Using the GNI and DMAPP APIs

S–2446–31

Version 3.1 Published June 2010 Supports the Cray Linux Environment (CLE) 3.1 release and the System Management Workstation (SMW) 5.1 release.

Contents

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Page

Tables

Page

Figures

Introduction [1]

This guide includes reference information for the Generic Network Interface (GNI) and Distributed Shared Memory Application (DMAPP) APIs. The intended audience are programmers who are developing system software such as Partitioned Global Address Space (PGAS) compilers and communication libraries that use the Cray Gemini based system interconnection network to transfer data between processors on a Cray XE system.

The Cray Gemini application-specific integrated circuit (ASIC) provides an interface between the processors and the interconnection network. The ASIC provides the address translation mechanism, communication modes, and low-latency synchronization necessary to support the abstraction of a global, shared address space across the entire machine.

Each ASIC includes two network interface controllers (NICs), and an embedded interconnection switch (router). Each NIC is an independent, addressable endpoint in the network, therefore a single ASIC supports two nodes.

The Cray Gemini based system interconnection network and its associated software provides the following features:

- Support for message passing, one-sided operations, and global address space programming models.
- Global synchronization. Global timing information is passed through the high-speed network to synchronize the scheduler interrupts and time-of-day clocks in all the processors.
- Gather/scatter performance. A symmetric address translation mode allows access to all nodes in a job without needing to modify the fast memory access (FMA) window. This reduces processor and network overhead on operations involving a small amount of data on a large number of nodes. Network packet overhead is reduced so that network efficiency is high during these operations.
- Flat collectives. Support for atomic memory operations plus efficient scatters allows collectives to be programmed in a vector-like manner to scale much better than typical message-based algorithms.
- End-to-end data protection. Hardware support is provided so that all packets between the sender and receiver receive a cyclic redundancy check (CRC) to detect data corruption. Further, link-level data is resent if an error occurs while data is transiting a link.

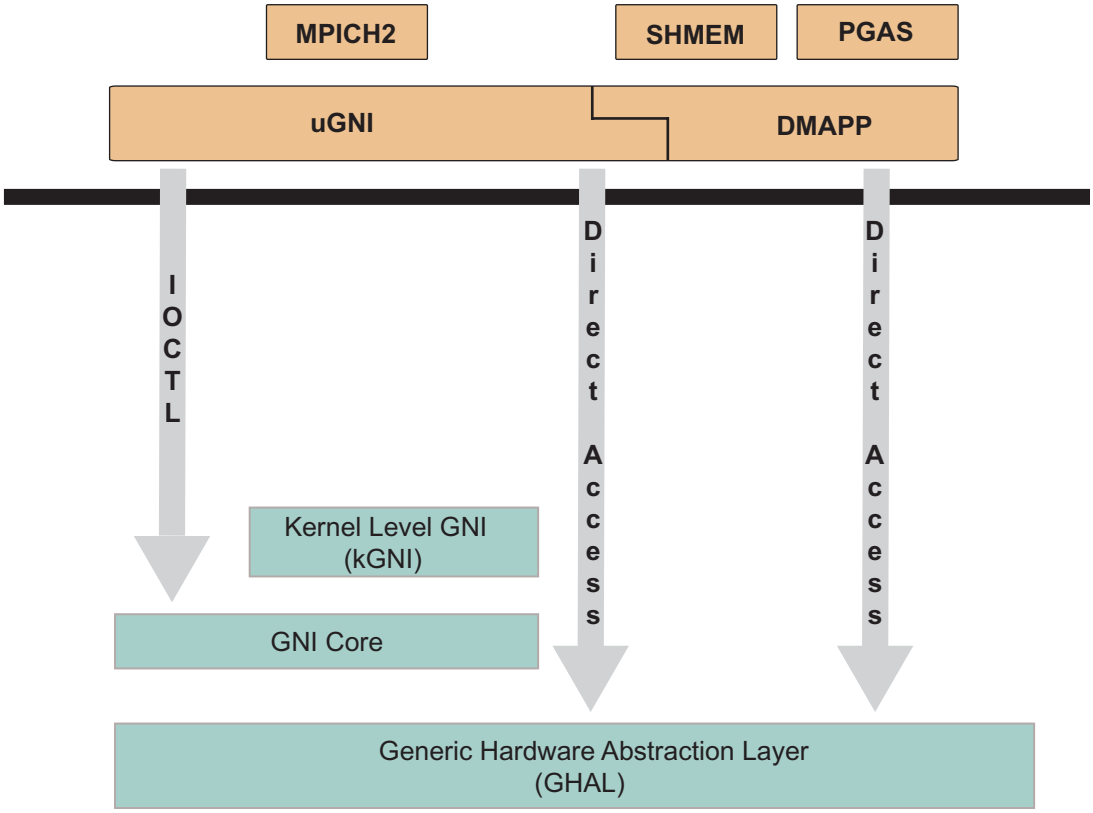
- Network routing allows you to add and delete nodes from the network while it is running.
- Flexible memory mapping. The Memory Relocation Table (MRT) allows software to use a contiguous address space when directly accessing memory allocated by your program.
- I/O performance enhanced by RDMA transfers from I/O adapters to remote memory throughout the system.
- Adaptive routing may be used for most network data, reducing sensitivity to network hot spots.

1.1 Software Stack

uGNI and DMAPP provide low-level communication services to user-space software. uGNI directly exposes the communications capabilities of the Cray Gemini ASIC, and is extensively described in Part I, *The GNI API*. DMAPP implements a logically shared, distributed memory (DM) programming model, and is extensively described in Part II, *The DMAPP API*.

The uGNI and DMAPP APIs allow system software to realize as much of the hardware performance of the Gemini network ASIC as possible while being reasonably portable to its successors.

Figure 1. GNI and DMAPP Software Layers



kGNI is a kernel module that presents to kernel-space code an API similar to that of uGNI. The GNI Core provides low-level services to both uGNI and kGNI. kGNI and GNI Core are both in the kGNI module.

The Generic Hardware Abstraction Layer (GHAL) isolates all software from the hardware specifics of theCray network application-specific integrated circuit (ASIC). These components are not described further in this book.

Layered on top of uGNI and DMAPP are portable communication libraries (such as MPICH2 and Cray SHMEM) and the Partitioned Global Address Space (PGAS) compilers (such as UPC and Coarray Fortran). These software components are extensively described in other books available from Cray Inc.

uGNI and DMAPP are packaged as libraries available with the Cray Linux Environment (CLE) 3.1 release and are installed in `/opt/cray/ugni` and `/opt/cray/dmapp`.

Part I: The GNI API

About the GNI API [2]

The GNI API includes two sets of function calls. User-level high performance applications use uGNI functions while kernel-level drivers use kGNI functions. This document describes the functionality of the uGNI set of function calls, focusing on their direct interaction with the NIC.

2.1 Functional Overview

A high performance user-level application would use the uGNI API to accomplish the following tasks in order to establish communication among its instances:

- Establish a communication domain and attach it to an NIC device
- Create one or more completion queues (CQs)
- Register memory for use by the Cray Gemini network ASIC
- Create logical endpoints
- Use Fast Memory Access (FMA) or Remote Direct Memory Access (RDMA) to communicate between endpoints.
- De-register memory to free up resource

Each of these tasks comprises a category of API functions as described in the following sections.

2.1.1 Establish Communication Domain

A Communication Domain is a software construct which defines a group of endpoints which can intercommunicate. The domain creation step establishes a unique set of domain properties including a unique identifier, which is used by the application to reference a particular instance of a communication domain.

The communication domain allows an application to enforce a protection scheme across all of its network transactions. The application attaches the domain to a specific NIC device.

Logical Endpoints are created within the communication domain and represent a virtual interface into the network. Communication takes place between endpoints on local and remote peers, where each endpoint is bound tightly to exactly one other endpoint. Logical endpoints may be used for initiating transactions. An application posts transaction requests to an endpoint to invoke communication through that endpoint. See Communication Domain on page 33.

2.1.2 Create Completion Queue (CQ)

Completion Queues (CQ) provide an event notification mechanism. For example, an application may use them to track the progress of Fast Memory Access (FMA) or Block Transfer Engine (BTE) requests, or to notify a remote node that data has been delivered to local memory.

An application must first initialize a CQ to obtain a completion queue handle, which is used for subsequent CQ references. An application then associates the CQ with the logical endpoints and with registered memory segments to be used for future data transactions. After initiating transactions between endpoints, an application references the associated CQ to track various events related to transaction completion, messaging notifications, and errors.

Completion queues have a fixed size which is specified when they are created. When the queue is full, it is said to be in the overrun state. CQEs received when the CQ is full are discarded.

Local completion queues track the completion of operations initiated on local endpoints. They are linked to these endpoints by being specified as a parameter to `EpCreate()`. See `EpCreate` on page 51.

Receive completion queues notify the application of completion of operations initiated on remote endpoints targeting local registered memory. They are linked with this memory by being specified as a parameter to `MemRegister()`. See `MemRegister` on page 44.

See Completion Queue Management on page 41.

2.1.3 Register Memory

Memory allocated by an application must be *registered* before it can be given to a peer as a destination buffer or used as a source buffer for most uGNI transactions.

Registration associates a specific memory segment (described by a pointer and a size) with an NIC that will be performing transactions from/to this memory. Memory can be registered with multiple NICs at the same time; it is up to the application to ensure that the NICs do not use the memory simultaneously.

When an application registers a memory segment, it receives a memory handle for subsequent references to that segment. The application attaches that segment to an NIC and must keep track of the handles for each attached NIC and de-register the associated memory when no longer needed. See Memory Registration on page 43.

2.1.4 Create Logical Endpoints

Before instances of an application can start communicating with each other, the logical local and remote endpoints have to be created. Endpoint properties include the handle of the NIC device used for this connection, the remote PE, and the local CQ. Applications usually synchronize among instances before attempting to bind endpoints. When no longer needed, the application must unbind the endpoint explicitly through function call or implicitly by destroying the endpoint. See Logical Endpoint on page 51.

2.1.5 Transfer Data

There are two mechanisms for accessing remote memory on another node — *Fast Memory Access* (FMA), and *Block Transfer Engine* (BTE).

For some transfer operations, the GNI kernel driver first exchanges *datagrams*, which contain messaging parameters, to initialize communication between PEs.

2.1.5.1 Fast Memory Access (FMA)

Use FMA primarily for the efficient transfer of small, possibly non-contiguous blocks of data between local and remote memory. For example, use FMA for the short inter-process data transfers typical of one-sided communication in models like Cray SHMEM, UPC or Coarray Fortran.

To send and receive short messages between endpoints an application must first initialize an endpoint with communication parameters and pre-registered buffers required for performing FMA transactions. An application then calls a send function with pointer, length and control information. An application calls a receive function to obtain a pointer to the header of the next available message for a given connection.

Either the application process messages immediately or copies them to another buffer. The application must release the message buffer when it is no longer needed. See FMA Short Messaging on page 67.

To access *Distributed Memory* (DM), moving user data between local and remote memory, an application prepares a Transaction Request Descriptor which has properties such as type (PUT/GET), CQ, data source and destination, and length. To post the transaction, an application passes the pointer to a Transaction Request Descriptor to the `PostFma` function. See FMA DM on page 66.

Prepare a transaction request for atomic memory operation (AMO) by specifying the remote node, the AMO command to execute, operands, and other fields depending on the syntax of the AMO command.

GNI implements FMA through a set of memory windows that enable data to be moved by the processor directly from user space, through the Cray Gemini ASIC, to the network.

2.1.5.2 Block Transfer Engine (BTE)

The BTE functionality, which is implemented on the ASIC, is primarily intended for large asynchronous data transfers between nodes. More time is required to set up data for a transfer, than for an FMA transfer, but once initiated, there is no further involvement by the processor core.

An application can instruct the Block Transfer Engine (BTE) to perform an RDMA PUT operation, which instructs the BTE to move data from local to a remote memory, or an RDMA GET operation, which instructs the BTE to move data from remote to local memory and to notify a source and destination upon completion. These functions write block transfer descriptors to a queue in the NIC, and the BTE services the requests asynchronously. Block transfer descriptors use privileged state, so access to the BTE is gated through the kernel. Due to the overhead of accessing the BTE through the operating system, the BTE mechanism is more appropriate for larger messages.

PUT/GET transactions require a pointer and a memory domain handle to identify the data source and destination, data length and return a transaction ID. These operations use several modes, some of which are appropriate for kernel-level applications because they bypass memory registration. Other modes are targeted for user applications which control data ordering, event notification, and synchronization. See RDMA (BTE) on page 73.

2.1.6 Process Completion Queue

The calling process must poll a completion queue for a completion entry to discover information about an event generated by the NIC device, i.e. messaging and data transactions. If a new completion entry is found, the application processes status information and event data.

Any type of an error in the network that leads to data loss will result in the NIC generating an interrupt and delivering an error into a corresponding completion queue.

To avoid dropped completion notifications, applications should make sure that the number of operations posted on Endpoints attached to a `src_cq_handle` does not exceed the completion queue capacity at any time. See Completion Queue Processing on page 74.

2.2 Restrictions

The total number of GNI application processes running on a given node should be limited to the number of CPU cores of the node.

The application is required to be statically or dynamically linked to Cray XE libraries and complied by the Cray PGAS group of languages.

2.3 Compiling

Applications must be compiled using a Cray PGAS compiler (i.e. UPC or Coarray Fortran) and linked with libraries provided on the CLE 3.1 System.

To compile and link an executable file which uses the GNI libraries:

```
cc -dynamic -lugni -o test test.c
```

GNI API Reference [3]

This chapter contains reference information for functions, structures, and enumerations contained in the GNI API. Your application must include the `gni_pub.h` file when using this API.

3.1 Naming Conventions

The GNI API defines four types of entities: functions, types, return codes and constants. User-level functions start with `GNI_` and use mixed upper and lower case. Kernel-level functions start with `gni_` and use lower case with underscores to separate words.

Only user-level GNI functions (uGNI) are documented in this guide.

3.2 Communication Domain

3.2.1 `CdmCreate`

The `GNI_CdmCreate` function creates an instance of the communication domain.

3.2.1.1 Synopsis

```
gni_return_t GNI_CdmCreate (
  IN uint32_t inst_id,
  IN uint8_t ptag,
  IN uint32_t cookie,
  IN uint32_t modes,
  OUT gni_cdm_handle_t *cdm_handle)
```

3.2.1.2 Parameters

inst_id	Rank of the instance in the job.
ptag	Protection tag.
cookie	Unique identifier generated by the system. Along with *ptag*, the *cookie* identifies the communication domain.

modes The modes bit mask. The following flags are used for this parameter:

- One of the following flags (the flags are mutually exclusive):
  - GNI_CDM_MODE_FORK_NOCOPY
  - GNI_CDM_MODE_FORK_PARTCOPY
  - GNI_CDM_MODE_FORK_FULLCOPY
- GNI_CDM_MODE_CACHED_AMO_ENABLED
- GNI_CDM_MODE_DUAL_EVENTS

  Must be used when local and global completion events are needed for RDMA post operations.
- GNI_CDM_MODE_FAST_DATAGRAM_POLL
- One of the following flags (the flags are mutually exclusive):
  - GNI_CDM_MODE_ERR_NO_KILL
  - GNI_CDM_MODE_ERR_ALL_KILL

cdm_handle Returns a pointer to a handle for the communication domain object. The handle is used by other functions to specify a particular instance of the communication domain.

3.2.1.3 Return Codes

GNI_RC_SUCCESS

The operation completed successfully.

GNI_RC_INVALID_PARAM

One of the input parameters was invalid.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

3.2.2 CdmDestroy

The GNI_CdmDestroy function destroys the instance of the communication domain and removes associations between the calling process and the Gemini NIC devices that were established by the corresponding GNI_CdmAttach function.

3.2.2.1 Synopsis

```
gni_return_t GNI_CdmDestroy (
  IN gni_cdm_handle_t cdm_handle)
```

3.2.2.2 Parameters

cdm_handle The communication domain handle.

3.2.2.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_INVALID_PARAM`

The caller specified an invalid communication domain handle.

3.2.3 `CdmGetNicAddress`

The `CdmGetNicAddress` function reads the `/sys/class/gemini/ghalX/address` file, where X is the *device_id*.

3.2.3.1 Synopsis

```
gni_return_t GNI_CdmGetNicAddress (
  IN uint32_t device_id,
  OUT uint32_t *address,
  OUT uint32_t *cpu_id )
```

3.2.3.2 Parameters

device_id The device identifier. For example, the NIC `/dev/kgni1` has the *device_id*= `DEVICE_MINOR_NUMBER-GEMINI_BASE_MINOR_NUMBER`= 1.

address PE address of the NIC.

cpu_id ID of the first CPU in the slot directly connected to the NIC.

3.2.3.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_NO_MATCH`

The specified *device_id* does not exist.

3.2.4 CdmAttach

The CdmAttach function associates the communication domain with a Gemini NIC and provides a NIC handle to the upper layer protocol. A process cannot attach a single communication domain instance to a Gemini NIC more than once, but it can attach multiple communication domains to a single Gemini NIC.

3.2.4.1 Synopsis

```
gni_return_t GNI_CdmAttach (
  IN gni_cdm_handle_t cdm_handle,
  IN uint32_t device_id,
  OUT uint32_t *local_address,
  OUT gni_nic_handle_t *nic_handle)
```

3.2.4.2 Parameters

cdm_handle Communication domain handle.

device_id Device identifier for the Gemini NIC to which the communication domain attaches. The device id is the minor number for the device that is assigned to the device by the system when the device is created. To determine the device number, look in the /dev directory, which contains a list of devices. For a NIC, the device is listed as *kgniX*, where *X* is the device number.

local_address

Returns a pointer to the PE address for the NIC that this function has attached to the communication domain.

nic_handle Returns a pointer to a handle for the NIC. The handle is used by the API to specify an instance of a Gemini NIC.

3.2.4.3 Return Codes

GNI_RC_SUCCESS

The operation completed successfully.

GNI_RC_INVALID_PARAM

The caller specified an invalid communication domain handle.

GNI_RC_NO_MATCH

The specified *device_id* does not exist.

GNI_RC_ERROR_RESOURCE

The operation failed due to insufficient resources. To resolve this, verify that the FMA descriptors are available on the given NIC. The most likely cause of this error is that too many CDM domains got attached to the given NIC on that node.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

GNI_RC_INVALID_STATE

The caller attempted to attach a communication domain instance to the Gemini NIC device more than once.

GNI_RC_PERMISSION_ERROR

Insufficient permissions to perform the operation.

3.2.5 GetVersion

The GetVersion function returns the version number of the uGNI library.

3.2.5.1 Synopsis

```
gni_return_t GNI_GetVersion(
  OUT uint32_t   *version)
```

3.2.5.2 Parameters

version GNI version number.

3.2.5.3 Return Codes

GNI_RC_SUCCESS

Operation completed successfully.

GNI_RC_INVALID_PARAM

The version is undefined.

3.2.6 ConfigureNTT

The Node Translation Table (NTT) works in conjunction with the FMA mechanism to allow applications to employ logical network endpoints when addressing remote nodes. This facilitates efficient user-level access to FMA, as well as simplifying checkpoint/restart operations, etc. There are 8192 entries in the NTT for each NIC on the Cray Gemini network ASIC. Each entry contains 18 bits of data which is used to convert an application virtual PE into a 16-bit Network Endpoint ID and a 2-bit Gemini core (DstID). Bit 17 of the entry specifies bit 1 of the DstID field. The NTTConfig register controls the granularity for NTT addressing.

The `GNI_ConfigureNTT` function sets up entries in the NTT associated with a particular `/dev/kgni` device.

If the table field of the input *ntt_desc* is set to NULL, the NTT entries starting from *ntt_base* up to and including *ntt_base* + *ntt_desc->group_size* – 1 are reset to 0.

If the *ntt_base* is -1 and *ntt_desc->group_size* is -1, and the table field of *ntt_desc* is NULL, all entries of NTT allocations not currently in use will be reset to 0.

3.2.6.1 Synopsis

```
gni_return_t GNI_ConfigureNTT (
  IN uint32_t device_id,
  IN gni_ntt_descriptor_t *ntt_desc,
  INOUT uint32_t ntt_base )
```

3.2.6.2 Parameters

device_id The device identifier, for example, for `/dev/kgni1` it is *device_id* `= DEVICE_MINOR_NUMBER – GEMINI_BASE_MINOR_NUMBER = 1`

ntt_desc NTT configuration descriptor. Descriptions are set using the `gni_ntt_descriptor` structure which has the types found in Table 1.

Table 1. `gni_ntt_descriptor`

Type	Option	Description
`uint32_t`	*group_size*	Size of the NTT group to be configured.
`uint8_t`	*granularity*	NTT granularity.
`uint32_t`	*table*	Pointer to the array of new NTT values.
`uint8_t`	*flags*	Configuration flags.

ntt_base On input, base entry into NTT. On return, set to the base entry allocated by the driver.

