DAT_INVALID_PARAMETER         = 0x00060000,

   /* One of the parameters was invalid for this operation.
There are Event
     * Streams associated with the Event Dispatcher feeding
it.                  */
    DAT_INVALID_STATE             = 0x00070000,

    /* The size of the receiving buffer is too small for
sending buffer data.
    * The size of the local buffer is too small for the data
of the remote
    * buffer.
*/
    DAT_LENGTH_ERROR              = 0x00080000,

  /* The requested Model was not supported by the Provider.
*/
    DAT_MODEL_NOT_SUPPORTED       = 0x00090000,

    /* The specified IA name was not found in the list of
registered Providers. */
    DAT_PROVIDER_NOT_FOUND        = 0x000A0000,

    /* Protection violation for local or remote memory ac-
cess. Protection Zone
```

```
     * mismatch between an LMR of one of the local_iov seg-
ments and the local
     * End-
point.                                                        */
    DAT_PRIVILEGES_VIOLATION     = 0x000B0000,

   /* Privileges violation for local or remote memory access.
One of the LMRs
     * used in local_iov was either invalid or did not have
the local read
     * privi-
leges.                                                        */
    DAT_PROTECTION_VIOLATION     = 0x000C0000,

    /* The operation timed out without a notifica-
tion.                        */
    DAT_QUEUE_EMPTY              = 0x000D0000,

   /* The Event Dispatcher queue is full.
*/
    DAT_QUEUE_FULL               = 0x000E0000,

   /* The operation timed out. uDAPL ONLY
*/
    DAT_TIMEOUT_EXPIRED          = 0x000F0000,

   /* The provider name was already regis-
tered                                              */
   DAT_PROVIDER_ALREADY_REGISTERED      =0x00100000,

   /* The provider is "in-use" and cannot be closed at this
time             */
   DAT_PROVIDER_IN_USE                  =0x00110000,

   /* The requested remote address is not valid or not reach-
able */
   DAT_INVALID_ADDRESS                  =0x00120000,

   /* [Unix only] dat_evd_wait or dat_cno_wait has been in-
terrupted. */
   DAT_INTERRUPTED_CALL                 =0x00130000,
```

```
   /* No Connection Qualifiers are available*/
   DAT_CONN_QUAL_UNAVAILABLE            =0x00140000,

/* The specified IP Port was in use. */
    DAT_PORT_IN_USE               = 0x00160000,
/* The specified COMM not supported. */
    DAT_COMM_NOT_SUPPORTED        = 0x00170000,

#ifdef DAT_EXTENSIONS
/* The DAT extensions support. */
   DAT_EXTENSION_BASE                  = 0x10000000,
/* range 0x10000000 - 0x3FFF0000 is reserved for extensions
*/
#endif /* DAT_EXTENSIONS */

   /* Provider does not support the operation yet. */
   DAT_NOT_IMPLEMENTED                 = 0x3FFF0000

    } DAT_RETURN_TYPE;

typedef DAT_UINT32 DAT_RETURN;

/* Backward compatibility with DAT 1.0 */
#define DAT_NAME_NOT_FOUND DAT_PROVIDER_NOT_FOUND

/*
 * DAT_RETURN_SUBTYPE listing
 *
*/

typedef enum dat_return_subtype
    {
   /* First element is no subtype */
    DAT_NO_SUBTYPE,
    /* ABORT sub types */
    /* call was interrupted by a signal, or otherwise */
    DAT_SUB_INTERRUPTED,

    /* DAT_CONN_QUAL_IN_USE has no subtypes */
```

```
        /* INSUFFICIENT_RESOURCES subtypes */
        DAT_RESOURCE_MEMORY,
        DAT_RESOURCE_DEVICE,
        DAT_RESOURCE_TEP,/* transport endpoint, e.g. QP */
        DAT_RESOURCE_TEVD,/* transport EVD, e.g. CQ */
        DAT_RESOURCE_PROTECTION_DOMAIN,
       DAT_RESOURCE_MEMORY_REGION, /* HCA memory for LMR or RMR
*/
        DAT_RESOURCE_ERROR_HANDLER,
      DAT_RESOURCE_CREDITS, /* e.g outstanding RDMA Read credit
as target */
        DAT_RESOURCE_SRQ,


        /* DAT_INTERNAL_ERROR has no subtypes */

        /* INVALID_HANDLE subtypes */
        DAT_INVALID_HANDLE_IA,
        DAT_INVALID_HANDLE_EP,
        DAT_INVALID_HANDLE_LMR,
        DAT_INVALID_HANDLE_RMR,
        DAT_INVALID_HANDLE_PZ,
        DAT_INVALID_HANDLE_PSP,
        DAT_INVALID_HANDLE_RSP,
        DAT_INVALID_HANDLE_CR,
        DAT_INVALID_HANDLE_CNO,
        DAT_INVALID_HANDLE_EVD_CR,
        DAT_INVALID_HANDLE_EVD_REQUEST,
        DAT_INVALID_HANDLE_EVD_RECV,
        DAT_INVALID_HANDLE_EVD_CONN,
        DAT_INVALID_HANDLE_EVD_ASYNC,
        DAT_INVALID_HANDLE_SRQ,
        DAT_INVALID_HANDLE_CSP,

        DAT_INVALID_HANDLE1,
        DAT_INVALID_HANDLE2,
        DAT_INVALID_HANDLE3,
        DAT_INVALID_HANDLE4,
```

```
DAT_INVALID_HANDLE5,
DAT_INVALID_HANDLE6,
DAT_INVALID_HANDLE7,
DAT_INVALID_HANDLE8,
DAT_INVALID_HANDLE9,
DAT_INVALID_HANDLE10,


/* DAT_INVALID_PARAMETER subtypes */
DAT_INVALID_ARG1,
DAT_INVALID_ARG2,
DAT_INVALID_ARG3,
DAT_INVALID_ARG4,
DAT_INVALID_ARG5,
DAT_INVALID_ARG6,
DAT_INVALID_ARG7,
DAT_INVALID_ARG8,
DAT_INVALID_ARG9,
DAT_INVALID_ARG10,


/* DAT_INVALID_EP_STATE subtypes */
DAT_INVALID_STATE_EP_UNCONNECTED,
DAT_INVALID_STATE_EP_ACTCONNPENDING,
DAT_INVALID_STATE_EP_PASSCONNPENDING,
DAT_INVALID_STATE_EP_TENTCONNPENDING,
DAT_INVALID_STATE_EP_CONNECTED,
DAT_INVALID_STATE_EP_DISCONNECTED,
DAT_INVALID_STATE_EP_RESERVED,
DAT_INVALID_STATE_EP_COMPLPENDING,
DAT_INVALID_STATE_EP_DISCPENDING,
DAT_INVALID_STATE_EP_PROVIDERCONTROL,
DAT_INVALID_STATE_EP_NOTREADY,
DAT_INVALID_STATE_EP_RECV_WATERMARK,
DAT_INVALID_STATE_EP_PZ,
DAT_INVALID_STATE_EP_EVD_REQUEST,
DAT_INVALID_STATE_EP_EVD_RECV,
DAT_INVALID_STATE_EP_EVD_CONNECT,
DAT_INVALID_STATE_EP_UNCONFIGURED,
```

```
    DAT_INVALID_STATE_EP_UNCONFRESERVED,
    DAT_INVALID_STATE_EP_UNCONFPASSIVE,
    DAT_INVALID_STATE_EP_UNCONFTENTATIVE,

    DAT_INVALID_STATE_CNO_IN_USE,
    DAT_INVALID_STATE_CNO_DEAD,

   /* EVD states. Enabled/Disabled, Waitable/Unwaitable, and
         Notify/Solicited/Threshold are three orthogonal
bands of
         EVD state. The Threshold one is uDAPL specific.*/
    DAT_INVALID_STATE_EVD_OPEN,
   /* EVD can be either in enabled or disabled, but not both
or neither
         at the same time */
    DAT_INVALID_STATE_EVD_ENABLED,
    DAT_INVALID_STATE_EVD_DISABLED,
   /* EVD can be either in waitable or unwaitable, but not
both or neither
         at the same time */
    DAT_INVALID_STATE_EVD_WAITABLE,
    DAT_INVALID_STATE_EVD_UNWAITABLE,
   /* Do not release an EVD if it is in use                 */
    DAT_INVALID_STATE_EVD_IN_USE,

   /* EVD can be either in notify or solicited or threshold,
but not any pair,
   or all, or none at the same time. The threshold one is for
uDAPL only. */
    DAT_INVALID_STATE_EVD_CONFIG_NOTIFY,
    DAT_INVALID_STATE_EVD_CONFIG_SOLICITED,
    DAT_INVALID_STATE_EVD_CONFIG_THRESHOLD,
    DAT_INVALID_STATE_EVD_WAITER,
    DAT_INVALID_STATE_EVD_ASYNC,/* Async EVD required */

