Intel Virtual Interface (VI) Architecture *Developer's* Guide

Revision *1.0*
September 9, 1998

DISCLAIMERS

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

No license, expressed or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification. Intel does not warrant or represent that such use will not infringe such rights.

Nothing in this document constitutes a guarantee, warranty, or license, express or implied. Intel disclaims all liability for all such guaranties, warranties, and licenses, including but not limited to: fitness for a particular purpose; merchantability; non-infringement of intellectual property or other rights of any third party or of Intel; indemnity; and all others. The reader is advised that third parties may have intellectual property rights which may be relevant to this document and the technologies discussed herein, and is advised to seek the advice of competent legal counsel, without obligation to Intel.

Intel retains the right to make changes to this document at any time, without notice. Intel makes no warranty for the use of this document and assumes no responsibility for any errors, which may appear in the document, nor does it make a commitment to update the information contained herein.

The Intel Virtual Interface (VI) Architecture *Developer's* Guide may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Currently characterized errata are available on request.

Table of Contents

1. Introduction

1.1. Overview

The Intel VI Architecture Developer's Guide describes the Virtual Interface Provider Library (VIPL) and the VI Kernel Agent along with illustrative programming examples and application notes. VIPL is based on the example VI User Agent in Appendix A of the VI Architecture Specification Version 1.0, with annotations, errata corrections and proposed extensions in italics to provide a more complete interface for implementers and developers. The VI Kernel Agent is a component of the operating system required by the VI Architecture that subsumes the role of the device driver for the VI NIC and includes the kernel software needed to register communication memory and manage VIs.

1.2. Terminology

1.2.1. Acronyms and Abbreviations

API Application Programming Interface. A collection of function calls exported by libraries and/or services.

CRC Cyclic Redundancy Check. A number derived from, and stored or transmitted with, a block of data in order to detect corruption. By recalculating the CRC and comparing it to the value originally transmitted, the receiver can detect some types of transmission errors.

DMA Direct Memory Access. A facility that allows a peripheral device to read and write memory without intervention by the CPU.

IHV Independent Hardware Vendor. Any vendor providing hardware. Used synonymously at times with VI Hardware Vendor.

MTU Maximum Transfer Unit. The largest frame length that may be sent on a physical medium.

NIC Network Interface Controller. A NIC provides an electro-mechanical attachment of a computer to a network. Under program control, a NIC copies data from memory to the network medium, transmission, and from the medium to memory, reception, and implements a unique destination for messages traversing the network.

OSV Operating System Vendor. The software manufacturer of the operating system that is running on the node under discussion.

QOS Quality of Service. Metrics that predict the behavior, speed and latency of a given network connection.

SAN System Area Network. A high-bandwidth, low-latency network interconnecting nodes within a distributed computer system.

SAR Segmentation and Re-assembly. The process of breaking data to be transferred into quantities that are less than or equal to the MTU, transmitting them across the network and then reassembling them at the receiving end to reconstruct the original data.

TCP/IP Transmission Control Protocol/Internet Protocol. A standard networking protocol developed for LANs and WANs. This is the standard communication protocol used in the Internet.

VM Virtual Memory. The address space available to a process running in a system with a memory management unit (MMU). The virtual address space is usually divided into pages, each consisting of 2^N bytes. The bottom N address bits (the offset within a page) are left unchanged, indicating the offset within a page, and the upper bits give a (virtual) page number that is mapped by the MMU to a physical page address. This is recombined with the offset to give the address of a location in physical memory.

1.2.2. Industry Terms

Callback A scheme used in event-driven programs where the program registers a function, called the callback handler, for a certain event. The program does not call the callback handler directly. Rather, when the event occurs, the handler is invoked asynchronously, possibly with arguments describing the event.

Data Payload The amount of data, not including any control or header information, that can be carried in one packet.

Frame One unit of data encapsulated by a physical network protocol header and/or trailer. The header generally provides control and routing information for directing the frame through the network fabric. The trailer generally contains control and CRC data for ensuring packets are not delivered with corrupted contents.

Link A full duplex channel between any two network fabric elements, such as nodes, routers or switches.

Message An application-defined unit of data interchange. A primitive unit of communication between cooperating sequential processes.

Message Latency
The elapsed time from the initiation of a message send operation until the receiver is notified that the entire message is present in its memory.

Message Overhead
The sum of the times required to initiate transmission of a message, notify the receiver that the message is available, and the non-bandwidth dependent latencies (e.g. time for a NIC to process data) incurred in moving a message from the source to the destination.

Network Fabric
The collection of routers, switches, connectors, and cables that connects a set of nodes.

Network Partition
A network partition is when a network of nodes breaks into two (or more) separate subnetworks where no communication can occur between the subnetworks.

Node A computer attached by a NIC to one or more links of a network, and forming the origin and/or destination of messages within the network.

Packet A primitive unit of data interchange between nodes, comprised of a set of data segments transmitted in an ordered stream. A packet may be sent as a single frame, or may be fragmented into smaller units (cells) such that cells for various packets may be interleaved in the fabric but the transmission order of cells for a packet is preserved and manifest as a contiguous unit at a receiving node.

Server The class of computers that emphasize I/O connectivity and centralized data storage capacity to support the needs of other, typically remote, client computers.

Workstation, or Client
The class of computers that emphasize numerical and/or graphic performance and provide an interface to a human being.

1.2.3. VI Architecture Terms

The following terms are introduced in this document.

Address Segment
The second of the three segments that comprise a remote-DMA operation Descriptor, specifying the memory region to access on the target.

Communication Memory
Any region of a process' memory that is registered with the VI Provider to serve for storage of Descriptors and/or as communication buffers; i.e., any region of a process' memory that will be accessed by the VI NIC.

Connection An association between a pair of VIs such that messages sent using either VI arrives at the other VI. A VI is either unconnected, or connected to one and only one other VI.

Control Segment
The first component of a Descriptor containing information regarding the type of VI NIC data movement operation to be performed, the status of a completed VI NIC data movement operation, and the location of the next Descriptor on a Work Queue.

Completion Queue
A queue containing information about completed Descriptors. Used to create a single point of completion notification for multiple queues.

Completion Queue Entry
A single data structure on a Completion Queue that describes a completed Descriptor. This entity contains sufficient information to determine the queue that holds the completed Descriptor.

Data Segment A component of a Descriptor specifying one memory region for the VI NIC to use as a communication buffer.

Descriptor A data structure recognized by the VI NIC that describes a data movement request. A Descriptor is organized as a list of segments. A Descriptor is comprised of a control segment followed by an optional address segment and an arbitrary number of data segments. The data segments describe a communication buffer gather or scatter list for a VI NIC data movement operation.

Doorbell A mechanism for a process to notify the VI NIC that work has been placed on a Work Queue. The Doorbell mechanism must be protected by the operating system—i.e., for address protection, only the operating system should be able to establish a Doorbell—and the VI NIC must be able to identify the owner of a VI by the use of its Doorbell.

Done The state of a Descriptor when the VI NIC has completed processing it. *This term is not used in this document, refer to Section 5.5 for a description of Descriptor states.*

Immediate Data
Data contained in a Descriptor that is sent along with the data to the remote node and placed in the remote node's pre-posted Receive Queue Descriptor.

Kernel Agent A component of the operating system required by the VI Architecture that subsumes the role of the device driver for the VI NIC and includes the kernel software needed to register communication memory and manage VIs.

Memory Handle
A programmatic construct that represents a process's authorization to specify a memory region to the VI NIC. A memory handle is created by the VI Kernel Agent when a process registers communication memory. A process must supply a corresponding Memory Handle with any virtual address to qualify it to the VI NIC. The VI NIC will not perform an access to a virtual address if the supplied memory handle does not agree with the memory region containing the virtual address or if the memory region is registered to a process other than the process that owns a Virtual Interface (VI).

Memory Protection Attributes
The access rights for RDMA granted to VIs and to Memory Regions.

Memory Protection Tag
A unique identifier generated by the VI Provider for use by the VI Consumer. Memory Protection Tags are associated with VIs and Memory Regions to define the access permission the VI has to a memory region.

Memory Region
An arbitrary sized region of a process's virtual address space registered as communication memory such that it can be directly accessed by the VI NIC.

Memory Registration
The act of creating a memory region. The memory registration operation returns a Memory Handle that the process is required to provide with any virtual address within the memory region.

VI NIC Address
The logical network address of the VI NIC. This address is assigned to a VI NIC by the operating system and allows processes within a network to identify a remote node with respect to a VI NIC attachment of the remote node to the network.

NIC Handle A programmatic construct representing a process's authorization to perform communication operations using a local VI NIC.

Outstanding The state of a Descriptor after it has been posted on a Work Queue, but before it is Done. This state represents the interval of time between a process posting a Descriptor and the completion of the Descriptor by the VI NIC. *This term is not used in this document, refer to Section 5.5 for a description of Descriptor states.*

Peer A generic term for the process at the other end of a connection.

Post To place a Descriptor on a VI Work Queue.

RDMA Remote Direct Memory Access. A Descriptor operation whereby data in a local gather or scatter list is moved directly to or from a memory region on a remote node. A process authorizes remote access to its memory by creating a VI with remote-DMA operations enabled, connecting it to a remote VI, and making the memory handle for the memory region to be shared available to the peer that will perform the remote-DMA operation. There are two remote-DMA operations: write and read.

Receive Queue
One of the two queues associated with a VI. This queue contains Descriptors that describe where to place incoming data.

Reliable Delivery

The middle communication reliability level. Guarantees that all data submitted for transfer will arrive at its destination exactly once, intact and in the order submitted, in the absence of errors. The VI Provider must deliver an error to the VI Consumer if a transfer is lost, corrupted or delivered out of order.

Reliable Reception

The most reliable communication reliability level. A Descriptor is completed with a successful status only when the data has been delivered into the target memory location. If an error occurs that prevents a successful (in-order, intact and exactly once) delivery of the data into the target memory, the error is reported through the Descriptor status. Otherwise, a Reliable Reception VI behaves like a Reliable Delivery VI.

Retired The state of a Descriptor after the VI NIC completes the operation specified by the Descriptor, but before the done operation has been used to synchronize the process with the status stored in the Descriptor. *This term is not used in this document, refer to Section 5.5 for a description of Descriptor states.*

Send Queue One of the two queues associated with a VI. This queue contains Descriptors that describe the data to be transmitted.

Unreliable Delivery

The least reliable communication level. This level guarantees that a Send or RDMA Write is delivered at most once to the receiving VI and corrupted transfers are detected on the receiving side. Sends and RDMA Writes may be lost on an Unreliable Delivery VI. In addition, requests are not guaranteed to be delivered to the receiver in the order submitted by the sender. However, the order must adhere to the Descriptor processing ordering rules.

User Agent A software component that enables an Operating System communication facility to utilize a particular VI Provider. The VI User Agent abstracts the details of the underlying VI NIC hardware in accordance with an interface defined by the Operating System communication facility.

VI Virtual Interface. An interface between a VI NIC and a process allowing a VI NIC direct access to the process' memory. A VI consists of a pair of Work Queues—one for send operations and one for receive operations. The queues store a Descriptor between the time it is posted and the time it is Done. A pair of VIs are associated using the connect operation to allow packets sent at one VI to be received at the other.

VI Address The logical name for a VI. The VI address identifies a remote end-point to be associated with a local end-point using the connect-VI operation.

VI Application An application that uses the primitives provided by the VI User Agent.

VI Consumer A software process that communicates using a Virtual Interface. The VI Consumer typically consists of an application program, an Operating System communications facility, and a VI User Agent.

VI Handle A programmatic construct that represents a processes authorization to perform operations on a specific VI. A VI handle is returned by the operation that creates the VI and is supplied as an identifier parameter to the other VI operations.

VI Hardware Vendor
Anyone who produces a VI Architecture enabled NIC implementation. The vendor is responsible for providing the VI NIC, VI Kernel Agent and the VI User Agent.

VI NIC A Network Interface *Controller* that complies with the VI Architecture Specification.

VIPL *An implementation of the VI User Agent. VIPL is an acronym for Virtual Interface Provider Library.*

VI Provider The combination of a VI NIC and a VI Kernel Agent. Together, these two components instantiate a Virtual Interface.

Work Queue A posted list of Descriptors being processed by a VI NIC. Every VI has two Work Queues: a send queue and a receive queue. The combination of the Work Queue selected by a post operation and the operation type indicated by the Descriptor determine the exact type of data movement that the VI NIC will perform.

2. Virtual Interface Provider Library (VIPL)

2.1. Overview

This section describes a reference interface to the VI Architecture, referred to as VIPL. The material is presented in the form of groups of related routines, followed by definitions of data structures, constants and error codes. Semantic clarifications on specific routines are provided at the end of respective sections, whenever deemed necessary.

2.2. Conformance Phases

The conformance phases have been defined with input from ISVs and consultation with IHVs that are planning products developed to the VIPL interface. The ISV community can use the phases to determine which VI NIC products provide the functions needed for demonstrations and products and the VI NIC vendors can use this to understand the needs of the ISVs. The Intel VI conformance test suite will provide the means to report the highest phase of conformance to this guide attained by a VI NIC offering.

Three phases have been defined and have been named as follows: Early Adopter, Functional and Full Conformance. The Early Adopter phase has been broadly defined as the set of VIPL API calls and other functionality in the API required for demonstrations and application prototyping. The Functional phase contains the additional functions that ISVs plan to use to develop products. Finally, Full Conformance is defined to be the full set of API and functions defined in this guide.

The details of each of the phases are contained in the following sections.

2.2.1. Early Adopter

The following are the calls and functionality required to meet the requirements of the Early Adopter phase:

VipOpenNic
VipCloseNic

VipCreateVi
VipDestroyVi

VipConnectWait
VipConnectAccept
VipConnectReject
VipConnectRequest
VipDisconnect

VipQueryNic
VipQueryVi
VipQueryMem

VipPostSend
VipSendDone
VipSendWait
VipPostRecv
VipRecvDone
VipRecvWait

VipRegisterMem
VipDeregisterMem

Additional functionality requirements: Reliable Delivery and RDMA Write. (Although Reliable Delivery is not a requirement in the VI Architecture Specification, many of the ISVs have stated they require Reliable Delivery).

Note: It is acceptable to ignore the memory protection tags in the memory protection checks for the early adopter release.

2.2.2. Functional

VipConnectPeerRequest
VipConnectPeerDone
VipConnectPeerWait

VipCQDone
VipCQWait
VipCreateCQ
VipDestroyCQ
VipResizeCQ

VipCreatePtag
VipDestroyPtag

VipNSInit
VipNSGetHostByName
VipNSGetHostByAddr
VipNSShutdown

VipErrorCallback

Additional Requirements: Protection Tag support

2.2.3. Full Conformance

VipSendNotify
VipRecvNotify
VipCQNotify

VipSetViAttributes
VipSetMemAttributes

VipQuerySystemManagementInfo

Additional Requirements: Unreliable Delivery

RDMA Read capability is not a requirement to be conformant to any phase, but some ISVs have expressed an interest in using it.

2.3. Multi-Threading Considerations

Multi-threaded applications or transport layers above VIPL layer commonly employ locks to protect their own data structures. A non thread-safe version of VIPL can avoid overhead and deadlock conditions from redundant locking when working with these applications. Applications can determine whether the library supplied is thread-safe or not by checking the ThreadSafe field in the NIC Attributes. (Section 3.7.1 explains how to retrieve NIC Attributes). General guidelines for multi-threaded usage of a non thread-safe version of VIPL are described below.

Running on a multi-processor system in itself does not require locking as each VI is owned by a single process. Explicit locking is required only when multiple threads are accessing the same queue within a VI. For example, locks are required when two threads are sending to the same VI, but not if one thread is sending and the other is receiving. Explicit locking is also necessary between threads when manipulating the same completion queue. The VI provider must ensure that explicit application level locking is not required to ensure correct operation when multiple threads are accessing different queues within a VI. Since the creation and destruction of a VI affect all the associated queues, locks for all queues must be taken before performing these operations if multiple threads are accessing the same VI.

A non-thread-safe implementation of VIPL does not provide thread synchronization to access the VI work queues and completion queues in order to reduce the overheads in the speed paths of data movement operations. This is the only difference between thread-safe and non-thread-safe implementations of VIPL. Both thread-safe and non-thread-safe VIPL implementations are expected to provide thread synchronization for all other resources. This means the VI Application needs to only manage locks for the VI work queues and completion queues when using a non-thread-safe VIPL. For more information, see Section 6.4.

2.4. Reliability Considerations

Section 2.5.2 Reliable Delivery in the VI Architecture Specification is not clear regarding how asynchronous errors affect Descriptor processing, especially receive Descriptor processing for Reliable Delivery VIs. A condition for a Reliable Delivery VI is the guarantee that all data submitted for transfer will arrive exactly once, intact and in the order submitted. In order to meet this guarantee, the VI Provider must detect transfers that are corrupted or delivered out of order at the receiver in a synchronous manner. Note that detecting an out of order transfer will also catch a lost transfer.

When an error is detected, the VI transitions immediately to the Error state before processing any more Descriptors and the VI Provider drops the connection. All Descriptors pending or posted while in the Error state are marked completed in error. This ensures that no subsequent receive Descriptors complete successfully after the Descriptor where the error occurred.

It is possible for one or more send Descriptors to have completed successfully before receiving a non-local asynchronous error notification. The status of transfers initiated by these Descriptors is unknown.

The send side Descriptor processing is the primary difference between Reliable Delivery and Reliable Reception. Reliable Reception guarantees that no subsequent send Descriptors are processed after the Descriptor that generated the error.

3. VIPL Calls

3.1. Hardware Connection

3.1.1. VipOpenNic

Synopsis

```
VIP_RETURN
    VipOpenNic(
        IN      const VIP_CHAR      *DeviceName,
        OUT     VIP_NIC_HANDLE      *NicHandle
    )
```

Parameters

DeviceName: Symbolic name of the device (VI Provider instance) associated with the NIC.

NicHandle: Handle returned. The handle is used with the other functions to specify a particular instance of a VI NIC.

Description

VipOpenNic associates a process with a VI NIC, and provides a NIC handle to the VI Consumer. The NIC handle is used in subsequent functions in order to specify a particular NIC. *A process is allowed to open the same VI NIC multiple times. Each time a process calls VipOpenNic with the same device name, a different NIC handle is returned that references the same NIC*.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_ERROR_RESOURCE – An error was detected due to insufficient resources.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

3.1.2. VipCloseNic

Synopsis

```
VIP_RETURN
    VipCloseNic(
        IN      VIP_NIC_HANDLE      NicHandle
    )
```

Parameters

NicHandle: The NIC handle.

Description

VipCloseNic removes the association between the calling process and the VI NIC that was established via the corresponding *VipOpenNic* function.

When a VI NIC is closed, it is the responsibility of the VI Provider to clean up all resources associated with that NIC instance. This includes any resources allocated by the library and threads started on behalf of the NIC, such as error callback and notify threads.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – Caller specified an invalid NIC handle.

3.2. Endpoint Creation and Destruction

3.2.1. VipCreateVi

Synopsis

```
VIP_RETURN
        VipCreateVi(
                IN      VIP_NIC_HANDLE      NicHandle,
                IN      VIP_VI_ATTRIBUTES   *ViAttribs,
                IN      VIP_CQ_HANDLE       SendCQHandle,
                IN      VIP_CQ_HANDLE       RecvCQHandle,
                OUT     VIP_VI_HANDLE       *ViHandle
        )
```

Parameters

NicHandle: Handle of the associated VI NIC.

ViAttribs: The initial attributes to set for the new VI.

SendCQHandle: The handle of a Completion Queue. If a valid handle, the send Work Queue of this VI will be associated with the Completion Queue. If NULL, the send queue is not associated with any Completion Queue.

RecvCQHandle: The handle of a Completion Queue. If valid, the receive Work Queue of this VI will be associated with the Completion Queue. If NULL, the receive queue is not associated with any Completion Queue.

ViHandle: The handle for the newly created VI instance.

Description

VipCreateVi creates an instance of a Virtual Interface to the specified NIC.

The *ViAttribs* input parameter specifies the initial attributes for this VI instance.

The SendCQHandle and RecvCQHandle parameters allow the caller to associate the Work Queues of this VI with a Completion Queue. If one or both of the Work Queues are associated with a Completion Queue, the calling process cannot wait on that queue via *VipSendWait* or *VipRecvWait*.

When a new instance of a VI is created, it begins in the Idle state.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_ERROR_RESOURCE – *The operation failed due to* insufficient resources.

VIP_INVALID_PARAMETER – One of the input parameters was invalid.

VIP_INVALID_RELIABILITY_LEVEL – The requested reliability level attribute was invalid or not supported.

VIP_INVALID_MTU – The maximum transfer size attribute was invalid or not supported.

VIP_INVALID_QOS – The quality of service attribute was invalid or not supported.

VIP_INVALID_PTAG – The protection tag attribute was invalid.

VIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

3.2.2. VipDestroyVi

Synopsis

```
VIP_RETURN
    VipDestroyVi(
        IN    VIP_VI_HANDLE    ViHandle
    )
```

Parameters

ViHandle: The handle of the VI instance to be destroyed.

Description

VipDestroyVi tears down a Virtual Interface. A VI instance may only be destroyed if the VI is in the Idle state and all Descriptors on its work queues have been de-queued, otherwise an error is returned to the caller. Use of the destroyed handle in any subsequent operation will fail.

Returns

VIP_SUCCESS – Operation completed successfully

VIP_INVALID_PARAMETER – An invalid VI Handle was specified.

VIP_INVALID_STATE – The VI is not in the Idle state or there are still Descriptors posted on the work queues.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.3. Connection Management

This section illustrates the various connection management models available in VIPL and describes their semantics. Sections 3.3.1 to 3.3.4 provide the client/server connection management APIs. The client/server connection model is described in detail in the VI specification, and is intended for use by standard client/server applications. Section 3.3.5 describes the disconnect semantics for all the connection models. Sections 3.3.6 to 3.3.8 show the extended peer-to-peer connection management APIs. The peer-to-peer connection model is intended for use by applications having distributed modules that logically fit into a peer-to-peer relation rather than a client/server relation. The additional APIs for the peer-to-peer connection management model were added to VIPL based on feedback from application developers and ISVs. The client/server and peer-to-peer connection models are completely separate and applications must use the same model on both sides of the same connection. Both models can be used in an application as long as the rule requiring the same model on both sides of a connection is followed.

NOTE: In response to feedback from application developers and ISVs, the minimum value that is required for the maximum discriminator length is 16 bytes. This means that all applications are guaranteed that 16 byte discriminators can be used during connection setup for VIPL implementations that conform to this developer's guide.

The client-server connection model provides blocking semantics. The peer-to-peer connection model provides both blocking and non-blocking semantics. A typical peer-to-peer connection scenario starts with a peer initiating a connection request by calling VipConnectPeerRequest. In this scenario, there is no matching peer connection request with the given remote attributes at the

time the connection request is initiated by the first peer. At some point, before the timeout expires for the first peer's connect request, the second peer comes up and issues a VipConnectPeerRequest. The existence of a matching ConnectPeerRequest in the first peer is determined and acknowledged. Finally, the ConnectPeerRequest is accepted and both VI's transition to the Connected State. The timing diagram for this sequence is illustrated in Figure 1.

