INTERNATIONAL STANDARD

ISO/IEC 14165-331

First edition
2007-07

Information technology –
Fibre channel –

Part 331: Virtual interface (FC-VI)

Reference number
ISO/IEC 14165-331:2007(E)

IEC Central Office
3, rue de Varembé
CH-1211 Geneva 20
Switzerland
Email: inmail@iec.ch
Web: www.iec.ch

About the IEC

The International Electrotechnical Commission (IEC) is the leading global organization that prepares and publishes International Standards for all electrical, electronic and related technologies.

About IEC publications

The technical content of IEC publications is kept under constant review by the IEC. Please make sure that you have the latest edition, a corrigenda or an amendment might have been published.

- Catalogue of IEC publications: www.iec.ch/searchpub
The IEC on-line Catalogue enables you to search by a variety of criteria (reference number, text, technical committee,…). It also gives information on projects, withdrawn and replaced publications.

- IEC Just Published: www.iec.ch/online_news/justpub
Stay up to date on all new IEC publications. Just Published details twice a month all new publications released. Available on-line and also by email.

- Customer Service Centre: www.iec.ch/webstore/custserv
If you wish to give us your feedback on this publication or need further assistance, please visit the Customer Service Centre FAQ or contact us:

Email: csc@iec.ch
Tel.: +41 22 919 02 11
Fax: +41 22 919 03 00

INTERNATIONAL STANDARD

ISO/IEC 14165-331

First edition
2007-07

Information technology – Fibre channel –

Part 331: Virtual interface (FC-VI)

PRICE CODE X

For price, see current catalogue

CONTENTS

INFORMATION TECHNOLOGY – FIBRE CHANNEL –

Part 331: Virtual interface (FC-VI)

FOREWORD

1) ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards. Their preparation is entrusted to technical committees; any ISO and IEC member body interested in the subject dealt with may participate in this preparatory work. International governmental and non-governmental organizations liaising with ISO and IEC also participate in this preparation.

2) In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

3) The formal decisions or agreements of IEC and ISO on technical matters express, as nearly as possible, an international consensus of opinion on the relevant subjects since each technical committee has representation from all interested IEC and ISO member bodies.

4) IEC, ISO and ISO/IEC publications have the form of recommendations for international use and are accepted by IEC and ISO member bodies in that sense. While all reasonable efforts are made to ensure that the technical content of IEC, ISO and ISO/IEC publications is accurate, IEC or ISO cannot be held responsible for the way in which they are used or for any misinterpretation by any end user.

5) In order to promote international uniformity, IEC and ISO member bodies undertake to apply IEC, ISO and ISO/IEC publications transparently to the maximum extent possible in their national and regional publications. Any divergence between any ISO/IEC publication and the corresponding national or regional publication should be clearly indicated in the latter.

6) ISO and IEC provide no marking procedure to indicate their approval and cannot be rendered responsible for any equipment declared to be in conformity with an ISO/IEC publication.

7) All users should ensure that they have the latest edition of this publication.

8) No liability shall attach to IEC or ISO or its directors, employees, servants or agents including individual experts and members of their technical committees and IEC or ISO member bodies for any personal injury, property damage or other damage of any nature whatsoever, whether direct or indirect, or for costs (including legal fees) and expenses arising out of the publication of, use of, or reliance upon, this ISO/IEC publication or any other IEC, ISO or ISO/IEC publications.

9) Attention is drawn to the normative references cited in this publication. Use of the referenced publications is indispensable for the correct application of this publication.

10) Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 14165-331 was prepared by subcommittee 25: Interconnection of information technology equipment, of ISO/IEC joint technical committee 1: Information technology.

The list of all currently available parts of ISO/IEC 14165 series, under the general title *Information technology – Fibre channel,* can be found on the IEC web site.

This International Standard has been approved by vote of the member bodies, and the voting results may be obtained from the address given on the second title page.

This publication has been drafted in accordance with the ISO/IEC Directives, Part 2.

INTRODUCTION

This International Standard defines an upper-layer protocol within the domain of Fibre Channel, that is designed to permit efficient peer-to-peer or client-server messaging between nodes, and to comply with the Virtual Interface (VI) Architecture. Vendors that wish to implement devices that connect to FC-VI may follow the requirements of this and other normatively referenced standards to manufacture an FC-VI compliant device.