    DAT_INVALID_STATE_IA_IN_USE,

    DAT_INVALID_STATE_LMR_IN_USE,
    DAT_INVALID_STATE_LMR_FREE,
```

```
    DAT_INVALID_STATE_PZ_IN_USE,
    DAT_INVALID_STATE_PZ_FREE,

   /* DAT_INVALID_STATE_SRQ subtypes */
    DAT_INVALID_STATE_SRQ_OPERATIONAL,
    DAT_INVALID_STATE_SRQ_ERROR,
    DAT_INVALID_STATE_SRQ_IN_USE,

   /* DAT_LENGTH_ERROR has no subtypes
*/
   /* DAT_MODEL_NOT_SUPPORTED has no sub-
types                          */
   /* DAT_PRIVILEGES_VIOLATION subtypes
*/
   DAT_PRIVILEGES_READ,
    DAT_PRIVILEGES_WRITE,
    DAT_PRIVILEGES_RDMA_READ,
    DAT_PRIVILEGES_RDMA_WRITE,

    /* DAT_PROTECTION_VIOLATION subtypes */
    DAT_PROTECTION_READ,
    DAT_PROTECTION_WRITE,
    DAT_PROTECTION_RDMA_READ,
    DAT_PROTECTION_RDMA_WRITE,

    /* DAT_QUEUE_EMPTY has no subtypes */
    /* DAT_QUEUE_FULL has no subtypes */
    /* DAT_TIMEOUT_EXPIRED has no subtypes */
    /* DAT_PROVIDER_ALREADY_REGISTERED has no subtypes */
    /* DAT_PROVIDER_IN_USE has no subtypes */

   /* DAT_INVALID_ADDRESS subtypes */
   /* Unsupported addresses - those that are not Malformed,
but
   are incorrect for use in DAT (regardless of local routing
capabilities):
   IPv6 Multicast Addresses (ff/8)
   IPv4 Broadcast/Multicast Addresses */
   DAT_INVALID_ADDRESS_UNSUPPORTED,
```

```
   /* Unreachable addresses - A Provider might know that cer-
tain
   addresses are unreachable immediately. One example would
be
   an IPv6 addresses on an IPv4-only system.
   This can also be returned if it is known that there is no
route to the host.
   A Provider is not obligated to check for this condition.
*/
   DAT_INVALID_ADDRESS_UNREACHABLE,
   /* Malformed addresses - These cannot be valid in any con-
text.
    Those listed in RFC1884 section 2.3 as "Reserved" or "Un-
assigned". */
   DAT_INVALID_ADDRESS_MALFORMED,

   /* DAT_INTERRUPTED_CALL has no subtypes */
   /* DAT_CONN_QUAL_UNAVAILABLE has no subtypes */

   /* DAT_PROVIDER_NOT_FOUND subtypes. Erratta to the 1.1
spec    */
    DAT_NAME_NOT_REGISTERED,
    DAT_MAJOR_NOT_FOUND,
    DAT_MINOR_NOT_FOUND,
    DAT_THREAD_SAFETY_NOT_FOUND

} DAT_RETURN_SUBTYPE;

#endif /* _DAT_ERROR_H_ */
```

A.6 UDAT_VENDOR_SPECIFIC.H

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All
rights reserved.
 *
 * This Software is licensed under all of the following li-
censes:
 *
 * 1) under the terms of the "Common Public License 1.0". The
license is also
```

```
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 */
```

```
/******************************************************************
 *
 * HEADER: udat_vendor_specific.h
 *
 * PURPOSE: Vendor defined macros & support.
 *
 * Description: Header file for "uDAPL: User Direct Access
 *              ProgrammingLibrary, Version: 2.0"
 *
 * Mapping rules:
 *
 *
******************************************************************/

#ifndef _UDAT_VENDOR_SPECIFIC_H_
#define _UDAT_VENDOR_SPECIFIC_H_

#include <dat/dat_vendor_specific.h>

/* Vendor-specific extensions */

#if defined(_AMMASSO)

#elif defined(_BROADCOM)

#elif defined(_CISCO)

#elif defined(_IBM)

#elif defined(_INTEL)

#elif defined(_JNI)

#elif defined(_MELLANOX)

#elif defined(_MYRINET)
```

```
#elif defined(_NETEFFECT)

#elif defined(_QLOGIC)

#elif defined(_SILVERSTORM)

#elif defined(_VOLTAIRE)

#endif

#endif /* _UDAT_VENDOR_SPECIFIC_H_ */
```

A.7 DAT_VENDOR_SPECIFIC.H

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
 * This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * OR
 *
 * 2) under the terms of the "The BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
```

```
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
 */
/**********************************************************
 *
 * HEADER: dat_vendor_specific.h
 *
 * PURPOSE:
 *
 * Description: Header file for "DAPL: Direct Access Programming
 *       Library, Version: 2.0"
 *
 * Mapping rules:
 *
 **********************************************************/
```

```
#ifndef _DAT_VENDOR_SPECIFIC_H_
#define _DAT_VENDOR_SPECIFIC_H_

/* Vendor-specific extensions */

#if defined(_AMMASSO)

#elif defined(_BROADCOM)

#elif defined(_CISCO)

#elif defined(_IBM)

#elif defined(_INTEL)

#elif defined(_JNI)

#elif defined(_MELLANOX)

#elif defined(_MYRINET)

#elif defined(_NETEFFECT)

#elif defined(_QLOGIC)

#elif defined(_SILVERSTORM)

#elif defined(_VOLTAIRE)

#endif

#endif /* _DAT_VENDOR_SPECIFIC_H_ */
```

APPENDIX B: uDAPL-2.0 REGISTRATION HEADERS

B.1 DAT REGISTRY

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
 * This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "The BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the “GNU General Public License (GPL) Version 2”.
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
```

```
 *
 * Redistributions in binary form must reproduce both the
above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names
of other DAT
 * Collaborative contributors may be used to endorse or pro-
mote
 * products derived from this software without specific prior
written
 * permission.
 *
 */

/*************************************************************
******
 *
 * HEADER: dat_registry.h
 *
 * PURPOSE: DAT registration API signatures
 *
 * Description: Header file for "DAPL: Direct Access Program-
ming
 *    Library, Version: 2.0"
 *
 * Contains registration external reference signatures
 * for dat registry functions. This file is *only*
 * included by providers, not consumers.
 *
 * Mapping rules:
 * All global symbols are prepended with DAT_ or dat_
 * All DAT objects have an 'api' tag which, such as 'ep' or
'lmr' .
 * The method table is in the provider definition structure.
 *
 *

*************************************************************/
```

```
#ifndef _DAT_REGISTRY_H_
#define _DAT_REGISTRY_H_
#if defined(_UDAT_H_)
#include <dat/udat_redirection.h>
#elif defined(_KDAT_H_)
#include <dat/kdat_redirection.h>
#else
#error Must include udat.h or kdat.h
#endif

/*
 * dat registration API.
 *
 * Technically the dat_ia_open is part of the registration API. This
 * is so the registration module can map the device name to a provider
 * structure and then call the provider dat_ia_open function.
 * dat_is_close is also part of the registration API so that the
 * registration code can be aware when an ia is no longer in use.
 *
 */

extern DAT_RETURN dat_registry_add_provider(
   IN const    DAT_PROVIDER*,          /* provider */
   IN const    DAT_PROVIDER_INFO* );/* provider info */

extern DAT_RETURN dat_registry_remove_provider(
   IN const DAT_PROVIDER*,       /* provider */
   IN  const DAT_PROVIDER_INFO* );/* provider info */

/*
* Provider initialization APIs.
 *
 * Providers that support being automatically loaded by the Registry must
* implement these APIs and export them as public symbols.
*/
```

```
#define DAT_PROVIDER_INIT_FUNC_NAME dat_provider_init
#define DAT_PROVIDER_FINI_FUNC_NAME dat_provider_fini

#define DAT_PROVIDER_INIT_FUNC_STR   "dat_provider_init"
#define DAT_PROVIDER_FINI_FUNC_STR   "dat_provider_fini"

typedef void ( *DAT_PROVIDER_INIT_FUNC)(
   IN    const DAT_PROVIDER_INFO *,/* provider info */
   IN    const char *);          /* instance data */

typedef void ( *DAT_PROVIDER_FINI_FUNC)  (
   IN    const DAT_PROVIDER_INFO *);/* provider info */

typedef enum dat_ha_relationship
   {
   DAT_HA_FALSE,      /* two IAs are not related    */
   DAT_HA_TRUE,       /* two IAs are related        */
   DAT_HA_UNKNOWN,    /* relatioship is not known   */
   DAT_HA_CONFLICTING,     /* 2 IAs do not agree on the re-
lationship */
   DAT_HA_EXTENSION_BASE
   } DAT_HA_RELATIONSHIP;

extern DAT_RETURN dat_registry_providers_related (
         IN    const DAT_NAME_PTR,
         IN    const DAT_NAME_PTR,
         OUT   DAT_HA_RELATIONSHIP*);

#endif /* _DAT_REGISTRY_H_ */
```