3.2.6.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

`GNI_RC_PERMISSION_ERROR`

The process has insufficient permission to set up NTT resources.

`GNI_RC_ERROR_RESOURCE`

A hardware resource limitation prevents NTT setup.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

`GNI_RC_NO_MATCH`

The specified *device_id* does not exist.

3.2.7 `ConfigureJob`

The `GNI_ConfigureJob` function sets the configuration options for the job which include the device ID, the job ID, the protection tag, cookie, and limit values for the job. The user (ALPS) can call this function multiple times for the same Gemini interface. The driver looks up a triplet (*job_id+ptag+cookie*) and then adds a new entry into the list it maintains for each physical NIC, for every unique triplet. Each entry may have a non-unique *job_id* or *ptag* or *cookie*. Using the same *ptag* with a different *job_id* is illegal and such calls fail. This function must be called before `GNI_CdmAttach` for the CDM with the same *ptag+cookie*. Calling `GNI_ConfigureJob` for the same triplet has no effect, unless *limits* is non-NULL.

An application may also use this function to associate NTT resources with a job. The NTT resources would have been previously allocated by a call to `GNI_ConfigureNTT`. In this case, the application sets the *ntt_base* and *ntt_size* fields in the limits input. If the application expects the driver to clean up the NTT resources upon termination of the job, the application sets the *ntt_ctrl* field in the limits input to `GNI_JOB_CTRL_NTT_CLEANUP`. The application should not attempt to change *ntt_base* or *ntt_size* by calling `ConfigureJob` subsequently with different NTT parameters.

3.2.7.1 Synopsis

```
gni_return_t GNI_ConfigureJob (
  IN uint32_t device_id,
  IN uint64_t job_id,
  IN uint8_t ptag,
  IN uint32_t cookie,
  IN gni_job_limits_t *limits )
```

3.2.7.2 Parameters

device_id The device identifier, for example, for `/dev/kgni1` has *device_id* = `DEVICE_MINOR_NUMBER – GEMINI_BASE_MINOR_NUMBER = 1`.

job_id Job container identifier.

ptag Protection tag to be used by all applications in the given job container.

cookie Unique identifier. Assigned to all applications within the job container along with *ptag*.

limits When this argument is non-NULL, the driver takes all the limit values that are not set to `GNI_JOB_INVALID_LIMIT` and stores them into the table indexed by the *ptag*. These limits are imposed on all applications running within the given job container. If you set different limits for the same *ptag*, the driver overwrites previously set limits.

3.2.7.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

`GNI_RC_PERMISSION_ERROR`

The process has insufficient permission to configure job or no NTT entries exist for input *ntt_base*/*ntt_size* fields in the limits argument.

`GNI_RC_NO_MATCH`

The specified *device_id* does not exist or there are no NTT entries.

`GNI_RC_INVALID_STATE`

The caller attempted to use the same *ptag* with a different *job_id* or a different *cookie*.

`GNI_RC_ILLEGAL_OP`

The application is attempting to resize the NTT resources.

`GNI_RC_ERROR_RESOURCE`

A resource allocation error was encountered while trying to configure the job resources.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.3 Completion Queue Management

3.3.1 `CqCreate`

The `CqCreate` function creates a new completion queue. The caller must specify the minimum number of completion entries that the queue must contain in the *entry_count* parameter. To avoid dropped completion notifications, you should set up your application to verify that the number of operations posted on endpoints attached to a `cq_handle` do not exceed the completion queue capacity at any time.

The *event_handler* function, if specified, is called if (and only if) `CqGetEvent` or `CqWaitEvent` return with either `GNI_RC_SUCCESS` or `GNI_RC_TRANSACTION_ERROR`. The handler is invoked at some time between the time that the CQ entry arrives in the CQ, and the successful return of `GNI_CqGetEvent` or `GNI_CqWaitEvent`.

The user must call `GNI_CqGetEvent` or `GNI_CqWaitEvent` for each event deposited into the CQ, regardless of whether an *event_handler* is used.

Completion queues may be used for the receipt of locally generated events, such as those arising from GNI_Post style transactions or may be used for the receipt of remote events, but not both.

3.3.1.1 Synopsis

```
gni_return_t GNI_CqCreate (
  IN gni_nic_handle_t nic_handle,
  IN uint32_t entry_count,
  IN uint32_t delay_ count,
  IN uint32_t mode,
  IN void (*event_handler)(gni_cq_entry_t *, void *),
  IN void *context,
  OUT gni_cq_handle_t *cq_handle)
```

3.3.1.2 Parameters

nic_handle The handle of the associated Gemini NIC.

entry_count

The number of completion entries that this completion queue holds.

delay_count

The number of events the NIC allows before generating an interrupt. Setting this parameter to zero results in interrupt delivery with every event. When using this parameter, the *mode* parameter must be set to `GNI_CQ_BLOCKING`.

mode The mode of operation for the new completion queue. The following modes are used by this parameter:

- `GNI_CQ_BLOCKING`
- `GNI_CQ_NOBLOCK`

event_handler

User supplied callback function to be run for each CQ entry received in the CQ. The handler is supplied with two arguments: a pointer to the CQ entry, and a pointer to the context provided at CQ creation.

context User-supplied pointer that is passed to the handler callback function.

cq_handle Returns a pointer to the handle of the newly created completion queue.

3.3.1.3 Return Codes

`GNI_RC_SUCCESS`

A new completion queue was successfully created.

`GNI_RC_INVALID_PARAM`

One or more of the parameters was invalid.

`GNI_RC_ERROR_RESOURCE`

The completion queue could not be created due to insufficient resources.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.3.2 `CqDestroy`

The `CqDestroy` function destroys a specified completion queue. If any endpoints are associated with the completion queue, the completion queue is not destroyed and an error is returned.

3.3.2.1 Synopsis

```
gni_return_t GNI_CqDestroy (
  IN gni_cq_handle_t cq_handle)
```

3.3.2.2 Parameters

cq_handle The handle for the completion queue to be destroyed.

3.3.2.3 Return Codes

`GNI_RC_SUCCESS`

The completion queue was successfully destroyed.

`GNI_RC_INVALID_PARAM`

The `cq_handle` was invalid.

`GNI_RC_ERROR_RESOURCE`

The completion queue could not be destroyed because one or more endpoint instances are still associated with it. Use `EpDestroy` to destroy the endpoint instance, then try calling this function again.

3.4 Memory Registration

After an application allocates a memory region be used for data transfers, it must *register* the memory region to support the remote address translation and data protection mechanism.

Depending on the size of the allocated memory region, the registration function configures either GART entries on the AMD processor, or, in the case of huge pages, it configures entries in the Memory Relocation Table (MRT) on the NIC, to span the allocated memory region. Address translation uses one of these two translation mechanisms.

Registration also typically configures a Memory Domain Descriptor (MDD), which can be later referenced by its memory domain handle (MDH). When an NIC encounters an incoming local memory reference, the NIC uses the memory domain handle (MDH) as an index into a table of memory domain descriptors (MDDs) on the local PE. Each MDD provides the base address and bounds of a local memory region. The address contained within the incoming network packet is added to the base, and checked against the limit; the address is used as an offset into a local memory window defined by the MDD. This allows the local node to place the memory associated with a given MDD in any location in its local memory space using the associated MRT or GART entry.

MDHs are partially specified by the user and cannot be trusted by the NIC driver, so each MDD also contains a *protection tag* (PTag) which is assigned by the operating system and cannot be modified by the user. The PTag in an incoming memory reference is checked against a PTag stored in the referenced MDD, to verify that the reference is permitted to use that MDD.

3.4.1 Virtual Memory

The memory registration mechanism also supports a capability to use virtual Memory Domain Handles (vMDH), which supports Distributed Memory (DM) programming models.

For this discussion, a DM programming model is defined as a parallel job consisting of multiple independent processes distributed across one or more nodes of a Cray XE system. The processes may be executing the same application or different ones. At least one memory segment of equal size on each node is made remotely accessible by all of the processes in the job. To implement this model, the Virtual Memory Domain Handle Table (vMDHT) creates a relationship between the virtual MDH in the incoming network request and the actual memory domain handle to use in looking up the MDD associated with the incoming reference.

3.4.2 `MemRegister`

The `MemRegister` function allows a process to register a region of memory with the NIC. Before calling this function, the user must first allocate the memory.

The user may specify an arbitrary size region of memory, with arbitrary alignment, but the actual area of memory registered will be on MRT block granularity (or physical page granularity if MRT is not enabled for this process).

Users should usually choose a single-segment memory registration to register application memory, with multiple-segment registration being reserved for special cases. Using a single segment to register a memory region allows an application to use a virtual address in future transactions in and out of the registered region. Using multiple segments during the registration requires the application to use an offset within the registered memory region instead of a virtual address in all future transactions, where the registered region is aligned to MRT block size (or page size for non-MRT registrations).

A new memory handle is generated for each region of memory that is registered by a process. A length parameter of zero in any segment results in a `GNI_RC_INVALID_PARAM` error. While `GNI_MEM_USE_VMDH` flag is set, this function fails with `GNI_RC_ERROR_RESOURCE` return code if the memory domain descriptor block was never allocated using the `AllocMddResources` function or if the virtual MDH entry specified by *vmdh_index* is already in use.

3.4.2.1 Synopsis

```
gni_return_t GNI_MemRegister (
  IN gni_nic_handle_t nic_handle,
  IN uint64_t address,
  IN uint64_t length,
  IN gni_cq_handle_t dst_cq_handle,
  IN uint32_t flags,
  IN uint32_t vmdh_index,
  INOUT gni_mem_handle_t *mem_handle)
```

3.4.2.2 Parameters

nic_handle Handle of a currently open Gemini NIC.

address Starting address of the memory region to be registered.

length Length of the memory region to be registered, in bytes.

dst_cq_handle

If this value is not NULL, it specifies the completion queue to receive events related to the transactions initiated by the remote node into this memory region.

flags Attributes of the memory region. A combination of the following flags are used for this parameter:

`GNI_MEM_READWRITE`

The read/write attribute is associated with the memory region.

`GNI_MEM_READ_ONLY`

The read only attribute is associated with the memory region.

`GNI_MEM_USE_VMDH`

Directive to use virtual MDH while registering this memory region.

`GNI_MEM_USE_GART`

Directive to use GART while registering the memory region.

`GNI_MEM_STRICT_PI_ORDERING`

Instructs the NIC to enforce HT ordering for the memory region.

`GNI_MEM_PI_FLUSH`

Instructs the NIC to issue a `HT FLUSH` command prior to sending network responses for the memory region.

vmdh_index Specifies the index within the pre-allocated memory domain descriptor block that must be used for the registration. For example, when this parameter is set to `0`, it uses the first entry of the memory domain descriptor block. If set to `-1`, it relies on the GNI library to allocate the next available entry from the memory domain descriptor block.

mem_handle The new memory handle for the region.

3.4.2.3 Return Codes

`GNI_RC_SUCCESS`

The memory region was successfully registered.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

GNI_RC_ERROR_RESOURCE

The registration operation failed due to insufficient resources.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

GNI_RC_PERMISSION_ERROR

The user's buffer read/write permissions conflict with the flags argument.

3.4.3 MemRegisterSegments

The MemRegisterSegments function allows a process to register a memory region comprised of multiple memory segments with the NIC.

Multiple segment registration should be reserved for special cases. Single segment memory registration is the preferred method for memory registration. To register a single segment, use GNI_MemRegister.

The user may specify an arbitrary size region of memory, with arbitrary alignment, but the actual area of memory registered will be registered on MRT block granularity (or physical page granularity if MRT is not enabled for this process).

If an application registers multiple segments, it must use an offset within the registered memory region instead of a virtual address in all future transactions where registered region is aligned to MRT block size (or page size for non-MRT registrations). This is because a single memory domain is used for the registration of multiple segments and transactions must access memory for these segments as if it was contiguous.

A new memory handle is generated for each region of memory that is registered by a process. A length parameter of zero in any segment results in a GNI_RC_INVALID_PARAM error. While GNI_MEM_USE_VMDH flag is set, this function fails with a GNI_RC_ERROR_RESOURCE return code if memory domain descriptor block was never allocated using the AllocMddResources function or if the virtual MDH entry specified by *vmdh_index* is already in use.

3.4.3.1 Synopsis

```
gni_return_t GNI_MemRegisterSegments (
  IN gni_nic_handle_t nic_handle,
  IN gni_mem_segment_t *mem_segments,
  IN uint32_t segments_cnt,
  IN gni_cq_handle_t dst_cq_handle,
  IN uint32_t flags,
  IN uint32_t vmdh_index,
  INOUT gni_mem_handle_t *mem_handle)
```

3.4.3.2 Parameters

nic_handle Handle of a currently open Gemini NIC.

mem_segments

List of segments to register. Each element of the list consists of the starting address of the memory region and the length, in bytes. The list elements are set using the `gni_mem_segment` structure.

segment_cnt

Number of segments in the *mem_segments* list.

dst_cq_handle

If this value is not `NULL`, it specifies the completion queue to receive events related to the transactions initiated by the remote node into this memory region.

flags Attributes of the memory region. A combination of the following flags are used for this parameter:

`GNI_MEM_READWRITE`

The read/write attribute is associated with the memory region.

`GNI_MEM_READ_ONLY`

The read only attribute is associated with the memory region.

`GNI_MEM_USE_VMDH`

Directive to use virtual MDH while registering this memory region.

`GNI_MEM_USE_GART`

Directive to use GART while registering the memory region.

`GNI_MEM_STRICT_PI_ORDERING`

Instructs the NIC to enforce HT ordering for the memory region.

`GNI_MEM_PI_FLUSH`

Instructs the NIC to issue a `HT FLUSH` command prior to sending network responses for the memory region.

vmdh_index Specifies the index within the pre-allocated memory domain descriptor block that must be used for the registration. For example, when this parameter is set to `0`, it uses the first entry of the memory domain descriptor block. If set to `-1`, it relies on the GNI library to allocate the next available entry from the memory domain descriptor block.

mem_handle The new memory handle for the region.

3.4.3.3 Return Codes

`GNI_RC_SUCCESS`

The memory region was successfully registered.

`GNI_RC_INVALID_PARAM`

One on the parameters was invalid.

`GNI_RC_ERROR_RESOURCE`

The registration operation failed due to insufficient resources.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

`GNI_RC_PERMISSION_ERROR`

The user's buffer read/write permissions conflict with the flags argument.

3.4.4 `SetMddResources`

The `SetMddResources` function specifies the size of a contiguous block of MDD entries that can be used for future memory registrations.

3.4.4.1 Synopsis

```
gni_return_t GNI_SetMddResources (
  IN gni_nic_handle_t nic_handle,
  IN uint32_t num_entries
```

3.4.4.2 Parameters

nic_handle The handle for the NIC.

num_entries

Number of MDD entries in the block.

3.4.4.3 Return Codes

`GNI_RC_SUCCESS`

The block size was successfully specified.

`GNI_RC_INVALID_PARAM`

One or more of the parameters was invalid.

`GNI_RC_ERROR_NOMEM`

Insuficient memory to complete the operation.

3.4.5 `MemDeregister`

The `MemDeregister` function de-registers memory that was previously registered and unlocks the associated pages from physical memory. The contents and attributes of the region of memory being de-registered are not altered in any way.

3.4.5.1 Synopsis

```
gni_return_t GNI_MemDeregister (
  IN gni_nic_handle_t nic_handle,
  IN gni_mem_handle_t *mem_handle
```

3.4.5.2 Parameters

nic_handle The handle for the NIC that owns the memory region being deregistered.

mem_handle Memory handle for the region; obtained from a previous call to `MemRegister`.

3.4.5.3 Return Codes

`GNI_RC_SUCCESS`

The memory region was successfully de-registered.

`GNI_RC_INVALID_PARAM`

One or more of the parameters was invalid.

3.5 Logical Endpoint

3.5.1 `EpCreate`

The `EpCreate` function creates an instance of a logical endpoint. A new instance is always created in a non-bound state. A non-bound endpoint is able to exchange posted data with any bound remote endpoint within the same communication domain. An endpoint cannot be used to post RDMA or FMA transactions or to send short messages while it is in a non-bound state.

3.5.1.1 Synopsis

```
gni_return_t GNI_EpCreate (
  IN gni_nic_handle_t nic_handle,
  IN gni_cq_handle_t src_cq_handle,
  OUT gni_ep_handle_t *ep_handle)
```

3.5.1.2 Parameters

nic_handle Handle of the associated Gemini NIC.

src_cq_handle

Handle of the completion queue that is used by default to deliver events related to the transactions initiated by the local node.

ep_handle Returns a pointer to the handle of the newly created endpoint instance.

3.5.1.3 Return Codes

`GNI_RC_SUCCESS`

Operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.5.2 EpSetEventData

The EpSetEventData function allows the user to define the values that the EpBind function uses to generate CQ events. By default, EpBind uses the Communication Domain's *inst_id* as the event data for generating global and remote CQ events and the Endpoint's *remote_id* for generating local CQ events.

3.5.2.1 Synopsis

```
gni_return_t GNI_EpSetEventData (
  IN gni_ep_handle_t ep_handle,
  IN uint32_t local_event,
  IN uint32_t remote_event)
```

3.5.2.2 Parameters

ep_handle The handle of the endpoint instance to define event data.

local_event

The value to use when generating local CQ events.

remote_event

The value to use when generating global and remote CQ events.

3.5.2.3 Return Codes

GNI_RC_SUCCESS

Operation completed successfully.

GNI_RC_INVALID_PARAM

An invalid endpoint handle was specified.

3.5.3 EpBind

The EpBind function binds a logical endpoint to a specific remote address and remote instance within the communication domain. Once bound, the endpoint can be used to post RDMA and FMA transactions.

3.5.3.1 Synopsis

```
gni_return_t GNI_EpBind (
  IN gni_ep_handle_t ep_handle,
  IN uint32_t remote_addr,
  OUT uint32_t remote_id)
```

3.5.3.2 Parameters

ep_handle Handle of the endpoint instance to be bound.

remote_addr

Physical address of the Gemini NIC at the remote peer or NTT index, when NTT is enabled for the given communication domain.

remote_id User-specified ID of the remote instance in the job or unique identifier of the remote instance within the upper layer protocol domain.

3.5.3.3 Return Codes

`GNI_RC_SUCCESS`

Operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

`GNI_RC_ERROR_RESOURCE`

Failed due to insufficient resources.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.5.4 EpUnbind

The `EpUnbind` function unbinds a logical endpoint from the specific address and remote instance and releases any internal short message resources. A non-bound endpoint can exchange posted data with any bound remote endpoint within the same communication domain. An endpoint cannot be used to post RDMA, FMA transactions, or send short messages while it is in a non-bound state.

3.5.4.1 Synopsis

```
gni_return_t GNI_EpUnbind (
  IN gni_ep_handle_t ep_handle)
```

3.5.4.2 Parameters

ep_handle The handle of the endpoint instance to be unbound.

3.5.4.3 Return Codes

`GNI_RC_SUCCESS`

Operation completed successfully.

`GNI_RC_INVALID_PARAM`

An invalid endpoint handle was specified.

`GNI_RC_NOT_DONE`

The endpoint still has outstanding transaction requests or pending datagrams and cannot be unbound at this time. Retry unbinding later.

3.5.5 EpDestroy

The `EpDestroy` function destroys an endpoint, cancels any outstanding requests and releases short messaging resources.

3.5.5.1 Synopsis

```
gni_return_t GNI_EpDestroy (
  IN gni_ep_handle_t ep_handle)
```

3.5.5.2 Parameters

ep_handle The handle of the endpoint instance to be destroyed.

3.5.5.3 Return Codes

`GNI_RC_SUCCESS`

Operation completed successfully.

`GNI_RC_INVALID_PARAM`

An invalid endpoint handle was specified.

3.5.6 EpPostData

The `EpPostData` function posts a datagram to be exchanged with a remote, bound endpoint in the communication domain.

3.5.6.1 Synopsis

```
gni_return_t GNI_EpPostData (
  IN gni_ep_handle_t ep_handle,
  IN void *in_data,
  IN uint16_t data_len,
  IN void *out_buf,
  IN uint16_t buf_size)
```

3.5.6.2 Parameters

ep_handle	Handle of the local endpoint.
in_data	Pointer to the data to send.
data_len	Size of the data to send, in bytes.
out_buf	Pointer to the buffer that receives the incoming datagram.
buf_size	Size of the buffer for the incoming datagram, in bytes.

3.5.6.3 Return Codes

`GNI_RC_SUCCESS`

The posted datagram is queued.

`GNI_RC_INVALID_PARAM`

The specified endpoint handle is invalid.

`GNI_RC_ERROR_RESOURCE`

The system only allows a single outstanding datagram transaction for each endpoint. There is already a pending datagram for the specified endpoint

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete transaction.

`GNI_RC_SIZE_ERROR`

The size of the datagram is too large.