INFORMATION TECHNOLOGY – FIBRE CHANNEL –

Part 331: Virtual interface (FC-VI)

1 Scope

This part of ISO/IEC 14165 defines the Fibre Channel mapping protocol for the Virtual Interface (VI) Architecture (FC-VI). FC-VI defines the Fibre Channel Information Units in accordance with the VI Architecture model. FC-VI additionally defines how Fibre Channel services are used to perform the services required by the VI Architecture model of its network transport.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced documents (including any amendments) applies.

ISO/IEC 14165-122, *Information technology – Fibre channel – Arbitrated Loop-2 (FC-AL-2)*

ISO/IEC 14165-251, *Information technology – Fibre channel – Framing and Signalling Interface (FC-FS)* (To be published)

ISO/IEC 14165-414, *Information technology – Fibre channel – Generic services-4 (FC-GS-4)*

The following references for VI Architecture are the product of Intel, Microsoft and Compaq. The VI Architecture 1.0 specification is completely defined in these three documents. For the convenience of the reader they are added as supplementary documents.

Virtual Interface Architecture Specification, V1.0 (VI-ARCH)

Virtual Interface (VI) Architecture Developer's Guide, V1.0 (VI-DG)

Virtual Interface (VI) Architecture Developer's Guide Error Table Supplement, V1.0

IP Version 6 Addressing Architecture, RFC 2373, July 1998 (RFC2373)
(can be downloaded from the Internet)

3 Terms, definitions and abbreviations

3.1 FC-VI terms and definitions

3.1.1 completing a descriptor

a VI Provider completes a Descriptor by updating the status field and setting the Done bit

3.1.2 FC-VI connection
a VI Connection that is established and maintained between two FC-VI Ports

3.1.3 FC-VI connection point
the context used to listen for FC-VI Connection requests and responses within an FC-VI Port. It is bound to an IP address and a Discriminator

3.1.4 FC-VI connection setup
an FC-VI operation that consists of a Sequence of FC-VI Connection IUs that establish an FC-VI Connection

3.1.5 FC-VI disconnect
an FC-VI operation that consists of a Sequence of FC-VI Connection IUs that removes an FC-VI Connection or aborts a FC-VI Connection Setup

3.1.6 FC-VI endpoint
the context for a VI within an FC-VI Port. Each end of an FC-VI Connection is an FC-VI Endpoint

3.1.7 FC-VI message transfer
an FC-VI operation that consists of one or more FC-VI Message IUs to transfer a VI Message between FC-VI Ports

3.1.8 FC-VI port: a Fibre Channel Port that is capable of FC-VI operation and complies with this standard.

3.1.9 FC-VI provider
the hardware and software services that implement the transport dependent functions of a VI Provider over a Fibre Channel transport conforming to this standard

3.1.10 fully qualified message ID (FQMID)
the tuple of {FCVI_HANDLE, FCVI_MSG_ID, Exchange Context (F_CTL:23)} that uniquely identifies and routes each received FC-VI IU to the correct FC-VI Endpoint context within a FC-VI Port

3.1.11 host name
a symbolic name associated with a VI capable Node. The Host Name is represented as an ASCII character string to the VI Application

3.1.12 in-order fabric
a Fibre Channel configuration where the order of frame arrival at a receiving Port is identical to the transmission order at the originating Port. An Arbitrated Loop is one example of an In-Order Fabric

3.1.13 local
entity (Endpoint, Connectionpoint, Provider, etc.) at this end of a FC-VI Connection

3.1.14 out-of-order fabric
a Fibre Channel configuration where the order of frame arrival at a receiving Port may be different than the transmission order at the originating Port

3.1.15 remote
entity (Endpoint, Connectionpoint, Provider, etc.) at the other end of a FC-VI Connection

3.1.16 VI connection
connection between two VI Endpoints

3.1.17 VI endpoint
a pair of work queues and associated context visible to a VI Application. Also known as a VI

3.1.18 VI message
VI Application data that is transferred between FC-VI Ports over a previously established FC-VI Connection.

3.2 VI Definitions

The following VI Terms used in this standard are defined in the VI Architecture Specification and the VI Architecture Developer's Guide, provided as supplementary documents.