B.2 UDAT_REDIRECTION.H

```
/*
 * Copyright (c) 2002-2006, Network Appliance, Inc. All
rights reserved.
 *
 * This Software is licensed under all of the following li-
censes:
 *
```

```
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "The BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain both the above copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names of other DAT
 * Collaborative contributors may be used to endorse or promote
 * products derived from this software without specific prior written
 * permission.
 *
```

```
 */

/************************************************************
******
 *
 * HEADER: udat_redirection.h
 *
 * PURPOSE: User DAT macro definitions
 *
 * Description: Macros to invoke DAPL functions from the dat_
registry
 *
 * Mapping rules:
 *      All global symbols are prepended with DAT_ or dat_
 *     All DAT objects have an 'api' tag which, such as 'ep'
or 'lmr'
 *      The method table is in the provider definition struc-
ture.
 *
 *
 *

************************************************************/

#ifndef _UDAT_REDIRECTION_H_
#define _UDAT_REDIRECTION_H_


#define DAT_LMR_CREATE(ia,mem_type,reg_desc,len,pz,priv,\
                    va_type,lmr,lmr_context,rmr_context,reg_
len,reg_addr) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->lmr_create_func)(\
                (ia),\
                (mem_type),\
                (reg_desc),\
                (len),\
                (pz),\
                (priv),\
                (va_type),\
                (lmr),\
```

```
                (lmr_context),\
        (rmr_context),\
        (reg_len),\
                (reg_addr))

#define DAT_EVD_CREATE(ia,qlen,cno,flags,handle) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->evd_create_func)(\
                (ia),\
                (qlen),\
                (cno),\
                (flags),\
                (handle))

#define DAT_EVD_ENABLE(evd) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_enable_func)(\
                (evd))

#define DAT_EVD_WAIT(evd,timeout,threshold,event,nmore) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_wait_func)(\
                (evd),\
                (timeout),\
                (threshold),\
                (event),\
                (nmore))

#define DAT_EVD_DISABLE(evd) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_disable_func)(\
                (evd))

#define DAT_EVD_SET_UNWAITABLE(evd) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_set_unwaitable_
func)(\
                (evd))

#define DAT_EVD_CLEAR_UNWAITABLE(evd) \
       (*DAT_HANDLE_TO_PROVIDER(evd)->evd_clear_unwaitable_
func)(\
                (evd))
```

```
#define DAT_EVD_MODIFY_CNO(evd,cno) \
       (*DAT_HANDLE_TO_PROVIDER(evd)->evd_modify_cno_func)(\
                 (evd),\
                 (cno))

#define DAT_CNO_CREATE(ia,proxy,cno) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->cno_create_func)(\
                 (ia),\
                 (proxy),\
                 (cno))

#define DAT_CNO_FD_CREATE(ia,fd,cno) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->cno_fd_create_func)(\
                 (ia),\
                 (fd),\
                 (cno))

#define DAT_CNO_TRIGGER(cno, evd) \
         (*DAT_HANDLE_TO_PROVIDER(cno)->cno_trigger_func)(\
                 (cno),\
                 (evd))

#define DAT_CNO_MODIFY_AGENT(cno,proxy) \
         (*DAT_HANDLE_TO_PROVIDER(cno)->cno_modify_agent_
func)(\
                 (cno),\
                 (proxy))

#define DAT_CNO_QUERY(cno,mask,param) \
         (*DAT_HANDLE_TO_PROVIDER(cno)->cno_query_func)(\
                 (cno),\
                 (mask),\
                 (param))

#define DAT_CNO_FREE(cno) \
         (*DAT_HANDLE_TO_PROVIDER(cno)->cno_free_func)(\
                 (cno))

#define DAT_CNO_WAIT(cno,timeout,evd) \
```

```
        (*DAT_HANDLE_TO_PROVIDER(cno)->cno_wait_func)(\
                (cno),\
                (timeout),\
                (evd))



/*******************************************************************
 * FUNCTION PROTOTYPES
 *
 * User DAT function call definitions,
 *
 *******************************************************************/

typedef DAT_RETURN (*DAT_LMR_CREATE_FUNC) (
   IN      DAT_IA_HANDLE,          /* ia_handle             */
   IN      DAT_MEM_TYPE,           /* mem_type              */
   IN      DAT_REGION_DESCRIPTION,/* region_description
*/
   IN      DAT_VLEN,               /* length                */
   IN      DAT_PZ_HANDLE,          /* pz_handle             */
   IN      DAT_MEM_PRIV_FLAGS,     /* privileges            */
   IN      DAT_VA_TYPE,            /* va_type */
   OUT     DAT_LMR_HANDLE *,       /* lmr_handle            */
   OUT     DAT_LMR_CONTEXT *,      /* lmr_context           */
   OUT     DAT_RMR_CONTEXT *,      /* rmr_context           */
   OUT     DAT_VLEN *,             /* registered_length     */
   OUT     DAT_VADDR * );          /* registered_address    */

typedef DAT_RETURN (*DAT_LMR_QUERY_FUNC) (
   IN      DAT_LMR_HANDLE,/* lmr_handle            */
   IN      DAT_LMR_PARAM_MASK,/* lmr_param_mask        */
   OUT     DAT_LMR_PARAM *);/* lmr_param             */

/* Event functions */

typedef DAT_RETURN (*DAT_EVD_CREATE_FUNC) (
```

```
   IN      DAT_IA_HANDLE,        /* ia_handle           */
   IN      DAT_COUNT,            /* evd_min_qlen        */
   IN      DAT_CNO_HANDLE,       /* cno_handle          */
   IN      DAT_EVD_FLAGS,        /* evd_flags           */
   OUT     DAT_EVD_HANDLE * );   /* evd_handle          */

typedef DAT_RETURN (*DAT_EVD_MODIFY_CNO_FUNC) (
   IN      DAT_EVD_HANDLE,       /* evd_handle          */
   IN      DAT_CNO_HANDLE);      /* cno_handle          */

typedef DAT_RETURN (*DAT_CNO_CREATE_FUNC) (
   IN    DAT_IA_HANDLE,                /* ia_handle
*/
   IN    DAT_OS_WAIT_PROXY_AGENT,      /* agent
*/
   OUT   DAT_CNO_HANDLE *);            /* cno_handle
*/

typedef DAT_RETURN (*DAT_CNO_FD_CREATE_FUNC) (
   IN    DAT_IA_HANDLE,          /* ia_handle */
   OUT   DAT_FD *,               /* file descriptor */
   OUT   DAT_CNO_HANDLE *);      /* cno_handle          */

typedef DAT_RETURN (*DAT_CNO_TRIGGER_FUNC) (
   IN    DAT_CNO_HANDLE,         /* cno_handle */
   OUT   DAT_EVD_HANDLE *);      /* evd_handle          */

typedef DAT_RETURN (*DAT_CNO_MODIFY_AGENT_FUNC) (