3.5.7 EpPostDataWId

The EpPostDataWId function posts a datagram with a user-specified *datagram_id* to be exchanged with a remote endpoint in the communication domain. If the local endpoint is unbound, a datagram can be exchanged with any bound endpoint within the communication domain.

It is required that datagrams posted on unbound endpoints be associated with a *datagram_id*.

3.5.7.1 Synopsis

```
gni_return_t GNI_EpPostDataWId (
  IN gni_ep_handle_t ep_handle,
  IN void *in_data,
  IN uint16_t data_len,
  IN void *out_buf,
  IN uint16_t buf_size,
  IN uint64_t datagram_id)
```

3.5.7.2 Parameters

ep_handle Handle of the local endpoint.

in_data Pointer to the data to send.

data_len Size of the data to send.

out_buf Pointer to the buffer that receives the incoming datagram.

buf_size Size of the buffer for the incoming datagram.

datagram_id

Id associated with the datagram.

3.5.7.3 Return Codes

GNI_RC_SUCCESS

The posted datagram is queued.

GNI_RC_INVALID_PARAM

The specified endpoint handle is invalid, or an invalid value for the *datagram_id* was specified.

GNI_RC_ERROR_RESOURCE

The system only allows a single outstanding datagram transaction for each endpoint.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete transaction.

GNI_RC_SIZE_ERROR

The size of the datagram is too large.

3.5.8 EpPostDataTest

The EpPostDataTest function returns the state of the EpPostData transaction.

3.5.8.1 Synopsis

```
gni_return_t GNI_EpPostDataTest (
  IN gni_ep_handle_t ep_handle,
  OUT gni_ep_post_state_t *post_state,
  OUT uint32_t *remote_address,
  OUT uint32_t *remote_id)
```

3.5.8.2 Parameters

ep_handle Handle of the local endpoint.

post_state Returns a pointer to the state of the transaction. The following states are used for this parameter:

- GNI_POST_PENDING
- GNI_POST_COMPLETED
- GNI_POST_ERROR
- GNI_POST_TIMEOUT
- GNI_POST_TERMINATED
- GNI_POST_REMOTE_DATA

remote_address

Returns a pointer to the physical address of the Gemini NIC being used by the remote peer. The address is only valid if the *post_state* returns GNI_POST_COMPLETE.

remote_id Returns a pointer to the user specific ID of the remote instance in the job. The ID is only valid if the *post_state* returns GNI_POST_COMPLETE.

3.5.8.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_INVALID_PARAM`

An invalid endpoint handle was specified.

`GNI_RC_NO_MATCH`

No matching datagram was found.

`GNI_RC_SIZE_ERROR`

The size of the output buffer is too small for the received datagram.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.5.9 `EpPostDataTestById`

The `EpPostDataTestById` function returns the state of the `EpPostData` transaction with the specified *datagram_id*.

3.5.9.1 Synopsis

```
gni_return_t GNI_EpPostDataTestById (
  IN gni_ep_handle_t ep_handle,
  IN uint64_t datagram_id,
  OUT gni_ep_post_state_t *post_state,
  OUT uint32_t *remote_address,
  OUT uint32_t *remote_id)
```

3.5.9.2 Parameters

ep_handle Handle of the local endpoint. Must be the same as that used when posting the datagram using `EpPostDataWId`.

datagram_id

Id of the datagram to test for.

post_state Returns a pointer to the state of the transaction. The following states are used for this parameter:

- GNI_POST_PENDING
- GNI_POST_COMPLETED
- GNI_POST_ERROR
- GNI_POST_TIMEOUT
- GNI_POST_TERMINATED
- GNI_POST_REMOTE_DATA

remote_address

Returns a pointer to the physical address of the Gemini NIC being used by the remote peer. The address is only valid if the *post_state* returns GNI_POST_COMPLETE.

remote_id Returns a pointer to the user specific ID of the remote instance in the job. The ID is only valid if the *post_state* returns GNI_POST_COMPLETE.

3.5.9.3 Return Codes

GNI_RC_SUCCESS

The operation completed successfully.

GNI_RC_INVALID_PARAM

An invalid endpoint handle was specified.

GNI_RC_NO_MATCH

No matching datagram was found.

GNI_RC_SIZE_ERROR

The size of the output buffer is too small for the received datagram.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

3.5.10 EpPostDataWait

The EpPostDataWait function is used to determine the result of a previously posted EpPostData call on the specified endpoint, blocking the calling thread until the completion of the posted transaction or until the specified timeout expires.

3.5.10.1 Synopsis

```
gni_return_t GNI_EpPostDataWait(
  IN gni_ep_handle_t ep_handle,
  IN uint32_t timeout,
  OUT gni_post_state_t *post_state,
  OUT uint32_t *remote_address,
  OUT uint32_t *remote_id)
```

3.5.10.2 Parameters

ep_handle Handle of the local endpoint.

timeout The count that this function waits, in milliseconds, for transaction to complete. Set to (-1) if no timeout is desired. A timeout value of zero results in a GNI_RC_INVALID_PARAM error return.

post_state State of the transaction

remote_address

Physical address of the Gemini NIC at the remote peer. Valid only if *post_state* returned GNI_POST_COMPLETE.

remote_id User specific ID of the remote instance in the job (user). Unique address of the remote instance within the upper layer protocol domain (kernel). Valid only if *post_state* returned GNI_POST_COMPLETE.

3.5.10.3 Return Codes

GNI_RC_NO_MATCH

No matching datagram was found.

GNI_RC_SUCCESS

The transaction completed successfully.

GNI_RC_INVALID_PARAM

The specified endpoint handle is invalid or timeout was set to zero.

GNI_RC_TIMEOUT

The timeout expired before a successful completion of the transaction.

GNI_RC_SIZE_ERROR

Output buffer is too small for the size of the received datagram.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

3.5.11 EpPostDataWaitById

The EpPostDataWaitById function determines the result of a previously posted EpPostData call on the specified endpoint, blocking the calling thread until the transaction involving *datagram_id* has completed, or until the specified timeout expires.

3.5.11.1 Synopsis

```
gni_return_t GNI_EpPostDataWaitById (
  IN gni_ep_handle_t ep_handle,
  IN uint64_t datagram_id,
  IN uint32_t timeout,
  OUT gni_ep_post_state_t *post_state,
  OUT uint32_t *remote_address,
  OUT uint32_t *remote_id)
```

3.5.11.2 Parameters

ep_handle Handle of the local endpoint.

datagram_id

Id of the datagram to wait for.

timeout The length of time that this function waits, in milliseconds, for the transaction to complete. If a timeout is not needed, set this parameter to -1. If the timeout is set to zero, the system returns the GNI_RC_INVALID_PARAM error.

post_state Returns a pointer to the state of the transaction. The following states are used for this parameter:

- GNI_POST_PENDING
- GNI_POST_COMPLETED
- GNI_POST_ERROR
- GNI_POST_TIMEOUT
- GNI_POST_TERMINATED
- GNI_POST_REMOTE_DATA

remote_address

Returns a pointer to the physical address of the Gemini NIC being used by the remote peer. The address is only valid if the *post_state* returns GNI_POST_COMPLETE.

remote_id Returns a pointer to the user specific ID of the remote instance in the job. The ID is only valid if the *post_state* returns GNI_POST_COMPLETE.

3.5.11.3 Return Codes

`GNI_RC_SUCCESS`

The transaction completed successfully.

`GNI_RC_INVALID_PARAM`

An invalid endpoint handle was specified or timeout was set to zero, or invalid datagram id was specified.

`GNI_RC_TIMEOUT`

The timeout expired before a successful completion of the transaction.

`GNI_RC_SIZE_ERROR`

The size of the output buffer is too small for the received datagram.

`GNI_RC_NO_MATCH`

No matching datagram found.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.5.12 `EpPostDataCancel`

The `EpPostDataCancel` function cancels an outstanding post data transaction.

3.5.12.1 Synopsis

```
gni_return_t GNI_EpPostDataCancel (
  IN gni_ep_handle_t ep_handle)
```

3.5.12.2 Parameters

ep_handle Handle of the local endpoint.

3.5.12.3 Return Codes

`GNI_RC_SUCCESS`

The transaction cancellation was successful.

`GNI_RC_INVALID_PARAM`

The *ep_handle* parameter is invalid.

`GNI_RC_NO_MATCH`

No active post data transaction on the *ep_handle*.

3.5.13 EpPostDataCancelById

The EpPostDataCancelById function cancels an outstanding post data transaction with the specified datagram Id.

3.5.13.1 Synopsis

```
gni_return_t GNI_EpPostDataCancelById (
  IN gni_ep_handle_t ep_handle,
  IN uint64_t datagram_id)
```

3.5.13.2 Parameters

ep_handle Handle of the local endpoint.

datagram_id

Id of the datagram to cancel.

3.5.13.3 Return Codes

GNI_RC_SUCCESS

The transaction cancellation was successful.

GNI_RC_INVALID_PARAM

One of the input parameters are invalid.

GNI_RC_NO_MATCH

No active post data transaction with the specified Id on the *ep_handle*.

3.5.14 PostDataProbe

The PostDataProbe function returns the remote ID and remote address of the first datagram found in completed, timed out, or canceled state for the CDM associated with the input NIC handle. This function must be used in conjunction with GNI_EpPostDataTestById or GNI_EpPostDataWaitById to obtain data exchanged in the datagram transaction.

3.5.14.1 Synopsis

```
gni_return_t GNI_PostDataProbe (
  IN gni_nic_handle_t nic_handle,
  OUT uint32_t *remote_address,
  OUT uint32_t *remote_id)
```

3.5.14.2 Parameters

nic_handle Handle of the NIC associated with the CDM for which the datagram status is being probed.

remote_address

Physical address of the Gemini NIC at the remote peer with a datagram in the completed, timed-out or cancelled state. Valid only if the return value is `GNI_RC_SUCCESS`.

remote_id User specific ID of the remote instance in the upper layer protocol domain with a datagram in the completed, timed-out or cancelled state. Valid only if the return value is `GNI_RC_SUCCESS`.

3.5.14.3 Return Codes

`GNI_RC_SUCCESS`

A datagram in the completed, timed-out or cancelled state was found.

`GNI_RC_INVALID_PARAM`

An invalid *nic_handle*, *remote_addr* or *remote_id* was specified.

`GNI_RC_NO_MATCH`

No datagram in the completed, timed-out or cancelled state was found.

3.5.15 `PostDataProbeById`

The `PostDataProbeById` function returns the *datagram_id* of the first datagram found in completed, timed out, or canceled state for the CDM associated with the input NIC handle. This function should be used for probing for completion of datagrams that were previously posted using the `GNI_EpPostDataWId` function.

This function must be used in conjunction with `GNI_EpPostDataTestById` or `GNI_EpPostDataWaitById` to obtain the data exchanged in the datagram transaction.

3.5.15.1 Synopsis

```
gni_return_t GNI_PostDataProbeById (
  IN gni_nic_handle_t nic_handle,
  OUT uint64_t *datagram_id)
```

3.5.15.2 Parameters

nic_handle Handle of the NIC associated with the CDM for which the datagram status is being probed.

datagram_id

The datagram ID of the first datagram with a datagram ID specified found in the completed, timed-out or cancelled state. Valid only if the return value is `GNI_RC_SUCCESS`.

3.5.15.3 Return Codes

`GNI_RC_SUCCESS`

A datagram in the completed, timed-out or cancelled state was found.

`GNI_RC_INVALID_PARAM`

An invalid *nic_handle*, *datagram_id* was specified.

`GNI_RC_NO_MATCH`

No datagram with the specified ID found in the completed, timed-out or cancelled state.

3.5.16 `PostDataProbeWaitById`

The `PostDataProbeWaitById` function returns the post ID of the first datagram posted with a datagram ID found in completed, timed out, or canceled state for the CDM associated with the input *nic_handle*. This function must be used in conjunction with `gni_ep_postdata_test_by_id` or `gni_ep_postdata_wait_by_id` to obtain data exchanged in the datagram transaction.

3.5.16.1 Synopsis

```
gni_return_t GNI_PostDataProbeWaitById (
 IN gni_nic_handle_t nic_handle,
 IN uint32_t timeout,
 OUT uint64_t *datagram_id )
```

3.5.16.2 Parameters

nic_handle Handle of the NIC associated with the CDM for which the datagram status is being probed.

timeout The count that this function waits, in milliseconds, for transaction to complete. Set to (-1) if no timeout is desired.

datagram_id

The first datagram ID found in the completed, timed-out or cancelled state. Valid only if the return value is `GNI_RC_SUCCESS`.

3.5.16.3 Return Codes

`GNI_RC_SUCCESS`

A datagram with the specified id was found in the completed, timed-out or cancelled state.

`GNI_RC_INVALID_PARAM`

An invalid *nic_handle*, or *timeout* was specified.

`GNI_RC_TIMEOUT`

No datagram with a datagram ID specified and in the completed, timed-out or cancelled state was found before the timeout expired.

`GNI_RC_NO_MATCH`

No datagram with the specified ID found in the completed, timed-out or cancelled state.

3.6 FMA DM

3.6.1 `PostFma`

The `PostFma` function executes a data transaction (PUT, GET, or AMO) by storing into the directly mapped FMA window to initiate a series of FMA requests. It returns before the transaction is confirmed by the remote NIC.

Zero-length FMA PUT operations are supported. Zero-length FMA GET and zero-length FMA AMO operations are not supported

3.6.1.1 Synopsis

```
gni_return_t GNI_PostFma (
  IN gni_ep_handle_t ep_handle,
  IN gni_post_descriptor_t *post_descr)
```

3.6.1.2 Parameters

ep_handle Instance of a local endpoint.

post_descr Pointer to a descriptor to be posted.

3.6.1.3 Return Codes

`GNI_RC_SUCCESS`

The descriptor was successfully posted.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid.

`GNI_RC_ALIGNMENT_ERROR`

The posted source or destination data pointers or data length are not properly aligned. There are no alignment restrictions on PUTs. GETs require 4 byte-alignment. AMOs require 8 byte-alignment, except AAX which requires 16 byte-alignment.

`GNI_RC_ERROR_RESOURCE`

The transaction request failed due to insufficient resources.

3.7 FMA Short Messaging

3.7.1 `SmsgInit`

The `SmsgInit` function configures the short messaging protocol on the given endpoint. Short messaging buffers must be zeroed before calling `SmsgInit`.

3.7.1.1 Synopsis

```
gni_return_t GNI_SmsgInit (
  IN gni_ep_handle_t ep_handle,
  IN gni_smsg_attr_t *local_smsg_attr,
  IN gni_smsg_attr_t *remote_smsg_attr)
```

3.7.1.2 Parameters

ep_handle Instance of an endpoint.

local_smsg_attr

Pointer to a list of local parameters used for short messaging. Parameter values are defined using the `gni_smsg_attr` structure.

remote_smsg_attr

Pointer to a list of remote parameters used for short messaging provided by a peer. Parameter values are defined using the `gni_smsg_attr` structure.

3.7.1.3 Return Codes

`GNI_RC_SUCCESS`

Operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters was invalid.

`GNI_RC_INVALID_STATE`

Endpoint is not bound.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to allocate short message internal structures.

3.7.2 SmsgSend

The `SmsgSend` function sends a message to the remote peer by copying it into the preallocated remote buffer space using the FMA mechanism. It returns before the delivery is confirmed by the remote NIC. When the endpoint is set with the `GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT` flag, the system attempts to retransmit messages when certain transaction failures occur. This `SmsgSend` function is a non-blocking call.

3.7.2.1 Synopsis

```
gni_return_t GNI_SmsgSend (
  IN gni_ep_handle_t ep_handle,
  IN void *header,
  IN uint32_t header_length,
  IN void *data,
  IN uint32_t data_length,
  IN uint32_t *msg_id)
```

3.7.2.2 Parameters

ep_handle An instance of an endpoint.

header A pointer to the header of a message.

header_length

The length of the header in bytes.

data A pointer to the payload of the message.

data_length

The length of the payload in bytes.

msg_id Identifier for application to track transaction. Only valid for short messaging using `GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT` type, otherwise ignored.

3.7.2.3 Return Codes

`GNI_RC_SUCCESS`

The transmission has been initiated.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid or the endpoint is not initialized for short messaging.

`GNI_RC_NOT_DONE`

No credits available to send the message.

`GNI_RC_ERROR_RESOURCE`

The operation failed due to insufficient memory.

3.7.3 `SmsgSendWTag`

The `SmsgSendWTag` function sends a tagged message to the remote peer, by copying it into the pre-allocated remote buffer space using the FMA mechanism. It returns before the delivery is confirmed by the remote NIC. When the endpoint is set with `GNI_SMSG_MBOX_AUTO_RETRANSMIT` type, the system attempts to re-transmit for certain transaction failures. This is a non-blocking call.

3.7.3.1 Synopsis

```
gni_return_t GNI_SmsgSendWTag(
  IN gni_ep_handle_t ep_hndl,
  IN void *header,
  IN uint32_t header_length,
  IN void *data,
  IN uint32_t data_length,
  IN uint32_t msg_id,
  IN uint8_t tag)
```

3.7.3.2 Parameters

ep_hndl An instance of an endpoint.

header A pointer to the header of a message.

header_length

The length of the header in bytes.

data A pointer to the payload of the message.

data_length

The length of the payload in bytes.

msg_id Identifier for application to track transaction. Only valid for short messaging using `GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT` type, otherwise ignored.

tag Tag associated with the short message.

3.7.3.3 Return Codes

`GNI_RC_SUCCESS`

The transmission was initiated.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid or the endpoint is not initialized for short messaging.

`GNI_RC_NOT_DONE`

No credits available to send the message.

`GNI_RC_ERROR_RESOURCE`

The operation failed due to insufficient memory.

3.7.4 SmsgGetNext

The SmsgGetNext function returns a pointer to the header of the newly arrived message and makes this message current. You can set up your application to copy the message out of the mailbox or process it immediately. This is a non-blocking call.

3.7.4.1 Synopsis

```
gni_return_t GNI_SmsgGetNext (
  IN gni_ep_handle_t ep_handle,
  OUT void **header)
```

3.7.4.2 Parameters

ep_handle Instance of an endpoint.

header Pointer to the header of the newly arrived message.

3.7.4.3 Return Codes

GNI_RC_SUCCESS

The new message arrived successfully.

GNI_RC_INVALID_PARAM

The endpoint handle was invalid or the endpoint is not initialized for short messaging.

GNI_RC_NOT_DONE

There are no new messages available.

3.7.5 SmsgGetNextWTag

The SmsgGetNextWTag function returns a pointer to the header of the newly arrived message and makes this message current if the input tag matches the tag of the newly arrived message. An application may decide to copy the message header out of the mailbox or process the header immediately. This is a non-blocking call.

3.7.5.1 Synopsis

```
gni_return_t GNI_SmsgGetNextWTag (
  IN gni_ep_handle_t  ep_hndl,
  OUT void          **header,
  INOUT uint8_t       *tag)
```

3.7.5.2 Parameters

ep_handle Instance of an endpoint.

header Pointer to the header of the newly arrived message.

tag On input, a pointer to the value of the remote event to be matched. A wildcard value of `GNI_SMSG_ANY_TAG` is used to match any tag value of the incoming message. The value is set to that of the matching remote event on output.

3.7.5.3 Return Codes

`GNI_RC_SUCCESS`

The new message arrived successfully.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid or the endpoint is not initialized for short messaging.

`GNI_RC_NOT_DONE`

There are no new messages available.

`GNI_RC_NO_MATCH`

The message is available, but the tag of the message does not match the value supplied in the *tag* argument.

3.7.6 `SmsgRelease`

The `SmsgRelease` function releases the current message buffer. It must be called only after `GetNext` has returned `GNI_RC_SUCCESS`. This is a non-blocking call. The message returned by the `GetNext` function must be copied out or processed prior to making this call.

3.7.6.1 Synopsis

```
gni_return_t GNI_SmsgRelease (
  IN gni_ep_handle_t ep_handle)
```

3.7.6.2 Parameters

ep_handle Instance of an endpoint.

3.7.6.3 Return Codes

`GNI_RC_SUCCESS`

The current message is successfully released.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid or the endpoint is not initialized for short messaging.

`GNI_RC_NOT_DONE`

There is no current message. The `GetNext` function must return `GNI_RC_SUCCESS` before calling this function.

3.8 RDMA (BTE)

3.8.1 `PostRdma`

The `PostRdma` function adds a descriptor to the tail of the RDMA queue and returns immediately.

3.8.1.1 Synopsis

```
gni_return_t GNI_PostRdma (
  IN gni_ep_handle_t ep_handle,
  IN gni_post_descriptor_t *post_descr)
```

3.8.1.2 Parameters

ep_handle Instance of a local endpoint.

post_descr Pointer to the descriptor to be posted to the queue.

3.8.1.3 Return Codes

`GNI_RC_SUCCESS`

The descriptor was successfully posted.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid.

`GNI_RC_ALIGNMENT_ERROR`

Posted source, destination data pointers, or data length are not properly aligned.

GNI_RC_ERROR_RESOURCE

The transaction request could not be posted due to insufficient resources.