Client-Server

Control Segment

Descriptor

Discriminator

Peer

Peer-to-Peer

RDMA

RDMA Read

RDMA Write

Reliability Level

Reliable Delivery

Reliable Reception

Send

Unreliable Delivery

VI

VI Address

VI Application

VI Handle

VI NIC

VI Provider

3.3 Abbreviations

D_ID	Destination_Identifier [ISO/IEC 14165-251]
ELS	Extended Link Service [ISO/IEC 14165-251]
FC	Fibre Channel [ISO/IEC 14165-251]
FC-FS	The architecture specified by the Fibre Channel standard [ISO/IEC 14165-251]
FC-PH	The architecture specified by the Fibre Channel standard [ISO/IEC 14165-251]
FCP	Fibre Channel Protocol for SCSI [ISO/IEC 14776-222]
FC-4	Fibre Channel Layer 4 mapping layer [ISO/IEC 14165-251]
IU	Information Unit [ISO/IEC 14165-251]
S_ID	Source_Identifier [ISO/IEC 14165-251]
ULP	Upper Layer Protocol [ISO/IEC 14165-251]

3.4 Editorial conventions

Definitions, conventions, abbreviations, acronyms and symbols applicable to this standard are provided, unless they are identical to that described in any referenced standard, in which case they are included by reference. Some definitions from the glossary or body of other standards are included here for easy reference.

In this Standard, a number of conditions, mechanisms, sequences, parameters, events, states, or similar terms are printed with the following conventions:

– the first letter of each word in uppercase and the rest lowercase (e.g., Exchange, Class, etc.).

Such terms and words have special meaning and are defined in other standards. All terms and words not conforming to the convention noted above have the normal technical English meanings.

Numbered items in this Standard do not represent any priority. Any priority is explicitly indicated.

In case of any conflict between text, figure, table and state diagram, the state diagram, then table, then figure, and finally, text takes precedence. Exceptions to this convention are indicated in the appropriate subclauses.

The term “shall” is used to indicate a mandatory rule. If such a rule is not followed, the results are unpredictable unless indicated otherwise.

The fields or control bits which are not applicable shall be set to zero.

If a field or a control bit in a frame is specified as not meaningful, the entity which receives the frame shall not check that field or control bit.

Hexadecimal notation

Hexadecimal notation is used to represent fields. For example, a four-byte FCVI_HANDLE field containing a binary value of 00000000 11111111 10011000 11111010 is shown in hexadecimal format as ‘00FF98FAh’.

Binary notation

Binary notation is used to represent fields. For example, a one-byte Match Address Code Point in a FARP-REQ containing a binary value of 0000 0010 is shown in binary format as ‘00000010b’.

Not Equal

The symbol “!=” means “not equal”.

4 Structure and concepts

4.1 Fibre channel structure and concepts

Fibre Channel (FC) is logically a point-to-point serial data channel. FC architecture has been designed so that it may be implemented with high performance hardware that requires little real-time software management. The Fibre Channel Physical (FC-FS) transport described in ISO/IEC 14165-251 performs those functions required to transfer data from one N_Port to another. The FC-PH transport may be treated as a very powerful delivery service with information grouping and multiple defined classes of service.

A switching fabric allows communication among more than two N_Ports.

The Fibre Channel Arbitrated Loop (FC-AL) is an alternate N_Port interconnect architecture that uses FC mechanisms to transfer data between an NL_Port selected by the arbitration process and any of the other NL_Ports on the loop. Once communication is opened between two NL_Ports, standard FC frames are used to provide an FC-PH compliant delivery service. An Arbitrated Loop may be attached to a Fabric by an FL_Port.

An FC-4 mapping layer defines the steps required to perform the functions defined by a ULP. The protocol is defined in terms of the stream of FC IUs generated by a pair of N_Ports that support the FC-4. In this document, N_Ports and NL_Ports capable of supporting FC-VI transactions are collectively referred to as FC-VI Ports.

The number of Exchanges that may simultaneously be open between two FC-VI Ports is defined by the FC-PH implementation. The architectural limit for this value is 65,535. The maximum number of active Sequences that may simultaneously be open between two FC-VI Ports is limited by FC-PH to no more than 255. To allow management Exchanges to be originated, a certain number of extra Exchanges and at least one extra SEQ_ID should always be available.

FC-FS allows management protocols above the FC-PH interface to perform link management functions. The standard FC-PH primitive sequences, link management protocols, and basic and extended link services are used as required by FC-VI Endpoints.

4.2 FC-VI structure and concepts

FC-VI defines the mapping of VI Messages and VI Connections to FC-PH. FC-VI is based on the method of first establishing an FC-VI Connection between two FC-VI Ports, and then sending VI Messages over the FC-VI Connection. FC-VI defines the Information Units (IUs) that are required to establish and remove FC-VI Connections, and the IUs that are required to send VI Messages.