   IN    DAT_CNO_HANDLE,               /* cno_handle
*/
   IN    DAT_OS_WAIT_PROXY_AGENT);     /* agent
*/

typedef DAT_RETURN (*DAT_CNO_QUERY_FUNC) (
   IN      DAT_CNO_HANDLE,             /* cno_handle
*/
   IN      DAT_CNO_PARAM_MASK,         /* cno_param_mask
*/
   OUT     DAT_CNO_PARAM * );          /* cno_param
*/
```

```
typedef DAT_RETURN (*DAT_CNO_FREE_FUNC) (
   IN DAT_CNO_HANDLE);/* cno_handle          */

typedef DAT_RETURN (*DAT_CNO_WAIT_FUNC) (
   IN     DAT_CNO_HANDLE,              /* cno_handle
*/
   IN     DAT_TIMEOUT,                 /* tim-
eout              */
   OUT    DAT_EVD_HANDLE *);           /* evd_handle
*/

typedef DAT_RETURN (*DAT_EVD_ENABLE_FUNC) (
   IN      DAT_EVD_HANDLE);/* evd_handle          */

typedef DAT_RETURN (*DAT_EVD_WAIT_FUNC) (
   IN     DAT_EVD_HANDLE,         /* evd_handle          */
   IN     DAT_TIMEOUT,            /* timeout             */
   IN     DAT_COUNT,              /* threshold           */
   OUT    DAT_EVENT *,            /* event               */
   OUT    DAT_COUNT * );          /* N more events       */

typedef DAT_RETURN (*DAT_EVD_DISABLE_FUNC) (
   IN      DAT_EVD_HANDLE);/* evd_handle          */

typedef DAT_RETURN (*DAT_EVD_SET_UNWAITABLE_FUNC) (
   IN DAT_EVD_HANDLE); /* evd_handle */

typedef DAT_RETURN (*DAT_EVD_CLEAR_UNWAITABLE_FUNC) (
   IN DAT_EVD_HANDLE); /* evd_handle */



#include <dat/dat_redirection.h>
struct dat_provider
    {
    const char *                        device_name;
    DAT_PVOID                           extension;

    DAT_IA_OPEN_FUNC                    ia_open_func;
    DAT_IA_QUERY_FUNC                   ia_query_func;
```

```
    DAT_IA_CLOSE_FUNC                    ia_close_func;

    DAT_SET_CONSUMER_CONTEXT_FUNC     set_consumer_context_
func;
    DAT_GET_CONSUMER_CONTEXT_FUNC     get_consumer_context_
func;
    DAT_GET_HANDLE_TYPE_FUNC          get_handle_type_func;

    DAT_CNO_CREATE_FUNC               cno_create_func;
/* udat only */
    DAT_CNO_MODIFY_AGENT_FUNC            cno_modify_agent_
func;          /* udat only */
    DAT_CNO_QUERY_FUNC                cno_query_func;
/* udat only */
    DAT_CNO_FREE_FUNC                 cno_free_func;
/* udat only */
    DAT_CNO_WAIT_FUNC                 cno_wait_func;
/* udat only */

    DAT_CR_QUERY_FUNC                    cr_query_func;
    DAT_CR_ACCEPT_FUNC                   cr_accept_func;
    DAT_CR_REJECT_FUNC                   cr_reject_func;
    DAT_CR_HANDOFF_FUNC              cr_handoff_func;


    DAT_EVD_CREATE_FUNC        evd_create_func;
    DAT_EVD_QUERY_FUNC         evd_query_func;

    DAT_EVD_MODIFY_CNO_FUNC           evd_modify_cno_func;
/* udat only */
    DAT_EVD_ENABLE_FUNC               evd_enable_func;
/* udat only */
    DAT_EVD_DISABLE_FUNC              evd_disable_func;
/* udat only */
    DAT_EVD_WAIT_FUNC                 evd_wait_func;
/* udat only */

    DAT_EVD_RESIZE_FUNC                  evd_resize_func;
    DAT_EVD_POST_SE_FUNC                   evd_post_se_func;
    DAT_EVD_DEQUEUE_FUNC                   evd_dequeue_func;
    DAT_EVD_FREE_FUNC                      evd_free_func;
```

```
    DAT_EP_CREATE_FUNC                  ep_create_func;
    DAT_EP_QUERY_FUNC                   ep_query_func;
    DAT_EP_MODIFY_FUNC                  ep_modify_func;
    DAT_EP_CONNECT_FUNC                 ep_connect_func;
    DAT_EP_DUP_CONNECT_FUNC       ep_dup_connect_func;
    DAT_EP_DISCONNECT_FUNC             ep_disconnect_func;
    DAT_EP_POST_SEND_FUNC               ep_post_send_func;
    DAT_EP_POST_RECV_FUNC               ep_post_recv_func;
    DAT_EP_POST_RDMA_READ_FUNC    ep_post_rdma_read_func;
    DAT_EP_POST_RDMA_WRITE_FUNC ep_post_rdma_write_func;
    DAT_EP_GET_STATUS_FUNC              ep_get_status_
func;
    DAT_EP_FREE_FUNC                    ep_free_func;


    DAT_LMR_CREATE_FUNC                 lmr_create_func;
    DAT_LMR_QUERY_FUNC          lmr_query_func;
    DAT_LMR_FREE_FUNC               lmr_free_func;

    DAT_RMR_CREATE_FUNC                 rmr_create_func;
    DAT_RMR_QUERY_FUNC                  rmr_query_func;
    DAT_RMR_BIND_FUNC                   rmr_bind_func;
    DAT_RMR_FREE_FUNC                   rmr_free_func;

    DAT_PSP_CREATE_FUNC                 psp_create_func;
    DAT_PSP_QUERY_FUNC                  psp_query_func;
    DAT_PSP_FREE_FUNC                   psp_free_func;

    DAT_RSP_CREATE_FUNC                 rsp_create_func;
    DAT_RSP_QUERY_FUNC                    rsp_query_func;
    DAT_RSP_FREE_FUNC                     rsp_free_func;

    DAT_PZ_CREATE_FUNC                  pz_create_func;
    DAT_PZ_QUERY_FUNC                   pz_query_func;
    DAT_PZ_FREE_FUNC                    pz_free_func;

/* dat-1.1 */
   DAT_PSP_CREATE_ANY_FUNC psp_create_any_func;/* dat-1.1
*/
```

```
   DAT_EP_RESET_FUNC                     ep_reset_func;/*
dat-1.1 */

/* udat-1.1 */
   DAT_EVD_SET_UNWAITABLE_FUNC evd_set_unwaitable_func;
/* udat-1.1 */
   DAT_EVD_CLEAR_UNWAITABLE_FUNC evd_clear_unwaitable_func;
/* udat-1.1 */

/* dat-1.2 */
    DAT_LMR_SYNC_RDMA_READ_FUNC  lmr_sync_rdma_read_func;
    DAT_LMR_SYNC_RDMA_WRITE_FUNClmr_sync_rdma_write_func;

    DAT_EP_CREATE_WITH_SRQ_FUNC ep_create_with_srq_func;
    DAT_EP_RECV_QUERY_FUNC       ep_recv_query_func;
    DAT_EP_SET_WATERMARK_FUNC    ep_set_watermark_func;
    DAT_SRQ_CREATE_FUNC          srq_create_func;
    DAT_SRQ_FREE_FUNC            srq_free_func;
    DAT_SRQ_POST_RECV_FUNC       srq_post_recv_func;
    DAT_SRQ_QUERY_FUNC           srq_query_func;
    DAT_SRQ_RESIZE_FUNC          srq_resize_func;
    DAT_SRQ_SET_LW_FUNC          srq_set_lw_func;

/* DAT 2.0 functions */
    DAT_CSP_CREATE_FUNC                 csp_create_func;
    DAT_CSP_QUERY_FUNC                  csp_query_func;
    DAT_CSP_FREE_FUNC                   csp_free_func;

    DAT_EP_COMMON_CONNECT_FUNC   ep_common_connect_func;
    DAT_RMR_CREATE_FOR_EP_FUNC rmr_create_for_ep_func;

   DAT_EP_POST_SEND_WITH_INVALIDATE_FUNC     ep_post_send_
with_invalidate_func;
    DAT_EP_POST_RDMA_READ_TO_RMR_FUNC
ep_post_rdma_read_to_rmr_func;

   DAT_CNO_FD_CREATE_FUNC       cno_fd_create_func;
   DAT_CNO_TRIGGER_FUNC         cno_trigger_func;

   DAT_IA_HA_RELATED_FUNC       ia_ha_related_func;
```

```
#ifdef DAT_EXTENSIONS
   DAT_HANDLE_EXTENDEDOP_FUNC    handle_extendedop_func;
#endif
};

#endif /* _UDAT_REDIRECTION_H_ */
```