GNI_RC_ERROR_NOMEM

Insufficient memory to complete the operation.

GNI_RC_PERMISSION_ERROR

The user's buffer R/W permissions conflict with the access type.

3.9 Completion Queue Processing

3.9.1 CqTestEvent

The CqTestEvent function polls the specified completion queue for a completion entry. If a completion entry is found, it returns GNI_RC_SUCCESS, unless the CQ is overrun (full), in which case it returns GNI_RC_ERROR_RESOURCE. If no completion entry is found, GNI_RC_NOT_DONE is returned. No processing of new entries is performed by this function.

3.9.1.1 Synopsis

```
GNI_CqTestEvent (
  IN gni_cq_handle_t cq_handle)
```

3.9.1.2 Parameters

cq_handle The handle for the completion queue.

3.9.1.3 Return Codes

GNI_RC_SUCCESS

A completion entry was found on the completion queue.

GNI_RC_NOT_DONE

No new completion entries are on the completion queue.

GNI_RC_INVALID_PARAM

The completion queue handle was invalid.

GNI_RC_ERROR_RESOURCE

CQ is in an overrun (full) state and CQ entries may have been lost.

3.9.2 CqGetEvent

The CqGetEvent function returns information about the next event by polling the specified completion queue for a completion entry. If a completion entry is found, it returns the event data stored in the entry. CqGetEvent is a non-blocking call.

It is up to the calling process to subsequently invoke the appropriate function to dequeue the completed descriptor. CqGetEvent only de-queues the completion entry from the completion queue.

3.9.2.1 Synopsis

```
gni_return_t GNI_CqGetEvent (
  IN gni_cq_handle_t cq_handle,
  OUT gni_cq_entry_t *event_data)
```

3.9.2.2 Parameters

cq_handle The handle for the completion queue.

event_data Returns a pointer to a new event entry data if the return status indicates success. If not successful, then nothing is returned by this parameter.

3.9.2.3 Return Codes

GNI_RC_SUCCESS

A completion entry was found on the completion queue.

GNI_RC_NOT_DONE

No new completion entries are on the completion queue.

GNI_RC_INVALID_PARAM

The completion queue handle was invalid.

GNI_RC_ERROR_RESOURCE

The completion queue is in an overrun (full) state and completion queue events may have been lost.

GNI_RC_TRANSACTION_ERROR

A network error was encountered while processing a transaction.

3.9.3 CqWaitEvent

The CqWaitEvent function polls the specified completion queue for a completion entry. If CqWaitEvent finds a completion entry, it immediately returns event data.

If no completion entry is found, the caller is blocked until a completion entry is generated, or until the *timeout* value expires. The completion queue must be created with the GNI_CQ_BLOCKING mode set in order to be able to block on it.

3.9.3.1 Synopsis

```
gni_return_t GNI_CqWaitEvent (
  IN gni_cq_handle_t cq_handle,
  IN uint64_t timeout,
  OUT gni_cq_entry_t *event_data)
```

3.9.3.2 Parameters

cq_handle The handle for the completion queue.

timeout The number of milliseconds to block before returning to the caller; set this to -1 if no timeout is desired.

event_data Returns a pointer to a new event entry data if the return status indicates success. If not successful, then nothing is returned by this parameter.

3.9.3.3 Return Codes

GNI_RC_SUCCESS

A completion entry was found on the completion queue.

GNI_RC_TIMEOUT

The request timed out and no completion entry was found.

GNI_RC_INVALID_PARAM

The completion queue handle was invalid.

GNI_RC_ERROR_RESOURCE

The completion queue was not created in the GNI_CQ_BLOCKING mode.

GNI_RC_TRANSACTION_ERROR

A network error was encountered while processing a transaction.

3.9.4 CqVectorWaitEvent

The CqVectorWaitEvent function polls the specified completion queues for a completion entry. If CqVectorWaitEvent finds a completion entry, it immediately returns event data.

If no completion entry is found, the caller is blocked until a completion entry is generated, or until the *timeout* value expires. The completion queues must be created with the GNI_CQ_BLOCKING mode set in order to be able to block on it.

3.9.4.1 Synopsis

```
gni_return_t GNI_CqVectorWaitEvent (
  IN gni_cq_handle_t *cq_handls,
  IN uint32_t num_cqs
  IN uint64_t timeout,
  OUT gni_cq_entry_t *event_data,
  OUT uint32_t *which)
```

3.9.4.2 Parameters

cq_handls Array of handles for the completion queues.

num_cqs Number of completion queue handles.

timeout The number of milliseconds to block before returning to the caller; set this to -1 if no timeout is desired.

event_data Returns a pointer to a new event entry data if the return status indicates success. If not successful, then nothing is returned by this parameter.

which Returns the index of the CQ within the *cq_handls* array which returned *event_data* on success. Undefined otherwise.

3.9.4.3 Return Codes

GNI_RC_SUCCESS

A completion entry was found on the completion queue.

GNI_RC_TIMEOUT

The request timed out and no completion entry was found.

GNI_RC_INVALID_PARAM

One of the completion queue handles was invalid.

GNI_RC_ERROR_RESOURCE

One of the completion queues was not created in the GNI_CQ_BLOCKING mode.

GNI_RC_TRANSACTION_ERROR

A network error was encountered while processing a transaction.

3.9.5 GetCompleted

The GetCompleted function gets the next completed post descriptor from the specified completion queue. The descriptor is removed from the head of the queue and the address of the descriptor is returned.

A GNI_RC_DESCRIPTOR_ERROR is returned if the transaction has failed and the error was reported by the CqGetEvent function in the *event_data* parameter. In this case, the error information from the event_data is copied to the status field of the descriptor. GetCompleted is a non-blocking call.

3.9.5.1 Synopsis

```
gni_return_t GNI_GetCompleted (
  IN gni_cq_handle_t cq_handle,
  IN gni_cq_entry_t event_data,
  OUT gni_post_descriptor _t **post_descr)
```

3.9.5.2 Parameters

cq_handle Handle for the completion queue.

event_data The event returned by the CqGetEvent function.

post_desc Returns a pointer to the address of the descriptor that has completed.

3.9.5.3 Return Codes

GNI_RC_SUCCESS

A completed descriptor was returned with a successful completion status.

GNI_RC_DESCRIPTOR_ERROR

If the corresponding post queue (FMA, RDMA or AMO) is empty, the descriptor pointer is set to NULL, otherwise, a completed descriptor is returned with an error completion status.

GNI_RC_INVALID_PARAM

The CQ handle was invalid.

GNI_RC_TRANSACTION_ERROR

A completed descriptor was returned with a network error status.

3.9.6 PostCqWrite

The PostCqWrite function executes a CQ write transaction to a remote CQ. It returns before the transaction is confirmed by the remote NIC.

3.9.6.1 Synopsis

```
gni_return_t GNI_PostCqWrite (
  IN gni_ep_handle_t ep_handle,
  IN gni_post_descriptor_t *post_descr)
```

3.9.6.2 Parameters

ep_handle Instance of a local endpoint.

post_descr Pointer to a descriptor to be posted.

3.9.6.3 Return Codes

`GNI_RC_SUCCESS`

The descriptor was successfully posted.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid.

`GNI_RC_RESOURCE_ERROR`

Insufficient were resources available to initialize the endpoint.

3.9.7 `CqErrorStr`

The `CqErrorStr` function decodes the error status encoded in a CQ entry by the hardware.

3.9.7.1 Synopsis

```
gni_return_t GNI_CqErrorStr (
  IN gni_cq_entry_t entry,
  OUT void *buffer,
  IN uint32_t length)
```

3.9.7.2 Parameters

entry CQ entry with error status to decode.

buffer Pointer to the buffer where the error code is returned.

length Length of the buffer in bytes.

3.9.7.3 Return Codes

`GNI_RC_SUCCESS`

The completion queue was successfully destroyed.

GNI_RC_INVALID_PARAM

The cq_handle was invalid.

GNI_RC_SIZE_ERROR

The supplied buffer is too small to contain the error code.

3.9.8 CqErrorRecoverable

The CqErrorRecoverable function translates any error status encoded by the hardware in a completion queue entry into a recoverable or unrecoverable flag for application usage.

3.9.8.1 Synopsis

```
gni_return_t GNI_CqErrorRecoverable (
  IN gni_cq_entry_t cq_handle,
  OUT uint32_t *recoverable)
```

3.9.8.2 Parameters

entry Completion queue entry with error status to be decoded.

recoverable

Pointer to the integer flag that will contain the decoded result.

3.9.8.3 Return Codes

GNI_RC_SUCCESS

The entry was successfully decoded.

GNI_RC_INVALID_PARAM

Invalid input parameter.

GNI_RC_INVALID_STATE

The completion queue entry translates to an undefined state.

3.10 Error Handling

3.10.1 SubscribeErrors

The SubscribeErrors function creates an error event queue. When this function returns, events start reporting immediately. The error mask, *mask*, determines which errors are reported. See gni_error_mask on page 96.

Privileged users, such as superusers, can pass in NULL for *nic_handle* which causes the passed in *device_id* to be used instead. This allows privileged users to subscribe to errors without a CDM being attached. By default, if no *nic_handle* is passed in, then errors are captured for all *ptag*s.

3.10.1.1 Synopsis

```
gni_return_t GNI_SubscribeErrors(
  IN  gni_nic_handle_t  nic_handle,
  IN  uint32_t          device_id,
  IN  gni_error_mask_t  mask,
  IN  uint32_t          EEQ_size,
  OUT gni_err_handle_t *err_handle)
```

3.10.1.2 Parameters

nic_handle The handle of the associated NIC.

device_id The device identifier, for privileged mode (when NULL is passed in for *nic_handle*).

mask The error mask with corresponding bits set for notification.

EEQ_size Size of the EEQ. The queue size uses a default of 64 entries if a value of `0` is passed in.

err_handle This handle is returned to identify the instance in uGNI.

3.10.1.3 Return Codes

`GNI_RC_SUCCESS`

The operation completed successfully.

`GNI_RC_INVALID_PARAM`

One of the input parameters is invalid, or a non-privileged user is trying to subscribe without a communication domain.

`GNI_RC_NO_MATCH`

Specified *device_id* does not exist.

`GNI_RC_ERROR_RESOURCE`

The event queue could not be created due to insufficient resources.

`GNI_RC_ERROR_NOMEM`

Insufficient memory to complete the operation.

3.10.2 ReleaseErrors

The ReleaseErrors function releases the error event notification and cleans up the memory resources for the event queue.

3.10.2.1 Synopsis

```
gni_return_t GNI_ReleaseErrors(
  IN gni_err_handle_t err_handle)
```

3.10.2.2 Parameters

err_handle The handle of the subscribed error events.

3.10.2.3 Return Codes

GNI_RC_SUCCESS

The descriptor was successfully posted.

GNI_RC_INVALID_PARAM

One of the input parameters was invalid.

GNI_RC_NOT_DONE

A thread is still waiting on the event queue.

3.10.3 GetErrorMask

The GetErrorMask function returns the error mask associated with an error handle. The mask determines which error events are delivered. See gni_error_mask on page 96.

3.10.3.1 Synopsis

```
gni_return_t GNI_GetErrorMask(
  IN  gni_err_handle_t  err_handle,
  OUT gni_error_mask_t *mask)
```

3.10.3.2 Parameters

err_handle The handle of the subscribed error events.

mask The pointer to copy the mask value to.

3.10.3.3 Return Codes

`GNI_RC_SUCCESS`

The descriptor was successfully posted.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid.

3.10.4 `SetErrorMask`

The `SetErrorMask` function sets a new error mask for matching events.

3.10.4.1 Synopsis

```
gni_return_t GNI_SetErrorMask(
  IN gni_err_handle_t  err_handle,
  IN gni_error_mask_t  mask_in,
  IN gni_error_mask_t *mask_out)
```

3.10.4.2 Parameters

err_handle The handle of the subscribed error events.

mask_in The error mask with corresponding bits set for notification.

mask_out The pointer to copy the pre-set mask value to.

3.10.4.3 Return Codes

`GNI_RC_SUCCESS`

The descriptor was successfully posted.

`GNI_RC_INVALID_PARAM`

The endpoint handle was invalid.

3.10.5 `GetErrorEvent`

The `GetErrorEvent` function gets an error event, if available.

3.10.5.1 Synopsis

```
gni_return_t GNI_GetErrorEvent(
  IN gni_err_handle_t   err_handle,
  IN gni_error_event_t *event)
```

3.10.5.2 Parameters

err_handle The handle of the subscribed error events.

event The pointer to the buffer to copy the event into.

3.10.5.3 Return Codes

`GNI_RC_SUCCESS`

A completed descriptor was returned with a successful completion status.

`GNI_INVALID_PARAMETER`

The endpoint handle was invalid.

`GNI_RC_NOT_DONE`

No event was found in the event queue.

3.10.6 `WaitErrorEvents`

The `WaitErrorEvents` function blocks waiting forever when waiting for one event to occur. When that one event is triggered, it delays returning to try and coalesce error events. The *timeout* value is specified in number of milliseconds. The number of events copied are stored in the *num_events* structure.

3.10.6.1 Synopsis

```
gni_return_t GNI_WaitErrorEvents(
  IN  gni_err_handle_t    err_handle,
  IN  gni_error_event_t *events,
  IN  uint32_t             events_size,
  IN  uint32_t             timeout,
  OUT uint32_t            *num_events)
```

3.10.6.2 Parameters

err_handle The handle of the subscribed error events.

events The pointer to an array of events structures that will be filled in on a successful return. This pointer must be a valid memory location since the events will be copied from the EEQ.

events_size

The size of the array passed in from the events pointer.

timeout After first event is triggered, time to wait for subsequent events.

num_events The number of events copied into the events buffer.

3.10.6.3 Return Codes

GNI_RC_SUCCESS

The operation completed successfully.

GNI_RC_INVALID_PARAM

One of the input parameters was invalid.

GNI_RC_TIMEOUT

The request timed out and the event array was not filled all the way.

GNI_RC_NOT_DONE

The wait was interrupted by the system.

GNI_RC_PERMISSION_ERROR

The events pointer cannot be written to.

3.10.7 SetErrorPtag

The SetErrorPtag function sets the protection tag for an error handler. This is a privileged operation.

3.10.7.1 Synopsis

```
gni_return_t GNI_SetErrorPtag(
  IN  gni_err_handle_t  err_handle,
  IN  uint8_t           ptag)
```

3.10.7.2 Parameters

err_handle The handle of the subscribed error events.

ptag The protect tag to set for matching error events.

3.10.7.3 Return Codes

GNI_RC_SUCCESS

The descriptor was successfully posted.

GNI_RC_INVALID_PARAM

The endpoint handle was invalid.

GNI_RC_PERMISSION_ERROR

Only the superuser can set *ptag* to something other than the communication domain.

3.11 Other

3.11.1 GetNumLocalDevices

The GetNumLocalDevices function returns the number of NIC devices.

3.11.1.1 Synopsis

```
gni_return_t GNI_GetNumLocalDevices  (
 OUT int *num_devices)
```

3.11.1.2 Parameters

num_devices

Number of NICs on node.

3.11.1.3 Return Codes

GNI_RC_SUCCESS

Number of devices was returned successfully.

GNI_RC_INVALID_PARAM

One or more of the parameters was invalid.

GNI_RC_ERROR_RESOURCE

Gemini support missing from kernel.

3.11.2 GetLocalDeviceIds

The GetLocalDeviceIds function returns an array of local NIC devices.

3.11.2.1 Synopsis

```
gni_return_t GNI_GetLocalDeviceIds (
 IN int len
 OUT int *device_id)
```

3.11.2.2 Parameters

len number of entries in *device_ids*.

device_ids pointer to array of local NIC devices.

3.11.2.3 Return Codes

`GNI_RC_SUCCESS`

Number of devices was returned successfully.

`GNI_RC_INVALID_PARAM`

One or more of the parameters was invalid.

`GNI_RC_ERROR_RESOURCE`

Gemini support missing from kernel.

3.12 Ennumerations

3.12.1 gni_cq_mode

The `gni_cq_mode` enumeration defines the modes of operation to use for the completion queue. The flags from this enumeration are used for the `CqCreate` function *mode* parameter.

3.12.1.1 Synopsis

```
typedef enum gni_cq_mode {
  GNI_CQ_NOBLOCK = 0,
  GNI_CQ_BLOCKING
} gni_cq_mode_t;
```

3.12.1.2 Constants

`GNI_CQ_NOBLOCK`

Indicates that the CQ instance does not need to be configured in the blocking mode.

`GNI_CQ_BLOCKING`

Indicates that the CQ instance should be able to operate in the blocking mode.

3.12.2 gni_fma_cmd_type

The `gni_fma_cmd_type` enumeration defines the commands to use for the FMA. The FMA command is set using the `amo_cmd` member of the `gni_post_descriptor` structure, which is used by the `GNI_PostRdma`, `GNI_PostFma`, and `GNI_GetCompleted` functions.

3.12.2.1 Synopsis

```
typedef enum gni_fma_cmd_type {
  GNI_FMA_GET            = 0x000,
  GNI_FMA_PUT            = 0x100,
  GNI_FMA_PUT_MSG        = 0x110,
  GNI_FMA_ATOMIC_FADD    = 0x008,
  GNI_FMA_ATOMIC_FADD_C  = 0x018,
  GNI_FMA_ATOMIC_FAND    = 0x009,
  GNI_FMA_ATOMIC_FAND_C  = 0x019,
  GNI_FMA_ATOMIC_FOR     = 0x00A,
  GNI_FMA_ATOMIC_FOR_C   = 0x01A,
  GNI_FMA_ATOMIC_FXOR    = 0x00B,
  GNI_FMA_ATOMIC_FXOR_C  = 0x01B,
  GNI_FMA_ATOMIC_FAX     = 0x00C,
  GNI_FMA_ATOMIC_FAX_C   = 0x01C,
  GNI_FMA_ATOMIC_CSWAP   = 0x00D,
  GNI_FMA_ATOMIC_CSWAP_C = 0x01D,
  GNI_FMA_ATOMIC_ADD     = 0x108,
  GNI_FMA_ATOMIC_ADD_C   = 0x118,
  GNI_FMA_ATOMIC_AND     = 0x109,
  GNI_FMA_ATOMIC_AND_C   = 0x119,
  GNI_FMA_ATOMIC_OR      = 0x10A,
  GNI_FMA_ATOMIC_OR_C    = 0x11A,
  GNI_FMA_ATOMIC_XOR     = 0x10B,
  GNI_FMA_ATOMIC_XOR_C   = 0x11B,
  GNI_FMA_ATOMIC_AX      = 0x10C,
  GNI_FMA_ATOMIC_AX_C    = 0x11C,
} gni_fma_cmd_type_t;
```

3.12.2.2 Constants

`GNI_FMA_GET`

Reserved for use by GNI.

`GNI_FMA_PUT`

Reserved for use by GNI.

`GNI_FMA_PUT_MSG`

Reserved for use by GNI.

`GNI_FMA_ATOMIC_FADD`

Indicates an atomic fetch and `ADD` command.

`GNI_FMA_ATOMIC_FADD_C`

Indicates a cached atomic fetch and `ADD` command.

`GNI_FMA_ATOMIC_FAND`

Indicates an atomic fetch and `AND` command.

GNI_FMA_ATOMIC_FAND_C

Indicates a cached atomic fetch and AND command.

GNI_FMA_ATOMIC_FOR

Indicates an atomic fetch and OR command.

GNI_FMA_ATOMIC_FOR_C

Indicates a cached atomic fetch and OR command.

GNI_FMA_ATOMIC_FXOR

Indicates an atomic fetch and XOR command.

GNI_FMA_ATOMIC_FXOR_C

Indicates a cached atomic fetch and XOR command.

GNI_FMA_ATOMIC_FAX

Indicates an atomic fetch, AND and XOR command.

GNI_FMA_ATOMIC_FAX_C

Indicates a cached atomic fetch, AND and XOR command.

GNI_FMA_ATOMIC_CSWAP

Indicates an atomic compare and swap command.

GNI_FMA_ATOMIC_CSWAP_C

Indicates a cached atomic compare and swap command.

GNI_FMA_ATOMIC_ADD

Indicates an atomic ADD command.

GNI_FMA_ATOMIC_ADD_C

Indicates a cached atomic ADD command.

GNI_FMA_ATOMIC_AND

Indicates an atomic AND command.

GNI_FMA_ATOMIC_AND_C

Indicates a cached atomic AND command.

GNI_FMA_ATOMIC_OR

Indicates an atomic OR command.

GNI_FMA_ATOMIC_OR_C

Indicates a cached atomic OR command.

GNI_FMA_ATOMIC_XOR

Indicates an atomic XOR command.

GNI_FMA_ATOMIC_XOR_C

Indicate a cached atomic XOR command.

GNI_FMA_ATOMIC_AX

Indicates an atomic AND and XOR command.

GNI_FMA_ATOMIC_AX_C

Indicates an cached atomic AND and XOR command.