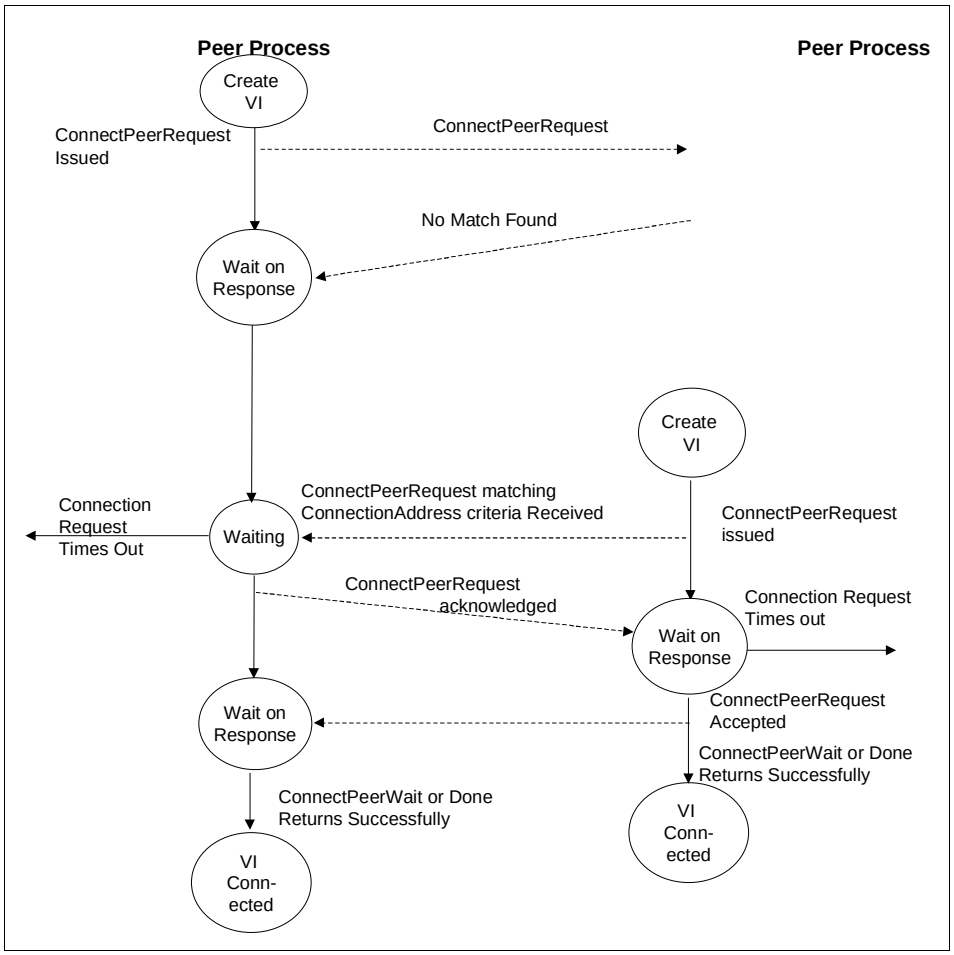

Figure 1: Peer-to-Peer Connection Model Timing Diagram

The detailed state diagram for a client/server connection model is provided in the VI Architecture Specification. A VI may be in one of four states throughout its life. The four states are Idle, Pending Connect, Connected, and Error. Transitions between states are driven by requests issued by the VI Consumer and network events. Requests that are not valid while a VI is in a given state, such as submitting a connect request while in the Pending Connect state, must be returned with an error by the VI Provider. The VI state diagram for peer-to-peer connections is exactly the same as the state diagram for client server connections except that the API call that causes state transitions between the Idle state and the Pending Connect state and between the Pending Connect state and the Connected state are peer-to-peer connection calls. The state diagram for the peer-to-peer connection model is shown in Figure 2.

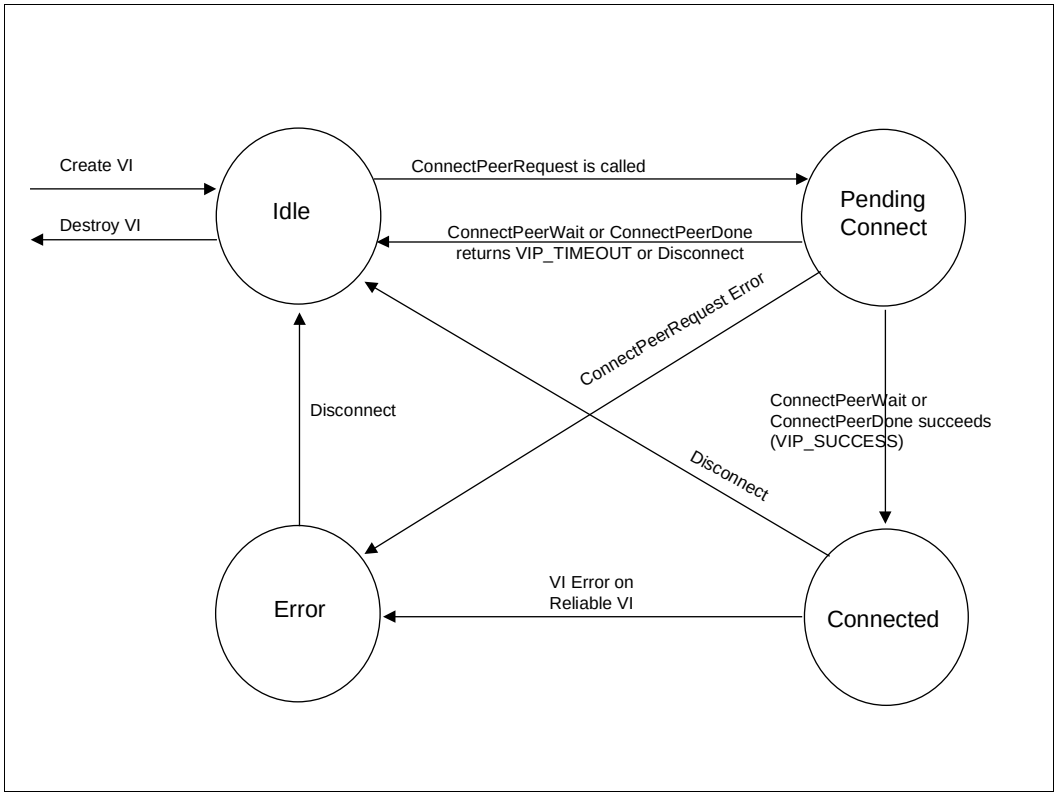

Figure 2: Peer-to-Peer Connection Model State Diagram

Net Address Matching Rules

This section describes the rules for matching connection requests for the client/server and the peer-to-peer model. Note that the matching rules for the two models are different.

In the client/server model, the client request must specify a host and a discriminator in the RemoteAddr parameter. The server waits for a connection request from any client node that matches the discriminator portion of the net address specified in RemoteAddr by the client application. The server application may accept or reject a connection request based on the client's remote address and other criteria, such as the VI attributes.

The peer-to-peer connection model requires that both the address and the discriminator portions of the net address specified in RemoteAddr match in order to complete the connection.

3.3.1. VipConnectWait

Synopsis

```
VIP_RETURN
        VipConnectWait(
                IN      VIP_NIC_HANDLE      NicHandle,
                IN      VIP_NET_ADDRESS     *LocalAddr,
                IN      VIP_ULONG           Timeout,
                OUT     VIP_NET_ADDRESS     *RemoteAddr,
                OUT     VIP_VI_ATTRIBUTES   *RemoteViAttribs,
                OUT     VIP_CONN_HANDLE     *ConnHandle
        )
```

Parameters

NicHandle: Handle for an instance of a VI NIC.

LocalAddr: Local network address on which to wait. *Only the discriminator portion of the net address is used to determine if the request matches. The host address portion must match the local NIC address.*

Timeout: The count, in milliseconds, that *VipConnectWait* will wait to complete before returning to the caller. VIP_INFINITE if no time-out is desired. A timeout of zero indicates immediate return.

RemoteAddr: The remote network address *(host address and discriminator)* that is requesting a connection. *The value of the network address returned is the same as the LocalAddr parameter supplied to the matching VipConnectRequest.*

RemoteViAttribs: The attributes of the remote VI endpoint that is requesting the connection.

ConnHandle: A handle to an opaque connection object subsequently used in calls to *VipConnectAccept* and *VipConnectReject*.

Description

VipConnectWait is used to look for incoming connection requests on the server side of the client/server connection model.

The caller passes in a local network address that is used to filter incoming connection requests. The format of the network address is VI Provider specific.

If a matching connection request is not found immediately, *VipConnectWait* will wait for a request until the Timeout period has expired.

If a connection request is found that matches the *discriminator* in the LocalAddress, the caller is returned the remote *network* address *including the discriminator* that is requesting a connection. The attributes of the remote endpoint that is requesting the connection and a connection handle to be used in subsequent calls to *VipConnectAccept* or *VipConnectReject* *are also returned to the caller.*

If the host portion of the LocalAddr parameter does not match the local NIC address (the LocalNicAddress field of the NicAttributes structure), then a VIP_INVALID_PARAMETER error is returned.

Returns

VIP_SUCCESS – The operation has successfully found a connection request.

VIP_TIMEOUT – The operation timed out, no connection request was found.

VIP_ERROR_RESOURCE – The operation failed due to a resource limit.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

3.3.2. VipConnectAccept

Synopsis

```
VIP_RETURN
        VipConnectAccept(
                IN      VIP_CONN_HANDLE   ConnHandle,
                IN      VIP_VI_HANDLE     ViHandle
        )
```

Parameters

ConnHandle: A handle to an opaque connection object created by *VipConnectWait*.

ViHandle: Instance of a local VI endpoint.

Description

VipConnectAccept is used to accept a connection request and associate the connection with a local VI endpoint. *This function is called on the server side of the client/server connection model.*

The caller passes in the handle of an Idle, unconnected VI endpoint to associate with the connection request. If the attributes of the local VI endpoint conflict with those of the remote endpoint, *VipConnectAccept* will fail. It is the function of the VI Provider to determine if the connection should succeed based on the attributes of the two endpoints. *Note: The VI attributes that must match when establishing a connection are ReliabilityLevel, MaxTransferSize and QoS.*

If *VipConnectAccept* fails, no explicit notification is sent to the remote end. The caller may choose to modify the attributes of the local VI endpoint, and try again. In order to reject a connection request, the VI Consumer must explicitly reject the connection request via the *VipConnectReject* function.

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed.

Returns

VIP_SUCCESS – The connection was successfully established.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_INVALID_ RELIABILITY_LEVEL – The reliability level attribute of the local endpoint conflicted with the reliability level of the remote VI.

VIP_INVALID_MTU – The maximum transfer size attribute of the local endpoint conflicted with the maximum transfer size of the remote endpoint.

VIP_INVALID_QOS – The quality of service attribute was invalid, or conflicted with the remote endpoint.

VIP_TIMEOUT - The connection could not be completed, possibly due to a timeout failure on the matching VipConnectRequest on the client node.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

VIP_INVALID_STATE - The specified VI endpoint is not in the Idle state.

VIP_NOT_REACHABLE - A network partition was detected.

3.3.3. VipConnectReject

Synopsis

```
VIP_RETURN
    VipConnectReject(
        IN      VIP_CONN_HANDLE    ConnHandle
    )
```

Parameters

ConnHandle: A handle to an opaque connection object created by *VipConnectWait*.

Description

VipConnectReject is used to reject a connection request. *This function is called on the server side of the client/server connection model.* Notification is sent to the remote end that the associated connection request was rejected.

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed.

Returns

VIP_SUCCESS – The operation completed successfully.

VIP_INVALID_PARAMETER – The ConnHandle parameter was invalid.

VIP_ERROR_RESOURCE - The operation failed due insufficient resources.

VIP_NOT_REACHABLE - A network partition was detected.

3.3.4. VipConnectRequest

Synopsis

```
VIP_RETURN
    VipConnectRequest(
        IN      VIP_VI_HANDLE       ViHandle,
        IN      VIP_NET_ADDRESS     *LocalAddr,
        IN      VIP_NET_ADDRESS     *RemoteAddr,
        IN      VIP_ULONG           Timeout,
        OUT     VIP_VI_ATTRIBUTES   *RemoteViAttribs
    )
```

Parameters

ViHandle: Handle for the local VI endpoint.

LocalAddr: Local network address. *The local address is used solely for naming purposes and is not used for matching by VipConnectWait. The host address portion of this network address must match the client NIC address. The discriminator portion is passed unchanged to the RemoteAddr structure returned from the matching VipConnectWait.*

RemoteAddr: The remote network address. *The remote network address must contain both the host address and discriminator.*

Timeout: The count, in milliseconds, that *VipConnectRequest* will wait for connection to complete before returning to the caller, VIP_INFINITE if no time-out is desired. A timeout value of zero is invalid.

RemoteViAttribs: The attributes of the remote endpoint if successful.

Description

VipConnectRequest requests that a connection be established between the local VI endpoint, and a remote endpoint. *This function is called on the client side of the client/server connection model.* The user specifies a local and remote network address for the connection. *Only the remote network address is used by the server to match to a VipConnectWait.*

When a connection is successfully established, the local address is bound to the local VI endpoint, and the attributes of the remote endpoint are returned to the caller. The attributes of the

remote endpoint allow the caller to determine whether the indicated RDMA operations can be executed on the resulting connection.

If the remote end rejects the connection *explicitly by calling VipConnectReject,* a rejection error is returned. If a connection cannot be established before the specified Timeout period, a timeout error is returned. Specifying a timeout value of zero is invalid and will result in an immediate *VIP_INVALID_PARAMETER* error. *If a VipConnectAccept or the VipConnectReject response is not returned within the specified timeout period (including a non-functional server or interconnect), VIP_TIMEOUT is returned after the timeout period expires. If the server is responding but is not waiting for a connection that matches the discriminator specified in RemoteAddr, or is not in the right state to handle connection requests, a VIP_NO_MATCH error is returned.*

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned immediately if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed.

If the host portion of the LocalAddr parameter does not match the local NIC address (the LocalNicAddress field of the NicAttributes structure), then a VIP_INVALID_PARAMETER error is returned.

Returns

VIP_SUCCESS – The connection was successfully established.

VIP_TIMEOUT – The connection operation timed out or the server is not functional.

VIP_ERROR_RESOURCE – The connection operation failed due to *insufficient resources*.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_REJECT – The connection was rejected by the remote end.

VIP_NO_MATCH - The server is up and is not waiting for a connection request with the specified discriminator.

VIP_INVALID_STATE - The specified VI endpoint is not in the Idle state.

VIP_NOT_REACHABLE - A network partition was detected.

3.3.5. VipDisconnect

Synopsis

```
VIP_RETURN
        VipDisconnect(
                IN  VIP_VI_HANDLE     ViHandle
        )
```

Parameters

ViHandle: Instance of a connected Virtual Interface endpoint.

Description

VipDisconnect is used to terminate a connection. When the local endpoint is disconnected, it stops processing of all posted Descriptors, all *pending (not completed*) Descriptors are marked completed because of disconnection *with a Descriptor Flushed error,* and the local endpoint transitions to the Idle state. *When* the remote endpoint causes a connection to terminate by closing the endpoint or by calling *VipDisconnect*, an asynchronous error callback happens indicating the disconnected connection. *The node that initiated the VipDisconnect will not get an error callback.* The local client should call *VipDisconnect* to reset the disconnected connection from *the* Error state to *the* Idle state.

VipDisconnect can be called in any VI state to cause pending Descriptors on the VI to be completed with the Descriptor Flushed error bit set and transition the VI to the Idle state. Specifically, VipDisconnect can be called in the Idle state to force pending receive Descriptors to be completed in error so they can be de-queued prior to calling VipDestroyVi.

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed.

Returns

VIP_SUCCESS – The disconnect was successful.

VIP_INVALID_PARAMETER – The ViHandle parameter was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

VIP_NOT_REACHABLE - A network partition was detected.

3.3.6. VipConnectPeerRequest

Synopsis

```
VIP_RETURN
    VipConnectPeerRequest(
        IN    VIP_VI_HANDLE      ViHandle,
        IN    VIP_NET_ADDRESS    *LocalAddr,
        IN    VIP_NET_ADDRESS    *RemoteAddr,
        IN    VIP_ULONG          Timeout
    )
```

Parameters

ViHandle: *Handle for the local VI endpoint.*

Local Addr: *Local network address. The local address is used solely for administrative purposes and is not used for matching connection requests. The host portion of this network address must match the NIC's address.*

RemoteAddr: *The remote network address.*

Timeout: *The count, in milliseconds, that VipConnectPeerRequest will wait for connection to complete. VIP_INFINITE if no time-out is desired. A timeout value of zero results in a VIP_INVALID_PARAMETER error return.*

Description

VipConnectPeerRequest posts a request that a connection be established between the local VI endpoint and a remote VI endpoint. The user specifies local and remote network addresses as part of the connection request. This call returns as soon as the connection request is initiated. The caller of VipConnectPeerRequest can check the status of the connection request or wait for connection completion by calling VipConnectPeerDone or VipConnectPeerWait respectively.

If a connection is successfully established, the local address is bound to the local VI endpoint and the attributes of the remote VI endpoint are returned to the caller when the connect completion status is delivered. If a connection cannot be established before the specified Timeout period, a VIP_TIMEOUT error is returned in the return status of VipConnectPeerDone or VipConnectPeerWait.

If the host portion of the LocalAddr parameter does not match the local NIC address (specified in the LocalNicAddress field of the NicAttributes structure), then a VIP_INVALID_PARAMETER error is returned.

Returns

VIP_SUCCESS - The connection request was queued.

VIP_ERROR_RESOURCE - The request failed due to insufficient resources.

VIP_INVALID_STATE - The specified VI endpoint is not in the Idle state.

VIP_INVALID_PARAMETER - One of the parameters was invalid.

3.3.7. VipConnectPeerDone

Synopsis

```
VIP_RETURN
    VipConnectPeerDone(
        IN   VIP_VI_HANDLE          ViHandle,
        OUT  VIP_VI_ATTRIBUTES      *RemoteViAttribs
    )
```

Parameters

ViHandle: Handle for the local VI endpoint.

RemoteVIAttribs: The attributes of the remote VI endpoint if the connection was successfully completed.

Description

VipConnectPeerDone is called by a client to determine the results of a previously posted VipConnectPeerRequest call on the specified VI handle, without blocking the calling thread.

If a connection is successfully established, the attributes of the remote endpoint are returned to the caller. The attributes of the remote endpoint allow the caller to determine whether/which RDMA operations can be executed on the resulting connection. Note: The VI attributes that must match are ReliabilityLevel, MaxTransferSize and QoS.

If the connection was not successfully established within the Timeout period specified in VipConnectPeerRequest, a VIP_TIMEOUT error is returned. VipConnectPeerDone returns VIP_NOT_DONE if the connection operation is still in progress.

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed.

Returns

VIP_SUCCESS – The connection completed successfully. Attributes of the remote endpoint are returned through the second function parameter.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_INVALID_ RELIABILITY_LEVEL – The reliability level attribute of the local endpoint conflicted with the reliability level of the remote VI.

VIP_INVALID_MTU – The maximum transfer size attribute of the local endpoint conflicted with the maximum transfer size of the remote endpoint.

VIP_INVALID_QOS – The quality of service attribute was invalid, or conflicted with the remote endpoint.

VIP_TIMEOUT – The connection request timeout expired before a successful connection completion

VIP_NOT_DONE - The connection operation is still in progress.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

VIP_INVALID_STATE - The specified VI endpoint is not in the Pending Connect state.

VIP_NOT_REACHABLE - A network partition was detected.

3.3.8. VipConnectPeerWait

Synopsis

```
VIP_RETURN
    VipConnectPeerWait(
        IN    VIP_VI_HANDLE          ViHandle,
        OUT   VIP_VI_ATTRIBUTES      *RemoteViAttribs
    )
```

Parameters

ViHandle: Handle for the local VI endpoint.

RemoteVIAttribs: The attributes of the remote VI endpoint if the connection was successfully completed

Description

VipConnectPeerWait is used by a client to determine the results of a previously posted VipConnectPeerRequest call on the specified VI handle, blocking the calling thread until the result of the connection request is available.

If a connection is successfully established, the attributes of the remote endpoint are returned to the caller. The attributes of the remote endpoint allow the caller to determine whether/which RDMA operations can be executed on the resulting connection. Note: The VI attributes that must match when establishing a connection are ReliabilityLevel, MaxTransferSize and QoS.

If the connection was not successfully established within the Timeout period specified in VipConnectPeerRequest, a VIP_TIMEOUT error is returned. VipConnectPeerWait blocks until a connection is established, the timeout expires or an error is detected.

For NICs that can detect network partitions, a VIP_NOT_REACHABLE error is returned if a network partition is detected. For Reliable Delivery and Reliable Reception, the VI transitions to the Error State and any pending receive descriptors are marked as flushed

Returns

VIP_SUCCESS – The connection completed successfully; Attributes of the remote endpoint are returned through the second function parameter.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_INVALID_ RELIABILITY_LEVEL – The reliability level attribute of the local endpoint conflicted with the reliability level of the remote VI.

VIP_INVALID_MTU – The maximum transfer size attribute of the local endpoint conflicted with the maximum transfer size of the remote endpoint.

VIP_INVALID_QOS – The quality of service attribute was invalid, or conflicted with the remote endpoint.

VIP_TIMEOUT – The connection request timeout expired before a successful connection completion.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

VIP_INVALID_STATE - The specified VI endpoint is not in the Pending Connect state.

VIP_NOT_REACHABLE - A network partition was detected.

3.4. Memory protection and registration

3.4.1. VipCreatePtag

Synopsis

```
VIP_RETURN
        VipCreatePtag(
                IN      VIP_NIC_HANDLE              NicHandle,
                OUT     VIP_PROTECTION_HANDLE       *Ptag
        )
```

Parameters

NicHandle: The NIC handle associated with the protection tag.

Ptag: The new protection tag.

Description

The *VipCreatePtag* function creates a new protection tag for the calling process. The protection tag is subsequently associated with VI endpoints via the *VipCreateVi* function, as well as memory regions via the *VipRegisterMem* function. A process may request multiple protection tags.

For all memory references by the VI Provider, including Descriptors and message buffers, the protection tag of the VI instance, and the memory region, must match in order to pass the memory protection check.

The Protection Tag is an element in the VI attributes data structure and the Memory Region Attributes data structure. The protection tag of a memory region and/or a VI can be changed by changing their attributes.

Returns

VIP_SUCCESS – The memory protection tag was successfully created.

VIP_ERROR_RESOURCE – The operation failed due to *insufficient resources*.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

3.4.2. VipDestroyPtag

Synopsis

```
VIP_RETURN
        VipDestroyPtag(
                IN      VIP_NIC_HANDLE              NicHandle,
                IN      VIP_PROTECTION_HANDLE       Ptag
        )
```

Parameters

NicHandle: The NIC handle associated with the protection tag.

Ptag: The protection tag.

Description

The *VipDestroyPtag* function destroys a protection tag.

If the specified protection tag is associated with either a VI instance or a registered memory region at the time of the call, an error is returned.

Returns

VIP_SUCCESS – The memory protection tag was successfully destroyed.

VIP_ERROR_RESOURCE – A VI instance or a registered memory region is still associated with the specified protection tag *or the operation failed due to insufficient resources.*

VIP_INVALID_PARAMETER – One of the input parameters was invalid.

3.4.3. VipRegisterMem

Synopsis

```
VIP_RETURN
    VipRegisterMem(
        IN    VIP_NIC_HANDLE          NicHandle,
        IN    VIP_PVOID               VirtualAddress,
        IN    VIP_ULONG               Length,
        IN    VIP_MEM_ATTRIBUTES      *MemAttribs,
        OUT   VIP_MEM_HANDLE          *MemoryHandle
    )
```

Parameters

NicHandle: Handle for a currently open NIC.

VirtualAddress: Starting address of the memory region to be registered.

Length: The length, in bytes, of the memory region.

MemAttribs: The memory attributes to associate with the memory region.

MemoryHandle: If successful, the new memory handle for the region, otherwise NULL.

Description

VipRegisterMem allows a process to register a region of memory with a VI NIC. Memory used to hold Descriptors or data buffers must be registered with this function.

The user may specify an arbitrary size region of memory, with arbitrary alignment, but the actual area of memory registered will be registered on page granularity. Registered pages are locked into physical memory.