B.3 DAT_REDIRECTION.H

```
/*
 *
 * Copyright (c) 2002-2006, Network Appliance, Inc. All rights reserved.
 *
 * This Software is licensed under all of the following licenses:
 *
 * 1) under the terms of the "Common Public License 1.0". The license is also
 *    available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/cpl.php.
 *
 * 2) under the terms of the "The BSD License". The license is also available
 *    from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/bsd-license.php.
 *
 * 3) under the terms of the "GNU General Public License (GPL) Version 2".
 *  The license is also available from the Open Source Initiative, see
 *    http://www.opensource.org/licenses/gpl-license.php.
 *
 * Licensee has the right to choose one of the above licenses.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
```

```
 *
 * Redistributions of source code must retain both the above
copyright
 * notice and one of the license notices.
 *
 * Redistributions in binary form must reproduce both the
above copyright
 * notice, one of the license notices in the documentation
 * and/or other materials provided with the distribution.
 *
 * Neither the name of Network Appliance, Inc. nor the names
of other DAT
 * Collaborative contributors may be used to endorse or pro-
mote
 * products derived from this software without specific prior
written
 * permission.
 *
 */

/************************************************************
******
 *
 * HEADER: dat_redirection.h
 *
 * PURPOSE: Defines the common redirection macros
 *
 * Description: Macros to invoke DAPL functions from the dat_
registry
 *
 * Mapping rules:
 * All global symbols are prepended with DAT_ or dat_
 * All DAT objects have an 'api' tag which, such as 'ep' or
'lmr'
 * The method table is in the provider definition structure.
 *
************************************************************/

#ifndef _DAT_REDIRECTION_H_
#define _DAT_REDIRECTION_H_
```

```
typedef struct dat_provider DAT_PROVIDER;

#ifndef DAT_HANDLE_TO_PROVIDER

/* A utility macro to fetch the Provider Library for any ob-
ject
 *
 * An alternate version could be defined for single library
systems.
 * it would look something like:
 *     extern const struct dat_ia my_single_ia_provider;
 *     #define DAT_HANDLE_TO_PROVIDER(ignore)
 *     &my_single_ia_provider
 *
 * This would allow a good compiler to avoid indirection
 * overhead when making function calls.
 */

#define DAT_HANDLE_TO_PROVIDER(handle) (*(DAT_PROVIDER
**)(handle))
#endif

#define DAT_IA_QUERY (ia,evd,ia_msk,ia_ptr,p_msk,p_ptr) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->ia_query_func)(\
                (ia),\
                (evd),\
                (ia_msk),\
                (ia_ptr),\
                (p_msk),\
                (p_ptr))

#define DAT_SET_CONSUMER_CONTEXT (handle,context) \
        (*DAT_HANDLE_TO_PROVIDER(handle)->set_consumer_
context_func)(\
                (handle),\
                (context))

#define DAT_GET_CONSUMER_CONTEXT (handle,context) \
        (*DAT_HANDLE_TO_PROVIDER(handle)->get_consumer_
context_func)(\
```

```
                (handle),\
                (context))

#define DAT_GET_HANDLE_TYPE (handle,handle_type) \
        (*DAT_HANDLE_TO_PROVIDER(handle)->get_handle_type_
func)(\
                (handle),\
                (handle_type))

#define DAT_CR_QUERY (cr,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(cr)->cr_query_func)(\
                (cr),\
                (mask),\
                (param))

#define DAT_CR_ACCEPT (cr,ep,size,pdata) \
        (*DAT_HANDLE_TO_PROVIDER(cr)->cr_accept_func)(\
                (cr),\
                (ep),\
                (size),\
                (pdata))

#define DAT_CR_REJECT (cr,size,pdata) \
        (*DAT_HANDLE_TO_PROVIDER(cr)->cr_reject_func)(\
                (cr),\
                (size),\
                (pdata))

#define DAT_CR_HANDOFF(cr,qual) \
   (*DAT_HANDLE_TO_PROVIDER(cr)->cr_handoff_func)(\
         (cr),                   \
         (qual))

#define DAT_EVD_QUERY (evd,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_query_func)(\
                (evd),\
                (mask),\
                (param))
```

```
#define DAT_EVD_RESIZE (evd,qsize) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_resize_func)(\
                (evd),\
                (qsize))

#define DAT_EVD_POST_SE (evd,event) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_post_se_func)(\
                (evd),\
                (event))

#define DAT_EVD_DEQUEUE (evd,event) \
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_dequeue_func)(\
                (evd),\
                (event))

#define DAT_EVD_FREE (evd)\
        (*DAT_HANDLE_TO_PROVIDER(evd)->evd_free_func)(\
              (evd))

#define DAT_EP_CREATE (ia,pz,in_evd,out_evd,connect_
evd,attr,ep) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->ep_create_func)(\
                (ia),\
                (pz),\
                (in_evd),\
                (out_evd),\
                (connect_evd),\
                (attr),\
                (ep))

#define DAT_EP_CREATE_WITH_SRQ (ia,pz,in_evd,out_evd, \
   connect_evd,srq,attr,ep) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->ep_create_with_srq_
func)(\
                (ia),\
                (pz),\
                (in_evd),\
                (out_evd),\
                (connect_evd),\
```

```
                (srq),\
                (attr),\
                (ep))

#define DAT_EP_QUERY (ep,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_query_func)(\
                (ep),\
                (mask),\
                (param))

#define DAT_EP_MODIFY (ep,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_modify_func)(\
                (ep),\
                (mask),\
                (param))

#define DAT_EP_CONNECT (ep,ia_addr,conn_qual,\
   timeout,psize,pdata,qos,flags) \
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_connect_func)(\
                (ep),\
                (ia_addr),\
                (conn_qual),\
                (timeout),\
                (psize),\
                (pdata),\
                (qos),\
                (flags))

#define DAT_EP_COMMON_CONNECT (ep,addr,\
   timeout,psize,pdata) \
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_common_connect_
func)(\
                (ep),\
                (addr),\
                (timeout),\
                (psize),\
                (pdata))
```

```
#define DAT_EP_DUP_CONNECT (ep,dup,timeout,psize,pdata,qos)
\
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_dup_connect_func)(\
                  (ep),\
                  (dup),\
                  (timeout),\
                  (psize),\
                  (pdata),\
                  (qos))

#define DAT_EP_DISCONNECT (ep,flags) \
         (*DAT_HANDLE_TO_PROVIDER(ep)->ep_disconnect_func)(\
                   (ep),\
                   (flags))

#define DAT_EP_POST_SEND (ep,size,lbuf,cookie,flags) \
         (*DAT_HANDLE_TO_PROVIDER(ep)->ep_post_send_func)(\
                  (ep),\
                  (size),\
                  (lbuf),\
                  (cookie),\
                  (flags))

#define DAT_EP_POST_SEND_WITH_INVALIDATE( \
      ep,size,lbuf,cookie,flags,inv_flag,rmr_context) \
         (*DAT_HANDLE_TO_PROVIDER(ep)-> \
                 ep_post_send_with_invalidate_func)(\
                  (ep),\
                  (size),\
                  (lbuf),\
                  (cookie),\
                  (flags), \
                  (inv_flag), \
                  (rmr_context))

#define DAT_EP_POST_RECV (ep,size,lbuf,cookie,flags) \
         (*DAT_HANDLE_TO_PROVIDER(ep)->ep_post_recv_func)(\