3.12.3 gni_post_state

The gni_post_state enumeration defines the flags for the post state for datagram transactions between the endpoints on a local and a remote peer that are in the same communication domain. A pointer to the post state is returned by the EpPostDataTest and EpPostDataWait functions when testing the success of an EpPostData operation.

3.12.3.1 Synopsis

```
typedef enum gni_post_state{
  GNI_POST_PENDING,
  GNI_POST_COMPLETED,
  GNI_POST_ERROR,
  GNI_POST_TIMEOUT,
  GNI_POST_TERMINATED,
  GNI_POST_REMOTE_DATA
} gni_post_state_t;
```

3.12.3.2 Constants

GNI_POST_PENDING

Indicates the post is pending.

GNI_POST_COMPLETED

Indicates that the data exchange completed successfully.

GNI_POST_ERROR

Indicates the post did not complete due to an error.

GNI_POST_TIMEOUT

Indicates the post did not complete and timed out.

GNI_POST_TERMINATED

Indicates the post did not complete because it was terminated.

GNI_POST_REMOTE_DATA

Indicates receipt of the remote data, but the remote peer did not acknowledge getting the data from the local side.

3.12.4 gni_post_type

The gni_post_type enumeration defines the values to use for the post transaction. The constant values for this enumeration are used by the type member of the gni_post_descriptor structure, which is used by the GNI_PostRdma, GNI_PostFma, and GNI_GetCompleted functions.

3.12.4.1 Synopsis

```
typedef enum gni_post_type {
  GNI_POST_RDMA_PUT = 1,
  GNI_POST_RDMA_GET,
  GNI_POST_FMA_PUT,
  GNI_POST_FMA_PUT_W_SYNCFLAG,
  GNI_POST_FMA_GET,
  GNI_POST_AMO
} gni_post_type_t;
```

3.12.4.2 Constants

GNI_POST_RDMA_PUT

Indicates an RDMA PUT transaction.

GNI_POST_RDMA_GET

Indicates an RDMA GET transaction.

GNI_POST_FMA_PUT

Indicates an FMA PUT transaction.

GNI_POST_FMA_PUT_W_SYNCFLAG

Indicates an FMA PUT transaction with a synchronization flag.

GNI_POST_FMA_GET

Indicates an FMA GET transaction.

GNI_POST_AMO

Indicates an AMO transaction.

3.12.5 gni_return

The gni_return enumeration defines the values to use for return values.

3.12.5.1 Synopsis

```
typedef enum gni_return {
  GNI_RC_SUCCESS = 0,
  GNI_RC_NOT_DONE,
  GNI_RC_INVALID_PARAM,
  GNI_RC_ERROR_RESOURCE,
  GNI_RC_TIMEOUT,
  GNI_RC_PERMISSION_ERROR,
  GNI_RC_DESCRIPTOR_ERROR,
  GNI_RC_ALIGNMENT_ERROR,
  GNI_RC_INVALID_STATE,
  GNI_RC_NO_MATCH,
  GNI_RC_SIZE_ERROR,
  GNI_RC_TRANSACTION_ERROR,
  GNI_RC_ILLEGAL_OP
} gni_return_t;
```

3.12.5.2 Constants

GNI_RC_SUCCESS

The operation was successful.

GNI_RC_NOT_DONE

The operation is not permitted.

GNI_RC_INVALID_PARAM

One or more of the parameters was invalid.

GNI_RC_ERROR_RESOURCE

Typically, this error means there are insufficient resources or the wrong resources available to complete the operation.

GNI_RC_TIMEOUT

The request timed out.

GNI_RC_PERMISSION_ERROR

The process does not have the correct permissions to complete the operation.

GNI_RC_DESCRIPTOR_ERROR

If the corresponding post queue (FMA, RDMA or AMO) is empty, the descriptor pointer is set to NULL, otherwise, a completed descriptor is returned with an error completion status.

GNI_RC_ALIGNMENT_ERROR

Posted source or destination data pointers or data length are not properly aligned.

GNI_RC_INVALID_STATE

The caller attempted to attach a communication domain instance to the Gemini NIC device more than once.

GNI_RC_NO_MATCH

There is no match between the requested item and available items.

GNI_RC_SIZE_ERROR

The supplied buffer is too small to contain the error code.

GNI_RC_TRANSACTION_ERROR

Error in processing post data transaction.

GNI_RC_ILLEGAL_OP

The operation being attempted is illegal.

3.12.6 gni_smsg_type

The gni_smsg_type enumeration defines the values to use for the short messaging type. The constant values for this enumeration are used in the msg_type member of the gni_smsg_attr structure.

3.12.6.1 Synopsis

```
typedef enum gni_smsg_type {
  GNI_SMSG_TYPE_INVALID = 0,
  GNI_SMSG_TYPE_MBOX,
  GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT
} gni_smsg_type_t;
```

3.12.6.2 Constants

`GNI_SMSG_TYPE_INVALID`

Indicates that the short message type is invalid.

`GNI_SMSG_TYPE_MBOX`

Indicates the `MBOX` short messaging type.

`GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT`

Indicates that the system attempts to retransmit the message for certain transaction failures.

3.13 Structures

3.13.1 `gni_error_event`

3.13.1.1 Synopsis

```
typedef struct gni_error_event {
  uint16_t error_code;
  uint8_t  error_category;
  uint8_t  ptag;
  uint32_t serial_number;
  uint64_t timestamp;
  uint64_t info_mmrs[4];
} gni_error_event_t;
```

3.13.1.2 Members

`error_code` Identifies the error which caused the event. Used by GNI for problem reporting. Codes will not be interpreted by uGNI user.

`error_category`

Errors are divided into 6 categories:

- `CRITICAL_ERR`

  Caused by uncorrectable memory errors, an invalid hardware configuration, or other hardware issues. In most cases, future use of the NIC is unreliable and a node reboot may be required.

- `TRANSACTION_ERR`

  Caused by errors in a specific transaction sequence, likely due to a software issue. A node reboot is not required.

- `ADDR_TRANS_ERR`

  There were errors in the node address translation and/or memory address translation for a specific transaction. A node reboot is not required.

- `TRANSIENT_ERR`

  There may be transient issues with network, memory, or resource availability (i.e. no free descriptors). Software should often be able to recover from these errors by reissuing the transaction.

- `CORRECTABLE_MEM_ERR`

  Benign from a system perspective, but should be monitored by HSS and accounted for.

- `INFO_ERR`

  An event occurred which is not necessarily an error condition.

`ptag` PTag responsible for error, when applicable.

`serial_number`

This is a semi-unique identifier for the error. An application can use this to match errors entered into the HSS logs. However, some OS errors come outside the normal error reporting path, so they will have a zero for a serial number.

`timestamp` Time the error was reported.

`info_mmrs` Some errors gather additional information from other registers in the hardware which may be useful information in problem reports. Not used by the uGNI user.

3.13.2 gni_error_mask

The mask value can be a bitwise OR of the error categories as defined by the ERRMASK flags found in gni_pub.h.

3.13.2.1 Synopsis

```
typedef uint8_t gni_error_mask_t;

#define GNI_ERRMASK_CORRECTABLE_MEMORY   (1 << 0)
#define GNI_ERRMASK_CRITICAL             (1 << 1)
#define GNI_ERRMASK_TRANSACTION          (1 << 2)
#define GNI_ERRMASK_ADDRESS_TRANSLATION  (1 << 3)
#define GNI_ERRMASK_TRANSIENT            (1 << 4)
#define GNI_ERRMASK_INFORMATIONAL        (1 << 5)
#define GNI_ERRMASK_DIAG_ONLY            (1 << 6)
```

3.13.3 gni_cq_entry

The event data returned by CqGetEvent, CqWaitEvent, and CqVectorWaitEvent functions is used as input to event processing functions and error decoding functions. gni_pub.h defines event types.

3.13.3.1 Synopsis

```
typedef uint64_t gni_cq_entry_t;

#define GNI_CQ_EVENT_TYPE_POST  0x0ULL
#define GNI_CQ_EVENT_TYPE_SMSG  0x1ULL
#define GNI_CQ_EVENT_TYPE_DMAPP 0x2ULL
#define GNI_CQ_EVENT_TYPE_PRV   0x3ULL
```

3.13.4 gni_job_limits

The gni_job_limits structure defines job parameters and limits. This structure is used by the GNI_ConfigureJob function.

3.13.4.1 Synopsis

```
typedef struct gni_job_limits {
  int32_t  mdd_limit;
  int32_t  mrt_limit;
  int32_t  gart_limit;
  int32_t  fma_limit;
  int32_t  bte_limit;
  int32_t  cq_limit;
  int32_t  ntt_ctrl;
  int32_t  ntt_base;
  int32_t  ntt_size;
} gni_job_limits_t;
```

3.13.4.2 Members

`mdd_limit`	Number of MDDs associated with the given *ptag*.
`mrt_limit`	Number of MRT entries used by MDDs with the given *ptag*.
`gart_limit`	Number of GART entries used by MDDs with the given *ptag*.
`fma_limit`	Number of FMA descriptors associated with the given *ptag*.
`bte_limit`	Number of outstanding BTE descriptors with the given source *ptag*.
`cq_limit`	Number of CQ descriptors associated with the given *ptag*.
`ntt_ctrl`	NTT control flag. The only flag that can be used for this parameter is `GNI_JOB_CTRL_NTT_CLEANUP` which is a directive for the driver to cleanup NTT at the end of the job.
`ntt_base`	Base entry into NTT.
`ntt_size`	Size of the NTT.

3.13.5 gni_mem_segment

The `gni_mem_segment` structure defines the address and length of a memory segment. This structure is used by the `MemRegisterSegments` function.

3.13.5.1 Synopsis

```
typedef struct gni_mem_segment {
  uint64_t        address;
  uint64_t        length;
} gni_mem_segment_t;
```

3.13.5.2 Members

`address`	Address of the segment.
`length`	Size of the segment in bytes.

3.13.6 gni_ntt_descriptor

The `gni_ntt_descriptor` structure defines configuration options that can be set in NTT. This structure is used by the `GNI_CdmCreate` and `GNI_ConfigureNTT` functions.

3.13.6.1 Synopsis

```
typedef struct gni_ntt_descriptor {
  uint32_t        group_size;
  uint8_t         granularity;
  uint32_t        *table;
  uint8_t         flags;
} gni_ntt_descriptor_t;
```

3.13.6.2 Members

`group_size` Size of the NTT group to configure.

`granularity`

NTT granularity.

`table` Pointer to the array of new NTT values.

`flags` Configuration flags.

3.13.7 gni_post_descriptor

The `gni_post_descriptor` structure defines the transaction descriptors. This structure is used by the `GetCompleted`, `PostFMA`, and `PostRDMA` functions.

3.13.7.1 Synopsis

```
typedef struct gni_post_descriptor {
  void                *next_descr;
  void                *prev_descr;
  uint64_t             post_id;
  uint64_t             status;
  uint16_t             cq_mode_complete;
  gni_post_type_t      type;
  uint16_t             cq_mode;
  uint16_t             dlvr_mode;
  uint64_t             local_addr;
  gni_mem_handle_t     local_mem_hndl;
  uint64_t             remote_addr;
  gni_mem_handle_t     remote_mem_hndl;
  uint64_t             length;
  uint16_t             rdma_mode;
  gni_cq_handle_t      src_cq_hndl;
  uint64_t             sync_flag_value;
  uint64_t             sync_flag_addr;
  gni_fma_cmd_type_t   amo_cmd;
  uint64_t             first_operand;
  uint64_t             second_operand;
  uint64_t             cqwrite_value;
} gni_post_descriptor_t;
```

3.13.7.2 Members

`next_descr` Reserved for use by GNI.

`prev_descr` Reserved for use by GNI.

`post_id` Reserved for use by GNI.

`status` Reserved for use by GNI.

`cq_mode_complete`

Reserved for use by GNI.

`type` Required. The type of transaction. The following types are used for this member:

- `GNI_POST_RDMA_PUT`
- `GNI_POST_RDMA_GET`
- `GNI_POST_FMA_PUT`
- `GNI_POST_FMA_PUT_W_SYNCFLAG`
- `GNI_POST_FMA_GET`
- `GNI_POST_AMO`
- `GNI_POST_CQWRITE`

`cq_mode` Required. Instructs the Gemini NIC to generate completion events. Only `GNI_CQMODE_GLOBAL_EVENT` and `GNI_CQMODE_REMOTE_EVENT` can be requested for `FMA_PUT`, `FMA_GET` and `AMO` transactions. The following modes are used for this member:

- `GNI_CQMODE_LOCAL_EVENT`

  Can be used only for BTE transactions, and causes an event to be delivered to the local endpoint's CQ when the local BTE engine has finished handling that descriptor.

- `GNI_CQMODE_GLOBAL_EVENT`

  Can be specified for FMA and BTE transactions, and causes an event to be delivered to the local endpoint's CQ when the data successfully arrives at its destination (either local or remote, depending on the operation).

- `GNI_CQMODE_REMOTE_EVENT`

  Can be used for FMA and BTE transactions, and causes an event to be delivered to the CQ associated with the remote memory registration when the transaction completes.

- `GNI_CQMODE_SILENT`

  Generate no completion events to any associated CQ (local or remote).

- `GNI_CQMODE_DUAL_EVENTS( GNI_CQMODE_LOCAL_EVENT | GNI_CQMODE_GLOBAL_EVENT )`

`dlvr_mode` Required. Applications must reset the delivery mode to zero before using a default mode when adaptive routing and hashing are enabled.

- `GNI_DLVRMODE_PERFORMANCE`
- `GNI_DLVRMODE_NO_ADAPT`
- `GNI_DLVRMODE_NO_HASH`
- `GNI_DLVRMODE_NO_RADAPT`
- `GNI_DLVRMODE_IN_ORDER ( GNI_DLVRMODE_NO_ADAPT | GNI_DLVRMODE_NO_HASH )`

`local_addr` Required. The address of the region on the local node. This is the source for PUT and the target for GET operations. It must be a 4-byte aligned for GET operations and 8-byte aligned for AMOs.

`local_mem_hndl`

The local memory handle. This member is not required for FMA PUT and AMOs with PUT semantics.

`remote_addr`

The address of the remote region. This is the target for PUTs and source for GETs. Must be 4-byte aligned for GET operations and 8-byte aligned for AMOs.

`remote_mem_hndl`

Remote memory handle.

`length` Number of bytes to move. Must be a multiple of 4-bytes for GETs and multiple of 8-bytes for AMOs.

`rdma_mode` There are two modes used for this member:

- `GNI_RDMAMODE_PHYS_ADDR`

  If set, the kernel-level application uses a physical address for the `local_addr` field.

- `GNI_RDMAMODE_FENCE`

  If set, causes the completion processing of the transaction descriptor to be delayed until all network responses, associated with the current descriptor as well as all responses associated with previously processed descriptors of the same BTE channel, have been received. Processing of the next descriptor for the channel does not start until the write-back of the current transmit transaction descriptor is issued.

`src_cq_hndl`

If set, the NIC delivers the source completion events related to this transaction to the specified completion queue instead of the default one.

`sync_flag_value`

Synchronization value.

`sync_flag_addr`

Local to deliver synchronization value.

`amo_cmd` AMO command for the transaction.

`first_operand`

First operand required by the AMO command.

`second_operand`

Second operand required by the AMO command.

`cqwrite_value`

Value to use for a CQ write. Only six least significant bytes is available to software.

3.13.8 gni_smsg_attr

The `gni_smsg_attr` structure defines the attributes for short messaging. This structure is used by the `SmsgInit` function.

3.13.8.1 Synopsis

```
typedef struct gni_smsg_attr {
  gni_smsg_type_t         msg_type;
  void                    *msg_buffer;
  uint32_t                buff_size;
  gni_mem_handle_t        mem_hndl;
  uint32_t                mbox_offset;
  uint32_t                mbox_maxcredit;
  uint32_t                msg_maxsize;
} gni_smsg_attr_t;
```

3.13.8.2 Members

`msg_type` The type of short message buffering method to use. This member uses the following message types:

- `GNI_SMSG_TYPE_MBOX`
- `GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT`

For both of these types, the buffer space for incoming messages is associated with a single remote endpoint. The `GNI_SMSG_TYPE_MBOX_AUTO_RETRANSMIT` type supports automatic retransmission of short messages by the GNI library in the event of transient network faults.

`msg_buffer` A pointer to the beginning of the memory region used for message buffers. Individual message buffers may be associated with different endpoints.

`buff_size` Size of the message buffer in bytes for this endpoint.

`mem_hndl` Memory handle for the memory region used for message buffers.

`mbox_offset`

Offset from `msg_buffer` in bytes indicating the base address for the message buffer associated with this endpoint.

`mbox_maxcredit`

The maximum number of messages that can be buffered in the message buffer.

`msg_maxsize`

The maximum size of the short message which can be received for this endpoint.

3.13.9 gni_smsg_handle

The `gni_smsg_handle` structure is reserved for use by the GNI infrastructure.

Part II: The DMAPP API

About the DMAPP API [4]

DMAPP is a communication library which supports a logically shared, distributed memory (DM) programming model. DMAPP provides remote memory access (RMA) between processes within a job in a one-sided manner. One-sided remote memory access requests require no active participation by the process at the remote node; synchronization functions may be used to determine when side-effects of locally initiated requests are available.

DMAPP is typically not used directly within user application software. The DMAPP API allows one-sided communication libraries (such as Cray SHMEM), and PGAS compilers (such as Coarray Fortran and UPC), implemented on top of DMAPP, to realize much of the hardware performance of the Cray Gemini based system interconnection network while being reasonably portable to its successors.

4.1 DMAPP Programming Model

Cray has supported various forms of logically shared, distributed memory (DM) programming models since the introduction of the Cray T3D. In this model, a group of processes typically run the same executable in parallel. For purposes of this discussion, such a group of related processes is termed a *job*. Although each process in a job executes in its own address space, it can access certain memory segments of other processes in the same job. This parallel model is sometimes referred to as *Single Program Multiple Data* (SPMD).

DMAPP supports SPMD style parallel jobs, not *Multiple Program Multiple Data* (MPMD) parallel jobs. Although multiple DMAPP applications can be launched together via the ALPS `aprun` command, data exchange between processes running different applications must use some other communication paradigm.

Usually the number of processes executing the application does not change over the course of a job. Processes are sometimes termed PEs (processing elements). Although each PE executes in its own address space, it can access certain memory segments of other PEs in a one-sided (PUT/GET) manner using PGAS compiler constructs or by invoking function calls to libraries (such as Cray SHMEM) supporting one-sided programming models.

4.2 DMAPP Applications and Fork

The behavior of DM applications with respect to fork depends on which application memory segments are selected for export at link time. Fork should be avoided if the application's stack and/or local heap are exported. The static data segment is shared between a PE and any forked children if this segment is exported. The symmetric heap is shared between PEs and any forked children.

Other DMAPP resources are also shared between a PE and any forked children. Child processes are not new PEs in the DM job. As with multi-threaded PEs, the application is responsible for setting up mutual exclusion regions around DMAPP calls.

4.3 DMAPP Applications and Threads

PEs within a DM application can be multi-threaded. However, unless the user specifies a concurrency level larger than one during DMAPP initialization, none of the functions in the DMAPP API should be considered thread-safe. Even if a concurrency level larger than one is specified, none of the functions in the DMAPP API are reentrant. For instance, they cannot be called from within a signal handler.

4.4 DMAPP Applications and File Descriptors

Handling of file descriptors in a DM application on Gemini is similar to that on Cray X2. Each PE maintains its own private file descriptors.

4.5 DMAPP Application Intra-node Communication

Data exchange between PEs on a node use the Gemini network interface. It is up to the DM model implementation to optimize for intra-node communication, if desirable.

4.6 Compiling and Launching DMAPP Applications

DMAPP applications must be linked using Cray-supplied compiler/linker scripts.

DM applications must be launched using ALPS `aprun` command.

To create and run a statically linked executable, which uses less than 2GB memory per PE, execute the following commands:

```
cc -o dmapp_put.x dmapp_put.c
export XT_SYMMETRIC_HEAP_SIZE=2600M
aprun -n 192 -N 8 ./dmapp_put.x
```

See `dmapp_put.c` on page 193.

4.7 Resiliency

DMAPP does not support error recovery in the presence of link failures. It is up to the application to deal with such error, if so desired.

4.8 DMAPP Remote Memory Access

Remotely accessible memory segments in a PE can be classified as either *symmetric* or *non-symmetric*.

The address of an object within a *symmetric* memory segment of a PE[X] has a known relationship to the address of this same object in the address space of another PE[Y] in the same job. Objects within these symmetric memory segments on PE[X] can be accessed in a one-sided manner by PE[Y] using address information generated locally on PE[Y].

Objects within *non-symmetric* memory segments on PE[X], can only be accessed in a one-sided manner by a second PE[Y], using address information generated by PE[X] and communicated to PE[Y].