Each VI Message is transferred in one Exchange, and each FC-VI Connection is established in one Exchange (Client-Server) or two Exchanges (Peer-to-Peer). As required by FC-FS, each FC-VI Information Unit (IU) is constructed from one Sequence. One or more IUs are grouped together to form one Exchange. FC-VI defines FC-VI Connection IUs to establish and remove FC-VI Connections, and defines FC-VI Message IUs to send VI Messages.

FC-VI defines a FCVI_HANDLE to specify the destination VI for the VI Message. FC-VI Ports shall send FC-VI Connection IUs between them to establish the FCVI_HANDLEs. An FC-VI Port may choose its own FCVI_HANDLE for each of its FC-VI Endpoints. The FC-VI Port is required to use the FCVI_HANDLE specified by the other FC-VI Port when transmitting a VI Message. The FC-VI Provider may use any appropriate method to map a FCVI_HANDLE to a VI Handle as used by a VI Application, since FC-VI does not

define a specific mapping between FCVI_HANDLEs and VI Handles. An FCVI_HANDLE shall be unique for all open FC-VI Connections for an FC-VI Port.

FC-VI defines a FC-VI Message ID (FCVI_MSG_ID). The FCVI_MSG_ID is set to one by an FC-VI Port when it sends the first IU of its first VI Message on a new FC-VI Connection. The FC-VI Port that receives the VI Message shall echo the same FCVI_MSG_ID in any Response IU. Each FC-VI Port shall increment the FCVI_MSG_ID value by one for every VI Message sent within a VI to facilitate the detection of lost VI Messages at the receiver. A separate FCVI_MSG_ID count is kept for each direction of a VI. See Clause 8 for a complete description of FC-VI error detection and recovery.

VI Reliability Levels are defined with respect to VI Message delivery. FC-VI supports all three levels of reliability (Unreliable Delivery, Reliable Delivery, Reliable Reception) and all connection models (Client-Server, Peer-to-Peer) defined by the VI Architecture. FC-VI supports all VI data transfer models (Send, RDMA Write and RDMA Read) defined by the VI Architecture.

FC-VI supports unacknowledged (class 2) and acknowledged (class 3) classes of service. FC-VI supports Arbitrated Loop, Fabric, Loop attached Fabric, and Point-to-Point topologies. FC-VI supports an In-Order Fabric and an Out-of-Order Fabric.

NOTE FC-VI may support additional classes of service other than class 2 and class 3. However, no consideration was given to other classes of service during the creation of this standard.

The VI Architecture does not define the format of how VI Addresses are mapped to transport specific addresses. The FC-VI mapping of VI to Fibre Channel associates an IP address with a FC-VI Port. The method of assigning IP addresses to FC-VI Ports is outside the scope of this standard. FC-VI allows more than one IP address to be assigned to the same FC-VI Port. FC-VI also allows the same IP address to be assigned to more than one FC-VI Port. The format of an IP address for FC-VI shall use the format specified by RFC 2373.

FC-VI provides methods to map an N_Port Identifier to an IP address assigned to a FC-VI Port. The Fibre Channel Name Server of the Directory Service as defined in FC-GS-2 is the preferred method to map an N_Port Identifier to an IP address for Fabric topologies. FC-VI additionally specifies the Fibre Channel FARP ELS to resolve IP addresses to N_Port Identifiers for configurations that do not support a Name Server.

Figure 1 illustrates the addressing objects associated with an FC-VI Port.

Before VI Messages are sent, an FC-VI Connection shall be established between a pair of FC-VI Endpoints. An FC-VI Endpoint is an FC-4/ULP construct for a VI Endpoint. A VI Endpoint is created when a VI Application executes the VipCreateVI call (see VI-ARCH).

An FC-VI Port may need to discover the N_Port Identifier of the FC-VI Port to which it wishes to send a connection request to. This is accomplished by querying the Name Server or issuing a FARP-REQ. The process of establishing an FC-VI Connection is defined as an FC-VI Connection Setup. FC-VI defines a four phase sequencing of IUs within a single Exchange to establish a connection. The minimum number of phases to support the VI Architecture is three. FC-VI adds an additional phase to terminate the Connection Setup Exchange at the Exchange originator to facilitate error recovery.