                  (ep),\
                  (size),\
```

```
                    (lbuf),\
                    (cookie),\
                    (flags))

#define DAT_EP_POST_RDMA_READ
(ep,size,lbuf,cookie,rbuf,flags) \
          (*DAT_HANDLE_TO_PROVIDER(ep)->ep_post_rdma_read_
func)(\
                    (ep),\
                    (size),\
                    (lbuf),\
                    (cookie),\
                    (rbuf),\
                    (flags))

#define DAT_EP_POST_RDMA_READ_TO_RMR (ep, lbuf, cookie,
rbuf, flags) \
          (*DAT_HANDLE_TO_PROVIDER(ep)->ep_post_rdma_read_to_
rmr_func)(\
                    (ep),\
                    (lbuf),\
                    (cookie),\
                    (rbuf),\
                    (flags))

#define DAT_EP_POST_RDMA_WRITE
(ep,size,lbuf,cookie,rbuf,flags) \
          (*DAT_HANDLE_TO_PROVIDER(ep)->ep_post_rdma_write_
func)(\
                    (ep),\
                    (size),\
                    (lbuf),\
                    (cookie),\
                    (rbuf),\
                    (flags))

#define DAT_EP_GET_STATUS (ep,ep_state,recv_idle,request_
idle) \
          (*DAT_HANDLE_TO_PROVIDER(ep)->ep_get_status_func)(\
                    (ep),\
                    (ep_state),\
```

```
                (recv_idle),\
                (request_idle))

#define DAT_EP_FREE (ep)\
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_free_func)(\
                (ep))

#define DAT_EP_RESET (ep)\
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_reset_func)(\
                (ep))

#define DAT_EP_RECV_QUERY (ep,nbuf_alloc,buf_span)\
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_recv_query_func)(\
                (ep),\
                (nbuf_alloc),\
                (buf_span))

#define DAT_EP_SET_WATERMARK (ep,soft_wm,hard_wm)\
        (*DAT_HANDLE_TO_PROVIDER(ep)->ep_set_watermark_
func)(\
                (ep),\
                (soft_wm),\
                (hard_wm))

#define DAT_LMR_QUERY (lmr,mask,param)\
        (*DAT_HANDLE_TO_PROVIDER(lmr)->lmr_query_func)(\
                (lmr),\
                (mask),\
                (param))

#define DAT_LMR_FREE (lmr)\
        (*DAT_HANDLE_TO_PROVIDER(lmr)->lmr_free_func)(\
                (lmr))

#define DAT_LMR_SYNC_RDMA_READ (ia,lbuf,size)\
        (*DAT_HANDLE_TO_PROVIDER(ia)->lmr_sync_rdma_read_
func)(\
                (ia),\
                (lbuf),\
```

```
                (size))

#define DAT_LMR_SYNC_RDMA_WRITE (ia,lbuf,size)\
        (*DAT_HANDLE_TO_PROVIDER(ia)->lmr_sync_rdma_write_
func)(\
                (ia),\
                (lbuf),\
                (size))

#define DAT_RMR_CREATE (pz,rmr) \
         (*DAT_HANDLE_TO_PROVIDER(pz)->rmr_create_func)(\
                (pz),\
                (rmr))

#define DAT_RMR_CREATE_FOR_EP (pz,rmr) \
         (*DAT_HANDLE_TO_PROVIDER(pz)->rmr_create_for_ep_
func)(\
                (pz),\
                (rmr))

#define DAT_RMR_QUERY (rmr,mask,param) \
         (*DAT_HANDLE_TO_PROVIDER(rmr)->rmr_query_func)(\
                (rmr),\
                (mask),\
                (param))

#define DAT_RMR_BIND (rmr,lmr,lmr_triplet,mem_priv,va_
type,ep,cookie,flags,context) \
        (*DAT_HANDLE_TO_PROVIDER(rmr)->rmr_bind_func)(\
                (rmr),\
                (lmr),\
                (lmr_triplet),\
                (mem_priv),\
                (va_type),\
                (ep),\
                (cookie),\
                (flags),\
                (context))

#define DAT_RMR_FREE (rmr)\
```

```
        (*DAT_HANDLE_TO_PROVIDER(rmr)->rmr_free_func)(\
                (rmr))

#define DAT_PSP_CREATE (ia,conn_qual,evd,flags,handle) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->psp_create_func)(\
                (ia),\
                (conn_qual),\
                (evd),\
                (flags),\
                (handle))

#define DAT_PSP_CREATE_ANY (ia,conn_qual,evd,flags,handle)
\
       (*DAT_HANDLE_TO_PROVIDER(ia)->psp_create_any_func)(\
                (ia),\
                (conn_qual),\
                (evd),\
                (flags),\
                (handle))

#define DAT_PSP_QUERY (psp,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(psp)->psp_query_func)(\
                (psp),\
                (mask),\
                (param))

#define DAT_PSP_FREE (psp)\
        (*DAT_HANDLE_TO_PROVIDER(psp)->psp_free_func)(\
                (psp))

#define DAT_RSP_CREATE (ia,conn_qual,ep,evd,handle) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->rsp_create_func)(\
                (ia),\
                (conn_qual),\
                (ep),\
                (evd),\
                (handle))

#define DAT_RSP_QUERY (rsp,mask,param) \
```

```
        (*DAT_HANDLE_TO_PROVIDER(rsp)->rsp_query_func)(\
                (rsp),\
                (mask),\
                (param))

#define DAT_RSP_FREE (rsp)\
        (*DAT_HANDLE_TO_PROVIDER(rsp)->rsp_free_func)(\
                (rsp))

#define DAT_CSP_CREATE(ia, comm, addr, evd, handle) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->csp_create_func)(\
                (ia),\
                (comm),\
                (addr),\
                (evd),\
                (handle))

#define DAT_CSP_QUERY(csp, mask, param) \
        (*DAT_HANDLE_TO_PROVIDER(csp)->csp_query_func)(\
                (csp),\
                (mask),\
                (param))

#define DAT_CSP_FREE(csp)\
        (*DAT_HANDLE_TO_PROVIDER(csp)->csp_free_func)(\
                (csp))

#define DAT_PZ_CREATE (ia,pz) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->pz_create_func)(\
                (ia),\
                (pz))

#define DAT_PZ_QUERY (pz,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(pz)->pz_query_func)(\
                (pz),\
                (mask),\
                (param))

#define DAT_PZ_FREE (pz) \
```

```
        (*DAT_HANDLE_TO_PROVIDER(pz)->pz_free_func)(\
                (pz))

#define DAT_SRQ_CREATE (ia,pz,attr,srq) \
        (*DAT_HANDLE_TO_PROVIDER(ia)->srq_create_func)(\
                (ia),\
                (pz),\
                (attr),\
                (srq))

#define DAT_SRQ_SET_LW (srq,lw) \
        (*DAT_HANDLE_TO_PROVIDER(srq)->srq_set_lw_func)(\
                (srq),\
                (lw))

#define DAT_SRQ_FREE (srq) \
        (*DAT_HANDLE_TO_PROVIDER(srq)->srq_free_func)(\
                (srq))

#define DAT_SRQ_QUERY (srq,mask,param) \
        (*DAT_HANDLE_TO_PROVIDER(srq)->srq_query_func)(\
                (srq),\
                (mask),\
                (param))

#define DAT_SRQ_RESIZE (srq,qsize) \
        (*DAT_HANDLE_TO_PROVIDER(srq)->srq_resize_func)(\
                (srq),\
                (qsize))

#define DAT_SRQ_POST_RECV (srq,size,lbuf,cookie) \
       (*DAT_HANDLE_TO_PROVIDER(srq)->srq_post_recv_func)(\
                (srq),\
                (size),\
                (lbuf),\
                (cookie))

#define DAT_IA_HA_RELATED (ia, name, answer) \
         (*DAT_HANDLE_TO_PROVIDER(ia)->ia_ha_related) (\
```

```
         (ia), \
         (name), \
         (answer))


#ifdef DAT_EXTENSIONS
   /* generic extended op */
#define  DAT_HANDLE_EXTENDEDOP (handle,op,args) \
         (*DAT_HANDLE_TO_PROVIDER(handle)->extendedop_func)
(\
               (handle), \
               (op),       \
               (args))
#endif

/***********************************************************
******
 * FUNCTION PROTOTYPES
***********************************************************
*****/

typedef DAT_RETURN (*DAT_IA_OPEN_FUNC) (
   IN      const DAT_NAME_PTR,/* provider             */
   IN      DAT_COUNT,/* asynch_evd_min_qlen  */
   INOUT   DAT_EVD_HANDLE *,/* asynch_evd_handle    */