The DMAPP implementation on the Cray XE itself does not guarantee symmetry of the symmetric heap. It is up to the DM model implementation to guarantee the symmetry of the symmetric heap or any other symmetric regions other than the statically linked data segment.

For most DM model implementations, symmetric regions are the statically linked data segment and a symmetric heap segment.

Preparing memory segments of a DM application for remote memory access is handled by DMAPP startup code. Segments which may be exported include the static data segment. The symmetric heap is always exported. At runtime, application software can determine which segments of the address space are exported using query functions. See `dmapp_get_jobinfo` on page 122.

Each exported memory segment has an associated `dmapp_seg_desc_t`. See `dmapp_seg_desc` on page 118.

You should be aware that there are trade-offs in requesting that various program segments be exported.

Since the AMD64 processor cannot be used effectively to directly load/store from exported memory on remote nodes, DMAPP provides an API for interfacing to the remote memory access (RMA) hardware mechanisms. The DMAPP RMA functions can be divided into the following categories:

- One-sided RMA functions
- RMA synchronization functions

All one-sided RMA functions (PUT type, GET type and atomic memory operations) belong to one of the following three categories:

blocking (no suffix)

The process returns from the function only after the side-effects of the remote memory access are globally visible in the system.

non-blocking (`_nb` suffix)

A synchronization ID (*syncid*) is returned to the process. The effects of the remote memory access are only assured to be globally visible in the system after the application has determined via a synchronization call (`dmapp_syncid_test` or `dmapp_syncid_wait`) whether the *syncid* has been retired.

non-blocking implicit (`_nbi` suffix)

No synchronization ID is returned to the process, the effects of the remote memory access are only assured to be globally visible in the system following a call to `dmapp_gsync_test` or `dmapp_gsync_wait`. For performance reasons, this mode is recommended for applications with many small messages, where blocking calls or using individual *syncids* would be expensive.

One-sided remote memory access requests require no active participation by PEs at the remote node. Remote memory segments which are targets of operations with put semantics or sources of operations with get semantics must have been exported at job startup.

The maximum number of concurrent, non-blocking requests allowable can be set by the application at DMAPP initialization. If the application attempts to initiate more non-blocking requests than this maximum, DMAPP returns an error.

There are no ordering guarantees as to completion of different non-blocking RMA requests initiated by a PE.

4.9 DMAPP API

DMAPP provides a C interface for applications. Most DMAPP functions return a status value indicating success or failure of the call. In the case of non-blocking RMA functions, this status does not indicate whether or not the remote memory access request completed successfully; it simply indicates whether the transfer request was initiated successfully. See Chapter 5, DMAPP API Reference on page 117.

4.9.1 Initialization and Query Functions

Before using any other DMAPP functions, an application must call `dmapp_init`to request and initialize resources. See `dmapp_init` on page 121.

After the last call to any other DMAPP functions, an application must call `dmapp_finalize` to return resources. See `dmapp_finalize` on page 121.

The query function `dmapp_get_jobinfo` returns general information about the job, such as the DMAPP version, number of PEs in the job, and symmetric heap and data segment locations. See `dmapp_get_jobinfo` on page 122.

A process can set RMA attributes to control the way that DMAPP handles various RMA requests. Some attributes can be set only during initialization. They will be referred to as static attributes. Others can be set multiple times over the course of the job, and will be referred to as dynamic attributes. Setting dynamic attributes does not affect RMA requests previously issued by the PE, only subsequent RMA requests. Dynamic attributes include when to switch from CPU-based mechanisms for handling RMA requests to using CPU offload mechanisms. See `dmapp_set_rma_attrs` on page 123 and `dmapp_get_rma_attrs` on page 122.

4.9.2 One-sided RMA Functions

RMA functions share some common arguments. For non-blocking explicit functions, the `syncid` argument supplies a pointer to a location in local memory which will be used by DMAPP for storing synchronization related information.

The remote address for either PUT or GET style operations is specified by a virtual address, a segment descriptor, and the remote PE.

When the virtual address is generated locally by the initiating PE, as is the case when working with symmetric data objects, the target segment descriptor supplied in the `job_info` structure returned by `dmapp_get_job_info` may be used. If the virtual address was obtained from the remote PE via some external pointer-passing mechanism, the segment descriptor from the remote PE must be used.

In addition to some common arguments, each of the data motion functions can operate on 1, 4 (DWORD), 8 (QWORD), or 16 (DQWORD) byte data types. Hardware will work most efficiently with requests that are at least DWORD aligned.

4.9.2.1 Contiguous Functions

The PUT functions store a contiguous block of data, starting at local memory address *source_addr*, into a contiguous block at a remote address.

For more detailed information, see `dmapp_put_nb` on page 123, `dmapp_put_nbi` on page 125, and `dmapp_put` on page 126.

The GET functions load a contiguous block of data starting from a remote source address to a contiguous block starting at local memory address *target_addr*. Note that zero-length GETs are not supported.

For both PUT and GET functions, the remote address is specified by the triplet consisting of a virtual address, segment descriptor, and processor. The *nelems* parameter specifies the number of elements of type *type* to transfer. The memory region described by the remote address and *nelems* must reside in an exported memory of the remote PE.

For further information, see `dmapp_get_nb` on page 127, `dmapp_get_nbi` on page 128, and `dmapp_get` on page 129.

4.9.2.2 Strided Functions

The strided PUT functions deliver data starting at a local memory address, using a specified stride, to a remote target address, using a separately specified stride.

The strided GET functions load data starting from a remote memory address using a specified stride and copy the data to a local memory address using a separately specified stride. For more information, see `dmapp_iget_nb` on page 134, `dmapp_iget_nbi` on page 135, and `dmapp_iget` on page 136.

For both strided PUT and GET functions, the remote address is specified by the triplet consisting of a virtual address, segment descriptor, and PE. The remote memory region described by the remote address, stride, and the number of elements to transfer must reside in an exported segment of remote memory.

For more information, see `dmapp_iput_nb` on page 130, `dmapp_iput_nbi` on page 131, and `dmapp_iput` on page 132.

4.9.2.3 Scatter/Gather Functions

The scatter functions PUT elements of a contiguous block of data in local memory to a remote memory location using multiple offset values.

The remote address for each element is specified by the triplet consisting of the virtual address, segment descriptor and target process, plus an offset value.

The memory region defined by the remote address, the largest offset in the array, and the number of elements transferred must be within an exported memory segment of the remote memory.

These functions are also referred to as *indexed PUT functions* and begin with the string `dmapp_ixput`. For more detail, see `dmapp_ixput_nb` on page 137, `dmapp_ixput_nbi` on page 139 , and `dmapp_ixput` on page 140.

The gather functions GET separate elements from remote memory locations placed at various offsets, to contiguous local memory. The remote address for each element is specified by the triplet consisting of the virtual address, segment descriptor and remote process, plus an offset value.

The memory region defined by the remote address, the largest offset in the array, and the number of elements transferred must be within an exported memory segment of the remote memory.

The Indexed GET functions are `dmapp_ixget_nb` on page 141, `dmapp_ixget_nbi` on page 142, `dmapp_ixget` on page 143.

4.9.2.4 PE-strided Functions

The following functions provide PUT (broadcast), GATHER, and SCATTER to remotely accessible addresses across a set of PEs in a DMAPP job. Note that none of these are collective operations. These routines are best used when a small amount of data needs to be broadcast, scattered to, or collected from a set of PEs.

The PUT (broadcast) functions with indexed PE stride deliver data from local memory to the remote memory of multiple PEs within a DMAPP job. When the transfer is complete, each remote PE will have a copy of the contents of the original source buffer. See `dmapp_put_ixpe_nb` on page 145, `dmapp_put_ixpe` on page 147, `dmapp_put_ixpe_nbi` on page 146.

The GATHER functions with indexed PE stride gather data from the remote memory of multiple PEs within a DMAPP job, and concatenate it in local memory. The remote address ranges must be exported for each PE and the remote addresses must be symmetric. See `dmapp_gather_ixpe_nb` on page 152, `dmapp_gather_ixpe_nbi` on page 154, and `dmapp_gather_ixpe` on page 155.

The SCATTER functions with indexed PE stride deliver data starting at an address in local memory to multiple remote PEs within a DMAPP job. Each target PE receives a different portion of the local source data. See `dmapp_scatter_ixpe_nb` on page 148, `dmapp_scatter_ixpe_nbi` on page 150, and `dmapp_scatter_ixpe` on page 151.

For all PE-strided functions, the remote address must be a symmetric address; it must lie in the statically linked data segment or the symmetric heap.

4.9.2.5 DMAPP AMO Functions

In addition to PUT and GET RMA functionality, DMAPP also provides support for using atomic memory operation (AMO) RMA requests. The set of AMO functions are modeled on the set provided on the Cray X2 systems. AMOs can be used both for synchronization and at-memory style operations. AMOs are restricted to operating on 8-byte (also referred to as *qword*, *qw*, or *quad word*) data types, located in a remote PE.

As with RMA functions, the remote memory location must reside in an exported memory segment of remote PE.

Table 1. AMO Instructions Supported by Gemini

Command	Description	Data Returned in Response
AFADD	Atomic Fetch and ADD	yes
AFAX	Atomic Fetch and XOR	yes
ACSWAP	Compare and swap	yes
AADD	Atomic ADD	no
AFAND	Atomic fetch and AND	yes
AFOR	Atomic fetch and OR	yes
AFXOR	Atomic fetch and XOR	yes
AAND	Atomic AND	no
AOR	Atomic OR	no
AXOR	Atomic XOR	no

The scalar-type blocking and non-blocking Atomic Memory Operations (AMO) functions where no result is returned are named `dmapp_`*op*`_qw`, `dmapp_`*op*`_qw_nb`, `dmapp_`*op*`_qw_nbi`, where *op* is either `AADD`, `AAND`, `AOR`, or `AXOR`. For more detail on all AMO functions, see `dmapp_aadd_qw_nb` on page 156 through `dmapp_acswap_qw` on page 185.

The scalar-type blocking and non-blocking Atomic Memory Operations (AMO) functions in which the result is returned in the response are named `dmapp_`*op*`_qw`, `dmapp_`*op*`_qw_nb`, `dmapp_`*op*`_qw_nbi`, where *op* is either `ACSWAP`, `AFADD`, `AFAND`, `AFAX`, `AFOR`, or `AFXOR`.

4.9.2.6 DMAPP Synchronization Functions

DMAPP applications use synchronization functions to determine when locally initiated, non-blocking RMA requests have completed.

A process can determine when the effects of a non-blocking **explicit** RMA function call are globally visible in the system by using `dmapp_sync_test()` or `dmapp_sync_wait()`, both of which return information about the specified `syncid`.

`dmapp_sync_test()` immediately returns a value indicating whether all RMA requests associated with `syncid` have completed. `dmapp_sync_wait()`, only returns after all RMA requests associated with `syncid` have completed. See `dmapp_syncid_test` on page 186, and `dmapp_syncid_wait` on page 187.

A process can determine when the side effects of one or more non-blocking **implicit** RMA function calls are globally visible in the system by using `dmapp_gsync_test()` or `dmap_gsync_wait()` functions, both of which refer to all remote memory accesses associated with previously issued non-blocking implicit RMA requests; therefore, a `syncid` is not relevant.

`dmapp_gsync_test()` immediately returns with a value indicating whether all remote memory accesses associated with previously issued non-blocking implicit RMA function calls are globally visible in the system. `dmap_gsync_wait()` only returns after all non-blocking implicit RMA requests are globally visible in the system. See `dmapp_gsync_test` on page 188 and `dmapp_gsync_wait` on page 188.

4.9.3 Symmetric Heap Functions

DMAPP provides routines for allocating and releasing symmetric heap memory. The DMAPP application is responsible for preserving symmetry of this heap memory. This is achieved by ensuring that all PEs in a job make the same calls to the symmetric heap management functions in the same sequence, involving the same amount of memory. The DMAPP application controls the size of the symmetric heap at startup.

```
void *dmapp_sheap_malloc(IN size_t size)
```

This function allocates *size* bytes memory from the symmetric heap. Equality of addresses across PEs is not guaranteed.

```
void *dmapp_sheap_realloc(IN void *ptr,IN size_t size)
```

This function changes the size of the block to which *ptr* points to the *size* (in bytes) specified by *size*. Equality of addresses across PEs is not guaranteed.

```
void dmapp_sheap_free(IN void *ptr)
```

This function frees a block of memory previously allocated by `dmapp_sheap_malloc` or `dmapp_sheap_realloc`.

DMAPP API Reference [5]

This chapter contains reference information for enumerations, structures, and functions contained in the DMAPP API. Your application must include the `dmapp.h` file when using this API.

5.1 DMAPP Enumerations

5.1.1 `dmapp_type`

The `dmapp_type_t` enumeration defines the valid types supplied by the type input parameter to all data motion functions.

5.1.1.1 Synopsis

```
typedef enum dmapp_type {
  DMAPP_DQW = 0,
  DMAPP_QW,
  DMAPP_DW,
  DMAPP_BYTE
} dmapp_type_t;
```

5.1.1.2 Constants

`DMAPP_DQW` Indicates a double quad (16 byte) word.

`DMAPP_QW` Indicates a quad (8 byte) word.

`DMAPP_DW` Indicates a double (4 byte) word.

`DMAPP_BYTE` Indicates a byte. This option does not provide good performance.

5.1.2 `dmapp_routing_type`

The `dmapp_routing_type_t` enumeration defines the valid routing modes to be supplied to the relaxed_ordering fields of the RMA attributes structure`dmapp_rma_attrs_t`.

```
typedef enum uint8_t {
        DMAPP_ROUTING_IN_ORDER = 0,  /* hash off, adapt off */
        DMAPP_ROUTING_DETERMINISTIC, /* hash on,  adapt off */
        DMAPP_ROUTING_ADAPTIVE       /* hash off, adapt on  */
} dmapp_routing_type_t;
```

Note that DMAPP_ROUTING_IN_ORDER enforces the strictest routing method and is not recommended when performance is desirable.

5.2 DMAPP Structures

5.2.1 dmapp_seg_desc

The dmapp_seg_desc structure is a memory segment descriptor, with an address and length.

5.2.1.1 Synopsis

```
typedef struct dmapp_seg_desc {
  void             *addr;
  size_t           len;
  gni_mem_handle_t memhndl;
  uint16_t         flags;
  void             *reserved;
} dmapp_seg_desc_t;
```

5.2.1.2 Members

addr	A pointer to the address for the memory segment.
len	The currently mapped size of the segment, in bytes.
memhndl	Memory handle for the region; automatically obtained at initialization or obtained from a previous call to MemRegister.
flags	For internal use only.
reserved	For internal use only.

5.2.2 dmapp_jobinfo

The dmapp_jobinfo structure contains general information relevant to the job.

5.2.2.1 Synopsis

```
typedef struct dmapp_jobinfo {
  int              version;
  int              hw_version;
  int              npes;
  dmapp_pe_t       pe;
  dmapp_seg_desc_t data_seg;
  dmapp_seg_desc_t sheap_seg;
} dmapp_jobinfo_t;
```

5.2.2.2 Members

`version` The version of DMAPP that this job uses.

`hw_version` The hardware version of the system. The current version is `DMAPP_GNI_HW_MAJOR_GEMINI`.

`npes` The number of processing elements in use for the entire job.

`pe` The processing element number, in `[0, npes-1]`.

`data_seg` The data segment in memory that this job is using.

`sheap_seg` The symmetric heap memory that this job is using.

5.2.3 dmapp_rma_attrs

The `dmapp_rma_attrs` structure sets RMA attributes to control the way in which DMAPP handles various RMA requests. Some attributes can be set during initialization only, others can be set multiple times over the course of a job.

5.2.3.1 Synopsis

```
typedef struct dmapp_rma_attrs {
  uint32_t max_outstanding_nb;
  uint32_t offload_threshold;
  uint8_t put_relaxed_ordering;
  uint8_t get_relaxed_ordering;
  uint8_t max_concurrency;
} dmapp_rma_attrs_t;
```

5.2.3.2 Members

`max_outstanding_nb`

The maximum number of outstanding non-blocking requests supported. You can only specify this flag during initialization. The following is the range of valid values to be supplied:

`[DMAPP_MIN_OUTSTANDING_NB, .., DMAPP_MAX_OUTSTANDING_NB]`

Setting the value to one of the extremes may lead to a slowdown. The recommended value is `DMAPP_DEF_OUTSTANDING_NB`. Users can experiment with the value to find the optimal setting for their application.

`offload_threshold`

The threshold, in bytes, for switching between CPU-based mechanisms and CPU off-load mechanisms. This value can be specified at any time and can use any value. The default setting is `DMAPP_OFFLOAD_THRESHOLD`. Very small of very large settings may lead to suboptimal performance. The default value is 4k bytes. In order register memory in the MRT, allocate huge pages. Consider how to best set this threshold. While a threshold increase may increase CPU availability, it may also increase transfer latency due to BTE involvement.

put_relaxed_ordering

Specifies the type of routing to be used. Applies to RMA requests with PUT semantics and all AMOs. The default is `DMAPP_ROUTING_DETERMINISTIC`. The value can be specified at any time. Note that `DMAPP_ROUTING_IN_ORDER` may result in poor performance. Valid settings are:

- `DMAPP_ROUTING_IN_ORDER`
- `DMAPP_ROUTING_DETERMINISTIC`
- `DMAPP_ROUTING_ADAPTIVE`

`get_relaxed_ordering`

Specifies the type of routing to be used. Applies to RMA requests with GET semantics. The default is `DMAPP_ROUTING_ADAPTIVE`. The value can be specified at any time. Note that `DMAPP_ROUTING_IN_ORDER` may result in poor performance. Valid settings are:

- `DMAPP_ROUTING_IN_ORDER`
- `DMAPP_ROUTING_DETERMINISTIC`
- `DMAPP_ROUTING_ADAPTIVE`

`max_concurrency`

The maximum number of threads that can access DMAPP. You can only use this when thread-safety is enabled. The default is 1. You can only specify this during initialization and it must be `>= 1`.

5.2.4 `dmapp_syncid`

The `dmapp_syncid` structure contains a pointer to the synchronization ID that is used by a non-blocking explicit RMA function.

5.2.4.1 Synopsis

```
typedef struct dmapp_syncid *dmapp_syncid_handle_t;
```

5.3 DMAPP Functions

5.3.1 dmapp_init

The dmapp_init function initializes resources for a DMAPP job. All DMAPP applications must call this function before using other DMAPP functions. In a threaded application, this function should only be called once.

After the last call to any other DMAPP functions, an application must call a finalization function: dmapp_return_t dmapp_finalize(void).

5.3.1.1 Synopsis

```
dmapp_return_t dmapp_init(
  IN  dmapp_rma_attrs_t *requested_attrs,
  OUT dmapp_rma_attrs_t *actual_attrs);
```

5.3.1.2 Parameters

requested_attrs

Pointer to the desired job attributes. See dmapp_rma_attrs on page 119.

actual_attrs

The actual job attributes.

5.3.1.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more parameters are invalid.

DMAPP_RC_RESOURCE_ERROR

An error occurred during initialization.

5.3.2 dmapp_finalize

The dmapp_finalize function synchronizes and cleans up DMAPP resources. All DMAPP applications must call this function when it has finished using all DMAPP functions.

5.3.2.1 Synopsis

```
dmapp_return_t dmapp_finalize(void);
```

5.3.2.2 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

5.3.3 dmapp_get_jobinfo

The dmapp_get_jobinfo function returns a pointer to the data structure dmapp_jobinfo on page 118 which contains general information about the job.

5.3.3.1 Synopsis

```
dmapp_return_t dmapp_get_jobinfo(
  OUT dmapp_jobinfo_t *info);
```

5.3.3.2 Parameters

info Returns a pointer to the current information about the job.

5.3.3.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

The input parameter is invalid.

5.3.4 dmapp_get_rma_attrs

The dmapp_get_rma_attrs function returns RMA attributes of a DMAPP job.

5.3.4.1 Synopsis

```
dmapp_return_t dmapp_get_rma_attrs(
  OUT dmapp_rma_attrs_t *attrs);
```

5.3.4.2 Parameters

attrs Current RMA attributes of the job.

5.3.4.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

Input parameter is invalid.

5.3.5 dmapp_set_rma_attrs

The `dmapp_set_rma_attrs` function sets dynamic RMA attributes for a DMAPP job.

5.3.5.1 Synopsis

```
dmapp_return_t dmapp_set_rma_attrs(
  IN  dmapp_rma_attrs_t *requested_attrs,
  OUT dmapp_rma_attrs_t *actual_attrs);
```

5.3.5.2 Parameters

requested_attrs

Pointer to desired job attributes.

actual_attrs

Returns a pointer to the actual job attributes.

5.3.5.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

5.3.6 dmapp_put_nb

The `dmapp_put_nb` function is a non-blocking explicit PUT. `dmapp_put_nb` stores a contiguous block of data, starting at the address indicated by *source_addr*, from local memory into a contiguous block at a remote address. The remote address is specified by the triplet virtual address *target_addr*, segment descriptor *target_seg*, and the target process *target_pe*. *nelems* specifies the number of elements of type *type* to be transferred. The memory region defined by *target_addr* and *nelems* must be within an exported memory segment of *target_pe*.