An FC-VI Provider listens for incoming connection requests on an FC-VI Connectionpoint on behalf of a VI Application. An FC-VI Provider creates an FC-VI Connectionpoint when a VI Application executes a Vip-

ConnectWait for a Client-Server connection or VipConnectPeerRequest for a Peer-to-Peer connection (see VI-DG).

The FC-VI Provider listens for incoming connection requests on a specific IP address and a Discriminator bound to the FC-VI Connectionpoint. A Discriminator is typically unique per FC-VI Connectionpoint, but it is not required to be so. When the connection successfully completes, the connection originator FC-VI Endpoint is bound to the connection responder FC-VI Endpoint.

An incoming Connect Request, FCVI_CONNECT_RQST, contains both the Local and Remote IP addresses and Discriminators. A connection match exists if the IP address and Discriminator in the FCVI_CONNECT_RQST match the IP address and Discriminator in the FC-VI Connectionpoint of the Connect Request responder. See Section 6.7 in the VI Architecture Developer's Guide for the exact matching rules for Connection Setup.

Each FC-VI Endpoint communicates a FCVI_HANDLE to the other FC-VI Endpoint during FC-VI Connection Setup. Once the FC-VI Connection is established, the FCVI_HANDLE is used to route all VI Messages to the proper FC-VI Endpoint within an FC-VI Port. IP addresses and Discriminators are not used for VI Message transfer. During an FC-VI Connection Setup, the FC-VI Provider shall guarantee that the Local FC-VI Endpoint is bound to the connection and enabled for reception of Messages before any Connect Response is issued.

The VI Architecture defines two methods to establish a connection. A Client-Server Connection Setup is similar to the Sockets model - a Server waits for connection requests from a Client. A Peer-to-Peer Connection Setup involves two Peers attempting to establish a connection with each other at about the same time.