The memory attributes include the Protection Tag, and the RDMA enable bits that are initially associated with the memory region.

Descriptors and data buffers contained within registered memory can be used by any VI with a matching protection tag that is owned by the process**.** A new memory handle is generated for each region of memory that is registered by a process.

The EnableRdmaWrite memory attribute can be used to ensure that no remote process can modify a region of memory, this could be particularly useful to protect regions of memory that contain Descriptors (control information). The EnableRdmaRead parameter can be used to ensure that no remote process can read a particular region of memory.

Note that the implementation of *VipRegisterMem* should always check for read-only pages of memory and not allow modification to those pages by the VI Hardware.

A Length parameter of zero will result in a VIP_INVALID_PARAMETER error.

The contents of the memory region being registered are not altered. The memory region must have been previously allocated by the VI Consumer.

Returns

VIP_SUCCESS – The memory region was successfully registered.

VIP_ERROR_RESOURCE – The registration operation failed due to *insufficient resources*.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_INVALID_PTAG – The protection tag attribute was invalid.

VIP_INVALID_RDMAREAD – the attributes requested the memory region be enabled for RDMA Read, but the VI Provider does not support it.

3.4.4. VipDeregisterMem

Synopsis

```
VIP_RETURN
    VipDeregisterMem(
        IN    VIP_NIC_HANDLE     NicHandle,
        IN    VIP_PVOID          VirtualAddress,
        IN    VIP_MEM_HANDLE     MemoryHandle
    )
```

Parameters

NicHandle: The handle for the NIC that owns the memory region being de-registered.

VirtualAddress: Address of the region of memory to be de-registered.

MemoryHandle: Memory handle for the region; obtained from a previous call to *VipRegisterMem*.

Description

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

Returns

VIP_SUCCESS – The memory region was successfully de-registered.

VIP_INVALID_PARAMETER – One or more of the parameters was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.5. Data transfer and completion operations

3.5.1. VipPostSend

Synopsis

```
VIP_RETURN
        VipPostSend(
                IN      VIP_VI_HANDLE       ViHandle,
                IN      VIP_DESCRIPTOR      *DescriptorPtr,
                IN      VIP_MEM_HANDLE      MemoryHandle
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Pointer to a Descriptor to be posted on the send queue.

MemoryHandle: The handle for the memory region of the Descriptor being posted.

Description

VipPostSend adds a Descriptor to the tail of the send queue of a VI, notifies the NIC that a new Descriptor is available, and returns immediately.

Returns

VIP_SUCCESS – The send Descriptor was successfully posted.

VIP_INVALID_PARAMETER – The VI handle was invalid.

3.5.2. VipSendDone

Synopsis

```
VIP_RETURN
        VipSendDone(
                IN      VIP_VI_HANDLE       ViHandle,
                OUT     VIP_DESCRIPTOR      **DescriptorPtr
        )
```

Parameters

ViHandle: Instance of a Virtual Interface.

DescriptorPtr: Address of the Descriptor that has completed, if any.

Description

VipSendDone checks the Descriptor at the head of the send queue to see if it has been marked complete. If the operation has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is returned. *A VIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status or the send queue is empty. If the send queue is empty, DescriptorPtr is set to NULL. VipSendDone is a non-blocking call. In particular, VipSendDone is not allowed to block behind a VipSendWait, even in a thread-safe implementation.*

Returns

VIP_SUCCESS – A completed Descriptor was returned with a successful completion status.

VIP_DESCRIPTOR_ERROR - If the send queue is empty, the Descriptor pointer is set to NULL, otherwise, a completed Descriptor is returned with an error completion status.

VIP_NOT_DONE – No completed Descriptor was found.

VIP_INVALID_PARAMETER – The VI handle was invalid.

3.5.3. VipSendWait

Synopsis

```
VIP_RETURN
    VipSendWait(
        IN    VIP_VI_HANDLE      ViHandle,
        IN    VIP_ULONG          TimeOut,
        OUT   VIP_DESCRIPTOR     **DescriptorPtr
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

TimeOut: The count, in milliseconds, before control is returned to the caller, VIP_INFINITE if no time-out is desired.

DescriptorPtr: Address of the Descriptor that has completed.

Description

VipSend Wait checks the Descriptor on the head of the send queue to see if it has been marked complete. If the send has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is immediately returned. *A VIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status. If the send queue is empty a VIP_DESCRIPTOR_ERROR is returned and DescriptorPtr is set to NULL.*

If the Descriptor at the head of the send queue has not been marked complete, *VipSendWait* blocks the caller until a Descriptor is completed, or until the specified timeout has expired.

VipSendWait cannot be used to block on a send queue that has been associated with a completion queue. Refer to *VipCQWait* for a more complete description.

Returns

VIP_SUCCESS – A completed Descriptor was found on the send queue with a successful completion status.

VIP_DESCRIPTOR_ERROR - If the send queue is empty, the Descriptor pointer is set to NULL, otherwise a completed Descriptor is returned with an error completion status.

VIP_INVALID_PARAMETER – The VI handle was invalid.

VIP_TIMEOUT – The timeout expired and no completed Descriptor was found.

VIP_ERROR_RESOURCE – This send queue is associated with a completion queue or the operation failed due to insufficient resources.

3.5.4. VipPostRecv

Synopsis

```
VIP_RETURN
        VipPostRecv(
                IN      VIP_VI_HANDLE       ViHandle,
                IN      VIP_DESCRIPTOR      *DescriptorPtr,
                IN      VIP_MEM_HANDLE      MemoryHandle
        )
```

Parameters

ViHandle:	Instance of a Virtual Interface.
DescriptorPtr:	Pointer to a Descriptor to be posted on the receive queue.
MemoryHandle:	The handle for the memory region of the Descriptor being posted.

Description

VipPostRecv adds a Descriptor to the tail of the receive queue of the specified VI, notifies the NIC that a new Descriptor is available, and returns immediately.

Returns

VIP_SUCCESS – The receive Descriptor was successfully posted.

VIP_INVALID_PARAMETER – The VI handle was invalid.

3.5.5. VipRecvDone

Synopsis

```
VIP_RETURN
        VipRecvDone(
                IN      VIP_VI_HANDLE       ViHandle,
                OUT     VIP_DESCRIPTOR      **DescriptorPtr
        )
```

Parameters

ViHandle:	Instance of a Virtual Interface.
DescriptorPtr:	Address of the Descriptor that has completed, if any.

Description

VipRecvDone checks the Descriptor on the head of the receive queue to see if it has been marked complete. If the receive has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is returned*. A VIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status or the receive queue is empty. If the receive queue is empty, DescriptorPtr is set to NULL. VipRecvDone is a non-blocking call. In particular, VipRecvDone is not allowed to block behind a VipRecvWait, even in a thread-safe implementation.*

Returns

VIP_SUCCESS – A completed receive Descriptor was returned with a successful completion status.

VIP_DESCRIPTOR_ERROR - If the receive queue is empty, the Descriptor pointer is set to NULL, otherwise a completed Descriptor is returned with an error completion status.

VIP_NOT_DONE – No completed Descriptor was found.

VIP_INVALID_PARAMETER – The VI handle was invalid.

3.5.6. VipRecvWait

Synopsis

```
VIP_RETURN
    VipRecvWait(
        IN      VIP_VI_HANDLE       ViHandle,
        IN      VIP_ULONG           Timeout,
        OUT     VIP_DESCRIPTOR      **DescriptorPtr
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

TimeOut: The count, in milliseconds, before control is returned to the caller, VIP_INFINITE if no time-out is desired.

DescriptorPtr: Address of the Descriptor that has completed.

Description

VipRecvWait checks the Descriptor on the head of the receive queue to see if it has been marked complete. If the receive has completed, the Descriptor is removed from the head of the queue and the address of the Descriptor is immediately returned. *A VIP_DESCRIPTOR_ERROR is returned if the operation completed with errors returned in the Descriptor status or the receive queue is empty. If the receive queue is empty, DescriptorPtr is set to NULL.* If the Descriptor at the head of the receive queue has not been marked complete, *VipRecvWait* blocks the caller until a Descriptor is completed, or until the specified timeout has expired.

VipRecvWait cannot be used to block on a receive queue that has been associated with a completion queue. Refer to *VipCQWait* for a more complete description.

Returns

VIP_SUCCESS – A completed receive Descriptor was found on the receive queue with a successful completion status.

VIP_DESCRIPTOR_ERROR - If the receive queue is empty, the Descriptor pointer is set to NULL, otherwise a completed Descriptor is returned with an error completion status.

VIP_INVALID_PARAMETER – The VI handle was invalid.

VIP_TIMEOUT – The timeout expired and no completed Descriptor was found.

VIP_ERROR_RESOURCE – This receive queue is associated with a completion queue *or the operation failed due to insufficient resources.*

3.5.7. VipCQDone

Synopsis

```
VIP_RETURN
        VipCQDone(
                IN      VIP_CQ_HANDLE       CQHandle,
                OUT     VIP_VI_HANDLE       *ViHandle,
                OUT     VIP_BOOLEAN         *RecvQueue
        )
```

Parameters

CQHandle: The handle of the Completion Queue.

ViHandle: The handle of the VI endpoint associated with the completion, if the return status indicates success. Undefined otherwise.

RecvQueue: If *VIP_TRUE*, indicates that the completion was associated with the receive queue of the VI. If *VIP_FALSE*, *indicates that the completion was associated with the send queue of the VI. Undefined if the returned status does not indicate success.*

Description

VipCQDone polls the specified Completion Queue for a completion entry (a completed operation). If a completion entry is found, it returns the VI handle, along with a flag to indicate whether the completed Descriptor resides on the send or receive queue*. VipCQDone is a non-blocking call. In particular, VipCQDone is not allowed to block behind a VipCQWait, even in a thread-safe implementation.*

It is up to the calling process to subsequently invoke the appropriate function to actually de-queue the completed Descriptor. The completed Descriptor may only be de-queued by the function *VipSendDone* or *VipRecvDone*. *VipCQDone* only dequeues the completion entry from the Completion Queue.

It is possible for a process to have multiple threads, some of which are waiting for completions on a Completion Queue, and others polling the Work Queues of an associated VI. In this case, the caller must be prepared for the case where the Completion Queue indicated a completion, but a subsequent call to de-queue the Descriptor fails.

If a process has associated a Work Queue of a VI instance with a Completion Queue, it may not block directly on that Work Queue via the *VipSendWait* or *VipRecvWait* functions.

Returns

VIP_SUCCESS – A completion entry was found on the Completion Queue.

VIP_NOT_DONE – No completion entries are on the Completion Queue.

VIP_INVALID_PARAMETER – The Completion Queue handle was invalid.

3.5.8. VipCQWait

Synopsis

```
VIP_RETURN
        VipCQWait(
                IN      VIP_CQ_HANDLE       CQHandle,
                IN      VIP_ULONG           TimeOut,
                OUT     VIP_VI_HANDLE       *ViHandle,
                OUT     VIP_BOOLEAN         *RecvQueue
        )
```

Parameters

CQHandle: The handle of the Completion Queue.

TimeOut: The number of milliseconds to block before returning to the caller, VIP_INFINITE if no time-out is desired.

ViHandle: Returned to the caller. The handle of the VI endpoint associated with the completion if returned status indicates success.

RecvQueue: If *VIP_TRUE*, indicates that the completion was associated with the receive queue of the VI. If *VIP_FALSE*, indicates that the completion was associated with the send queue of the VI. Undefined if the returned status does not indicate success.

Description

VipCQWait polls the specified completion queue for a completion entry (a completed operation). If a completion entry was found, it immediately returns the VI handle, along with a flag to indicate the send or receive queue, where the completed Descriptor resides.

If no completion entry is found, the caller is blocked until a completion entry is generated, or until the TimeOut value expires.

It is up to the calling process to subsequently invoke the appropriate function to actually de-queue the completed Descriptor. The completed Descriptor may only be de-queued by the function *VipSendDone* or *VipRecvDone*. *VipCQWait only dequeues the completion entry from the Completion Queue.*

It is possible for a process to have multiple threads, some of which are checking for completions on a *Completion Queue,* and others polling the work queues of an associated VI directly. In this case, the caller must be prepared for the case where the *Completion Queue* indicated a completion, but a subsequent call to de-queue the Descriptor fails.

If a process has associated a work queue of a VI instance with a completion queue, it may not block directly on that work queue via the *VipSendWait* or *VipRecvWait* functions. If this is attempted, the function returns *VIP_ERROR_RESOURCE.*

Returns

VIP_SUCCESS – A completion entry was found on the *Completion Queue*.

VIP_INVALID_PARAMETER – The *Completion Queue* handle was invalid.

VIP_TIMEOUT – The request timed out and no completion entry was found.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.5.9. VipSendNotify

Synopsis

```
VIP_RETURN
    VipSendNotify(
        IN    VIP_VI_HANDLE      ViHandle,
        IN    VIP_PVOID          Context,
        IN  void(*Handler)(
            IN    VIP_PVOID          Context,
            IN    VIP_NIC_HANDLE     NicHandle,
            IN    VIP_VI_HANDLE      ViHandle,
            IN    VIP_DESCRIPTOR     *DescriptorPtr
        )
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

Context: Data to be passed through to the Handler as a parameter. Not used by the VI Provider.

Handler: Address of the user-defined function to be called when a single Descriptor completes. This function is not guaranteed to run in the context of the calling thread. The handler function is called with the following input parameters:

- Context: Data passed through from the function call. Not used by the VI Provider.
- NicHandle: Handle of the NIC.
- ViHandle: Instance of a Virtual Interface.
- DescriptorPtr: Address of the Descriptor that has completed.

Description

VipSendNotify is used by the VI Consumer to request that *a* Handler routine be called when a Descriptor completes.

VipSendNotify checks the Descriptor on the head of the send queue to see if it has been marked complete. If the Descriptor has completed, it is removed from the head of the queue and the Handler *associated with the Descriptor* is invoked with the address of the completed Descriptor as a parameter.

If the Descriptor at the head of the send queue has not been marked complete, *VipSendNotify* will enable interrupts for the given VI Send Queue. When a Descriptor is completed, the Handler will be invoked with the address of the completed Descriptor as a parameter.

This registration is only associated with the VI Send Queue for a single completed Descriptor. In order for the Handler to be invoked multiple times, the function must be called multiple times. *Multiple handlers can be registered at one time and will be queued in Descriptor order.*

Destruction of the VI will result in cancellation of any pending function calls.

VipSendNotify cannot be used to block on a send queue that has been associated with a Completion Queue. Refer to *VipCQNotify* for a more complete description.

If the send queue is empty at the time that VipSendNotify is called, the function returns VIP_DESCRIPTOR_ERROR. The Handler is called for outstanding Notify requests with the Descriptor pointer set to NULL if the send queue becomes empty.

Returns

VIP_SUCCESS – The routine was successfully completed.

VIP_INVALID_PARAMETER – The VI handle was invalid or the function call address was invalid.

VIP_DESCRIPTOR_ERROR - The send queue was empty when VipSendNotify was called.

VIP_ERROR_RESOURCE – The send queue of the VI is associated with a Completion Queue *or the operation failed due to insufficient resources.*

3.5.10. VipRecvNotify

Synopsis

```
VIP_RETURN
    VipRecvNotify(
        IN    VIP_VI_HANDLE      ViHandle,
        IN    VIP_PVOID          Context,
        IN  void(*Handler)(
            IN    VIP_PVOID          Context,
            IN    VIP_NIC_HANDLE     NicHandle,
            IN    VIP_VI_HANDLE      ViHandle,
            IN    VIP_DESCRIPTOR     *DescriptorPtr
        )
    )
```

Parameters

ViHandle: Instance of a Virtual Interface.

Context: Data to be passed through to the Handler as a parameter. Not used by the VI Provider.

Handler: Address of the user-defined function to be called when a single Descriptor completes. This function is not guaranteed to run in the context of the calling thread. The handler function is called with the following input parameters:

- Context: Data passed through from the function call. Not used by the VI Provider.
- NicHandle: Handle of the NIC.
- ViHandle: Instance of a Virtual Interface.
- DescriptorPtr: Address of the Descriptor that has completed.

Description

VipRecvNotify is used by the VI Consumer to request that *a* Handler routine be called when a Descriptor completes.

VipRecvNotify checks the Descriptor on the head of the receive queue to see if it has been marked complete. If the Descriptor has completed, it is removed from the head of the queue and the Handler *associated with the Descriptor* is invoked with the address of the completed Descriptor as a parameter.

If the Descriptor at the head of the receive queue has not been marked complete, *VipRecvNotify* will enable interrupts for the given VI Receive Queue. When a Descriptor is completed, the Handler will be invoked with the address of the completed Descriptor as a parameter.

This registration is only associated with the VI Receive Queue for a single completed Descriptor. In order for the Handler to be invoked multiple times, the function must be called multiple times. *Multiple handlers can be registered at one time and will be queued in Descriptor order.*

Destruction of the VI will result in cancellation of any pending function calls.

VipRecvNotify cannot be used to block on a receive queue that has been associated with a Completion Queue. Refer to *VipCQNotify* for a more complete description.

If the receive queue is empty at the time VipRecvNotify is called, the function returns VIP_DESCRIPTOR_ERROR. The Handler is called for outstanding Notify requests with the Descriptor pointer set to NULL if the receive queue becomes empty.

Returns

VIP_SUCCESS – The routine was successfully completed.

VIP_INVALID_PARAMETER – The VI handle was invalid or the function call address was invalid.

VIP_DESCRIPTOR_ERROR - The receive queue was empty when VipRecvNotify was called.

VIP_ERROR_RESOURCE – The receive queue of the VI is associated with a Completion Queue *or the operation failed due to insufficient resources.*

3.5.11. VipCQNotify

Synopsis

```
VIP_RETURN
        VipCQNotify(
                IN      VIP_CQ_HANDLE       CQHandle,
                IN      VIP_PVOID           Context,
                IN  void(*Handler)(
                        IN      VIP_PVOID           Context,
                        IN      VIP_NIC_HANDLE      NicHandle,
                        IN      VIP_VI_HANDLE       ViHandle,
                        IN      VIP_BOOLEAN         RecvQueue
                )
)
```

Parameters

CQHandle: Instance of a Completion Queue.

Context: Data to be passed through to the Handler as a parameter. Not used by the VI Provider.

Handler: Address of the user-defined function to be called when a single Descriptor completes. This function is not guaranteed to run in the context of the calling thread. The handler function is called with the following input parameters:

- Context: Data passed through from the function call. Not used by the VI Provider.
- NicHandle: Handle of the NIC.
- ViHandle: Instance of a Virtual Interface.

RecvQueue: *VIP_TRUE* indicates that the completion was associated with the receive queue of the VI. *VIP_FALSE* indicates that the completion was associated with the send queue of the VI.

Description

VipCQNotify is used by the VI Consumer to request that *a* Handler routine be called when a Descriptor completes on a VI Work Queue that is associated with a Completion Queue.

VipCQNotify checks the Entry on the head of the Completion queue to see if it indicates that a Descriptor has been marked complete. If there is an entry, the Entry is removed from the Completion Queue and the Handler *associated with the Entry* is invoked*. The* ViHandle and RecvQueue *are* set appropriately to indicate to the VI Consumer which Work Queue contains the completed Descriptor.

If there is no valid Completion Queue Entry, *VipCQNotify* enables interrupts for the given Completion Queue. When a Completion Queue Entry is generated, the handler will be invoked.

This registration is only associated with the Completion Queue for a single entry. In order for the Handler to be invoked multiple times, the function must be called multiple times. *Multiple handlers can be registered at one time and will be queued in Completion Queue Entry order.*

Destruction of the Completion Queue will result in cancellation of any pending function calls.

Returns

VIP_SUCCESS – The routine was successfully completed.

VIP_INVALID_PARAMETER – The VI handle, the CQ Handle or the function call address was invalid.

VIP_ERROR_RESOURCE – The operation failed due to insufficient resources.

3.5.12. Notify Semantics

This is a description of the recommended semantics for the notify calls VipSendNotify, VipRecvNotify and VipCQNotify. These semantics were selected because they can be implemented in a simple and straightforward manner by VI Providers and provide predictable operation.

A single, dedicated thread per queue is used for all notify operations on a specified send, receive or completion queue. This thread is started on the first notify call for the queue and persists throughout the life of the associated queue. With this model, the handler routine will always run in the context of the dedicated notify thread for the queue and not in the context of the calling thread.

Notify calls should not be used concurrently with either Done or Wait calls on the same work queue. If a Notify is outstanding when a Done or a Wait call is made, the order in which completed Descriptors are associated with the calls is indeterminate.

3.6. Completion Queue Management

3.6.1. VipCreateCQ

Synopsis

```
VIP_RETURN
    VipCreateCQ(
        IN      VIP_NIC_HANDLE      NicHandle,
        IN      VIP_ULONG           EntryCount,
        OUT     VIP_CQ_HANDLE       *CQHandle
    )
```

Parameters

NicHandle: The handle of the associated NIC.

EntryCount: The number of completion entries that this Completion Queue will hold.

CQHandle: Returned to the caller. The handle of the newly created Completion Queue.

Description

VipCreateCQ creates a new Completion Queue. The caller must specify the *minimum number of* completion entries that the queue must contain. If successful, it returns a handle to the newly created Completion Queue. *A C*ompletion queue *is* created with *at least the specified number of completion entries.* To avoid dropped completion notifications, applications should make sure that *the number of* operations posted on send/receive queues attached to a completion queue do*es* not exceed the completion queue capacity at any time. A common technique to deal with this in multi-threaded environments is *to use* atomic increment/decrement variables to keep track of the available space in completion queues.

Returns

VIP_SUCCESS – A new Completion Queue was successfully created.

VIP_INVALID_PARAMETER – The NIC handle was invalid.

VIP_ERROR_RESOURCE – The Completion Queue could not be created due to insufficient resources.

3.6.2. VipDestroyCQ

Synopsis

```
VIP_RETURN
    VipDestroyCQ(
        IN    VIP_CQ_HANDLE    CQHandle
    )
```

Parameters

CQHandle: The handle of the Completion Queue to be destroyed.

Description

VipDestroyCQ destroys a specified Completion Queue. If any VI Work Queues are associated with the Completion Queue, the Completion Queue is not destroyed and an error is returned.

Returns

VIP_SUCCESS – The Completion Queue was successfully destroyed.

VIP_INVALID_PARAMETER – The Completion Queue handle was invalid.

VIP_ERROR_RESOURCE – The Completion Queue could not be destroyed because the Work Queues of one or more VI instances are still associated with it.

3.6.3. VipResizeCQ

Synopsis

```
VIP_RETURN
    VipResizeCQ(
        IN    VIP_CQ_HANDLE    CQHandle,
        IN    VIP_ULONG        EntryCount
    )
```

Parameters

CQHandle: The handle of the Completion Queue to be resized.

EntryCount: The new number of completion entries that the Completion Queue must hold.

Description

VipResizeCQ modifies the size of a specified Completion Queue by specifying the new *minimum* number of completion entries that it must hold. This function is useful when the potential number of completion entries that could be placed on this queue changes dynamically.

Returns

VIP_SUCCESS – The Completion Queue was successfully resized.

VIP_INVALID_PARAMETER – The Completion Queue handle was invalid.

VIP_ERROR_RESOURCE – The Completion Queue could not be resized because of insufficient resources.

3.7. Querying Information

3.7.1. VipQueryNic

Synopsis

```
VIP_RETURN
    VipQueryNic(
        IN    VIP_NIC_HANDLE      NicHandle,
        OUT   VIP_NIC_ATTRIBUTES  *NicAttribs
    )
```

Parameters

NicHandle: The handle of a VI NIC.

NicAttribs: Returned to the caller, contains NIC-specific information.