5.3.6.1 Synopsis

```
dmapp_return_t dmapp_put_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  void                  *source_addr,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.6.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe The target processing element.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type The type of elements to transfer.

syncid Pointer to the synchronization ID.

5.3.6.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.7 dmapp_put_nbi

The dmapp_put_nbi function is a non-blocking implicit PUT. dmapp_put_nbi stores a contiguous block of data, starting at the address indicated by *source_addr*, from local memory into a contiguous block at a remote address. The remote address is specified by the triplet virtual address *target_addr*, segment descriptor *target_seg*, and the target process *target_pe*. *nelems* specifies the number of elements of type *type* to be transferred. The memory region defined by *target_addr* and *nelems* must be within an exported memory segment of *target_pe*.

5.3.7.1 Synopsis

```
dmapp_return_t dmapp_put_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  void             *source_addr,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.7.2 Parameters

target_addr

Pointer to the address of a target buffer.

target_seg Pointer to a segment descriptor of a target buffer.

target_pe The target processing element.

source_addr

Address of the source buffer.

nelems The number of elements to transfer.

type The type of elements to transfer.

5.3.7.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_RESOURCE_ERROR

A resource error occurred.

5.3.8 dmapp_put

The dmapp_put function is a blocking PUT. dmapp_put stores a contiguous block of data, starting at the address indicated by *source_addr*, from local memory into a contiguous block at a remote address. The remote address is specified by the triplet virtual address *target_addr*, segment descriptor *target_seg*, and the target process *target_pe*. *nelems* specifies the number of elements of type *type* to be transferred. The memory region defined by *target_addr* and *nelems* must be within an exported memory segment of *target_pe*.

5.3.8.1 Synopsis

```
dmapp_return_t dmapp_put(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  void             *source_addr,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.8.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe The target processing element.

source_addr

Pointer to the address of the source buffer.

nelems The number of elements to transfer.

type The type of elements to transfer.

5.3.8.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_RESOURCE_ERROR

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.9 `dmapp_get_nb`

The `dmapp_get_nb` function is a non-blocking explicit GET. `dmapp_get_nb` loads from a contiguous block of data starting from a remote source address and returning the data into a contiguous block starting at address *target_addr* in local memory. The remote address is specified by the triplet virtual address *source_addr*, segment descriptor *source_seg* and source process *source_pe*. The *nelems* parameter specifies the number of elements of type *type* to transfer. The memory region described by the remote address and *nelems* must reside in an exported memory of *source_pe*. Note that zero-length GETs are not supported.

5.3.9.1 Synopsis

```
dmapp_return_t dmapp_get_nb(
  IN  void                    *target_addr,
  IN  void                    *source_addr,
  IN  dmapp_seg_desc_t        *source_seg,
  IN  dmapp_pe_t              source_pe,
  IN  uint64_t                nelems,
  IN  dmapp_type_t            type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.9.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to a segment descriptor of a source buffer.

source_pe The source processing element.

nelems The number of elements to transfer.

type The type of elements to transfer.

syncid Pointer to a synchronization ID.

5.3.9.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.10 `dmapp_get_nbi`

The `dmapp_get_nbi` function is a non-blocking implicit GET. `dmapp_get_nbi` loads from a contiguous block of data starting from a remote source address and returning the data into a contiguous block starting at address *target_addr* in local memory. The remote address is specified by the triplet virtual address *source_addr*, segment descriptor *source_seg* and source process *source_pe*. The *nelems* parameter specifies the number of elements of type *type* to transfer. The memory region described by the remote address and *nelems* must reside in an exported memory of *source_pe*. Note that zero-length GETs are not supported.

5.3.10.1 Synopsis

```
dmapp_return_t dmapp_get_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.10.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe The source processing element.

nelems The number of elements to transfer.

type The type of elements to transfer.

5.3.10.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

5.3.11 `dmapp_get`

The `dmapp_get` function is a blocking GET. `dmapp_get` loads from a contiguous block of data starting from a remote source address and returning the data into a contiguous block starting at address *target_addr* in local memory. The remote address is specified by the triplet virtual address *source_addr*, segment descriptor *source_seg* and source process *source_pe*. The *nelems* parameter specifies the number of elements of type *type* to transfer. The memory region described by the remote address and *nelems* must reside in an exported memory of *source_pe*. Note that zero-length GETs are not supported.

5.3.11.1 Synopsis

```
dmapp_return_t dmapp_get_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.11.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe The source processing element.

nelems The number of elements to transfer.

type The type of elements to transfer.

5.3.11.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.12 dmapp_iput_nb

The `dmapp_iput_nb` function is a non-blocking explicit strided PUT.

5.3.12.1 Synopsis

```
dmapp_return_t dmapp_iput_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  void                  *source_addr,
  IN  ptrdiff_t             tst,
  IN  ptrdiff_t             sst,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.12.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of a target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tst Target stride (>= 1).

sst Source stride (>= 1).

nelems Number of elements to transfer.

type Type of elements to transfer.

syncid Returns the synchronization ID.

5.3.12.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.13 `dmapp_iput_nbi`

The `dmapp_iput_nbi` function is a non-blocking implicit strided PUT.

5.3.13.1 Synopsis

```
dmapp_return_t dmapp_iput_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  void             *source_addr,
  IN  ptrdiff_t        tst,
  IN  ptrdiff_t        sst,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.13.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tst Target stride.

sst Source stride.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.13.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.14 `dmapp_iput`

The `dmapp_iput` function is a blocking strided PUT.

5.3.14.1 Synopsis

```
dmapp_return_t dmapp_iput(
  IN  void              *target_addr,
  IN  dmapp_seg_desc_t  *target_seg,
  IN  dmapp_pe_t        target_pe,
  IN  void              *source_addr,
  IN  ptrdiff_t         tst,
  IN  ptrdiff_t         sst,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.14.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tst Target stride.

sst Source stride.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.14.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.15 dmapp_iget_nb

The dmapp_iget_nb function is a non-blocking explicit strided GET.

5.3.15.1 Synopsis

```
dmapp_return_t dmapp_iget_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  ptrdiff_t             tst,
  IN  ptrdiff_t             sst,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.15.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

tst Target stride.

sst Source stride.

nelems Number of elements to transfer.

type Type of elements to transfer.

syncid Pointer to the synchronization ID.

5.3.15.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

The source or target buffer or length is not properly Dword (4 byte) aligned.

DMAPP_RC_RESOURCE_ERROR

A resource error occurred.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.16 dmapp_iget_nbi

The `dmapp_iget_nbi` function is a non-blocking implicit strided GET.

5.3.16.1 Synopsis

```
dmapp_return_t dmapp_iget_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  ptrdiff_t        tst,
  IN  ptrdiff_t        sst,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.16.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

tst Target stride.

sst Source stride.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.16.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

The source or target buffer or length is not properly Dword (4 byte) aligned.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.17 `dmapp_iget`

The `dmapp_iget` function is a blocking strided GET.

5.3.17.1 Synopsis

```
dmapp_return_t dmapp_iget(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  ptrdiff_t        tst,
  IN  ptrdiff_t        sst,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.17.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

tst Target stride (>= 1).

sst Source stride (>= 1).

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.17.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Source or target buffer or length not properly Dword (4 byte) aligned.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.18 `dmapp_ixput_nb`

The `dmapp_ixput_nb` function is a non-blocking explicit Indexed PUT.

5.3.18.1 Synopsis

```
dmapp_return_t dmapp_ixput_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  void                  *source_addr,
  IN  ptrdiff_t             *tidx,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.18.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tidx Pointer to an array of positive offsets into target buffer. Each element to be transferred *i*, where *i*=1, *nelems* is transferred to *target_addr + tidx(i)*. Note that the length of the array *tidx* should be equal to *nelems*, or a segmentation fault may occur.

nelems Number of elements to be transferred.

type Type of elements to be transferred.

syncid Synchronization ID.

5.3.18.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.19 dmapp_ixput_nbi

The dmapp_ixput_nbi function is a non-blocking implicit Indexed PUT.

5.3.19.1 Synopsis

```
dmapp_return_t dmapp_ixput_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  void             *source_addr,
  IN  ptrdiff_t        *tidx,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.19.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tidx Pointer to an array of positive offsets into the target buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.19.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.20 dmapp_ixput

The dmapp_ixput function is a blocking Indexed PUT.

5.3.20.1 Synopsis

```
dmapp_return_t dmapp_ixput(
  IN  void              *target_addr,
  IN  dmapp_seg_desc_t  *target_seg,
  IN  dmapp_pe_t        target_pe,
  IN  void              *source_addr,
  IN  ptrdiff_t         *tidx,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.20.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to a segment descriptor of the target buffer.

target_pe Target processing element.

source_addr

Pointer to the address of the source buffer.

tidx Pointer to an array of positive offsets into the target buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.20.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

DMAPP_RC_TRANSACTION_ERROR

A transaction error has occurred.

5.3.21 dmapp_ixget_nb

The dmapp_ixget_nb function is a non-blocking explicit Indexed GET.

GET data starting from a remote source address using offsets specified by the *sidx* array and returning the data into a contiguous block starting at address *target_addr* in local memory. The remote address is specified by the triplet virtual address *source_addr*, segment descriptor *source_seg* and source process *source_pe*. *nelems* specifies the number of elements of type *type* to be transferred. Offsets in the *sidx* array are in units of type. The memory region described by the remote address, *sidx* and *nelems* must reside in an exported memory of *source_pe*.

5.3.21.1 Synopsis

```
dmapp_return_t dmapp_ixget_nb(
  IN  void                   *target_addr,
  IN  void                   *source_addr,
  IN  dmapp_seg_desc_t       *source_seg,
  IN  dmapp_pe_t             source_pe,
  IN  ptrdiff_t              *sidx,
  IN  uint64_t               nelems,
  IN  dmapp_type_t           type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.21.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

sidx Pointer to an array of positive offsets into the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer. The DMAPP_BYTE type is not supported.

syncid Pointer to the synchronization ID.

5.3.21.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

The source or target buffer or length is not properly Dword (4 byte) aligned.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.22 `dmapp_ixget_nbi`

The `dmapp_ixget_nbi` function is a non-blocking implicit Indexed GET.

5.3.22.1 Synopsis

```
dmapp_return_t dmapp_ixget_nbi(
  IN  void              *target_addr,
  IN  void              *source_addr,
  IN  dmapp_seg_desc_t  *source_seg,
  IN  dmapp_pe_t        source_pe,
  IN  ptrdiff_t         *sidx,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.22.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

sidx Pointer to an array of positive offsets into the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer. The type `DMAPP_BYTE` is not supported.

5.3.22.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Source or target buffer or length not properly Dword (4 byte) aligned.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.23 `dmapp_ixget`

The `dmapp_ixget` function is a blocking indexed GET.

5.3.23.1 Synopsis

```
dmapp_return_t dmapp_ixget(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  ptrdiff_t        *sidx,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.23.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

sidx Pointer to an array of positive offsets into the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer. The `DMAPP_BYTE` type is not supported.

5.3.23.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Source or target buffer or length not properly Dword (4 byte) aligned.

`DMAPP_RC_RESOURCE_ERROR`

A resource error occurred.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.24 dmapp_put_ixpe_nb

The dmapp_put_ixpe_nb function is a non-blocking explicit PUT with indexed PE stride. It delivers data starting at *source_addr* in local memory to a list of target PEs *target_pe_list* starting at *target_addr* in their memories. *nelems* specifies the number of elements of type *type* to be PUT into each target PE. When the transfer is complete, each target PE will have a copy of the contents of the original source buffer. The address range specified by *target_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *target_pe_list*.

5.3.24.1 Synopsis

```
dmapp_return_t dmapp_put_ixpe_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            *target_pe_list,
  IN  uint32_t              num_target_pes,
  IN  void                  *source_addr,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.24.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe_list

Pointer to the list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

syncid Pointer to the synchronization ID.

5.3.24.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.25 `dmapp_put_ixpe_nbi`

The `dmapp_put_ixpe_nbi` function is a non-blocking implicit PUT with indexed PE stride.

It delivers data starting at *source_addr* in local memory to a list of target PEs *target_pe_list* starting at *target_addr*in their memories. *nelems* specifies the number of elements of type *type* to be PUT into each target PE. When the transfer is complete, each target PE will have a copy of the contents of the original source buffer. The address range specified by *target_addr* and *nelems* must reside in an exported memory segment in each PE in *target_pe_list*.

5.3.25.1 Synopsis

```
dmapp_return_t dmapp_put_ixpe_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       *target_pe_list,
  IN  uint32_t         num_target_pes,
  IN  void             *source_addr,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.25.2 Parameters

IN target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe_list

Pointer to a list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.25.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.26 `dmapp_put_ixpe`

The `dmapp_put_ixpe` function is a blocking PUT with indexed PE stride. It delivers data starting at *source_addr* in local memory to a list of target PEs *target_pe_list* starting at *target_addr*in their memories. *nelems* specifies the number of elements of type *type* to be PUT into each target PE. When the transfer is complete, each target PE will have a copy of the contents of the original source buffer. The address range specified by *target* and *nelems* must reside in an exported memory segment in each PE in *target_pe_list*. The remote address is specified by the target virtual address *target_addr*and the segment descriptor *target_seg*. The address range specified by *target_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *target_pe_list*.

5.3.26.1 Synopsis

```
dmapp_return_t dmapp_put_ixpe(
  IN  void              *target_addr,
  IN  dmapp_seg_desc_t  *target_seg,
  IN  dmapp_pe_t        *target_pe_list,
  IN  uint32_t          num_target_pes,
  IN  void              *source_addr,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.26.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe_list

Pointer to the list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.26.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.27 `dmapp_scatter_ixpe_nb`

The `dmapp_scatter_ixpe_nb` function is a non-blocking explicit scatter with indexed PE stride. The function delivers data to a list of target PEs in the *target_pe_list* starting at *target_addr* in their memories. *nelems* specifies the number of elements of *type* to be PUT into each target PE. A remote PE at some index I in the *target_pe_list* will receive elements I * *nelems* to (I+1) * *nelems* - 1.

Unlike the dmapp_put_ixpe function, the *source_addr* array specifies a *num_target_pes* * *nelems* * sizeof *(type*) array.

The remote address is specified by the virtual address *target_addr* and segment descriptor *target_seg*. The address range specified by *target_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *target_pe_list*.

5.3.27.1 Synopsis

```
dmapp_return_t dmapp_scatter_ixpe_nb(
  IN  void                    *target_addr,
  IN  dmapp_seg_desc_t        *target_seg,
  IN  dmapp_pe_t              *target_pe_list,
  IN  uint32_t                num_target_pes,
  IN  void                    *source_addr,
  IN  uint64_t                nelems,
  IN  dmapp_type_t            type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.27.2 Parameters

target_addr

Pointer to an address of the target buffer.

target_seg Pointer to a segment descriptor of the target buffer.

target_pe_list

Pointer to a list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

syncid Pointer to the synchronization ID.

5.3.27.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.28 dmapp_scatter_ixpe_nbi

The `dmapp_scatter_ixpe_nbi` function is a non-blocking implicit scatter with indexed PE stride. The function delivers data to a list of target PEs in the *target_pe_list* starting at *target*_addr in their memories. *nelems* specifies the number of elements of type to be PUT into each target PE. A remote PE at some index I in the *target_pe_list* will receive elements I * *nelems* to (I+1) * *nelems* - 1.

Unlike the `dmapp_put_ixpe` function, the *source_addr* array specifies a *num_target_pes* * *nelems* * `sizeof` *(type*) array.

The remote address is specified by the virtual address *target_addr* and segment descriptor *target_seg*. The address range specified by *target_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *target_pe_list*.

5.3.28.1 Synopsis

```
dmapp_return_t dmapp_scatter_ixpe_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       *target_pe_list,
  IN  uint32_t         num_target_pes,
  IN  void             *source_addr,
  IN  uint64_t         nelems,
  IN  dmapp_type_t     type);
```

5.3.28.2 Parameters

target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe_list

Pointer to the list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.28.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.29 `dmapp_scatter_ixpe`

The `dmapp_scatter_ixpe` function is a blocking scatter with indexed PE stride. The function delivers data to a list of target PEs in the *target_pe_list* starting at *target*_addr in their memories. *nelems* specifies the number of elements of type to be PUT into each target PE.

A remote PE at some index I in the *target_pe_list* will receive elements I * *nelems* to (I+1) **nelems*-1. The address range specified by *target_addr* and *nelems* must reside in an exported memory segment in each PE specified in *target_pe_list.*

Unlike the `dmapp_put_ixpe` function, the *source_addr* array specifies a *num_target_pes* * *nelems* * sizeof *(type*) array.

The remote address is specified by the virtual address *target_addr*and segment descriptor *target_seg*. The address range specified by *target_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *target_pe_list*.

5.3.29.1 Synopsis

```
dmapp_return_t dmapp_scatter_ixpe(
  IN  void              *target_addr,
  IN  dmapp_seg_desc_t  *target_seg,
  IN  dmapp_pe_t        *target_pe_list,
  IN  uint32_t          num_target_pes,
  IN  void              *source_addr,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.29.2 Parameters

IN target_addr

Pointer to the address of the target buffer.

target_seg Pointer to the segment descriptor of the target buffer.

target_pe_list

Pointer to the list of target processing elements.

num_target_pes

Number of target processing elements.

source_addr

Pointer to the address of the source buffer.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.29.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.30 `dmapp_gather_ixpe_nb`

The `dmapp_gather_ixpe_nb` function is a non-blocking explicit gather with indexed PE stride. Gather data starting at *source_addr* from the list of PEs specified by *source_pe_list* and concatenate the returned data in a buffer in local memory specified by *target_addr*. *nelems* specifies the number of elements of *type* collected from each PE.

The address range specified by *source_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *source_pe_list*.

5.3.30.1 Synopsis

```
dmapp_return_t dmapp_gather_ixpe_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            *source_pe_list,
  IN  uint32_t              num_source_pes,
  IN  uint64_t              nelems,
  IN  dmapp_type_t          type,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.30.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe_list

Pointer to the list of source processing elements.

num_source_pes

Number of source processing elements.

nelems Number of elements to transfer.

type Type of elements to transfer.

syncid Returns a pointer to the synchronization ID.

5.3.30.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.31 dmapp_gather_ixpe_nbi

The dmapp_gather_ixpe_nbi function is a non-blocking implicit gather with indexed PE stride. Gather data starting at *source_addr* from the list of PEs specified by *source_pe_list* and concatenate the returned data in a buffer in local memory specified by *target_addr*. *nelems* specifies the number of elements of *type* collected from each PE.

The address range specified by *source_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE in *source_pe_list*.

5.3.31.1 Synopsis

```
dmapp_return_t dmapp_gather_ixpe_nbi(
  IN  void              *target_addr,
  IN  void              *source_addr,
  IN  dmapp_seg_desc_t  *source_seg,
  IN  dmapp_pe_t        *source_pe_list,
  IN  uint32_t          num_source_pes,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.31.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe_list

Pointer to the list of source processing elements.

num_source_pes

Number of source processing elements.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.31.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.32 `dmapp_gather_ixpe`

The `dmapp_gather_ixpe` function is a blocking gather with indexed processing element stride. Gather data starting at *source_addr* from the list of PEs specified by *source_pe_list* and concatenate the returned data in a buffer in local memory specified by *target_addr*. *nelems* specifies the number of elements of *type* collected from each PE.

The address range specified by *source_addr* and *nelems* must reside in an exported, symmetric memory segment in each PE listed in *source_pe_list*.

5.3.32.1 Synopsis

```
dmapp_return_t dmapp_gather_ixpe(
  IN  void              *target_addr,
  IN  void              *source_addr,
  IN  dmapp_seg_desc_t  *source_seg,
  IN  dmapp_pe_t        *source_pe_list,
  IN  uint32_t          num_source_pes,
  IN  uint64_t          nelems,
  IN  dmapp_type_t      type);
```

5.3.32.2 Parameters

target_addr

Pointer to the address of the target buffer.

source_addr

Pointer to the address of the source buffer.

source_seg Pointer to the segment descriptor of the source buffer.

source_pe_list

Pointer to the list of source processing elements.

num_source_pes

Number of source processing elements.

nelems Number of elements to transfer.

type Type of elements to transfer.

5.3.32.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.33 `dmapp_aadd_qw_nb`

The `dmapp_aadd_qw_nb` function is a non-blocking explicit atomic `ADD`.

5.3.33.1 Synopsis

```
dmapp_return_t dmapp_aadd_qw_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.33.2 Parameters

target_addr

Pointer to the address of the target buffer (for Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Value to be added.

syncid Pointer to the synchronization ID.

5.3.33.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.34 `dmapp_aadd_qw_nbi`

The `dmapp_aadd_qw_nbi` function is a non-blocking implicit atomic `ADD`.