   OUT     DAT_IA_HANDLE *);/* ia_handle            */

typedef DAT_RETURN (*DAT_IA_OPENV_FUNC) (
   IN      const DAT_NAME_PTR,/* provider             */
   IN      DAT_COUNT,/* asynch_evd_min_qlen  */
   INOUT   DAT_EVD_HANDLE *,/* asynch_evd_handle    */
   OUT     DAT_IA_HANDLE *,/* ia_handle            */
   IN      DAT_UINT32,/* dat major version number */
   IN      DAT_UINT32,/* dat minor version number */
   IN      DAT_BOOLEAN);/* dat thread safety */

typedef DAT_RETURN (*DAT_IA_CLOSE_FUNC) (
   IN      DAT_IA_HANDLE,/* ia_handle            */
   IN      DAT_CLOSE_FLAGS );/* close_flags          */
```

```
typedef DAT_RETURN (*DAT_IA_QUERY_FUNC) (
   IN       DAT_IA_HANDLE,/* ia_handle            */
   OUT      DAT_EVD_HANDLE *,/* async_evd_handle     */
   IN       DAT_IA_ATTR_MASK,/* ia_attr_mask         */
   OUT      DAT_IA_ATTR *,/* ia_attr              */
   IN      DAT_PROVIDER_ATTR_MASK, /* provider_attr_mask  */
   OUT      DAT_PROVIDER_ATTR * );/* provider_attr       */

/* helper functions */

typedef DAT_RETURN (*DAT_SET_CONSUMER_CONTEXT_FUNC) (
   IN       DAT_HANDLE,/* dat_handle           */
   IN       DAT_CONTEXT);/* context              */

typedef DAT_RETURN (*DAT_GET_CONSUMER_CONTEXT_FUNC) (
   IN       DAT_HANDLE,/* dat_handle           */
   OUT      DAT_CONTEXT * );/* context              */

typedef DAT_RETURN (*DAT_GET_HANDLE_TYPE_FUNC) (
   IN       DAT_HANDLE,            /* dat_handle           */
   OUT      DAT_HANDLE_TYPE * );   /* dat_handle_type      */

/* CR functions */

typedef DAT_RETURN (*DAT_CR_QUERY_FUNC) (
   IN       DAT_CR_HANDLE,/* cr_handle            */
   IN       DAT_CR_PARAM_MASK,/* cr_param_mask        */
   OUT      DAT_CR_PARAM * );/* cr_param             */

typedef DAT_RETURN (*DAT_CR_ACCEPT_FUNC) (
   IN       DAT_CR_HANDLE,/* cr_handle            */
   IN       DAT_EP_HANDLE,/* ep_handle            */
   IN       DAT_COUNT,/* private_data_size    */
   IN       const DAT_PVOID );/* private_data         */

typedef DAT_RETURN (*DAT_CR_REJECT_FUNC) (
   IN       DAT_CR_HANDLE,         /* cr_handle */
   IN       DAT_COUNT,             /* private_data_size    */
   IN       const DAT_PVOID );     /* private_data         */
```

```
/* For DAT-1.1 this function is defined for both uDAPL and
kDAPL.
 * For DAT-1.0 it was only defined for uDAPL.
 */
typedef DAT_RETURN (*DAT_CR_HANDOFF_FUNC) (
    IN          DAT_CR_HANDLE,/* cr_handle            */
    IN          DAT_CONN_QUAL);/* handoff              */

/* EVD functions */

typedef DAT_RETURN (*DAT_EVD_RESIZE_FUNC) (
   IN      DAT_EVD_HANDLE,       /* evd_handle           */
   IN      DAT_COUNT );/* evd_min_qlen         */

typedef DAT_RETURN (*DAT_EVD_POST_SE_FUNC) (
   IN      DAT_EVD_HANDLE,       /* evd_handle           */
   IN      const DAT_EVENT * );   /* event                */

typedef DAT_RETURN (*DAT_EVD_DEQUEUE_FUNC) (
   IN      DAT_EVD_HANDLE,/* evd_handle           */
   OUT     DAT_EVENT * );/* event                */

typedef DAT_RETURN (*DAT_EVD_FREE_FUNC) (
   IN      DAT_EVD_HANDLE );/* evd_handle           */

typedef DAT_RETURN (*DAT_EVD_QUERY_FUNC) (
   IN      DAT_EVD_HANDLE,/* evd_handle           */
   IN      DAT_EVD_PARAM_MASK,/* evd_param_mask       */
   OUT     DAT_EVD_PARAM * );/* evd_param            */

/* EP functions */

typedef DAT_RETURN (*DAT_EP_CREATE_FUNC) (
   IN      DAT_IA_HANDLE,/* ia_handle            */
   IN      DAT_PZ_HANDLE,/* pz_handle            */
   IN      DAT_EVD_HANDLE,/* recv_completion_evd_handle */
   IN      DAT_EVD_HANDLE,/* request_completion_evd_handle
*/
```

```
   IN       DAT_EVD_HANDLE,/* connect_evd_handle   */
   IN       const DAT_EP_ATTR *,/* ep_attributes        */
   OUT      DAT_EP_HANDLE * );/* ep_handle            */

typedef DAT_RETURN (*DAT_EP_CREATE_WITH_SRQ_FUNC)(
        IN       DAT_IA_HANDLE,          /* ia_handle
*/
        IN       DAT_PZ_HANDLE,          /* pz_handle
*/
        IN       DAT_EVD_HANDLE,         /* recv_completion_
evd_handle */
        IN       DAT_EVD_HANDLE,         /* request_
completion_evd_handle */
        IN       DAT_EVD_HANDLE,         /* connect_evd_
handle   */
        IN       DAT_SRQ_HANDLE,          /* srq_handle
*/
        IN       const DAT_EP_ATTR *,    /* ep_at-
tributes        */
        OUT      DAT_EP_HANDLE * );/* ep_handle            */

typedef DAT_RETURN (*DAT_EP_QUERY_FUNC) (
   IN       DAT_EP_HANDLE,/* ep_handle            */
   IN       DAT_EP_PARAM_MASK,/* ep_param_mask        */
   OUT      DAT_EP_PARAM * );/* ep_param             */

typedef DAT_RETURN (*DAT_EP_MODIFY_FUNC) (
   IN       DAT_EP_HANDLE,/* ep_handle            */
   IN       DAT_EP_PARAM_MASK,/* ep_param_mask        */
   IN      const DAT_EP_PARAM * ); /* ep_param             */

typedef DAT_RETURN (*DAT_EP_CONNECT_FUNC) (
   IN       DAT_EP_HANDLE,/* ep_handle            */
   IN       DAT_IA_ADDRESS_PTR,/* remote_ia_address    */
   IN       DAT_CONN_QUAL,/* remote_conn_qual     */
   IN       DAT_TIMEOUT,/* timeout              */
   IN       DAT_COUNT,/* private_data_size    */
   IN       const DAT_PVOID,/* private_data         */
   IN       DAT_QOS,/* quality_of_service   */
   IN       DAT_CONNECT_FLAGS );/* connect_flags        */
```

```
typedef DAT_RETURN (*DAT_EP_COMMON_CONNECT_FUNC) (
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_IA_ADDRESS_PTR,/* remote_ia_address    */
    IN      DAT_TIMEOUT,/* timeout             */
    IN      DAT_COUNT,/* private_data_size    */
    IN      const DAT_PVOID ); /* private_data         */

typedef DAT_RETURN (*DAT_EP_DUP_CONNECT_FUNC) (
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_EP_HANDLE,/* ep_dup_handle         */
    IN      DAT_TIMEOUT,/* timeout             */
    IN      DAT_COUNT,/* private_data_size    */
    IN      const DAT_PVOID,/* private_data         */
    IN      DAT_QOS);/* quality_of_service   */

typedef DAT_RETURN (*DAT_EP_DISCONNECT_FUNC) (
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_CLOSE_FLAGS );/* close_flags          */

typedef DAT_RETURN (*DAT_EP_POST_SEND_FUNC) (
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_COUNT,/* num_segments          */
    IN      DAT_LMR_TRIPLET *,/* local_iov            */
    IN      DAT_DTO_COOKIE,/* user_cookie          */
    IN     DAT_COMPLETION_FLAGS ); /* completion_flags     */

typedef DAT_RETURN (*DAT_EP_POST_SEND_WITH_INVALIDATE_FUNC)
(
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_COUNT,/* num_segments          */
    IN      DAT_LMR_TRIPLET *,/* local_iov            */
    IN      DAT_DTO_COOKIE,/* user_cookie */
    IN      DAT_COMPLETION_FLAGS, /* completion_flags     */
    IN     DAT_BOOLEAN, /* invalidate_flag */
    IN    DAT_RMR_CONTEXT ); /* RMR context */

typedef DAT_RETURN (*DAT_EP_POST_RECV_FUNC) (
    IN      DAT_EP_HANDLE,/* ep_handle             */
    IN      DAT_COUNT,/* num_segments          */
```

```
   IN      DAT_LMR_TRIPLET *,/* local_iov           */
   IN      DAT_DTO_COOKIE,/* user_cookie          */
   IN     DAT_COMPLETION_FLAGS ); /* completion_flags    */

typedef DAT_RETURN (*DAT_EP_POST_RDMA_READ_FUNC) (
   IN      DAT_EP_HANDLE,/* ep_handle            */
   IN      DAT_COUNT,/* num_segments         */
   IN      DAT_LMR_TRIPLET *,/* local_iov           */
   IN      DAT_DTO_COOKIE,/* user_cookie          */
   IN     const DAT_RMR_TRIPLET *,/* remote_iov          */
   IN     DAT_COMPLETION_FLAGS ); /* completion_flags    */

typedef DAT_RETURN (*DAT_EP_POST_RDMA_READ_TO_RMR_FUNC) (
   IN      DAT_EP_HANDLE,/* ep_handle            */
   IN      const DAT_RMR_TRIPLET *,/* local_iov         */