Description

VipQueryNic returns information for a specific NIC instance. The information is returned in the *NicAttribs* data structure.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The NIC handle was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.2. VipSetViAttributes

Synopsis

```
VIP_RETURN
        VipSetViAttributes(
                IN      VIP_VI_HANDLE         ViHandle,
                IN      VIP_VI_ATTRIBUTES     *ViAttribs
        )
```

Parameters

ViHandle: The handle of a VI instance.

ViAttribs: The attributes to be set for the VI.

Description

VipSetViAttributes attempts to modify the attributes of a VI instance. If the VI Provider does not support the requested attributes, or if the VI is in a state that does not allow the attributes to be modified, then it returns an error.

Changing VI attributes is valid only when the VI is in the Idle state. An error is returned if VipSetViAttributes is called while the VI is in any other state. When an error is returned, all of the attributes remain unchanged. For instance, if only one of many requested attributes is invalid or not supported, an error is returned and none of the new attributes requested are updated.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – One of the input parameters was invalid.

VIP_INVALID_STATE – The VI is not in a state where the attributes can be modified.

VIP_INVALID_ RELIABILITY_LEVEL – The requested reliability level attribute was invalid or not supported.

VIP_INVALID_MTU – The maximum transfer size attribute was invalid or not supported.

VIP_INVALID_QOS – The quality of service attribute was invalid or not supported.

VIP_INVALID_PTAG – The protection tag attribute was invalid.

VIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.3. VipQueryVi

Synopsis

```
VIP_RETURN
        VipQueryVi(
                IN      VIP_VI_HANDLE         ViHandle,
                OUT     VIP_VI_STATE          *State,
                OUT     VIP_VI_ATTRIBUTES     *ViAttribs,
                OUT     VIP_BOOLEAN           *ViSendQEmpty,
                OUT     VIP_BOOLEAN           *ViRecvQEmpty
        )
```

Parameters

ViHandle: The handle of a VI instance.

State: The current state of the VI.

ViAttribs: Returned to caller, contains VI-specific information.

ViSendQEmpty: If VIP_TRUE, the send queue is empty.

ViRecvQEmpty: If VIP_TRUE, the receive queue is empty.

Description

VipQueryVi returns information for a specific VI instance. The VI Attributes data structure and the current VI State are returned.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The VI handle was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.4. VipSetMemAttributes

Synopsis

```
VIP_RETURN
    VipSetMemAttributes(
        IN    VIP_NIC_HANDLE        NicHandle,
        IN    VIP_PVOID             Address,
        IN    VIP_MEM_HANDLE        MemHandle,
        IN    VIP_MEM_ATTRIBUTES    *MemAttribs
    )
```

Parameters

NicHandle: The handle of the NIC where the memory region is registered.

Address: The base address of the memory region.

MemHandle: The handle of the memory region.

MemAttribs: The memory attributes to set for this memory region.

Description

VipSetMemAttributes modifies the attributes of a registered memory region. If the VI Provider does not support the requested attribute, it returns an error. Modifying the attributes of a memory region, while a data transfer operation is in progress that refers to that memory region, can result in undefined behavior, and should be avoided by the VI Consumer.

When an error is returned, all of the attributes remain unchanged. For instance if only one of many requested attributes is invalid or not supported, an error is returned and none of the new attributes requested are updated.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The Memory Handle or Address was invalid.

VIP_INVALID_PTAG – The protection tag attribute was invalid.

VIP_INVALID_RDMAREAD – The attributes requested support for RDMA Read, but the VI Provider does not support it.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.5. VipQueryMem

Synopsis

```
VIP_RETURN
    VipQueryMem(
        IN      VIP_NIC_HANDLE          NicHandle,
        IN      VIP_PVOID               Address,
        IN      VIP_MEM_HANDLE          MemHandle,
        OUT     VIP_MEM_ATTRIBUTES      *MemAttribs
    )
```

Parameters

NicHandle: The handle of the NIC where the memory region is registered.

Address: The base address of the memory region.

MemHandle: The handle of a memory region.

MemAttribs: The memory attributes of this memory region.

Description

VipQueryMem returns the attributes of a registered memory region to the caller.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The Memory Handle or Address was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.7.6. VipQuerySystemManagementInfo

Synopsis

```
VIP_RETURN
    VipQuerySystemManagementInfo(
        IN      VIP_NIC_HANDLE      NicHandle,
        IN      VIP_ULONG           InfoType,
        IN OUT  VIP_PVOID           SysManInfo
    )
```

Parameters

NicHandle: The handle of a VI NIC.

InfoType: Specifies a particular piece of system management information.

SysManInfo: Pointer to a system management information structure.

Description

VipQuerySystemManagementInfo returns system management information about the specified NIC. The InfoType parameter allows the caller to specify specific pieces of information. The

types of information that can be retrieved are VI Provider specific. The content of the System Management Information Structure is VI Provider specific.

The only defined InfoType is VIP_SMI_AUTODISCOVERY. This provides a mechanism for applications to access auto-discovery information to get the network addresses of other nodes that are attached to the SAN fabric.

The SysManInfo structure is interpreted as both an IN and OUT parameter for the network configuration request. The structure is as follows:

```
typedef struct {
        VIP_ULONG           NumberOfHops;
        VIP_NET_ADDRESS     **ADAddrArray;
        VIP_ULONG           NumAdAddrs;
} VIP_AUTODISCOVERY_LIST;
```

Structure Element Definitions

NumberOfHops: *Specifies the maximum "depth" of interest to the requester. Only network addresses within the specified "depth" are returned. Depth is defined as the number of links that are traversed by the auto-discovery operation. A depth of one is defined as the collection of nodes that are directly connected.*

ADAddrArray: *Pointer to an array of pointers to the actual VIP_NET_ADDRESS structures that will receive the discovered addresses. It is up to the caller of this routine to allocate the ADAddrArray and initialize the pointers to point to the appropriately sized VIP_NET_ADDRESS structures.*

The VIP_NET_ADDRESS structures must be large enough to hold an address for the specified NIC (specified by the NicAddressLen field of the NicAttributes structure). The HostAddress field must be long enough to hold the host portion of the address and HostAddressLen must specify a length equal to or greater than the number of bytes needed to specify the HostAddress. The DiscriminatorLen field and the discriminator portion of the HostAddress field are ignored.

NumAdAddrs: *Specifies the number of elements in the ADAddrArray. If the number of addresses discovered is larger than the number requested, then an error is returned and this field is updated to contain the number of entries required to return the full list of node addresses.*

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The NIC handle was invalid.

VIP_ERROR_RESOURCE - The number of entries specified to hold the auto-discovery information is less than the number of addresses that were discovered within the specified NumberOfHops or the operation failed due to insufficient resources.

3.8. Error handling

3.8.1. VipErrorCallback

Synopsis

```
VIP_RETURN
        VipErrorCallback(
                IN      VIP_NIC_HANDLE          NicHandle,
                IN      VIP_PVOID               Context,
                IN      void(*Handler)(
                        IN      VIP_PVOID                       Context,
                        IN      VIP_ERROR_DESCRIPTOR            *ErrorDesc
                )
        )
```

Parameters

NicHandle: Handle of the NIC

Context: Data to be passed through to the Handler as a parameter. Not used by the VI Provider.

Handler: Address of the user-defined function to be called when an asynchronous error occurs. This function is not guaranteed to run in the context of the calling thread. The error handler function is called with the following input parameters:

- Context: Data passed through from the function call. Not used by the VI Provider.
- ErrorDesc: The error Descriptor.

Description

VipErrorCallback is used by the VI Consumer to register an error handling function with the VI Provider. If the VI Consumer does not register an error handling function via this call, a default error handler will log the error. If an error handling function has been specified via the *VipErrorCallback* function, the default error handling function can be restored by calling *VipErrorCallback* with a NULL Handler parameter.

Asynchronous errors are those errors that cannot be reported back directly into a Descriptor. The following is a list of possible asynchronous errors:

- Post Descriptor Error – *This error occurs under the following conditions:*
  - *The virtual address and memory handle of the Descriptor was not valid when the Descriptor was posted.*
  - *The Next Address and/or Next Handle field was inadvertently modified after the Descriptor was posted.*
  - *The Descriptor address was not aligned on a 64-byte boundary.*
- Connection Lost – The connection on a VI was lost and the associated VI is in the error state.
- Receive Queue Empty – An incoming packet was dropped because the receive queue was empty.
- VI Overrun – The VI Consumer attempted to post too many Descriptors to a Work Queue of a VI.

- RDMA Write Protection Error – A protection error was detected on *an incoming* RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- RDMA Write Data Error – A data corruption error was detected on *an incoming* RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- RDMA Write Packet Abort – Indicates a partial packet was detected on *an incoming* RDMA Write operation. If the RDMA write operation contained immediate data, this status would be reported in the associated Descriptor, and not via the asynchronous error mechanism.
- *RDMA Transport Error - A transport error was detected on an incoming RDMA operation that does not consume a Descriptor.*
- RDMA Read Protection Error – A protection error was detected on *an incoming* RDMA Read operation.
- Completion Protection Error - When reporting completion, this could result from a user de-registering a memory region containing a Descriptor after the Descriptor was read by the hardware but before completion status was written.

  This error can also result if a Completion Queue becomes inaccessible to the hardware. In this case, an error will be generated for each VI that was associated with the Completion Queue. Note that the status (Done or Done with Errors) of the Descriptors on work queues associated with the Completion Queue may have already been written.
- *Catastrophic Error - The hardware has failed or has detected a fatal configuration problem.*

Post Descriptor Error, VI Overrun, Completion Protection Error and Catastrophic Error are catastrophic hardware errors. For more information on how these are handled, see Section 6.3.1 Catastrophic Hardware Error Handling.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – One or more of the input parameters were invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.9. Name Service

This section describes a set of routines to be used to resolve names and addresses for VIPL applications. The description is intended to capture the semantics of how VIPL applications would do simple name and address lookups It is not intended to specify how a name service is implemented. It is intended that a variety of nameservices (such as file based (e.g. /etc/hosts)) or network based (such as DNS, NIS, NDS, ActiveDirectory, etc) could be used to store the information that VIPL applications would request through this simple set of procedures.

3.9.1. VipNSInit

Synopsis

```
VIP_RETURN
    VipNSInit(
        IN   VIP_NIC_HANDLE     NicHandle,
        IN   VIP_PVOID          NSInitInfo
    )
```

Parameters

NicHandle: The VI NIC context in which to initialize the name service.

NSInitInfo: Initialization information for the name service. In the case of a file based name service, this could be the name of the alternate file to use for name lookups. A value of (VIP_PVOID)0 specifies default behavior.

Description

VipNSInit initializes the name service using the specified parameters. It is recommended that it be possible to successfully initialize the name service by specifying default behavior (e.g. set the NSInitInfo parameter to (VIP_PVOID) 0).

It is up to the provider of the name service routines to specify the format of the initialization parameter. It is recommended that file-based name service implementations use the initialization parameter to specify an alternate database file. It is also recommended that other name service implementations use string-based initialization parameters.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The NSInitInfo parameter or NicHandle parameter was invalid.

VIP_ERROR_NAMESERVICE – Name service for the specified NIC context is already initialized.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources or was unable to initialize resources.

3.9.2. VipNSGetHostByName

Synopsis

```
VIP_RETURN
    VipNSGetHostByName(
        IN      VIP_NIC_HANDLE      NicHandle,
        IN      VIP_CHAR            *Name,
        IN OUT  VIP_NET_ADDRESS     *Address,
        IN      VIP_ULONG           NameIndex,
    )
```

Parameters

NicHandle: The VI NIC context in which to resolve the name.

Name: The NULL terminated string that will be used to query the name service to find a matching address.

Address: The VIP_NET_ADDRESS structure that will receive the NIC address of the specified host. The HostAddress field must be at least the size specified by the NICAddressLen field in the VI_NIC_ATTRIBUTES structure.

NameIndex: The "index" of NIC address to return. If a node has multiple NICs on a particular SAN, the client application can start at a NameIndex of 0 and increment the index until an error is returned to fully enumerate all of the NIC addresses associated with a particular nodename

Description

VipNSGetHostByName maps string names to network addresses that are recognized by VipConnectRequest and VipConnectPeerRequest for connect operations. The name matching is case insensitive.

The name service is queried and returns an address in the context of the specified VI NIC. If this operation completes successfully, a valid VIP_NET_ADDRESS structure is contained in the network address structure specified by Address. The Address structure must be allocated by the caller with the HostAddressLen field initialized.

The NameIndex is used to allow the name service to associate multiple NIC addresses with the same name on a SAN (as specified by the NicHandle parameter). For example, if a node on a single SAN has two NICs to allow for twice the bandwidth and number of VIs than a single NIC, the first call to this routine (with a NameIndex value of 0) returns one NIC address and the second call (with a NameIndex value of 1) returns the other NIC address. A call to this routine with the same name but with a NameIndex value of 2 or more returns a VIP_ERROR_NAMESERVICE.

Returns

VIP_SUCCESS – Operation completed successfully

VIP_ERROR_NAMESERVICE - The name specified was not found by the name service, the NameIndex for the given name is invalid or the name service was not initialized for the specified NIC context.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.9.3. VipNSGetHostByAddr

Synopsis

```
VIP_RETURN
    VipNSGetHostByAddr (
        IN      VIP_NIC_HANDLE      NicHandle,
        IN      VIP_NET_ADDRESS     *Address,
        OUT     VIP_CHAR            *Name,
        IN OUT  VIP_ULONG           *NameLen
)
```

Parameters

NicHandle: A NicHandle for which the Address parameter is valid.

Address: The network address used to query the name service.

Name: A pointer to the array where the name associated with the specified Address returned by the name service will be stored. The array must be allocated by the caller of this routine.

NameLen: The size of the Name array. The caller of the routine sets this parameter to the size of the array allocated. The array allocated must have enough storage to include the NULL termination. It is updated by the routine to contain the actual length of the host name string (does not include the NULL termination character).

Description

VipNSGetHostByAddr maps the Address to a host name by querying the name service in the context of the specified NicHandle. The name associated with the Address is returned to the caller in the Name parameter. If the size specified by NameLen is not sufficient to hold the entire

name, a VIP_INVALID_PARAMETER is returned. VipNSGetHostByAddr returns the size of the array required to hold the entire string in NameLen so the caller can increase the size of the array and call VipNSGetHostByName again with the larger array to retrieve the name.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_ERROR_NAMESERVICE - The address specified was not found by the name service or the name service was not initialized for the specified NIC context.

VIP_INVALID_PARAMETER – One of the parameters was invalid. If the NameLen size set by the caller is too small to hold the entire name, the value returned in the NameLen field is the size of the array required to hold the name.

VIP_ERROR_RESOURCE - The operation failed due to insufficient resources.

3.9.4. VipNSShutdown

Synopsis

```
VIP_RETURN
    VipNSShutdown(
        IN      VIP_NIC_HANDLE      NicHandle
    )
```

Parameters

NicHandle: The VI NIC context to shutdown an already initialized name service.

Description

VipNSShutdown is called to indicate to the name service that no more name resolutions will be required. It is expected that programs that use the VI Provider name service will call this routine for each VipNSInit call made previously prior to normal shutdown.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_ERROR_NAMESERVICE - The name service was already shutdown or was not initialized for the specified NIC context.

VIP_INVALID_PARAMETER – An invalid NicHandle parameter was specified.

VIP_ERROR_RESOURCE – An error occurred while shutting down the name service, possibly due to insufficient resources available to process the shutdown request.

4. Data Structures and Values

4.1. Generic VIPL Types

The following are generic VIPL types that were defined for portability.

```
typedef     void *              VIP_PVOID;
typedef     int                 VIP_BOOLEAN;
typedef     char                VIP_CHAR;
typedef     unsigned char       VIP_UCHAR;
typedef     unsigned short      VIP_USHORT;
typedef     unsigned long       VIP_ULONG;
typedef     unsigned __int64    VIP_UINT64;
typedef     unsigned __int32    VIP_UINT32;
typedef     unsigned __int16    VIP_UINT16;
typedef     unsigned __int8     VIP_UINT8;
```

VIP_TRUE and VIP_FALSE have been defined to provide a common definition for TRUE and FALSE.

```
#define     VIP_TRUE        (1)
#define     VIP_FALSE       (0)
```

The following is an implementation convenience defining the memory alignment required by the NIC for descriptors, in bytes.

```
#define     VIP_DESCRIPTOR_ALIGNMENT    64
```

4.2. Return Codes

The various functions described herein return error or success codes of the type VIP_RETURN. The possible values for VIP_RETURN follow:

- VIP_SUCCESS – The function completed successfully.
- VIP_NOT_DONE – No Descriptors are completed on the specified queue.
- VIP_INVALID_PARAMETER – One or more input parameters were invalid.
- VIP_ERROR_RESOURCE – An error occurred due to insufficient resources *or resource conflict*.
- VIP_TIMEOUT – The request timed out before it could successfully complete.
- VIP_REJECT – A connection request was rejected by the remote end.
- VIP_INVALID_RELIABILITY_LEVEL – The reliability level attribute for a VI was invalid or not supported.
- VIP_INVALID_MTU – The maximum transfer size attribute for a VI was invalid or not supported.
- VIP_INVALID_QOS – The quality of service attribute for a VI was invalid or not supported.
- VIP_INVALID_PTAG – The protection tag attribute for a VI or a memory region was invalid.
- VIP_INVALID_RDMAREAD - A memory or VI attribute requested support for RDMA Read, but the VI Provider does not support it.

- *VIP_DESCRIPTOR_ERROR - A completed Descriptor was returned with errors in the completion status, or the VI work queue was empty.*
- *VIP_INVALID_STATE - The operation is not valid in the current VI state.*
- *VIP_ERROR_NAMESERVICE - An unexpected error occurred while initializing, shutting down or resolving a name/address in the name service.*
- *VIP_NO_MATCH - The local and remote connection discriminators do not match.*
- *VIP_NOT_REACHABLE - A network partition was detected while attempting to establish a VI connection.*

The declaration for VIP_RETURN is as follows:

```
typedef enum {
    VIP_SUCCESS,
    VIP_NOT_DONE,
    VIP_INVALID_PARAMETER,
    VIP_ERROR_RESOURCE,
    VIP_TIMEOUT,
    VIP_REJECT,
    VIP_INVALID_RELIABILITY_LEVEL,
    VIP_INVALID_MTU,
    VIP_INVALID_QOS,
    VIP_INVALID_PTAG,
    VIP_INVALID_RDMAREAD,
    VIP_DESCRIPTOR_ERROR,
    VIP_INVALID_STATE,
    VIP_ERROR_NAMESERVICE,
    VIP_NO_MATCH,
    VIP_NOT_REACHABLE
} VIP_RETURN
```

4.3. VI Descriptor

The VI Descriptor is the data structure that describes the system memory associated with a VI Packet. For data to be transmitted, it describes a gather list of buffers that contain the data to be transmitted. For data that is to be received, it describes a scatter list of buffers to place the incoming data. It also contains fields for control and status information, and has variants to accommodate send/receive operations as well as RDMA operations.

Descriptors are made up of three types of segments; control, address and data segments. The control segment is the first segment for all Descriptors. An address segment follows the control segment for Descriptors that describe RDMA operations. A variable number of data segments come last that describe the system buffer(s) on the local host.

```
typedef union {
        VIP_ADDRESS_SEGMENT     Remote;
        VIP_DATA_SEGMENT        Local;
} VIP_DESCRIPTOR_SEGMENT

typedef struct _VIP_DESCRIPTOR {
        VIP_CONTROL_SEGMENT     CS;
        VIP_DESCRIPTOR_SEGMENT DS[2];
} VIP_DESCRIPTOR
```

```
typedef union {
        VIP_UINT64      AddressBits;
        VIP_PVOID       Address;
} VIP_PVOID64

typedef struct {
        VIP_PVOID64             Next;
        VIP_MEM_HANDLE          NextHandle;
        VIP_UINT16              SegCount;
        VIP_UINT16              Control;
        VIP_UINT32              Reserved;
        VIP_UINT32              ImmediateData;
        VIP_UINT32              Length;
        VIP_UINT32              Status;
} VIP_CONTROL_SEGMENT

typedef struct {
        VIP_PVOID64             Data;
        VIP_MEM_HANDLE          Handle;
        VIP_UINT32              Reserved;
} VIP_ADDRESS_SEGMENT

typedef struct {
        VIP_PVOID64             Data;
        VIP_MEM_HANDLE          Handle;
        VIP_UINT32              Length;
} VIP_DATA_SEGMENT
```

The possible values for the Control field of the Control Segment are as follows:

```
#define     VIP_CONTROL_OP_SENDRECV             0x0000
#define     VIP_CONTROL_OP_RDMAWRITE            0x0001
#define     VIP_CONTROL_OP_RDMAREAD             0x0002
#define     VIP_CONTROL_OP_RESERVED             0x0003
#define     VIP_CONTROL_OP_MASK                 0x0003
#define     VIP_CONTROL_IMMEDIATE               0x0004
#define     VIP_CONTROL_QFENCE                  0x0008
#define     VIP_CONTROL_RESERVED                0xFFF0
```

The possible values for the Status field of the Control Segment are as follows:

```
#define     VIP_STATUS_DONE                     0x00000001
#define     VIP_STATUS_FORMAT_ERROR             0x00000002
#define     VIP_STATUS_PROTECTION_ERROR         0x00000004
#define     VIP_STATUS_LENGTH_ERROR             0x00000008
#define     VIP_STATUS_PARTIAL_ERROR            0x00000010
#define     VIP_STATUS_DESC_FLUSHED_ERROR       0x00000020
#define     VIP_STATUS_TRANSPORT_ERROR          0x00000040
#define     VIP_STATUS_RDMA_PROT_ERROR          0x00000080
#define     VIP_STATUS_REMOTE_DESC_ERROR        0x00000100
#define     VIP_STATUS_ERROR_MASK               0x000001FE
#define     VIP_STATUS_OP_SEND                  0x00000000
#define     VIP_STATUS_OP_RECEIVE               0x00010000
#define     VIP_STATUS_OP_RDMA_WRITE            0x00020000
#define     VIP_STATUS_OP_REMOTE_RDMA_WRITE     0x00030000
#define     VIP_STATUS_OP_RDMA_READ             0x00040000
#define     VIP_STATUS_OP_MASK                  0x00070000
#define     VIP_STATUS_IMMEDIATE                0x00080000
#define     VIP_STATUS_RESERVED                 0xFFF0FE00
```

4.4. Error Descriptor

The error Descriptor is used by the error handling routine *VipErrorCallback*. It is used to determine the layer of software or hardware that caused the failure, and all relevant information that is available about the error.

An error Descriptor is passed to the user supplied error handler that was registered via *VipErrorCallback*. The error Descriptor contains the following fields:

- NIC handle – Indicates the NIC, or VI Provider, that is reporting the error.
- VI handle – If non-NULL, refers to the VI instance related to the error.
- *CQ handle - If non-NULL, refers to the Completion Queue related to the error.*
- Descriptor pointer – If non-NULL, refers to the Descriptor related to the error.
- Operation code – Describes the operation being performed when the error was detected. This code is the same as the 'completed operation' code that is described in the Descriptor status field.
- Resource code – Allows the application to tell if the error was due to a NIC problem, VI problem, queue problem or Descriptor problem.
- Error code – A numeric code that identifies the specific error.