5.3.34.1 Synopsis

```
dmapp_return_t dmapp_aadd_qw_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.34.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Value to be added.

5.3.34.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.35 dmapp_aadd_qw

The `dmapp_aadd_qw` function is a blocking atomic `ADD`.

5.3.35.1 Synopsis

```
dmapp_return_t dmapp_aadd_qw(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.35.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Value to be added.

5.3.35.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

DMAPP_RC_TRANSACTION_ERROR

A transaction error has occurred.

5.3.36 dmapp_aand_qw_nb

The dmapp_aand_qw_nb function is a non-blocking explicit atomic AND.

5.3.36.1 Synopsis

```
dmapp_return_t dmapp_aand_qw_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.36.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the AND operation.

syncid Pointer to the synchronization ID.

5.3.36.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.37 dmapp_aand_qw_nbi

The `dmapp_aand_qw_nbi` function is a non-blocking implicit atomic AND.

5.3.37.1 Synopsis

```
dmapp_return_t dmapp_aand_qw_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.37.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the AND operation.

5.3.37.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.38 dmapp_aand_qw

The dmapp_aand_qw function is a blocking atomic AND.

5.3.38.1 Synopsis

```
dmapp_return_t dmapp_aand_qw(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.38.2 Parameters

target_addr

Pointer the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the AND operation.

5.3.38.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

DMAPP_RC_TRANSACTION_ERROR

A transaction error has occurred.

5.3.39 dmapp_aor_qw_nb

The `dmapp_aor_qw_nb` function is a non-blocking explicit atomic OR.

5.3.39.1 Synopsis

```
dmapp_return_t dmapp_aor_qw_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.39.2 Parameter

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the OR operation.

syncid Pointer to the synchronization ID.

5.3.39.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.40 dmapp_aor_qw_nbi

The `dmapp_aor_qw_nbi` function is a non-blocking implicit atomic OR.

5.3.40.1 Synopsis

```
dmapp_return_t dmapp_aor_qw_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.40.2 Parameter

IN target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the OR operation.

5.3.40.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.41 dmapp_aor_qw

The dmapp_aor_qw function is a blocking atomic OR.

5.3.41.1 Synopsis

```
dmapp_return_t dmapp_aor_qw(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.41.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the ORD operation.

5.3.41.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.42 dmapp_axor_qw_nb

The dmapp_axor_qw_nb function is a non-blocking explicit atomic XOR.

5.3.42.1 Synopsis

```
dmapp_return_t dmapp_axor_qw_nb(
  IN  void                  *target_addr,
  IN  dmapp_seg_desc_t      *target_seg,
  IN  dmapp_pe_t            target_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.42.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the XOR operation.

syncid Pointer to the synchronization ID.

5.3.42.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.43 dmapp_axor_qw_nbi

The dmapp_axor_qw_nbi function is a non-blocking implicit atomic XOR.

5.3.43.1 Synopsis

```
dmapp_return_t dmapp_axor_qw_nbi(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.43.2 Parameter

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for the XOR operation.

5.3.43.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.44 `dmapp_axor_qw`

The `dmapp_axor_qw` function is a blocking atomic XOR.

5.3.44.1 Synopsis

```
dmapp_return_t dmapp_axor_qw(
  IN  void             *target_addr,
  IN  dmapp_seg_desc_t *target_seg,
  IN  dmapp_pe_t       target_pe,
  IN  int64_t          operand);
```

5.3.44.2 Parameters

target_addr

Pointer to the address of the target buffer (Qword only).

target_seg Pointer to the segment descriptor for the target buffer.

target_pe Target processing element.

operand Operand for an XOR operation.

5.3.44.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.45 `dmapp_afadd_qw_nb`

The `dmapp_afadd_qw_nb` function is a non-blocking explicit atomic FADD.

5.3.45.1 Synopsis

```
dmapp_return_t dmapp_afadd_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.45.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for the FADD operation.

syncid Pointer to the synchronization ID.

5.3.45.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

5.3.46 `dmapp_afadd_qw_nbi`

The `dmapp_afadd_qw_nbi` function is a non-blocking implicit atomic FADD.

5.3.46.1 Synopsis

```
dmapp_return_t dmapp_afadd_qw_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.46.2 Parameters

target_addr

Pointer to the address of a target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for the FADD operation.

5.3.46.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.47 `dmapp_afadd_qw`

The `dmapp_afadd_qw` function is a blocking atomic FADD.

5.3.47.1 Synopsis

```
dmapp_return_t dmapp_afadd_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.47.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for the FADD operation.

5.3.47.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.48 `dmapp_afand_qw_nb`

The `dmapp_afand_qw_nb` function is a non-blocking explicit atomic FAND.

5.3.48.1 Synopsis

```
dmapp_return_t dmapp_afand_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.48.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for the FAND operation.

syncid Pointer to the synchronization ID.

5.3.48.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.49 `dmapp_afand_qw_nbi`

The `dmapp_afand_qw_nbi` function is a non-blocking implicit atomic FAND.

5.3.49.1 Synopsis

```
dmapp_return_t
dmapp_afand_qw_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.49.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Segment descriptor of source buffer.

source_pe Source processing element.

operand Operand for an FAND operation.

5.3.49.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.50 `dmapp_afand_qw`

The `dmapp_afand_qw` function is a blocking atomic FAND.

5.3.50.1 Synopsis

```
dmapp_return_t dmapp_afand_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.50.2 Parameters

target_add Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for an FAND operation.

5.3.50.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.51 `dmapp_afxor_qw_nb`

The `dmapp_afxor_qw_nb` function is a non-blocking explicit atomic FXOR.

5.3.51.1 Synopsis

```
dmapp_return_t dmapp_afxor_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.51.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for an FXOR operation.

syncid Pointer to the synchronization ID.

5.3.51.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.52 `dmapp_afxor_qw_nbi`

The `dmapp_afxor_qw_nbi` function is a non-blocking implicit atomic FXOR.

5.3.52.1 Synopsis

```
dmapp_return_t dmapp_afxor_qw_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.52.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for an FXOR operation.

5.3.52.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.53 `dmapp_afxor_qw`

The `dmapp_afxor_qw` function is a blocking atomic FXOR.

5.3.53.1 Synopsis

```
dmapp_return_t dmapp_afxor_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.53.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for an FXOR operation.

5.3.53.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.54 `dmapp_afor_qw_nb`

The `dmapp_afor_qw_nb` function is a non-blocking explicit atomic FOR.

5.3.54.1 Synopsis

```
dmapp_return_t dmapp_afor_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               operand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.54.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for a FOR operation.

syncid Pointer to the synchronization ID.

5.3.54.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.55 `dmapp_afor_qw_nbi`

The `dmapp_afor_qw_nbi` function is a non-blocking implicit atomic FOR.

5.3.55.1 Synopsis

```
dmapp_return_t dmapp_afor_qw_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.55.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for a FOR operation.

5.3.55.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.56 `dmapp_afor_qw`

The `dmapp_afor_qw` function is a blocking atomic FOR.

5.3.56.1 Synopsis

```
dmapp_return_t dmapp_afor_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          operand);
```

5.3.56.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

operand Operand for a FOR operation.

5.3.56.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.57 dmapp_afax_qw_nb

The dmapp_afax_qw_nb function is a non-blocking explicit atomic FAX.

5.3.57.1 Synopsis

```
dmapp_return_t dmapp_afax_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               andMask,
  IN  int64_t               xorMask,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.57.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

andMask Mask for an AND operation.

xorMask Mask for an XOR operation.

syncid Pointer to the synchronization ID.

5.3.57.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.58 dmapp_afax_qw_nbi

The dmapp_afax_qw_nbi function is a non-blocking implicit atomic FAX.

5.3.58.1 Synopsis

```
dmapp_return_t dmapp_afax_qw_nbi(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          andMask,
  IN  int64_t          xorMask);
```

5.3.58.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

andMask Mask for an AND operation.

xorMask Mask for an XOR operation.

5.3.58.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

DMAPP_RC_NO_SPACE

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for max_outstanding_nb in the job attributes.

5.3.59 dmapp_afax_qw

The dmapp_afax_qw function is a blocking atomic FAX.

5.3.59.1 Synopsis

```
dmapp_return_t dmapp_afax_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          andMask,
  IN  int64_t          xorMask);
```

5.3.59.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

andMask Mask for an AND operation.

xorMask Mask for an XOR operation.

5.3.59.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

DMAPP_RC_ALIGNMENT_ERROR

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.60 dmapp_acswap_qw_nb

The `dmapp_acswap_qw_nb` function is a non-blocking explicit atomic CSWAP.

5.3.60.1 Synopsis

```
dmapp_return_t dmapp_acswap_qw_nb(
  IN  void                  *target_addr,
  IN  void                  *source_addr,
  IN  dmapp_seg_desc_t      *source_seg,
  IN  dmapp_pe_t            source_pe,
  IN  int64_t               comperand,
  IN  int64_t               swaperand,
  OUT dmapp_syncid_handle_t *syncid);
```

5.3.60.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

comperand Operand against which to compare.

swaperand Operand to swap in.

syncid Pointer to a synchronization ID.

5.3.60.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.61 `dmapp_acswap_qw_nbi`

The `dmapp_acswap_qw_nbi` function is a non-blocking implicit atomic CSWAP.

5.3.61.1 Synopsis

```
dmapp_return_t dmapp_acswap_qw_nbi(
  IN  void              *target_addr,
  IN  void              *source_addr,
  IN  dmapp_seg_desc_t  *source_seg,
  IN  dmapp_pe_t        source_pe,
  IN  int64_t           comperand,
  IN  int64_t           swaperand);
```

5.3.61.2 Parameters

target_addr

pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

comperand Operand against which to compare.

swaperand Operand to swap in.

5.3.61.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

5.3.62 dmapp_acswap_qw

The `dmapp_acswap_qw` function is a blocking atomic CSWAP.

5.3.62.1 Synopsis

```
dmapp_return_t dmapp_acswap_qw(
  IN  void             *target_addr,
  IN  void             *source_addr,
  IN  dmapp_seg_desc_t *source_seg,
  IN  dmapp_pe_t       source_pe,
  IN  int64_t          comperand,
  IN  int64_t          swaperand);
```

5.3.62.2 Parameters

target_addr

Pointer to the address of the target buffer where the result is returned (Qword only).

source_addr

Pointer to the address of the source buffer (Qword only).

source_seg Pointer to the segment descriptor of the source buffer.

source_pe Source processing element.

comperand Operand against which to compare.

swaperand Operand to swap in.

5.3.62.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_ALIGNMENT_ERROR`

Target buffer not properly (Qword (8 byte)) aligned.

`DMAPP_RC_NO_SPACE`

The transaction request could not be completed due to insufficient resources. To resolve this error, synchronize more often, or if possible, increase the value for `max_outstanding_nb` in the job attributes.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.63 `dmapp_syncid_test`

The `dmapp_syncid_test` function tests *syncid* for completion. It sets *flag* to 1 if the remote memory accesses associated with *syncid* are globally visible in the system. If the RMA request associated with the *syncid* has not completed, *flag* is set to 0.

5.3.63.1 Synopsis

```
dmapp_return_t dmapp_syncid_test(
  INOUT dmapp_syncid_handle_t *syncid,
  OUT   int                   *flag);
```

5.3.63.2 Parameters

syncid Pointer to the syncid to test for completion.

flag Pointer to the flag indicating global visibility.

5.3.63.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.64 `dmapp_syncid_wait`

The `dmapp_syncid_wait` function is a wait for completion of request associated with *syncid*.

5.3.64.1 Synopsis

```
dmapp_return_t dmapp_syncid_wait(
  INOUT dmapp_syncid_handle_t *syncid);
```

5.3.64.2 Parameters

syncid The *syncid* to test for completion.

5.3.64.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

`DMAPP_RC_TRANSACTION_ERROR`

A transaction error has occurred.

5.3.65 dmapp_gsync_test

The dmapp_gsync_test function is a test for completion of issued nb implicit requests. It sets flag to 1 if remote memory accesses associated with previously issued non-blocking implicit RMA requests are globally visible in the system. Otherwise, flag is set to 0.

5.3.65.1 Synopsis

```
dmapp_return_t dmapp_gsync_test(
  OUT int *flag);
```

5.3.65.2 Parameters

flag Pointer to a flag indicating global visibility.

5.3.65.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_TRANSACTION_ERROR

A transaction error has occurred.

5.3.66 dmapp_gsync_wait

The dmapp_gsync_wait function forces a wait for completion of issued nb implicit requests. This is the blocking version of dmapp_gsync_test. The function only returns when all remote memory accesses associated with previously issued non-blocking implicit RMA requests are globally visible in the system.

5.3.66.1 Synopsis

```
dmapp_return_t dmapp_gsync_wait(void);
```

5.3.66.2 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_TRANSACTION_ERROR

A transaction error has occurred.

5.3.67 dmapp_sheap_malloc

The dmapp_sheap_malloc function allocates *size* bytes of memory from the symmetric heap. The space returned is left uninitialized. It cannot be assumed that the memory returned is zeroed out. There are no address equality guarantees across ranks.

5.3.67.1 Synopsis

```
void *dmapp_sheap_malloc(
  IN size_t size);
```

5.3.67.2 Parameters

size The size, in bytes, of memory to allocate from the symmetric heap.

5.3.68 dmapp_sheap_realloc

The dmapp_sheap_realloc function changes the size of the block to which *ptr* points to the size, in bytes, specified by *size*. The contents of the block are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the block is indeterminate. If *ptr* is a null pointer, dmapp_sheap_realloc behaves like dmapp_sheap_malloc for the specified size. If *size* is 0 and *ptr* is not a null pointer, the block to which it points is freed. Otherwise, if *ptr* does not match a pointer earlier returned by a symmetric heap function, or if the space has already been deallocated, dmapp_sheap_realloc returns a null pointer. If the space cannot be allocated, the block to which *ptr* points is unchanged.

5.3.68.1 Synopsis

```
void *dmapp_sheap_realloc(
  IN void   *ptr,
  IN size_t size);
```

5.3.68.2 Parameters

ptr Pointer to the block.

size The size, in bytes, of which to change the block.

5.3.69 dmapp_sheap_free

The dmapp_sheap_free function frees a block of memory previously allocated by dmapp_sheap_malloc or dmapp_sheap_realloc.

5.3.69.1 Synopsis

```
void dmapp_sheap_free(
  IN void   *ptr);
```

5.3.69.2 Parameters

ptr	Pointer to the block of memory previously allocated with `dmapp_sheap_malloc` or `dmapp_sheap_realloc`.

5.3.70 dmapp_mem_register

The `dmapp_mem_register` function dynamically registers a memory region, other than statically linked data segment or the symmetric heap, with the NIC.

The memory region is described by starting address *addr* and *length*. This memory could have been allocated from the private heap or using `mmap`. Memory registered by a call to `dmapp_mem_register` becomes remotely accessible and is assumed to be non-symmetric.

The function updates the content of the segment descriptor to reflect the actual starting address and length of the region which was registered. These values can differ from the input values due to rounding, for instance. Dynamically registered memory can only be remotely accessed by point-to-point RMA functions, not by PE-strided RMA functions.

5.3.70.1 Synopsis

```
dmapp_return_t dmapp_mem_register(
  IN  void              *addr,
  IN  uint64_t           length,
  INOUT  dmapp_seg_desc_t *seg_desc);
```

5.3.70.2 Parameters

addr	Points to starting address of the memory region to be registered. Must be non-NULL.
length	Length in bytes of the memory region in bytes. Must be > 0.
seg_desc	On input, points to segment descriptor and must be non-NULL. Function updates it to the actual starting address and length of the registered region.

5.3.70.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_RESOURCE_ERROR`

Unsuccessful memory registration or invalid memory handle.

`DMAPP_RC_INVALID_PARAM`

One or more input parameters is invalid.

5.3.71 dmapp_mem_unregister

The `dmapp_mem_unregister` function unregisters a memory region, other than statically linked data segment or the symmetric heap, on the fly, from the NIC. The memory region must previously have been registered by a call to `dmapp_mem_register`.

5.3.71.1 Synopsis

```
dmapp_return_t dmapp_mem_register(
    IN  dmapp_seg_desc_t *seg_desc);
```

5.3.71.2 Parameters

seg_desc Points to segment descriptor to deregister, which must have been registered with a call to `dmapp_mem_register.`

5.3.71.3 Return Codes

`DMAPP_RC_SUCCESS`

The operation completed successfully.

`DMAPP_RC_INVALID_PARAM`

Input parameter is invalid.

5.3.72 dmapp_segdesc_compare

The `dmapp_segdesc_compare` function compares two segment descriptors. If they describe the same memory region, flag is set to 1. If they describe different memory regions, flag is set to 0.

5.3.72.1 Synopsis

```
dmapp_return_t dmapp_segdesc_compare(
  IN  dmapp_seg_desc_t *seg_desc1,
  OUT dmapp_seg_desc_t *seg_desc2,
  INOUT int *flag);
```

5.3.72.2 Parameters

seg_desc1	Pointer to segment descriptor 1.
seg_desc2	Pointer to segment descriptor 2.
flag	set to 1 if both segment descriptors describe the same memory region, otherwise set to 0.

5.3.72.3 Return Codes

DMAPP_RC_SUCCESS

The operation completed successfully.

DMAPP_RC_INVALID_PARAM

One or more input parameters is invalid.

5.4 Environment Variables Which Affect DMAPP

5.4.1 XT_SYMMETRIC_HEAP_SIZE

Controls the size (in bytes) of the symmetric heap. One Mbyte is allocated for internal use only.

Default: 0 bytes for the user

5.4.2 DMAPP_ABORT_ON_ERROR

Allows a user to force a core dump upon error. This is supported during initialization and memory handling operations.

Default: not set

Sample Code [A]

A.1 dmapp_put.c

```
#include <stdio.h>
#include <unistd.h>
#include "pmi.h"
#include "dmapp.h"

#define MAX_NELEMS  (128L*1024L)
/* If necessary, run the job with fewer than the maximum number of cores
 * per node so that enough memory is available for each PE. */

int main(int argc,char **argv)
{
    int               pe = -1;
    int               npes = -1;
    int               target_pe;
    int               fail_count = 0;
    long              nelems = MAX_NELEMS;
    long              *source;
    long              *target;
    long              i;
    dmapp_return_t    status;
    dmapp_rma_attrs_t actual_args = {0}, rma_args = {0};
    dmapp_jobinfo_t   job;
    dmapp_seg_desc_t  *seg = NULL;

    /* Set the RMA parameters. */

    rma_args.put_relaxed_ordering = DMAPP_ROUTING_ADAPTIVE;
    rma_args.max_outstanding_nb = DMAPP_DEF_OUTSTANDING_NB;
    rma_args.offload_threshold = DMAPP_OFFLOAD_THRESHOLD;
    rma_args.max_concurrency = 1;

    /* Initialize DMAPP. */

    status = dmapp_init(&rma_args, &actual_args);
    if (status != DMAPP_RC_SUCCESS) {
        fprintf(stderr," dmapp_init FAILED: %d\n", status);
        exit(1);
    }

    /* Allocate and initialize the source and target arrays. */

    source = (long *)dmapp_sheap_malloc(nelems*sizeof(long));
    target = (long *)dmapp_sheap_malloc(nelems*sizeof(long));
    if ((source == NULL) || (target == NULL))  {
        fprintf(stderr," sheap_malloc'd failed src 0x%lx targ 0x%lx\n",
            (long)source, (long)target);
```

```
    exit(1);
}
for (i=0; i<nelems; i++) {
    source[i] = i;
    target[i] = -9L;
}

/* Wait for all PEs to complete array initialization. */

PMI_Barrier();

/* Get job related information. */

status = dmapp_get_jobinfo(&job);
if (status != DMAPP_RC_SUCCESS) {
    fprintf(stderr," dmapp_get_jobinfo FAILED: %d\n", status);
    exit(1);
}

pe = job.pe;
npes = job.npes;
seg = &(job.sheap_seg);

/* Send my data to my target PE. */

target_pe = npes - pe -1;
status = dmapp_put(target, seg, target_pe, source, nelems, DMAPP_QW);

if (status != DMAPP_RC_SUCCESS) {
    fprintf(stderr," dmapp_put FAILED: %d\n", status);
    exit(1);
}

/* Wait for all PEs to complete their PUT. */

PMI_Barrier();

/* Check the results. */

for (i=0; i<nelems; i++) {
    if ((target[i] != i) && (fail_count<10)) {
        fprintf(stderr," PE %d: target[%ld] is %ld, should be %ld\n",
        pe, i, target[i], (long)i);
        fail_count++;
    }
}

if (fail_count == 0) {
    fprintf(stderr," dmapp_put PASSED for PE %04d\n", pe);
}
else {
    fprintf(stderr," dmapp_put FAILED for PE %04d: "
        "%d or more wrong values\n", pe, fail_count);
}
```

```
    /* Finalize. */

    status = dmapp_finalize();
    return(0);
}
```