   IN      DAT_DTO_COOKIE,/* user_cookie          */
   IN     const DAT_RMR_TRIPLET *,/* remote_iov          */
   IN     DAT_COMPLETION_FLAGS ); /* completion_flags    */

typedef DAT_RETURN (*DAT_EP_POST_RDMA_WRITE_FUNC) (
   IN      DAT_EP_HANDLE,/* ep_handle            */
   IN      DAT_COUNT,/* num_segments         */
   IN      DAT_LMR_TRIPLET *,/* local_iov           */
   IN      DAT_DTO_COOKIE,/* user_cookie          */
   IN     const DAT_RMR_TRIPLET *, /* remote_iov         */
   IN     DAT_COMPLETION_FLAGS ); /* completion_flags    */

typedef DAT_RETURN (*DAT_EP_GET_STATUS_FUNC) (
   IN      DAT_EP_HANDLE,/* ep_handle            */
   OUT     DAT_EP_STATE *,/* ep_state             */
   OUT     DAT_BOOLEAN *,/* recv_idle            */
   OUT     DAT_BOOLEAN * );/* request_idle         */

typedef DAT_RETURN (*DAT_EP_FREE_FUNC) (
   IN      DAT_EP_HANDLE);/* ep_handle          */

typedef DAT_RETURN (*DAT_EP_RESET_FUNC) (
   IN      DAT_EP_HANDLE);/* ep_handle          */
```

```
typedef DAT_RETURN (*DAT_EP_RECV_QUERY_FUNC)(
   IN      DAT_EP_HANDLE,   /* ep_handle            */
   OUT    DAT_COUNT *,      /* nbufs_allocated*/
   OUT    DAT_COUNT *);     /* bufs_alloc_span*/

typedef DAT_RETURN (*DAT_EP_SET_WATERMARK_FUNC)(
   IN     DAT_EP_HANDLE,    /* ep_handle */
   IN     DAT_COUNT,        /* ep_soft_high_watermark */
   IN     DAT_COUNT );      /* ep_hard_high_watermark */

/* LMR functions */

typedef DAT_RETURN (*DAT_LMR_FREE_FUNC) (
   IN      DAT_LMR_HANDLE); /* lmr_handle           */

typedef DAT_RETURN (*DAT_LMR_SYNC_RDMA_READ_FUNC)(
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
   IN const DAT_LMR_TRIPLET *, /* local segments */
   IN     DAT_VLEN );       /* num_segments */

typedef DAT_RETURN (*DAT_LMR_SYNC_RDMA_WRITE_FUNC)(
        IN      DAT_IA_HANDLE,          /* ia_handle
*/
   IN const DAT_LMR_TRIPLET *, /* local segments */
   IN     DAT_VLEN );       /* num_segments */

/* RMR functions */

typedef DAT_RETURN (*DAT_RMR_CREATE_FUNC) (
   IN      DAT_PZ_HANDLE,/* pz_handle           */
   OUT     DAT_RMR_HANDLE *);/* rmr_handle          */

typedef DAT_RETURN (*DAT_RMR_CREATE_FOR_EP_FUNC) (
   IN      DAT_PZ_HANDLE, /* pz_handle           */
   OUT     DAT_RMR_HANDLE *); /* rmr_handle          */

typedef DAT_RETURN (*DAT_RMR_QUERY_FUNC) (
   IN      DAT_RMR_HANDLE,/* rmr_handle          */
```

```
    IN      DAT_RMR_PARAM_MASK,/* rmr_param_mask        */
    OUT     DAT_RMR_PARAM *);/* rmr_param         */

typedef DAT_RETURN (*DAT_RMR_BIND_FUNC) (
    IN      DAT_RMR_HANDLE,/* rmr_handle          */
    IN      DAT_LMR_HANDLE,/* lmr_handle          */
    IN     const DAT_LMR_TRIPLET *, /* lmr_triplet          */
    IN      DAT_MEM_PRIV_FLAGS,/* mem_priv            */
    IN      DAT_VA_TYPE,             /* va_type */
    IN      DAT_EP_HANDLE,/* ep_handle           */
    IN      DAT_RMR_COOKIE,/* user_cookie         */
    IN      DAT_COMPLETION_FLAGS,/* completion_flags     */
    OUT     DAT_RMR_CONTEXT * );/* context            */

typedef DAT_RETURN (*DAT_RMR_FREE_FUNC) (
    IN      DAT_RMR_HANDLE);/* rmr_handle          */

/* PSP functions */

typedef DAT_RETURN (*DAT_PSP_CREATE_FUNC) (
    IN      DAT_IA_HANDLE,/* ia_handle           */
    IN      DAT_CONN_QUAL,/* conn_qual           */
    IN      DAT_EVD_HANDLE,/* evd_handle          */
    IN      DAT_PSP_FLAGS,/* psp_flags           */
    OUT     DAT_PSP_HANDLE * );/* psp_handle          */

typedef DAT_RETURN (*DAT_PSP_CREATE_ANY_FUNC) (
    IN      DAT_IA_HANDLE,/* ia_handle           */
    OUT     DAT_CONN_QUAL *,/* conn_qual           */
    IN      DAT_EVD_HANDLE,/* evd_handle          */
    IN      DAT_PSP_FLAGS,/* psp_flags           */
    OUT     DAT_PSP_HANDLE * );/* psp_handle          */

typedef DAT_RETURN (*DAT_PSP_QUERY_FUNC) (
    IN      DAT_PSP_HANDLE,       /* psp_handle          */
    IN      DAT_PSP_PARAM_MASK,   /* psp_param_mask      */
    OUT     DAT_PSP_PARAM * );    /* psp_param          */

typedef DAT_RETURN (*DAT_PSP_FREE_FUNC) (
```

```
    IN      DAT_PSP_HANDLE );/* psp_handle           */

/* RSP functions */

typedef DAT_RETURN (*DAT_RSP_CREATE_FUNC) (
    IN      DAT_IA_HANDLE,/* ia_handle           */
    IN      DAT_CONN_QUAL,/* conn_qual           */
    IN      DAT_EP_HANDLE,/* ep_handle           */
    IN      DAT_EVD_HANDLE,/* evd_handle          */
    OUT     DAT_RSP_HANDLE * );/* rsp_handle           */

typedef DAT_RETURN (*DAT_RSP_QUERY_FUNC) (
    IN      DAT_RSP_HANDLE,       /* rsp_handle           */
    IN      DAT_RSP_PARAM_MASK,   /* rsp_param_mask       */
    OUT     DAT_RSP_PARAM * );    /* rsp_param           */

typedef DAT_RETURN (*DAT_RSP_FREE_FUNC) (
    IN      DAT_RSP_HANDLE );/* rsp_handle           */

/* CSP functions functions - DAT 2.0 */

typedef DAT_RETURN (*DAT_CSP_CREATE_FUNC) (
        IN      DAT_IA_HANDLE,          /* ia_handle */
        IN      DAT_COMM *,          /* communicator */
        IN      DAT_IA_ADDRESS_PTR, /* address */
        IN      DAT_EVD_HANDLE,         /* evd_handle */
        OUT     DAT_CSP_HANDLE * );/* csp_handle          */

typedef DAT_RETURN (*DAT_CSP_QUERY_FUNC) (
        IN      DAT_CSP_HANDLE,  /* csp_handle           */
        IN      DAT_CSP_PARAM_MASK,/* csp_param_mask      */
    OUT     DAT_CSP_PARAM * );   /* csp_param            */

typedef DAT_RETURN (*DAT_CSP_FREE_FUNC) (
      IN     DAT_CSP_HANDLE );      /* csp_handle         */

/* PZ functions */

typedef DAT_RETURN (*DAT_PZ_CREATE_FUNC) (
```

```
    IN      DAT_IA_HANDLE,/* ia_handle              */
    OUT     DAT_PZ_HANDLE * );/* pz_handle              */

typedef DAT_RETURN (*DAT_PZ_QUERY_FUNC) (
    IN      DAT_PZ_HANDLE,/* pz_handle              */
    IN      DAT_PZ_PARAM_MASK,/* pz_param_mask         */
    OUT     DAT_PZ_PARAM *);/* pz_param              */

typedef DAT_RETURN (*DAT_PZ_FREE_FUNC) (
    IN      DAT_PZ_HANDLE );/* pz_handle              */

/* SRQ functions */

typedef DAT_RETURN (*DAT_SRQ_CREATE_FUNC)(
    IN    DAT_IA_HANDLE,            /* ia_handle */
    IN    DAT_PZ_HANDLE,            /* pz_handle */
    IN    DAT_SRQ_ATTR *,           /* srq_attributes */
    OUT   DAT_SRQ_HANDLE *);        /* srq_handle */

typedef DAT_RETURN (*DAT_SRQ_SET_LW_FUNC)(
    IN    DAT_SRQ_HANDLE,           /* srq_handle */
    IN    DAT_COUNT );              /* srq_low_watermark*/

typedef DAT_RETURN (*DAT_SRQ_FREE_FUNC)(
    IN    DAT_SRQ_HANDLE );         /* srq_handle */

typedef DAT_RETURN (*DAT_SRQ_QUERY_FUNC)(
    IN    DAT_SRQ_HANDLE ,          /* srq_handle */
    IN    DAT_SRQ_PARAM_MASK,       /* srq_param_mask */
    OUT   DAT_SRQ_PARAM *);         /* srq_param*/

typedef DAT_RETURN (*DAT_SRQ_RESIZE_FUNC)(
    IN    DAT_SRQ_HANDLE,           /* srq_handle */
    IN    DAT_COUNT );              /* srq_queue_length*/

typedef DAT_RETURN (*DAT_SRQ_POST_RECV_FUNC)(
    IN    DAT_SRQ_HANDLE,           /* srq_handle */
    IN    DAT_COUNT,                /* num_segments */
    IN    DAT_LMR_TRIPLET *,        /* local_iov */
```

```
   IN     DAT_DTO_COOKIE );         /* user_cookie */

typedef DAT_RETURN (*DAT_IA_HA_RELATED) (
         IN     DAT_IA_HANDLE,           /* ia_handle */
         IN     const DAT_NAME_PTR,      /* ia name */
         OUT    DAT_BOOLEAN*);           /* answer */

#ifdef DAT_EXTENSION
typedef  int    DAT_EXTENDED_OP;
#include <stdarg.h>
typedef DAT_RETURN (*DAT_HANDLE_EXTENDEDOP_FUNC)(
   IN     DAT_HANDLE,        /* handle */
   IN     DAT_EXTENDED_OP,   /* extended op */
   IN     va_list );         /* arguments list */
#endif /* DAT_EXTENSION */
#endif /* _DAT_REDIRECTION_H_ */
```