The declaration of the error Descriptor is as follows:

```
typedef struct {
    VIP_NIC_HANDLE          NicHandle;
    VIP_VI_HANDLE           ViHandle;
    VIP_CQ_HANDLE           CQHandle;
    VIP_DESCRIPTOR          *DescriptorPtr;
    VIP_ULONG               OpCode;
    VIP_RESOURCE_CODE       ResourceCode;
    VIP_ERROR_CODE          ErrorCode;
} VIP_ERROR_DESCRIPTOR
```

Possible values for ResourceCode are:

```
typedef enum _VIP_RESOURCE_CODE {
    VIP_RESOURCE_NIC,
    VIP_RESOURCE_VI,
    VIP_RESOURCE_CQ,
    VIP_RESOURCE_DESCRIPTOR
}  VIP_RESOURCE_CODE
```

Possible values for ErrorCode follow, refer to the description of *VipErrorCallback* for a more complete description:

```
typedef enum _VIP_ERROR_CODE {
    VIP_ERROR_POST_DESC,
    VIP_ERROR_CONN_LOST,
    VIP_ERROR_RECVQ_EMPTY,
    VIP_ERROR_VI_OVERRUN,
    VIP_ERROR_RDMAW_PROT,
    VIP_ERROR_RDMAW_DATA,
    VIP_ERROR_RDMAW_ABORT,
    VIP_ERROR_RDMAR_PROT,
    VIP_ERROR_COMP_PROT,
    VIP_ERROR_RDMA_TRANSPORT,
    VIP_ERROR_CATASTROPHIC
}  VIP_ERROR_CODE
```

4.5. NIC Attributes

The NIC attributes structure is returned from the *VipQueryNic* function. It contains information related to an instance of a NIC within a VI Provider. All values that are returned in the NIC Attributes structure are static values that are set by the VI Provider at the time that it is initialized. It is not required that the VI Provider return dynamically updated values within this structure at run-time.

- Name – The symbolic name of the NIC device.
- HardwareVersion – The version of the VI Hardware.
- ProviderVersion – The version of the VI Provider.
- NicAddressLen – The length, in bytes, of the local NIC address.
- LocalNicAddress – Points to a constant array of bytes containing the NIC Address.
- ThreadSafe – Synchronization model (thread safe / not thread safe)
- MaxDiscriminatorLen – The maximum number of bytes that the VI Provider allows for a connection discriminator. VI Providers are required to handle discriminators of at least 16 bytes in length. *The value returned in this field must be at least 16 bytes.*
- MaxRegisterBytes – Maximum number of bytes that can be registered
- MaxRegisterRegions – Maximum number of memory regions that can be registered.
- MaxRegisterBlockBytes – Largest contiguous block of memory that can be registered, in bytes.
- MaxVI – Maximum number of VI instances supported by this VI NIC.
- MaxDescriptorsPerQueue – Maximum Descriptors per VI Work Queue supported by this VI Provider.
- MaxSegmentsPerDesc – Maximum data segments per Descriptor that this VI Provider supports. The address segment is included in this count.
- MaxCQ – Maximum number of Completion Queues supported.
- MaxCQEntries – The maximum number of Completion Queue entries that this VI NIC will support per Completion Queue.
- MaxTransferSize – The maximum transfer size, specified in bytes, supported by this VI NIC. The maximum transfer size is the amount of data that can be described by a single VI Descriptor.
- NativeMTU – The native MTU size, specified in bytes, of the underlying network. For frame-based networks, this could reflect its native frame size. For cell-based networks, it could reflect the MTU of the appropriate abstraction layer that it supports.
- MaxPTags – The maximum number of Protection Tags that is supported by this VI NIC. It is required that all VI Providers can support at least one Protection Tag for each VI supported.
- *ReliabilityLevelSupport - Indicates the reliability levels that are supported by this VI NIC.*
- *RDMAReadSupport - Indicates which reliability levels support RDMA Read operations. Zero or more bits may be set.*

The declaration of the NIC attributes structure is as follows:

```
typedef struct {
    VIP_CHAR              Name [64];
    VIP_ULONG             HardwareVersion;
    VIP_ULONG             ProviderVersion;
    VIP_UINT16            NicAddressLen;
    const VIP_UINT8       *LocalNicAddress;
    VIP_BOOLEAN           ThreadSafe;
    VIP_UINT16            MaxDiscriminatorLen;
    VIP_ULONG             MaxRegisterBytes;
    VIP_ULONG             MaxRegisterRegions;
    VIP_ULONG             MaxRegisterBlockBytes;
    VIP_ULONG             MaxVI;
    VIP_ULONG             MaxDescriptorsPerQueue;
    VIP_ULONG             MaxSegmentsPerDesc;
    VIP_ULONG             MaxCQ;
    VIP_ULONG             MaxCQEntries;
    VIP_ULONG             MaxTransferSize;
    VIP_ULONG             NativeMTU;
    VIP_ULONG             MaxPtags;
    VIP_RELIABILITY_LEVEL ReliabilityLevelSupport;
    VIP_RELIABILITY_LEVEL RDMAReadSupport;
} VIP_NIC_ATTRIBUTES
```

4.6. VI Attributes

The VI attributes contain VI specific information. The VI attributes are set when the VI is created by *VipCreateVi*, can be modified by *VipSetViAttributes*, and can be discovered by *VipQueryVi*. The VI attributes structure contains:

- ReliabilityLevel – Reliability level of the VI (unreliable service, reliable delivery, reliable reception). As an attribute of a VI, it is the requested class of service for the requested connection.
- MaxTransferSize – The requested maximum transfer size for this connection. The Transfer Size specifies the amount of payload data that can be transferred in a single VI packet.
- QoS – The requested quality of service for the connection
- Ptag – The protection tag to be associated with the VI.
- EnableRdmaWrite – If *VIP_TRUE*, accept RDMA Write operations on this VI from the remote end of a connection.
- EnableRdmaRead – If *VIP_TRUE*, accept RDMA Read operations on this VI from the remote end of a connection.

The declaration of the VIP_VI_ATTRIBUTES is as follows:

```
typedef struct {
    VIP_RELIABLITY_LEVEL      ReliabilityLevel;
    VIP_ULONG                 MaxTransferSize;
    VIP_QOS                   QoS;
    VIP_PROTECTION_HANDLE     Ptag;
    VIP_BOOLEAN               EnableRdmaWrite;
    VIP_BOOLEAN               EnableRdmaRead;
} VIP_VI_ATTRIBUTES

typedef VIP_USHORT        VIP_RELIABILITY_LEVEL;
```

The possible values for VIP_RELIABILITY_LEVEL are:

```
#define VIP_SERVICE_UNRELIABLE              0x01
#define VIP_SERVICE_RELIABLE_DELIVERY       0x02
#define VIP_SERVICE_RELIABLE_RECEPTION      0x04
```

The VIP_QOS *is currently undefined, but* is declared as:

```
typedef     VIP_PVOID   VIP_QOS;       /* details are not defined */
```

Note: The only VI attributes that are checked when establishing a connection are ReliabilityLevel, MaxTransferSize and QoS. The details for QoS have not been defined in the 1.0 VI Architecture Specification.

4.7. Memory Attributes

The memory attributes structure contains the attributes of registered memory regions. The attributes of a registered memory region are set by *VipRegisterMem*, can be modified by *VipSetMemAttributes*, and can be discovered by *VipQueryMem*. The memory attributes structure contains:

- Ptag – The protection tag to be associated with a registered memory region.
- EnableRdmaWrite – If VIP_TRUE, allow RDMA Write operations into this registered memory region.
- EnableRdmaRead – If VIP_TRUE, allow RDMA Read operations from this registered memory region.

```
typedef struct {
    VIP_PROTECTION_HANDLE   Ptag;
    VIP_BOOLEAN             EnableRdmaWrite;
    VIP_BOOLEAN             EnableRdmaRead;
} VIP_MEM_ATTRIBUTES
```

4.8. VI Endpoint State

The VI State (Idle, Pending Connect, Connected, and Error). The VI State is returned from the query VI function. The type for VI endpoint state is VIP_VI_STATE, the possible values are:

```
typedef enum {
    VIP_STATE_IDLE,
    VIP_STATE_CONNECTED,
    VIP_STATE_CONNECT_PENDING,
    VIP_STATE_ERROR
}  VIP_VI_STATE
```

4.9. VI Network Address

A VI Network Address holds the network specific address for a VI endpoint. Each VI Provider may have a unique network address format. It is composed of two elements, a host address and an endpoint discriminator. These elements are qualified with a byte length in order to maintain network independence.

```
typedef struct {
    VIP_UINT16      HostAddressLen;
    VIP_UINT16      DiscriminatorLen;
    VIP_UINT8       HostAddress[1];
} VIP_NET_ADDRESS
```

The HostAddress array contains the host address, followed by the endpoint discriminator.

Figure 3 depicts how a network address is laid out in memory.

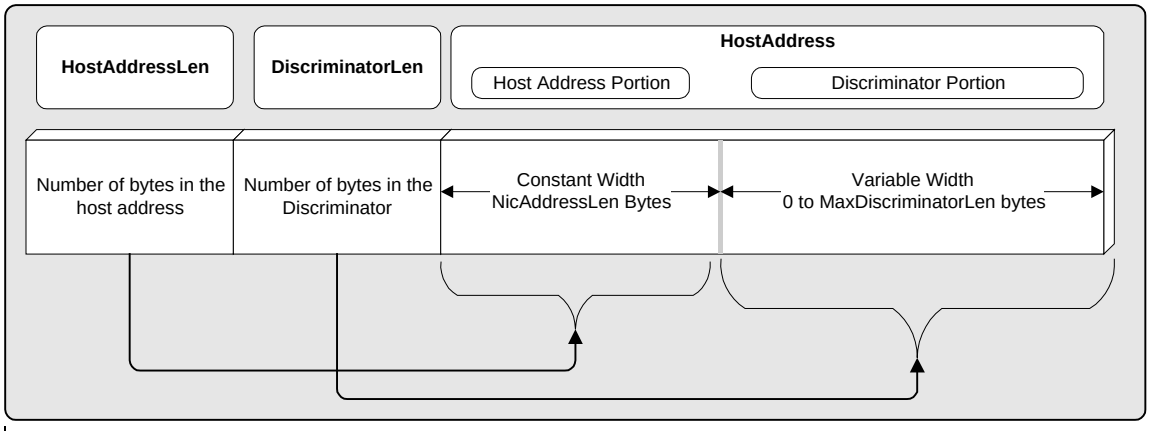

Figure 3: VIP_NET_ADDRESS memory layout

5. Descriptors

5.1. Descriptor Format Overview

This section describes the format for Descriptors. The NIC hardware is aware of this format, and cooperates with software in managing it. The format of a Descriptor is independent of physical media type.

Descriptors are in little-endian byte order. Any media or architecture that cannot support this byte order will require software or hardware translation of Descriptors and data. *Note that length fields are specified in number of bytes, unless explicitly stated otherwise.*

Descriptors are composed of segments. There are three types of segments: control, address and data. All segments of a single Descriptor must be in the order described below. Descriptors always begin with a Control Segment. The Control Segment contains control and status information, as well as reserved fields that are used for queuing.

An Address Segment follows the Control Segment for RDMA operations. This segment contains remote buffer address information for RDMA Read and RDMA Write operations.

The Data Segment contains information about the local buffers of a send, receive, RDMA Write, or RDMA Read operation. A Descriptor may contain multiple Data Segments.

The format of a send or receive Descriptor is shown in Figure 4. The format of an RDMA Descriptor is shown in Figure 5.

15:14	13:12	11:8	7:0	byte offset
control	seg count	memory handle	next descriptor virtual address	Control Segment
status		total length	immediate data / reserved	
seg length		memory handle	buffer virtual address	Data Segment

Figure 4: Send and Receive Descriptor Format

15:14	13:12	11:8	7:0	byte offset
control	seg count	memory handle	next descriptor virtual address	Control Segment
status		total length	immediate data / reserved	
reserved		remote memory handle	remote buffer virtual address	Address Segment
seg length		memory handle	buffer virtual address	Data Segment

Figure 5: RDMA Descriptor Format

5.2. Descriptor Control Segment

The fields of a Control Segment are described below. All Reserved fields must be zero or a format error will occur.

Next Descriptor Virtual Address (control segment bytes 7:0):

This field links a series of Descriptors to form the send and receive queues for a VI. The value is the virtual address of the next Descriptor on a queue. A VI *User Agent may fill* in this field in the Descriptor that is currently the tail of a queue to add a new Descriptor to the queue. *The VI Application should not expect this field to be returned intact upon completion of a VI operation.*

Next Descriptor Memory Handle (control segment bytes 11:8)

This field is the matching memory handle for the Next Descriptor Virtual Address. A VI *User Agent* fills in this field when it fills in the Next Descriptor Virtual Address field.

Descriptor Segment Count (control segment bytes 13:12)

This field contains the number of segments following the Control Segment, including the Address Segment, if present. A VI *Application* sets this field when formatting the Descriptor.

Control Field (control segment bytes 15:14)

This field contains control bits or information pertaining to the entire Descriptor. The VI *Application* sets the bits in this field when formatting the Descriptor. These bits indicate specific actions to be taken by the VI when processing the Descriptor.

This Control Field contains sub-fields, as follows:

Control field bits 1:0: Operation Type

Defines the operation for this Descriptor. Acceptable values are:

00: Indicates that this is a Send operation if this Descriptor is posted on the send queue. Indicates that this is a Receive operation if this Descriptor is posted on the receive queue.

01: Indicates that this is a RDMA Write Descriptor if posted on the send queue. This value is invalid if this Descriptor is posted on the receive queue, and will result in a format error.

10: Indicates that this is a RDMA Read Descriptor if posted on the send queue. This value is only valid if the underlying VI Provider supports RDMA Read operations. This value is invalid if this Descriptor posted on the receive queue, and will result in a format error.

11: This value is undefined and will result in a format error.

Control field bit 2: Immediate Data Indication

If this bit is set, it indicates that there is valid data in the Immediate Data field of this Descriptor.

If this is a Send Descriptor, it indicates that the data in the Immediate Data field is to be transferred to the corresponding Receive Descriptor on the remote end of the connection.

If this is an RDMA Write Descriptor, it indicates that the data in the Immediate Data field is to be transferred to the corresponding Receive Descriptor on the remote end of the connection. Normally RDMA Writes do not consume Descriptors on the remote node, but Immediate Data will cause this to happen.

This bit is ignored for RDMA Read operations. Immediate Data is not transferred with RDMA Read operations. This will not result in a format error. The Immediate Data is simply ignored.

If this is a pending Receive Descriptor, this bit is ignored*. A format error will not be generated if this bit is set.*

Control field bit 3: Queue Fence

The Queue Fence bit, when set, inhibits processing of the Descriptor until all previous RDMA Read operations on the same queue are complete.

If this is a pending Receive Descriptor, this bit is ignored. A format error will not be generated if this bit is set.

Control field bits 15:4: Reserved

These bits are reserved for future use. They must be set to zero by the VI Application or a format error will occur.

Reserved (control segment bytes 19:16):

This field is a reserved field. It must be set to zero by the VI Application or a format error will result.

Immediate Data (control segment bytes 23:20):

This field allows 32 bits of data to be transferred from the Descriptor of a Send or RDMA Write operation to a corresponding Descriptor in the connected VI's Receive Queue. Immediate Data is used in conjunction with the Immediate Data Indication bit of the Control Field in the Control Descriptor.

This field is optionally set by the VI Application in the case of Send and RDMA Write operation and is returned to the VI Application in the case of Receives. The Immediate Data field is ignored for RDMA Read operations.

Length Field (control segment bytes 27:24)

This field contains the total length of the data described by the Descriptor. The VI Application sets this field when formatting the Descriptor. For send Descriptors *and RDMA Read requests*, this field must specify the sum of the Local Buffer Length fields of all Data Segments for the packet. For outstanding receive Descriptors, this field is undefined. The VI NIC will use the length parameters in the individual Data Segments when determining reception length.

Upon completion of data transfer, this field is set by the VI NIC to reflect the total number of bytes transferred from, in the case of a Send or RDMA Write, or to, in the case of Receive or RDMA Read, the Data Segment buffers. If the Descriptor completed with an error, the Length field is undefined.

Note that the 1.0 VI Architecture Specification is internally inconsistent. Section 6.4.1 Completing Descriptors by the VI Provider, states: "If a data transfer incurs a data overrun error, the Receive Descriptor's total length is set to zero, the data is undefined and may result in the VI transitioning to the Error state as per the discussions in Sections 2.5 and 5." This is inconsistent with the statement in the above paragraph: "If the Descriptor completed with an error, the Length field is undefined." For portability, the recommendation is to assume the Length field is undefined in the case of a data overrun. In order to conform to the specification, the VI Provider must set the length field to zero in the case of a data overrun error.

The length field does not include the length of the immediate data field.

When a receive Descriptor is completed for an RDMA Write with immediate data, the length field contains the number of bytes transferred.

Status (control segment bytes 31:28):

This field contains status information that is written by a VI NIC in order to complete a Descriptor. A VI *User Agent* polls for completion of a Descriptor by reading this field in the Work Queue completion model. In general, the format of the status field is that bits 0:15 allow the VI *User Agent* to easily check for successful completion or for completion in error. Bits 16:31 contain flags to provide additional information to the VI *User Agent.* The VI *User* Agent must set this field to zero before posting a Descriptor to the VI NIC.

The format for the Status Field is shown in Figure 6.

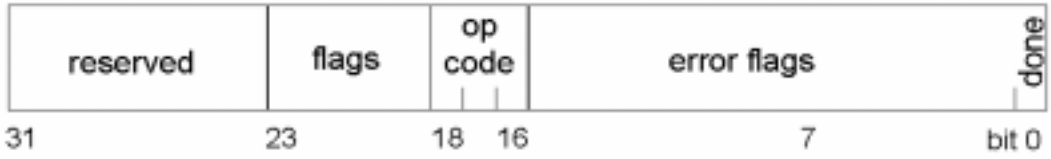

Figure 6: Status Field Format

The individual bits of the Status Field are defined as follows:

Status Field, bit 0: Done

This bit is set to 1 by a VI Provider to indicate that Descriptor execution has completed. Zeroes in bits 1 through 15 of the status field indicate successful completion. A 1 in any of the bits 1 through 15 of this field indicates that an error was detected during Descriptor execution.

This bit in the Descriptor is set according to the level of reliability of the Connection.

Status Field, bit 1: Local Format Error

This *bit* is set if the locally posted Descriptor has a format error. This includes errors such as invalid operation codes and reserved fields set by the software. It does not include errors covered by other error bits.

This bit is valid on all Descriptor and Connection types.

Status Field, bit 2: Local Protection Error

This *bit* is set if the locally posted Descriptor's data segment address and memory handle pair does not point to a protection table entry that is valid for the requested operation. This may indicate a bad memory handle, a bad virtual address, mismatched protection tags, insufficient rights for the requested operation*, or one or more of the specified data segments is not accessible due to a protection violation.*

This bit is also set in the Descriptor posted to the receive queue corresponding to an RDMA Write operation with Immediate data when a protection error occurs at the target of the RDMA Write.

This bit is valid on all Descriptor and Connection types.

Status Field, bit 3: Local Length Error

This bit is set under the following conditions for the Send and Receive queues for a VI.

For Descriptors posted on either the Send or Receive Queue:

- *The Segment Count field exceeds the capabilities of this NIC.*

Descriptors posted on the Send Queue:

- *The total length field in the control segment exceeds the maximum transfer size setting of this VI.*
- *The total length field in the control segment does not match the sum of the locally posted Descriptor's Data Segment lengths.*

Descriptors posted on the Receive Queue:

- *The sum of the locally posted Descriptor's Data Segment lengths was too small to receive the incoming packet.*
- *The length of the incoming send exceeds the maximum transfer size setting of this VI.*

This bit is valid on all Descriptor and Connection types.

Status Field, bit 4: Partial Packet Error

This bit will be set on a Descriptor posted to the send queue if an error was detected after a partial packet was put on the fabric. This bit will be set in conjunction with another bit that indicates the error causing the abort.

For Descriptors posted to the receive queue, this bit indicates and aborted or truncated packet was received.

This bit is valid on all Descriptor and Connection types.

Status Field, bit 5: Descriptor Flushed

This bit indicates that the Descriptor was flushed from the queue when the VI was disconnected. The VI may have been disconnected either explicitly or due to an error.

This bit is valid on all Descriptor and Connection types.

Status Field, bit 6: Transport Error

This bit is used to indicate that there was an unrecoverable data error, data could not be transferred, data was transferred but corrupted, the corresponding endpoint was not responding or VI NIC link problem. *For reliable delivery and reliable reception modes, a set bit indicates the VI has transitioned to Error State. On Unreliable connections, this bit is only valid on Receive Descriptors.* For reliable delivery connections, this bit is only valid on receive and RDMA read Descriptors. On reliable reception connections, this bit is valid on all types of Descriptors.

Status Field, bit 7: RDMA Protection Error

This bit is set if the source of the RDMA Read, or destination of an RDMA Write, had a protection error detected at the remote node. *Possible causes include a bad memory handle, a bad virtual address, mismatched protection tags, or insufficient rights for the requested operation.*

This bit is not valid for Descriptors on Unreliable Connections. For Reliable Delivery Connections, this bit is set only on RDMA Read Descriptors. On Reliable Reception Connections, this bit is set either on RDMA Read or RDMA Write Descriptors. This bit is not set on other Descriptor types.

Status Field, bit 8: Remote Descriptor Error

This bit is set if there was a length, format, or protection error in a Descriptor posted at the remote node. It is also set if there was no receive Descriptor posted for the incoming packet.

For Unreliable and Reliable Delivery Connections, this bit is not valid for any Descriptor posted. For Reliable Reception Connections, this bit is only set on Send Descriptors and RDMA Write Descriptors with Immediate Data.

Status Field, bits 15:9: Reserved Error Bits

These bits are reserved for future use and must be set to zero by VI Providers complying with this version of the specification.

Status Field, bits 18:16: Completed Operation Code

This field describes the type of operation completed for this Descriptor. The codes within this field are arranged such that the least significant bit (LSB) denotes the queue on which this operation completed. An LSB of zero denotes that the operation completed on the Send Queue, while an LSB of 1 denotes that the operation completed on the Receive Queue. The possible (binary) values are:

000b: Send operation completed.

001b: Receive operation completed.

010b: RDMA Write operation completed.

011b: Remote RDMA Write operation completed. This value indicates that an RDMA Write operation that was initiated on the remote end of the connection completed and consumed this Descriptor (implying that immediate data is available in the Immediate Data field).

100b: RDMA Read operation completed (if supported, otherwise undefined).

101b through 111b: are undefined.

Status Field, bit 19: Immediate Flag

This bit is set when the Immediate Data field is valid for a Descriptor on the Receive queue. The Immediate Flag is set at the completion of a Receive operation, or at the target side of a RDMA Write operation when a Receive Descriptor is consumed.

Status Field, bit 31:20 Reserved Flag Bits

These bits are reserved for future use and must be set to zero by VI Providers complying with this version of the specification.

5.3. Descriptor Address Segment

The second Segment in a Descriptor is the Address Segment. This segment is only included in RDMA operations. It is not included in normal Send operation nor in Receive Descriptors of any type, since all RDMA requests are posted to the Send queue.

The purpose of this segment is to identify to the VI NIC the virtual address on the remote node where the RDMA Data is to be read from or written to. The virtual address must reside in a Memory Region registered by the process associated with the remote VI. The remote virtual address and corresponding memory handle must be known to the local process before an RDMA request is initiated.

Remote Buffer Virtual Address (address segment bytes 7:0):

For an RDMA Write operation, this value specifies the virtual address of the destination buffer at the remote end of the connection. For RDMA Read operation, it specifies the source buffer at the remote end of the connection.

Remote Buffer Memory Handle (address segment bytes 11:8):

This field contains the memory handle that corresponds to the Remote Buffer Virtual Address.

Reserved (address segment bytes 15:12):

This field is reserved, and must be set to zero by the VI Consumer or the Descriptor will be completed in error due to a format error.

5.4. Descriptor Data Segment

Zero or more Data Segments can exist within a Descriptor.

Every VI NIC has a limit on the number of Data Segments that a Descriptor may contain. All VI NICs must be able to handle at least 252 Data Segments in a single Descriptor. Each VI Provider should supply a mechanism by which a VI Application can determine the maximum number of Data Segments supported by the Provider.

The minimum number of Data Segments that can be included in a Descriptor is zero. It is possible to send only Immediate Data in a Descriptor, although even that need not be sent.

The total sum of the buffer lengths described by the Data Segments in a Descriptor cannot exceed the *maximum transfer size* of the VI or a length error will result.

Local Buffer Virtual Address (data segment bytes 7:0):

This field contains the virtual address of the data buffer described by the segment. This field must be filled in by the VI Application.

Local Buffer Memory Handle (data segment bytes 11:8):

This field contains the corresponding Memory Handle for the Local Buffer Virtual Address.

Local Buffer Length (data segment bytes 15:12):

This field contains the length of the Local Buffer pointed to by the Local Buffer Virtual Address field. Zero is a valid value for this field.

5.5. Descriptor Handoff and Ownership

This section describes the handoff of the descriptor between the VI Application, the VI Library (VIPL) and the VI NIC from the point of view of the VI Application. The figures contained in this section show the descriptor at each stage of the handoff. The sections that are shaded are filled in by the current owner of the Descriptor.

VI Application

The Descriptor fields that the VI Application must fill in (the Immediate Data field is optional depending upon whether Immediate Data is to be sent with a send or RDMA write operation) are shaded in gray in Figure 7. On a send or RDMA write operation, the Application generates the data in the data buffers. The descriptor is passed to VIPL when the application calls VipPostSend or VipPostRecv.

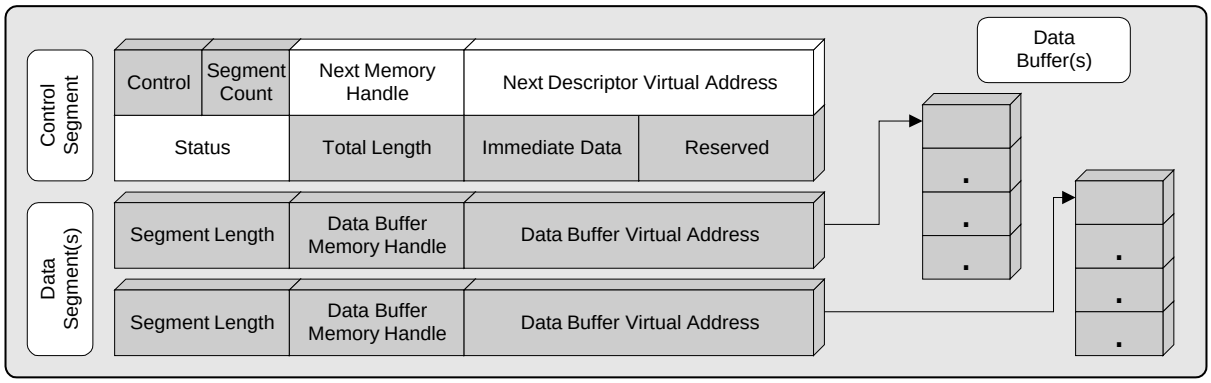

Figure 7: Descriptor fields updated by VI Application

VIPL

VIPL is responsible for setting the Status field to zero, enqueuing the Descriptor and writing the Doorbell token. The Descriptor fields that VIPL is responsible for updating are shown in gray in Figure 8. The VI NIC takes ownership of the Descriptor when the Doorbell token is posted to the Doorbell register on the VI NIC.

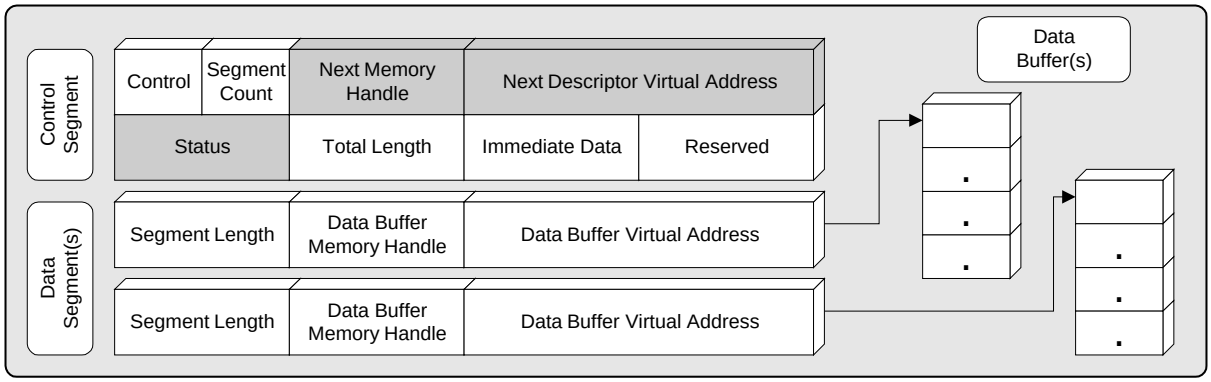

Figure 8: Descriptor fields updated by VIPL

VI NIC

The VI NIC "processes" the descriptor. The VI NIC writes the Immediate Data, Length and Status fields. When the VI NIC writes the Done bit, the Descriptor is handed off to VIPL.

Data buffers that are associated with Descriptors posted on a receive queue may have their contents modified by the VI NIC at any time while the VI NIC owns those Descriptors. The application can make no assumptions about the contents of data buffers that are associated with Descriptors that were completed in error or the contents of the unused portion of data buffers when the Descriptor completes successfully.

The Immediate Data field is only valid when the Immediate Flag is set in the status field and the Descriptor completes without error.

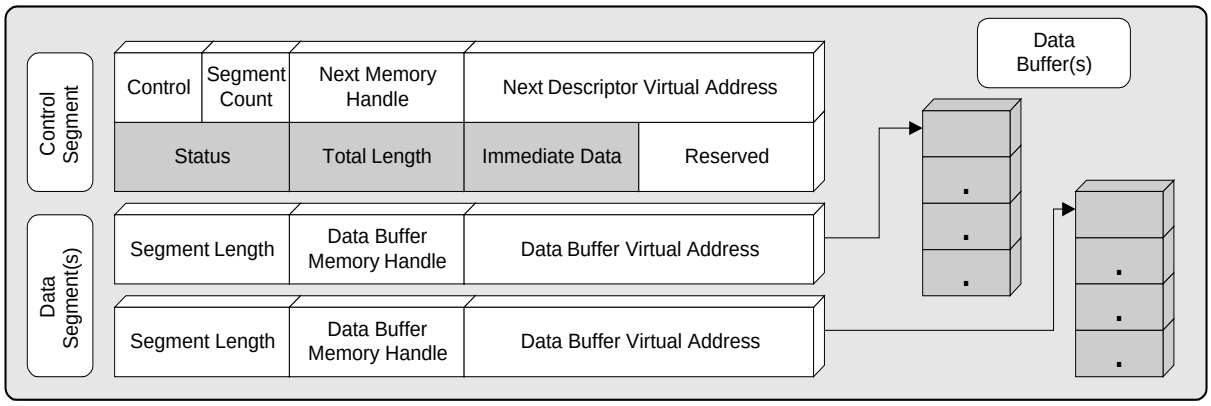

Figure 9: Descriptor fields updated by VI NIC on completion

VIPL

When the VI Application calls VipSendDone or VipRecvDone, VIPL checks the Done bit, dequeues the Descriptor and returns the updated Descriptor to the VI Application as shown in Figure 10. The VI application cannot use or make assumptions about the contents of the Next Handle and Virtual Address fields. (Note that for these discussions the Wait and Notify calls have the same effect as the Done calls).

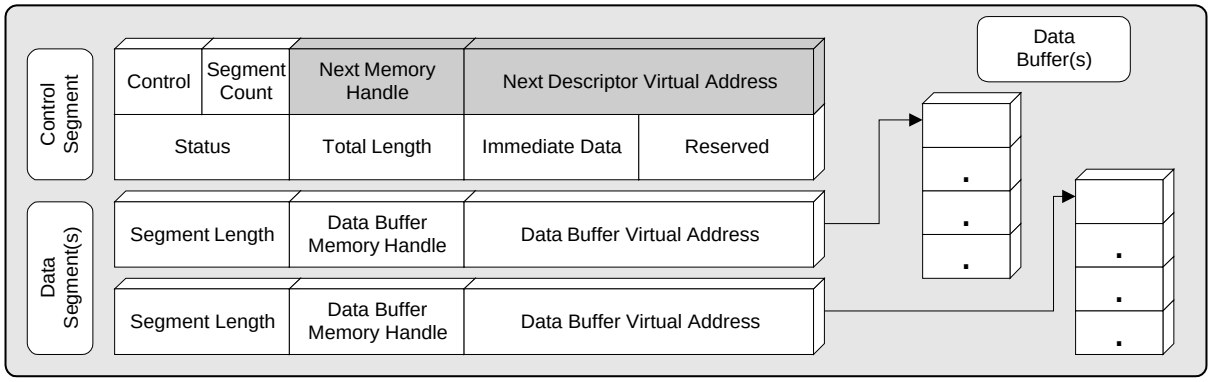

Figure 10: Descriptor fields updated by VIPL on completion

VI Application

The VI Application takes ownership of the Descriptor when the Descriptor pointer is returned by VipSendDone or VipRecvDone.

6. VI Provider Notes

This section contains clarifications and recommendations for VI Providers.

6.1. Completion Queue Ordering

To preserve the Completion Queue semantics, the Completion Queue entries must be written in the same order as the Descriptor entries on the VI work queues. For example, an RDMA Read could complete after a Send (or RDMA Write) operation even though the Descriptor for the RDMA Read is ahead of the Send (or RDMA Write) Descriptor in the VI work queue. In this case, the Completion Queue entry for the Send (or RDMA Write) operation must not appear on the Completion Queue before the entry for the completed RDMA Read.

6.2. Disconnect Notification

In Section 4.2, the VI Architecture Specification states the following: " A VI Provider may issue an asynchronous notification to the VI Consumer of a VI that has been disconnected by the remote end, but this feature is not a requirement. A VI Provider is required to detect that a VI is no longer connected and notify the VI Consumer. Minimally, the consumer must be notified upon the first data transfer operation that follows the disconnect."

In some cases, it is not possible to notify the consumer that the remote side has disconnected without an asynchronous error. For this reason, it is recommended that the VI Provider issue a Connection Lost error at both ends of the connection when the connection is dropped as a result of an error. It is also recommended that a Connection Lost error be issued when the corresponding endpoint requests the connection be broken.

Questions have been raised about the proper behavior if a VI that has been disconnected is associated with a Completion Queue. Section 4.2 of the VI Architecture Specification states: "A Disconnect request will result in the completion of all outstanding Descriptors on that VI endpoint. The Descriptors are completed with the appropriate error bit set". Section 6.4.1 of the VI Specification reads: "If a Completion Queue has been registered for the queue that this Descriptor is on, the VI NIC will place an entry on the Completion Queue that indicates the completed Descriptor's VI and queue". The only time that completed Descriptors would not appear in the Completion Queue is when a catastrophic error occurred such that the hardware could no longer update the Completion Queue.

6.3. Error Handling

For a detailed table depicting the details of error handling for each reliability level, please see the Error Supplement that accompanies this Developer's Guide.

6.3.1. Catastrophic Hardware Errors

The VI Architecture Specification does not specify how catastrophic hardware errors are handled. Catastrophic hardware errors are asynchronous errors that render the Work Queues and/or Completion Queues inoperable by the hardware. That is, the hardware is unable to read Descriptors, write status to Descriptors and/or write Completion Queue entries. The recommendation for handling catastrophic hardware errors is for the VI Provider software (VI Kernel Agent or VIPL) to clean up the VI Work Queues and transition the VI to the Error state for all reliability levels. The catastrophic asynchronous errors are Post Descriptor Error (due to all causes), VI Overrun Error and Catastrophic Error.

6.3.2. Completion Queue Overrun

For those hardware implementations where a Completion Queue overrun could result in data corruption, it is recommended that the VI Provider generate a VIP_ERROR_CATASTROPHIC asynchronous error and disconnect the VIs.

6.3.3. Connection Lost on an Unreliable Delivery VI

When a connection is lost on an Unreliable Delivery VI for any reason other than a call to VipDisconnect, an asynchronous error should be issued and transition the VI to the Error state. If VipDisconnect is called, the local endpoint breaks the connection and transitions the VI to the Idle state.

6.4. Thread Safety

The threads created within VIP to service Notify requests will need to synchronize the use of data structures between the Notify service threads entering VipSendNotify, VipRecvNotify and VipCQNotify. Even a non-thread-safe VIPL must ensure exclusive access to these data structures by implementing explicit mutual exclusion algorithms, since the user application has no control over the operation of the Notify service threads.

6.5. VI Device Name

Use of a common VI NIC device allows VI applications to use the same DeviceName parameter for VipOpenNic()across multiple NIC implementations. The recommended VI NIC device naming convention is the ASCII NULL terminated string, VINIC. The ASCII representation for a number can be appended to VINIC to allow multiple NICs from the same vendor to be installed in the system. For example, if two VI NICs are installed in the system, the first NIC is assigned the device name VINIC (VINIC is equivalent to VINIC0) and the second is named VINIC1. This is not intended to solve the problem where a user wants to install and run multiple VI NICs from different vendors at the same time.

6.6. Implications of Posting a Send Descriptor

Section 6.2 of the VI Architecture Specification reads: "Once a Descriptor has been prepared, the VI Consumer submits it to the VI NIC for processing by posting it to the appropriate VI Work Queue and ringing the queue's Doorbell." Section 6.3 reads: "Once a Descriptor has been posted to a queue, the VI NIC can begin processing it. Data is transferred between two connected VIs when a VI NIC processes a Descriptor or as the data arrives on the network."

The above two paragraphs indicate that on a send, data is transferred when the VI NIC processes a Descriptor and Descriptor processing can begin as soon as it has been posted on a queue. No other VIPL call is needed to initiate the data transfer.

6.7. Connection Management Clarification

This section contains details regarding comparison and exchange of net addresses during the connection negotiation process. The five diagrams and explanations that follow detail the portions of the addresses (both local and remote) and the VI attributes (both local and remote) that are compared and exchanged as part of the connection management process.

The five diagrams are:

1. *Local and remote address comparison and data exchange for peer-to-peer connections (VipConnectPeerRequest)*
2. *Local and remote address comparison and exchange for client/server connections (VipConnectWait and VipConnectRequest)*
3. *Local and remote VI attributes comparison and exchange for all connection types*
4. *Combined address and VI attribute comparison and data exchange diagram for peer-to-peer connections*
5. *Combined address and VI attribute comparison and data exchange diagram for client/server connections*

Address Comparison and Data Exchange for Peer-to-peer Connections

As shown in Figure 11, the address parameters are not exchanged for peer-to-peer connections. Since the peer-to-peer connection model is rendezvous based, the comparison of the host portions of the local and remote addresses and the discriminators in the remote addresses occurs after both peers have issued their connect requests. It is not specified where this comparison occurs. That is, the comparison can occur on either of the two peers involved or on another node that provides connection services.

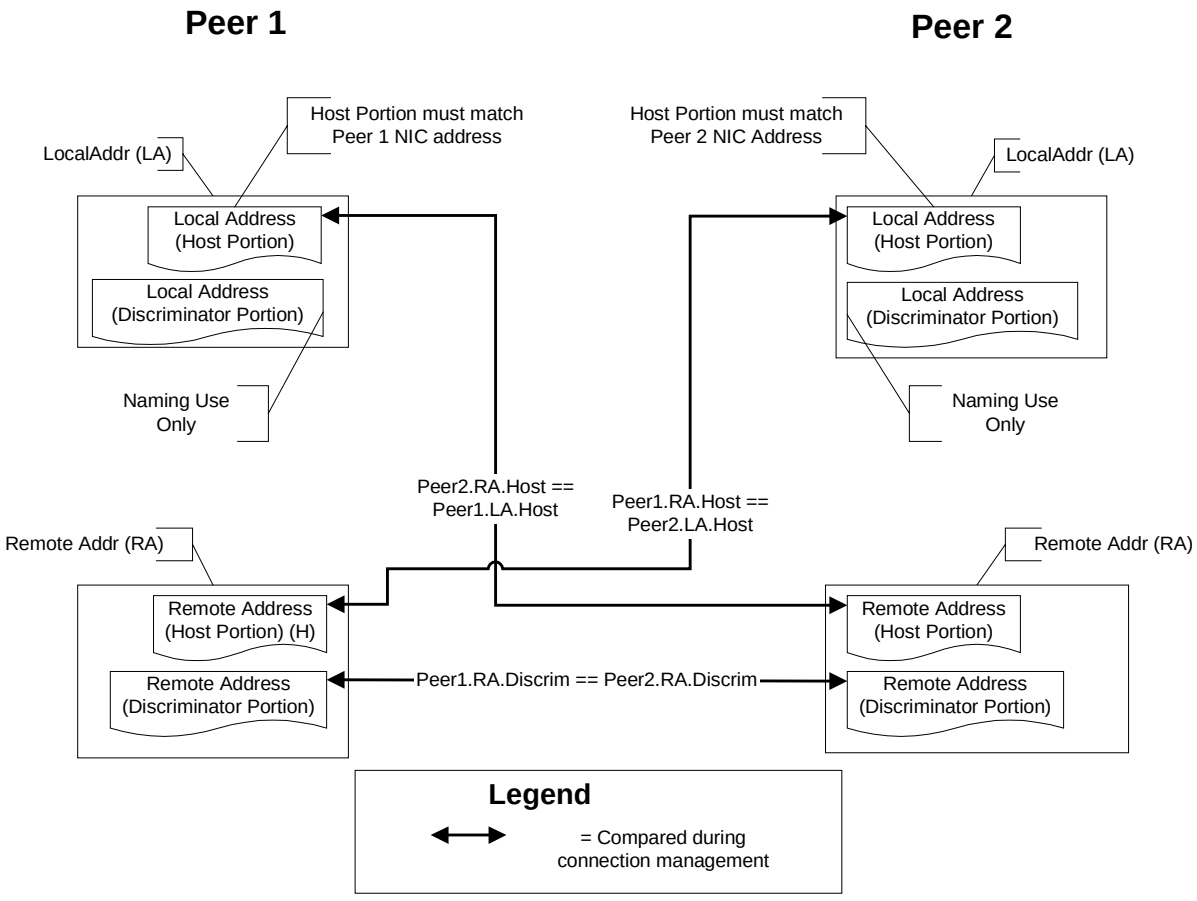

Figure 11: Address Comparison and Data Exchange for Peer-to-Peer

Address Comparison and Data Exchange for Client/Server Connections

The comparison of addresses follows a different pattern for client/server connections, as shown in Figure 12. Notice that the remote address for the server is above the local address on the Server side of the diagram. The client/server model requires the server VipConnectWait() to be issued prior to the client VipConnectRequest(). As mentioned in the previous section, the location of the comparison occurs is not specified. (Possible locations are the server node or some other node that provides connections services)

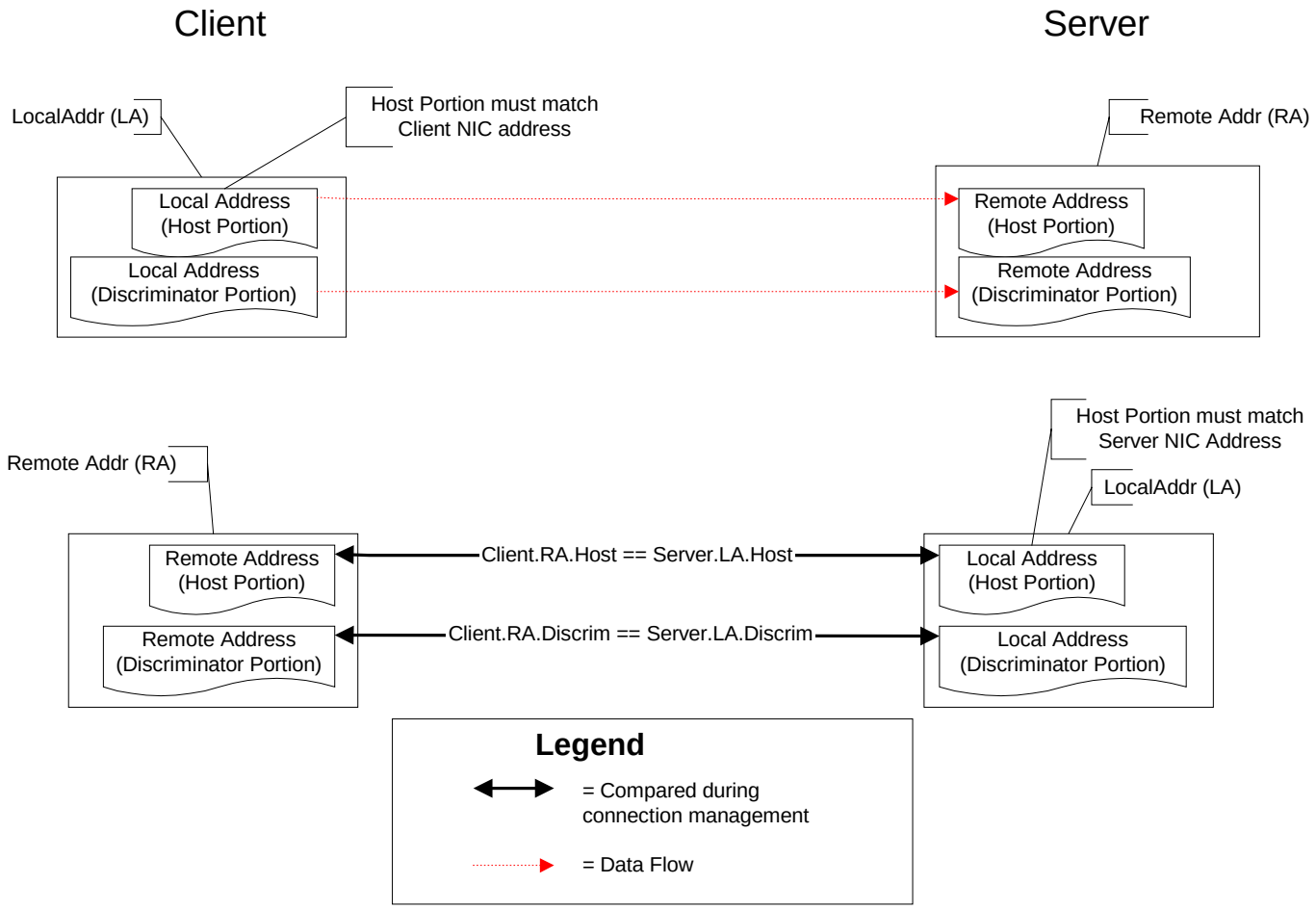

Figure 12: Address Comparison and Data Exchange for Client/Server

VI Attribute Comparison and Exchange

As detailed in Figure 13, the comparison of the VI attributes is the same for both connection types. The local attributes for the VIs involved in the connection are compared. Notice that the Ptag field of the VIP_VI_ATTRIBUTES structure is neither compared nor exchanged as part of the connection process. (Ptags are used only to validate the relationship between VIs and memory regions locally). Again, the location of the comparison is not specified, though it is likely that the location is coincident with the location where the address parameters are compared.

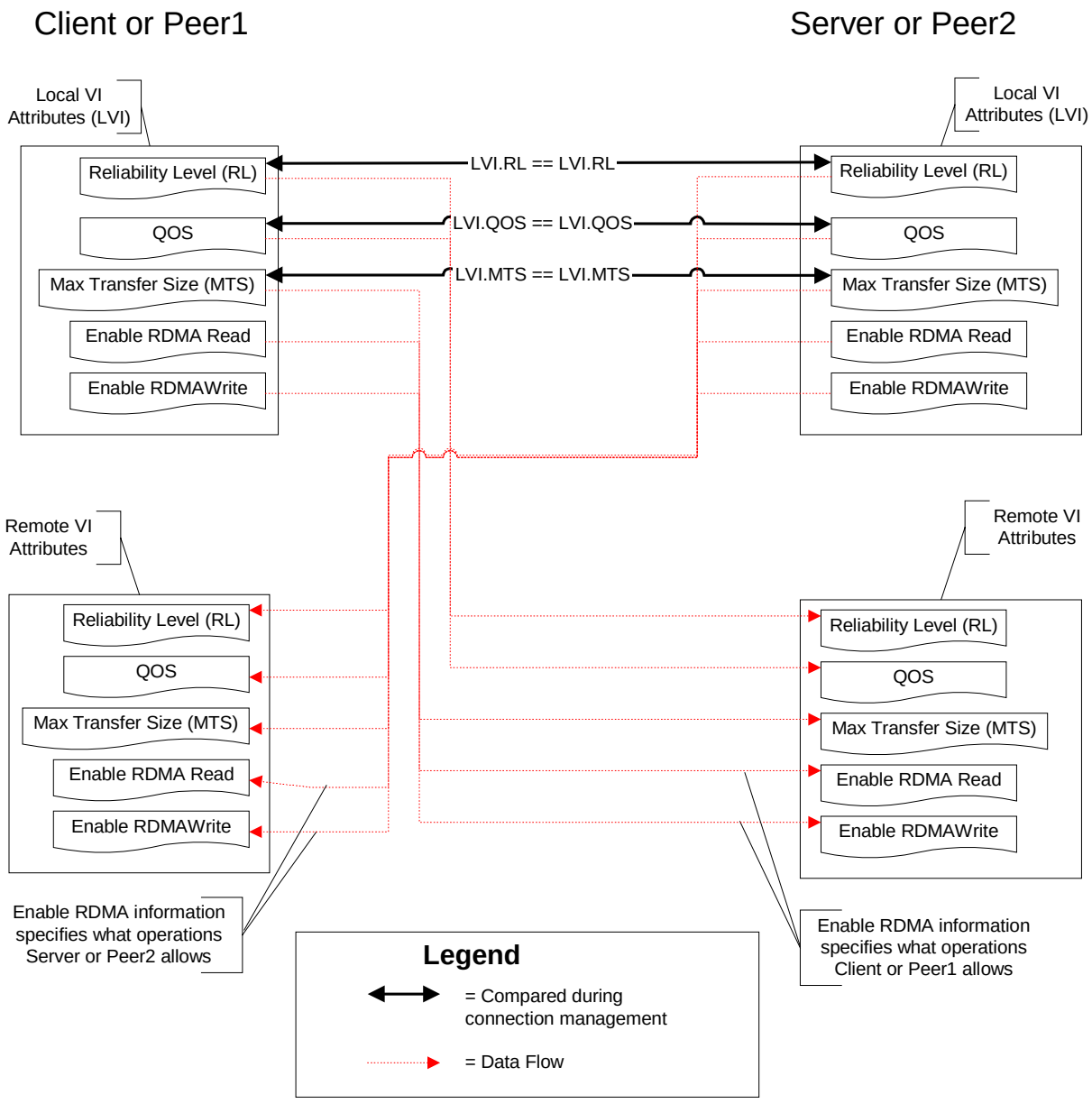

Figure 13: VI Attribute Comparison and Exchange

Peer-to-peer Combined Address and VI Attribute Comparison and Exchange

Figure 14 combines Figure 11 and Figure 13 and shows the ordering of the address and data comparisons described in the previous figures. Space limitations prevent the first two steps of the control flow shown in Figure 14 from fully documenting the correct behavior.

The diagram assuems that all peer-to-peer requests between two nodes (as identified by the tuple <Peer1.LA.Host, Peer2.LA.Host>) are tracked in a single "peer pool" and have their discriminators matched in an ordered way. Once two outstanding peer-to-peer connection requests are determined to have matching discriminators, the requests are removed from the "peer pool" and the rest of the control steps in the diagram are followed. Otherwise, the outstanding peer-to-peer connection requests remain in the "peer pool" until they have timed out. This is represented by the VIP_TIMEOUT transition in Figure 14.

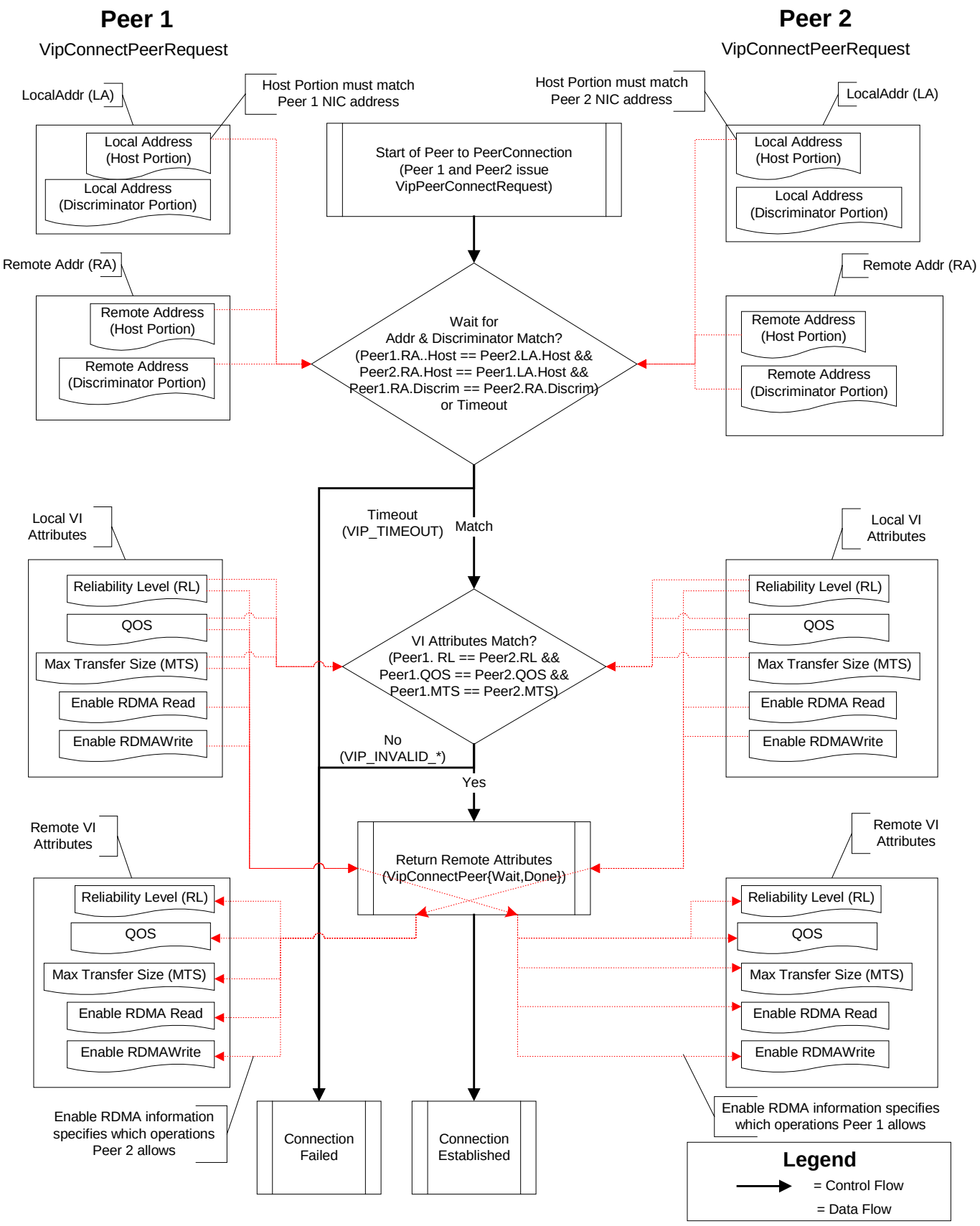

Figure 14: Peer-to-Peer Combined

Client Server Combined Address and VI Attribute Comparison and Exchange

Figure 15 combines Figure 12 and Figure 13 and shows the ordering of the address and data comparisons described in the previous figures.

The diagram assumes that all client/server connection requests between two nodes (as identified by the tuple <Client.Host, Server.Host>) are tracked in a single "client/server pool". Server connection requests are kept in the "client/server pool" until a matching client connection request arrives or the server request times out. Client connection requests are either immediately matched upon arrival in the "client/server pool" or they fail. (VipConnectRequest returns VIP_NO_MATCH). When a client connection request is matched to a server connection request, both are removed from the "client/server pool" and the subsequent control steps in Figure 15 are performed.

Space limitations prevent the transition from below the "VI Attributes Match?" decision to above the "Server Issues VipConnectAccept or VipConnectReject" from fully documenting the correct behavior.

This transition occurs when a server supplies a VI by calling VipConnectAccept with attributes that do not match the client VI's attributes. An error is returned on the server and client does not receive an error. In addition, if the server does not supply another VI with the correct attributes or rejects the request by calling VipConnectReject within the timeout period specified by the client, the client will time out and VipConnectRequest returns VIP_TIMEOUT). If the server attempts to accept the connection by calling VipConnectAccept after the client has timed out, the error returned is VIP_TIMEOUT. If the server attempts to reject the connection via VipConnectReject, then VIP_SUCCESS is returned, because the connection is not established and the client and server both agree that the connection request has failed.

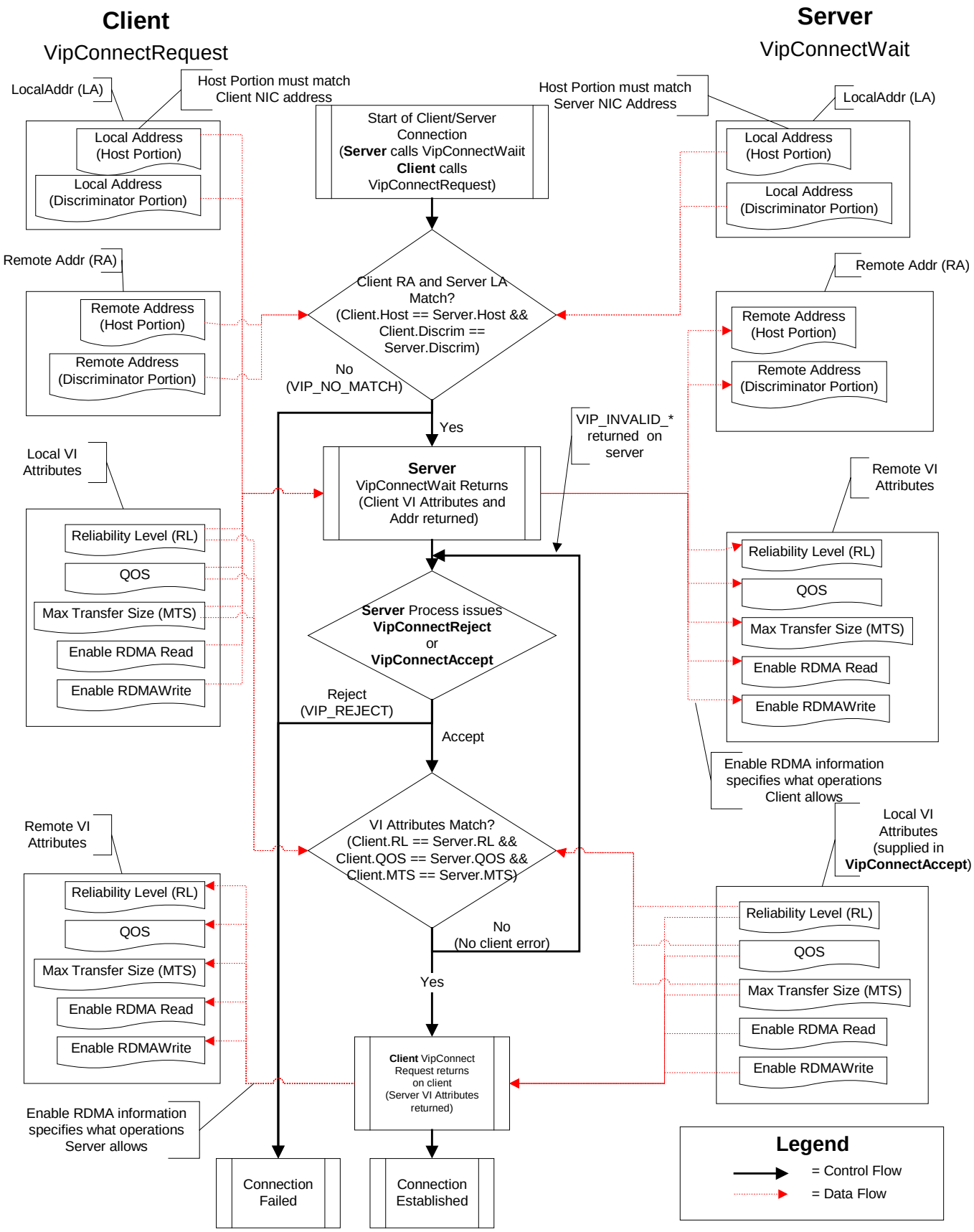

Figure 15: Client/Server Combined

6.8. VIP_NOT_REACHABLE

The error code, VIP_NOT_REACHABLE is an optional error return for VI connection operations for VI Providers that can detect a network partition. This error code is returned if a network partition is detected.

A permissible alternative for interconnects that cannot distinguish network partitions is to continue to attempt to make a connection until the timeout period expires and return a VIP_TIMEOUT error in the case of a network partition.

7. Application Notes

This section contains recommendations and warnings for application developers on using the VIPL interface.

7.1. Completion Queue Usage

It is recommended that the VI Application maintain a one-to-one mapping between entries on the Completion Queue and the Descriptors on the work queues associated with the Completion Queue. That is, the application should first check the Completion Queue for a completion prior to de-queuing the corresponding completed Descriptor from a work queue.

Furthermore, the application developer should take the appropriate precautions to avoid a Completion Queue overrun. The application developer will not be notified when a Completion Queue overrun occurs. When a Completion Queue overrun occurs, the one-to-one mapping of completion entries to completed Descriptors on the associated VI work queues will no longer be maintained and Completion Queue entries may be silently lost, corrupted or duplicated. The effect on the application of a Completion Queue overrun error is undefined.

7.2. Interaction of Notify, Done and Wait Calls

Notify calls should not be used concurrently with either Done or Wait calls on the same work queue. If a Notify is outstanding when a Done or a Wait call is made, the order in which completed Descriptors are associated with the calls is indeterminate.

Done and Wait calls should not be used concurrently even in a thread-safe implementation. If one thread invokes a Done call while another thread is waiting on the same VI work queue, the order in which completed Descriptors are associated with the calls is indeterminate.

7.3. Data Alignment

It is permissible to align data buffers on any arbitrary byte boundary. However, system performance may improve if data buffers are aligned on natural boundaries. The following is a list of alignments from best to least desirable: page, cache line, 8 byte and byte.

For the best future performance, the recommendation is to align data buffers on either 256 byte or page boundaries and Descriptors on 256 byte boundaries.

7.4. Connection Discriminator Usage

A connection discriminator can be interpreted as a port number, a Universally Unique Identifier (UUID) or as a sequence of arbitrary bytes. The VI Providers are expected to provide a minimum of 16 bytes as the MaxDiscriminatorLen to allow the application to interpret the discriminator as a UUID.

In the client/server connection model, the application must be aware that it is possible that more than one server process could be using the same discriminator. If more than one server process is using the same discriminator, it is indeterminate which server process will handle the incoming connection request from the client specifying that discriminator.

For portability it is recommended that the application use a different discriminator for each VipConnectRequest() call. Some VI Provider implementations may not allow multiple VipConnectRequest() calls with the same discriminator, even though they are connecting to different nodes.

7.5. Catastrophic Hardware Error Handling

If the application receives a notification of a catastrophic hardware error on a VI, the application should follow the steps outlined below:

1. *Call VipDisconnect to disconnect the VI.*
2. *Remove the Descriptors on both Work Queues associated with the VI using the Work Queue completion calls, VipSendDone and VipRecvDone.*
3. *Destroy the VI.*
4. *If a Completion Queue is associated with the VI that suffered the catastrophic error, destroy the CQ.*

The application must also be able to handle "stale" Completion Queue indications for VIs that suffer a catastrophic hardware error. That is, the application must be able to handle encountering Completion Queue indications for a VI that was destroyed when "normal" processing resumes.

If the application receives a notification of a catastrophic hardware error on a Completion Queue, the application should follow the steps outlined in the previous paragraph to clean up all of the VIs associated with the Completion Queue. When the VIs have been destroyed, the application should destroy the Completion Queue.

7.6. Error Reporting on Unreliable VIs

If a transport error occurs on an operation that does not consume a Descriptor for Unreliable VIs, an error will not be reported. (This includes transport errors that result in data corruption). Therefore, it is highly recommended that an application that uses RDMA Write operations on Unreliable VIs also specify Immediate Data to allow transport errors to be reported in the Descriptor status.

7.7. Interconnect Failure Detection

The VI Architecture Specification only guarantees that the application is notified of an interconnect failure when the next send Descriptor is processed. If the application has receive Descriptors posted and needs to know immediately that a failure has occurred, the application must provide the mechanism to detect the failure.

7.8. Blocked VipConnectWait

If a VipConnectWait call is blocked, the only way that the application can cancel the operation prior to the timeout expiration is to kill the thread that issued the call.

8. Programming Examples

8.1. Overview

This section describes a sample VI application that demonstrates the mechanics of writing a Client-Server application using the VIPL APIs described in the previous sections. The source and instructions on how to build and run the sample application can be found at the following web address: http://www.intel.com/design/servers/vi/developer/ia_imp_guide.htm#imp_guide. The sample application is intended for documenting important concepts when using the VIPL APIs, such as name service, memory management rules, connection semantics, data transfer operations, handling completion events, and managing multiple VI connections simultaneously. The sample application is composed of two parts. a) an EchoServer, and b) an EchoClient.

8.1.1. EchoServer

The EchoServer is the server side module of the application. This module allows a pre-specified maximum number of concurrent clients to connect to the server and run a simple Request/Reply protocol. Clients are free to connect to the server, use the echo-service of the server and disconnect from the server at any time. The server echoes any request message received back to the originating client as the reply message. The server uses a listen thread to accept new client connections. The Receive Queues of the connected VIs are attached to a global Completion Queue. The server runs a worker thread to constantly wait (poll) on the global completion queue and invokes a user-level handler function to process completed receive operations. The handler function dequeues the message received from the corresponding client and echoes the received request message back to the client. The server polls the corresponding Send Queue to wait for a send operation to complete because the Send Queues are not associated with any Completion Queue. This application illustrates the use of both the Work Queue Model and the Completion Queue Model for processing the Descriptors.

8.1.2. EchoClient

The EchoClient takes a server name and the number of Request/Reply iterations as the command line parameters. It is a single threaded application that connects to the target service address on the specified server, runs the Request/Reply loop over the specified iterations by using VIPL Data transfer and completion operations, and finally disconnects from the server.

8.1.3. Application Limitations:

The sample application makes several assumptions for the sake of simplicity and ease of understanding. The following list provides a brief description of some of the application's limitations.

- *All application messages are assumed to be less than or equal to the MaxTransferSize of the VI NIC.*
- *The sample application does not address issues like flow-control, error recovery, reliability etc.*
- *The application only uses Reliable Delivery VIs.*
- *Most of the error conditions are handled by using a MessageBox, rather than by using error handlers. In EchoServer, errors encountered during data transfer operations on a specific*

client connection result in disconnecting the erroneous connection, without affecting the service of any other client connections.

- *The application does not register an asynchronous error handler. The behavior of the default error handler is assumed. Real world applications with specific error logic should use appropriate error handlers.*
- *Since the VipConnectWait call blocks until a client requests a connection (or until timeout expires), multiple server listen threads are required to accept more than one client connection at a time. This sample application accepts only one client connection at a time. This does not mean that only one client can be connected to the server at any time.*
- *Only Client/Server connection semantics are used in the sample application. The Peer-to-Peer connection model is not used in the sample application.*
- *The application does not illustrate the use of Vip{Send,Recv,CQ}Notify routines.*
- *RDMA data-transfer operations are not used by this sample application.*

Standard disclaimers and Copyright notice described in Appendix (Section 6.1) applies to any code samples, programs, and application notes described in this document. This sample application targets a Windows™ NT 4.0 operating system environment and uses the standard Win32 APIs for requesting operating systems specific operations.

8.1.4. Helper Functions

For easier illustration, the sample application program is divided into a set of helper functions and the core client and server code. The helper functions abstract out various VIPL operations including endpoint initialization and shutdown, communication memory management, managing connections, setting up Descriptors, etc.

9. Future Considerations

This section contains issues that are under consideration for incorporation into a future release of the Intel VI Architecture Developer's Guide. These issues were brought up too late in the cycle to be resolved and incorporated into the first VI NIC products and the 1.0 version of the guide.

9.1. Discriminator Binding

Few restrictions exist on the usage of discriminators in the Intel VI Architecture Developer's Guide. The semantics were intentionally left broad because connection management can be implemented successfully with the minimal restrictions.

However, requests have been made to be able to "bind" discriminators, similar to the BSD sockets "bind" procedure call. This is an area to be investigated further. If required, API additions will be specified to provide clear semantics for discriminator usage across multiple implementations of VIPL.

9.2. VI Handle Extensions

A useful extension to the VI handle is to allow a user supplied handle to be associated with a VI handle along with a way to access the user handle from the VI handle. This provides an efficient mechanism for the application to find the data structure used to access the VI. A code fragment that shows how this feature could be used is shown below:

```
status = VipCQDone(CqHandle, &ViHandle, &RecvQueue);

// Look up the structure associated with the VI Handle

AppViStructP = LookUpAppViStruct(ViHandle);

// If implementation is not thread safe, lock access
if(RecvQueue)
{
      GrabViLock(AppViStructP->LockStructure);
      Status = VipRecvDone(ViHandle, &DescriptorPtr);
      ReleaseViLock(AppViStructP->LockStructure);
}
else
{
      GrabViLock(AppViStructP->LockStructure);
      Status = VipSendDone(ViHandle, &DescriptorPtr);
      ReleaseViLock(AppViStructP->LockStructure);
}
```

This example shows how a call such as `LookUpAppViStruct()` that returns the user defined pointer, `AppViStructP`, could be used. In this example, the data structure pointed to by `AppViStructP` is used for locking access to the queue. Associating the data structure with the VI Handle is a convenient way to keep track and access the application data structure for the VI.

Extensions to provide this functionality may be proposed in a future release of this guide.

9.3. Error Codes

A request was made to provide a mechanism to help application developers determine exactly what caused an error. For example, in the case of a VIP_INVALID_PARAMETER the error code does not reflect which parameter was invalid. Another example is VIP_ERROR_RESOURCE when it is used both for reporting resource conflicts and insufficient system resources. This is an area to investigate for the future.

9.4. Hardware Heartbeat NIC Attribute Field

A request was made to add a field to the NIC Attributes structure that the user application can use to determine whether a hardware implementation provides a "heartbeat" mechanism that automatically informs the application of hardware failures. The application could avoid duplicating the functionality if a mechanism was provided to determine whether the NIC already had a "heartbeat" mechanism built in and thus improve performance.

10. Appendix A - Include Files

10.1. Vipl.h

```
/*++

Copyright (c) 1996, 1997 Intel Corp. All Rights Reserved.

VIRTUAL INTERFACE ARCHITECTURE VIPL.H

Copyright (c) - 1998 Intel Corporation

Intel Corporation hereby grants a non-exclusive license under Intel's
copyright to copy, modify and distribute this software for any purpose
and without fee, provided that the above copyright notice and the
following paragraphs appear on all copies.

Intel Corporation makes no representation that this source code
is correct or is an accurate representation of any standard.

IN NO EVENT SHALL INTEL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, OR
SPECULATIVE DAMAGES, (INCLUDING WITHOUT LIMITING THE FORGOING,
CONSEQUENTIAL, INCIDENTAL AND SPECIAL DAMAGES) INCLUDING, BUT NOT
LIMITED TO INFRINGEMENT, LOSS OF USE, BUSINESS INTERRUPTIONS, AND LOSS
OF PROFITS, IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE
POSSIBILITY OF ANY SUCH DAMAGES.

INTEL CORPORATION SPECIFICALLY DISCLAIMS ANY WARRANTIES INCLUDING, BUT
NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR
A PARTICULAR PURPOSE AND NON-INFRINGEMENT.  THE SOFTWARE PROVIDED
HEREUNDER IS ON AN "AS IS" BASIS AND INTEL CORPORATION HAS NO
OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS OR
MODIFICATIONS.

Module Name:
      vipl.h

Abstract:

  vipl.h contains the complete user interface to the VIPL.

--*/

#ifndef     _VIPL_
#define     _VIPL_      1

/*****************************************************************************
 * Constants
*****************************************************************************/

#define IN
#define OUT
```

```
/***********************************************************************
 * Data Types
 ***********************************************************************/

/*
 * Generic types for portability
 */
typedef     void *            VIP_PVOID;
typedef     int               VIP_BOOLEAN;
typedef     char              VIP_CHAR;
typedef     unsigned char     VIP_UCHAR;
typedef     unsigned short    VIP_USHORT;
typedef     unsigned long     VIP_ULONG;

/*
 * The following are NT specific
 */
typedef     unsigned __int64  VIP_UINT64;
typedef     unsigned __int32  VIP_UINT32;
typedef     unsigned __int16  VIP_UINT16;
typedef     unsigned __int8   VIP_UINT8;

typedef     VIP_PVOID   VIP_QOS;        /* details are not defined */

/*
 * This constant is used only by the user and the VIPL library.
 */

#ifndef VIP_KERNEL

#define VIP_INFINITE (~(VIP_ULONG) 0)

#endif  /* !VIP_KERNEL */

/*
 * Constants used with VIP_BOOLEAN
 */
#define VIP_TRUE  (1)
#define VIP_FALSE (0)

/*
 * Handles are used for opaque objects.
 */
typedef VIP_PVOID       VIP_NIC_HANDLE;
typedef VIP_PVOID       VIP_VI_HANDLE;
typedef VIP_PVOID       VIP_PROTECTION_HANDLE;
typedef VIP_PVOID       VIP_CONN_HANDLE;
typedef VIP_PVOID       VIP_CQ_HANDLE;

/*
 * Handles used by the NIC hardware
 */
typedef VIP_UINT32      VIP_MEM_HANDLE;
```

```
/*
 * Recommended NIC name
 */
#define VINICBASENAME   "VINIC"

/*
 * VI Network Address
 *
 * A VI Network Address holds the network specific address for a VI
 * endpoint. Each VI Provider may have a unique network address format.
 * It is composed of two elements, a host address and an endpoint
 * discriminator. These elements are qualified with a byte length in
 * order to maintain network independence.
 */
typedef struct {
        VIP_UINT16      HostAddressLen;
        VIP_UINT16      DiscriminatorLen;
        VIP_UINT8       HostAddress[1];
} VIP_NET_ADDRESS;

typedef VIP_USHORT      VIP_RELIABILITY_LEVEL;

/*
 *  Bit values for VIP_RELIABILITY_LEVEL
 */
#define VIP_SERVICE_UNRELIABLE            0x01
#define VIP_SERVICE_RELIABLE_DELIVERY     0x02
#define VIP_SERVICE_RELIABLE_RECEPTION    0x04

/*
 * A VIP_NIC_ATTRIBUTES describes a nic.
 *
 * The NIC attributes structure is returned from the VipQueryNic
 * function. It contains information related to an instance of a NIC
 * within a VI Provider. All values that are returned in the NIC
 * Attributes structure are static values that are set by the VI
 * Provider at the time that it is initialized.
 */
typedef struct {
        VIP_CHAR                Name [64];
        VIP_ULONG               HardwareVersion;
        VIP_ULONG               ProviderVersion;
        VIP_UINT16              NicAddressLen;
        const VIP_UINT8         *LocalNicAddress;
        VIP_BOOLEAN             ThreadSafe;
        VIP_UINT16              MaxDiscriminatorLen;
        VIP_ULONG               MaxRegisterBytes;
        VIP_ULONG               MaxRegisterRegions;
        VIP_ULONG               MaxRegisterBlockBytes;
        VIP_ULONG               MaxVI;
        VIP_ULONG               MaxDescriptorsPerQueue;
        VIP_ULONG               MaxSegmentsPerDesc;
        VIP_ULONG               MaxCQ;
        VIP_ULONG               MaxCQEntries;
        VIP_ULONG               MaxTransferSize;
        VIP_ULONG               NativeMTU;
        VIP_ULONG               MaxPtags;
```

```
        VIP_RELIABILITY_LEVEL   ReliabilityLevelSupport;
        VIP_RELIABILITY_LEVEL   RDMAReadSupport;
} VIP_NIC_ATTRIBUTES;

/*
 * Memory alignment required by the NIC for descriptors, in bytes.
 */
#define VIP_DESCRIPTOR_ALIGNMENT    64

/*
 *  The address structure for registered memory accesses.
 */
typedef union {
     VIP_UINT64     AddressBits;
     VIP_PVOID      Address;
} VIP_PVOID64;

/*
 * The control portion of the descriptor.
 */
typedef struct {
      VIP_PVOID64       Next;
      VIP_MEM_HANDLE    NextHandle;
      VIP_UINT16        SegCount;
      VIP_UINT16        Control;
      VIP_UINT32        Reserved;
      VIP_UINT32        ImmediateData;
      VIP_UINT32        Length;
      VIP_UINT32        Status;
} VIP_CONTROL_SEGMENT;

/*
 * Descriptor RDMA Segment
 */
typedef struct {
      VIP_PVOID64       Data;
      VIP_MEM_HANDLE    Handle;
      VIP_UINT32        Reserved;
} VIP_ADDRESS_SEGMENT;

/*
 * Descriptor Data Segment
 */
typedef struct {
      VIP_PVOID64       Data;
      VIP_MEM_HANDLE    Handle;
      VIP_UINT32        Length;
} VIP_DATA_SEGMENT;

typedef union {
     VIP_ADDRESS_SEGMENT     Remote;
     VIP_DATA_SEGMENT        Local;
} VIP_DESCRIPTOR_SEGMENT;
```

```
/*
 * Complete VI Descriptor
 */
typedef struct _VIP_DESCRIPTOR {
      VIP_CONTROL_SEGMENT     CS;
      VIP_DESCRIPTOR_SEGMENT  DS[2];
} VIP_DESCRIPTOR;

/*
 * Bit field macros for descriptor control segment: Control field
 */
#define     VIP_CONTROL_OP_SENDRECV                   0x0000
#define     VIP_CONTROL_OP_RDMAWRITE                  0x0001
#define     VIP_CONTROL_OP_RDMAREAD                   0x0002
#define     VIP_CONTROL_OP_RESERVED                   0x0003
#define     VIP_CONTROL_OP_MASK                       0x0003
#define     VIP_CONTROL_IMMEDIATE                     0x0004
#define     VIP_CONTROL_QFENCE                        0x0008
#define     VIP_CONTROL_RESERVED                      0xFFF0

/*
 * Bit field macros for descriptor control segment: Status field
 */
#define     VIP_STATUS_DONE                           0x00000001
#define     VIP_STATUS_FORMAT_ERROR                   0x00000002
#define     VIP_STATUS_PROTECTION_ERROR               0x00000004
#define     VIP_STATUS_LENGTH_ERROR                   0x00000008
#define     VIP_STATUS_PARTIAL_ERROR                  0x00000010
#define     VIP_STATUS_DESC_FLUSHED_ERROR             0x00000020
#define     VIP_STATUS_TRANSPORT_ERROR                0x00000040
#define     VIP_STATUS_RDMA_PROT_ERROR                0x00000080
#define     VIP_STATUS_REMOTE_DESC_ERROR              0x00000100
#define     VIP_STATUS_ERROR_MASK                     0x000001FE

#define     VIP_STATUS_OP_SEND                        0x00000000
#define     VIP_STATUS_OP_RECEIVE                     0x00010000
#define     VIP_STATUS_OP_RDMA_WRITE                  0x00020000
#define     VIP_STATUS_OP_REMOTE_RDMA_WRITE           0x00030000
#define     VIP_STATUS_OP_RDMA_READ                   0x00040000
#define     VIP_STATUS_OP_MASK                        0x00070000
#define     VIP_STATUS_IMMEDIATE                      0x00080000
#define     VIP_STATUS_RESERVED                       0xFFF0FE00

/*
 * VIP_VI_STATE is used in the return of VipQueryVi.
 */
typedef enum {
      VIP_STATE_IDLE,
      VIP_STATE_CONNECTED,
      VIP_STATE_CONNECT_PENDING,
      VIP_STATE_ERROR
} VIP_VI_STATE;
```

```
/*
 * VIP_VI_ATTRIBUTES is used to create and query VIs.
 */
typedef struct {
      VIP_RELIABILITY_LEVEL         ReliabilityLevel;
      VIP_ULONG                     MaxTransferSize;
      VIP_QOS                       QoS;
      VIP_PROTECTION_HANDLE         Ptag;
      VIP_BOOLEAN                   EnableRdmaWrite;
      VIP_BOOLEAN                   EnableRdmaRead;
} VIP_VI_ATTRIBUTES;

/*
 * VIP_MEM_ATTRIBUTES is used to create and query Memory regions.
 */
typedef struct _VIP_MEM_ATTRIBUTES  {
      VIP_PROTECTION_HANDLE         Ptag;
      VIP_BOOLEAN                   EnableRdmaWrite;
      VIP_BOOLEAN                   EnableRdmaRead;
} VIP_MEM_ATTRIBUTES;

/*
 * Error Descriptor Error Codes
 */
typedef enum _VIP_ERROR_CODE {
      VIP_ERROR_POST_DESC,
      VIP_ERROR_CONN_LOST,
      VIP_ERROR_RECVQ_EMPTY,
      VIP_ERROR_VI_OVERRUN,
      VIP_ERROR_RDMAW_PROT,
      VIP_ERROR_RDMAW_DATA,
      VIP_ERROR_RDMAW_ABORT,
      VIP_ERROR_RDMAR_PROT,
      VIP_ERROR_COMP_PROT,
      VIP_ERROR_RDMA_TRANSPORT,
      VIP_ERROR_CATASTROPHIC
} VIP_ERROR_CODE;

/*
 * Error Descriptor Resource Codes
 */
typedef enum _VIP_RESOURCE_CODE {
      VIP_RESOURCE_NIC,
      VIP_RESOURCE_VI,
      VIP_RESOURCE_CQ,
      VIP_RESOURCE_DESCRIPTOR
} VIP_RESOURCE_CODE;
```

```
/*
 * Error Descriptor for Asynch. errors
 */
typedef struct _VIP_ERROR_DESCRIPTOR {
      VIP_NIC_HANDLE          NicHandle;
      VIP_VI_HANDLE           ViHandle;
      VIP_CQ_HANDLE           CQHandle;
      VIP_DESCRIPTOR          *DescriptorPtr;
      VIP_ULONG               OpCode;
      VIP_RESOURCE_CODE       ResourceCode;
      VIP_ERROR_CODE          ErrorCode;
} VIP_ERROR_DESCRIPTOR;

/*
 * Return codes from the functions.
 */
typedef enum {
      VIP_SUCCESS,
      VIP_NOT_DONE,
      VIP_INVALID_PARAMETER,
      VIP_ERROR_RESOURCE,
      VIP_TIMEOUT,
      VIP_REJECT,
      VIP_INVALID_RELIABILITY_LEVEL,
      VIP_INVALID_MTU,
      VIP_INVALID_QOS,
      VIP_INVALID_PTAG,
      VIP_INVALID_RDMAREAD,
      VIP_DESCRIPTOR_ERROR,
      VIP_INVALID_STATE,
      VIP_ERROR_NAMESERVICE,
      VIP_NO_MATCH,
      VIP_NOT_REACHABLE
 } VIP_RETURN;

/*
 * InfoType values
 */
#define VIP_SMI_AUTODISCOVERY   ((VIP_ULONG) 1)

/*
 * AutoDiscovery List fields
 */
typedef struct {
    VIP_ULONG        NumberOfHops;
    VIP_NET_ADDRESS  *ADAddrArray;
    VIP_ULONG        NumAdAddrs;
 } VIP_AUTODISCOVERY_LIST;

/*********************************************************************
 * Functions
*********************************************************************/

extern VIP_RETURN
VipOpenNic( IN  const VIP_CHAR      *DeviceName,
            OUT VIP_NIC_HANDLE      *NicHandle);
```

```
extern VIP_RETURN
VipCloseNic( IN VIP_NIC_HANDLE  NicHandle);

extern VIP_RETURN
VipCreateVi( IN     VIP_NIC_HANDLE      NicHandle,
             IN     VIP_VI_ATTRIBUTES   *ViAttribs,
             IN     VIP_CQ_HANDLE       SendCQHandle,
             IN     VIP_CQ_HANDLE       RecvCQHandle,
             OUT    VIP_VI_HANDLE       *ViHandle);

extern VIP_RETURN
VipDestroyVi( IN    VIP_VI_HANDLE   ViHandle);

extern VIP_RETURN
VipConnectWait( IN      VIP_NIC_HANDLE      NicHandle,
                IN      VIP_NET_ADDRESS     *LocalAddr,
                IN      VIP_ULONG           Timeout,
                OUT     VIP_NET_ADDRESS     *RemoteAddr,
                OUT     VIP_VI_ATTRIBUTES   *RemoteViAttribs,
                OUT     VIP_CONN_HANDLE     *ConnHandle);

extern VIP_RETURN
VipConnectAccept( IN    VIP_CONN_HANDLE ConnHandle,
                  IN    VIP_VI_HANDLE   ViHandle);

extern VIP_RETURN
VipConnectReject( IN    VIP_CONN_HANDLE ConnHandle);

extern VIP_RETURN
VipConnectRequest( IN   VIP_VI_HANDLE       ViHandle,
                   IN   VIP_NET_ADDRESS     *LocalAddr,
                   IN   VIP_NET_ADDRESS     *RemoteAddr,
                   IN   VIP_ULONG           Timeout,
                   OUT  VIP_VI_ATTRIBUTES   *RemoteViAttribs);

extern VIP_RETURN
VipDisconnect( IN    VIP_VI_HANDLE      ViHandle);

extern VIP_RETURN
VipCreatePtag( IN   VIP_NIC_HANDLE          NicHandle,
               OUT  VIP_PROTECTION_HANDLE   *Ptag);

extern VIP_RETURN
VipDestroyPtag( IN  VIP_NIC_HANDLE          NicHandle,
                IN  VIP_PROTECTION_HANDLE   Ptag);

extern VIP_RETURN
VipRegisterMem( IN  VIP_NIC_HANDLE      NicHandle,
                IN  VIP_PVOID           VirtualAddress,
                IN  VIP_ULONG           Length,
                IN  VIP_MEM_ATTRIBUTES  *MemAttribs,
                OUT VIP_MEM_HANDLE      *MemoryHandle);

extern VIP_RETURN
VipDeregisterMem( IN    VIP_NIC_HANDLE  NicHandle,
                  IN    VIP_PVOID       VirtualAddress,
                  IN    VIP_MEM_HANDLE  MemoryHandle);
```

```
extern VIP_RETURN
VipPostSend( IN VIP_VI_HANDLE   ViHandle,
             IN VIP_DESCRIPTOR  *DescriptorPtr,
             IN VIP_MEM_HANDLE  MemoryHandle);

extern VIP_RETURN
VipSendDone( IN     VIP_VI_HANDLE   ViHandle,
             OUT    VIP_DESCRIPTOR  **DescriptorPtr);

extern VIP_RETURN
VipSendWait( IN     VIP_VI_HANDLE   ViHandle,
             IN     VIP_ULONG       Timeout,
             OUT    VIP_DESCRIPTOR  **DescriptorPtr);

extern VIP_RETURN
VipPostRecv( IN VIP_VI_HANDLE   ViHandle,
             IN VIP_DESCRIPTOR  *DescriptorPtr,
             IN VIP_MEM_HANDLE  MemoryHandle);

extern VIP_RETURN
VipRecvDone( IN     VIP_VI_HANDLE       ViHandle,
             OUT    VIP_DESCRIPTOR      **DescriptorPtr);

extern VIP_RETURN
VipRecvWait( IN     VIP_VI_HANDLE   ViHandle,
             IN     VIP_ULONG       Timeout,
             OUT    VIP_DESCRIPTOR  **DescriptorPtr);

extern VIP_RETURN
VipCQDone( IN   VIP_CQ_HANDLE   CQHandle,
           OUT  VIP_VI_HANDLE   *ViHandle,
           OUT  VIP_BOOLEAN     *RecvQueue);

extern VIP_RETURN
VipCQWait( IN   VIP_CQ_HANDLE   CQHandle,
           IN   VIP_ULONG       Timeout,
           OUT  VIP_VI_HANDLE   *ViHandle,
           OUT  VIP_BOOLEAN     *RecvQueue);

extern VIP_RETURN
VipSendNotify( IN    VIP_VI_HANDLE  ViHandle,
               IN    VIP_PVOID      Context,
               IN    void(*Handler)(
                        IN  VIP_PVOID       Context,
                        IN  VIP_NIC_HANDLE  NicHandle,
                        IN  VIP_VI_HANDLE   ViHandle,
                        IN  VIP_DESCRIPTOR  *DescriptorPtr
               )
);
```

```
extern VIP_RETURN
VipRecvNotify( IN   VIP_VI_HANDLE   ViHandle,
               IN   VIP_PVOID       Context,
               IN   void(*Handler)(
                        IN  VIP_PVOID       Context,
                        IN  VIP_NIC_HANDLE  NicHandle,
                        IN  VIP_VI_HANDLE   ViHandle,
                        IN  VIP_DESCRIPTOR  *DescriptorPtr
               )
);

extern VIP_RETURN
VipCQNotify(   IN   VIP_CQ_HANDLE   CQHandle,
               IN   VIP_PVOID       Context,
               IN   void(*Handler)(
                        IN  VIP_PVOID       Context,
                        IN  VIP_NIC_HANDLE  NicHandle,
                        IN  VIP_VI_HANDLE   ViHandle,
                        IN  VIP_BOOLEAN     RecvQueue
               )
);

extern VIP_RETURN
VipCreateCQ( IN     VIP_NIC_HANDLE  NicHandle,
             IN     VIP_ULONG       EntryCount,
             OUT    VIP_CQ_HANDLE   *CQHandle);
extern VIP_RETURN
VipDestroyCQ( IN    VIP_CQ_HANDLE   CQHandle);

extern VIP_RETURN
VipResizeCQ( IN VIP_CQ_HANDLE   CQHandle,
             IN VIP_ULONG       EntryCount);

extern VIP_RETURN
VipQueryNic( IN     VIP_NIC_HANDLE      NicHandle,
             OUT    VIP_NIC_ATTRIBUTES  *NicAttribs);

extern VIP_RETURN
VipSetViAttributes(IN   VIP_VI_HANDLE       ViHandle,
                   IN   VIP_VI_ATTRIBUTES   *ViAttribs);

extern VIP_RETURN
VipQueryVi( IN  VIP_VI_HANDLE       ViHandle,
            OUT VIP_VI_STATE        *State,
            OUT VIP_VI_ATTRIBUTES   *ViAttribs,
            OUT VIP_BOOLEAN         *ViSendQEmpty,
            OUT VIP_BOOLEAN         *ViRecvQEmpty);

extern VIP_RETURN
VipSetMemAttributes( IN VIP_NIC_HANDLE      NicHandle,
                     IN VIP_PVOID           Address,
                     IN VIP_MEM_HANDLE      MemHandle,
                     IN VIP_MEM_ATTRIBUTES  *MemAttribs);
```

```
extern VIP_RETURN
VipQueryMem( IN     VIP_NIC_HANDLE           NicHandle,
             IN     VIP_PVOID                Address,
             IN     VIP_MEM_HANDLE           MemHandle,
             OUT    VIP_MEM_ATTRIBUTES       *MemAttribs);

extern VIP_RETURN
VipQuerySystemManagementInfo( IN     VIP_NIC_HANDLE  NicHandle,
                              IN     VIP_ULONG       InfoType,
                              OUT    VIP_PVOID       SysManInfo);

extern VIP_RETURN
VipErrorCallback( IN    VIP_NIC_HANDLE  NicHandle,
                  IN    VIP_PVOID       Context,
                  IN    void(*Handler)(
                           IN   VIP_PVOID               Context,
                           IN   VIP_ERROR_DESCRIPTOR    *ErrorDesc
                        )
);

/*********************************************************************
 * Peer-to-Peer Connection Model APIs
 *********************************************************************/

extern VIP_RETURN
VipConnectPeerRequest( IN   VIP_VI_HANDLE   ViHandle,
                       IN   VIP_NET_ADDRESS *LocalAddr,
                       IN   VIP_NET_ADDRESS *RemoteAddr,
                       IN   VIP_ULONG       Timeout);
extern VIP_RETURN
VipConnectPeerDone( IN  VIP_VI_HANDLE      ViHandle,
                    OUT VIP_VI_ATTRIBUTES  *RemoteViAttribs);

extern VIP_RETURN
VipConnectPeerWait( IN  VIP_VI_HANDLE       ViHandle,
                    OUT VIP_VI_ATTRIBUTES   *RemoteViAttribs);

/*********************************************************************
 * Name service APIs
 *********************************************************************/

extern VIP_RETURN
VipNSInit( IN   VIP_NIC_HANDLE  NicHandle,
           IN   VIP_PVOID       NSInitInfo);

extern VIP_RETURN
VipNSGetHostByName( IN  VIP_NIC_HANDLE  NicHandle,
                    IN  VIP_CHAR        *Name,
                    OUT VIP_NET_ADDRESS *Address,
                    IN  VIP_ULONG       NameIndex);

extern VIP_RETURN
VipNSGetHostByAddr( IN  VIP_NIC_HANDLE  NicHandle,
                    IN  VIP_NET_ADDRESS *Address,
                    OUT VIP_CHAR        *Name,
                    IN  OUT VIP_ULONG   *NameLen);
```

```
extern VIP_RETURN
VipNSShutdown( IN   VIP_NIC_HANDLE  NicHandle);

#endif /* _VIPL_ */
```

10.2. vipl.def

```
LIBRARY  vipl

EXPORTS
   VipOpenNic                    @1
   VipCloseNic                   @2
   VipCreateVi                   @3
   VipDestroyVi                  @4
   VipConnectWait                @5
   VipConnectAccept              @6
   VipConnectReject              @7
   VipConnectRequest             @8
   VipDisconnect                 @9
   VipConnectPeerRequest         @10
   VipConnectPeerDone            @11
   VipConnectPeerWait            @12
   VipCreatePtag                 @13
   VipDestroyPtag                @14
   VipRegisterMem                @15
   VipDeregisterMem              @16
   VipPostSend                   @17
   VipSendDone                   @18
   VipSendWait                   @19
   VipPostRecv                   @20
   VipRecvDone                   @21
   VipRecvWait                   @22
   VipCQDone                     @23
   VipCQWait                     @24
   VipSendNotify                 @25
   VipRecvNotify                 @26
   VipCQNotify                   @27
   VipCreateCQ                   @28
   VipDestroyCQ                  @29
   VipResizeCQ                   @30
   VipQueryNic                   @31
   VipSetViAttributes            @32
   VipQueryVi                    @33
   VipSetMemAttributes           @34
   VipQueryMem                   @35
   VipQuerySystemManagementInfo  @36
   VipErrorCallback              @37
   VipNSInit                     @38
   VipNSGetHostByName            @39
   VipNSGetHostByAddr            @40
   VipNSShutdown                 @41
```