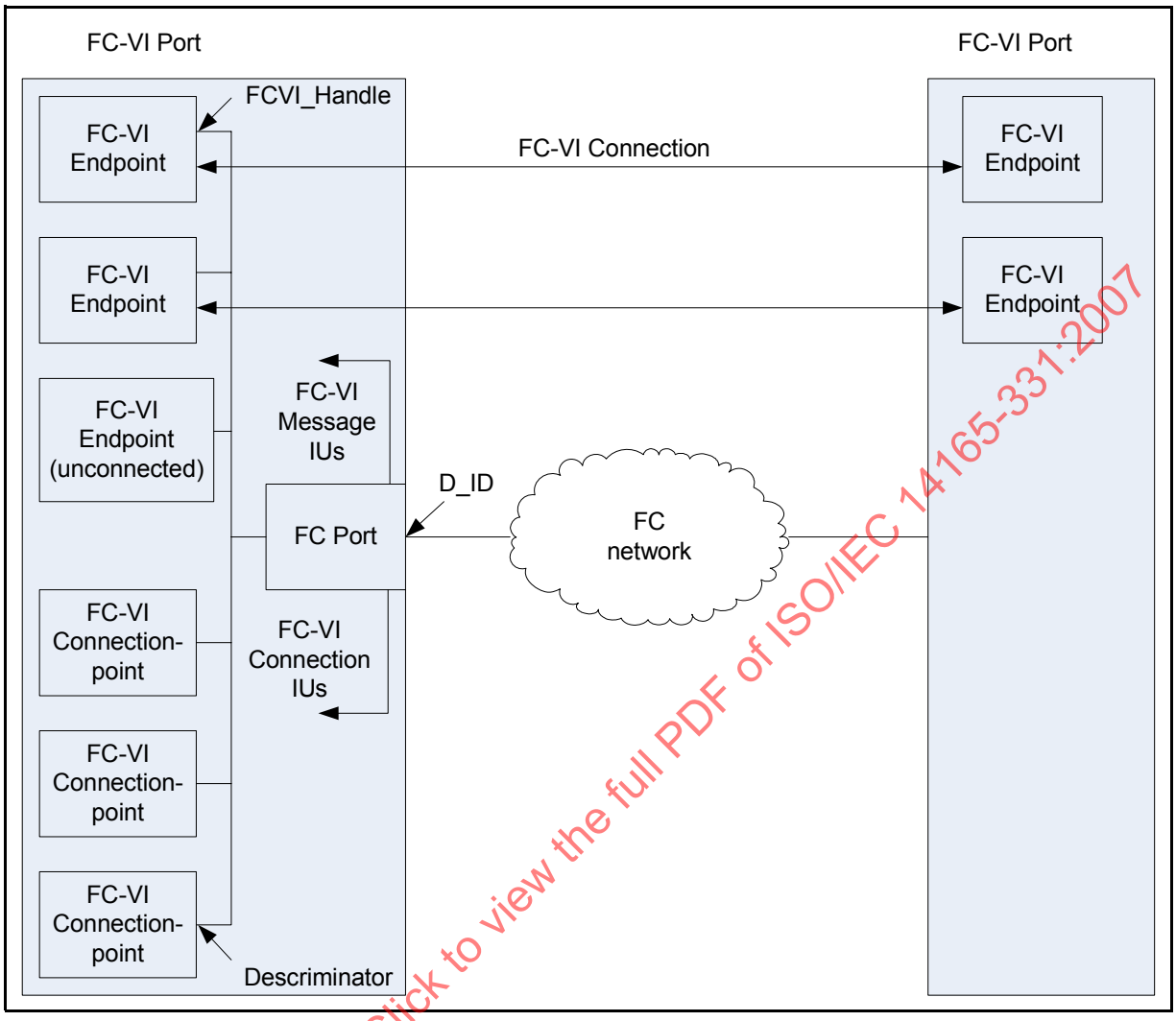


Figure 1 – FC-VI addressing objects

5 FC-VI protocol overview

5.1 FC-VI information units

The Information Units (IUs) used by FC-VI Ports and their characteristics are shown in Table 1. All FC-VI IUs shall be composed of Device_Data frames. All Connect Request/Response IUs are either Unsolicited Control or Solicited Control. This facilitates the FC-2 to route Message Request/Response IUs directly to specialized FC-VI data transfer hardware assists and route the Connect Request/Response IUs to a CPU software path.

The Table 1 column 'Mand/Opt' defines which IUs are Mandatory and which are Optional. The F/M/L column indicates whether the IU is the First, Middle or Last of the Exchange. FCVI_READ_RQST is optional since RDMA Read is optional. All Message Responses are optional, since they are only required for RDMA Read or Reliable Reception. All connection IUs are mandatory.

Table 1 – FC-VI Information unit summary

IU Category	FC-VI Information Unit Type	R_CTL	Device_Header Size	FC-VI Opcode	Payload Content	F/M/L	Mand / Opt
Message Request	FCVI_SEND_RQST	01h	32 Bytes	00h	Message Data	F/M/L	M
Message Request	FCVI_WRITE_RQST	01h	32 Bytes	01h	Message Data	F/M/L	M
Message Request	FCVI_READ_RQST	06h	32 Bytes	02h	none	F	O
Message Response	FCVI_SEND_RESP	07h	16 Bytes	08h	none	L	O
Message Response	FCVI_WRITE_RESP	07h	16 Bytes	09h	none	L	O
Message Response	FCVI_READ_RESP	01h	32 Bytes	0Ah	Message Data	M/L	O
Connect Request	FCVI_CONNECT_RQST	02h	32 Bytes	10h	Connect Info	F	M
Connect Request	FCVI_DISCONNECT_RQST	02h	32 Bytes	12h	none	F	M
Connect Response	FCVI_CONNECT_RESP1	03h	32 Bytes	18h	Connect Info	M	M
Connect Response	FCVI_CONNECT_RESP2	03h	32 Bytes	19h	none	M	M
Connect Response	FCVI_CONNECT_RESP3	03h	32 Bytes	1Ah	none	L	M
Connect Response	FCVI_DISCONNECT_RESP	03h	32 Bytes	1Bh	none	L	M

5.2 FC-VI message transfer operation

5.2.1 FC-VI message transfer

An FC-VI Message Transfer consists of a Message Request and Message Response, or a Message Request with no Message Response. A Message Request shall consist of one or more Message Request IUs of the same IU type. A Message Response shall consist of one or more Message Response IUs of the same IU type. If a response is required, the last request IU shall transfer Sequence Initiative. If no response is expected, the last request IU shall terminate the Exchange. If a response is required, the last response IU shall terminate the Exchange. Exchanges are dynamically created by the Fibre Channel transport mechanism to move the VI Message from one FC-VI Port to another. Utilizing the same Ex-

change for Message Request IUs and Message Response IUs (if needed) binds all Sequences and frames associated with one VI Message.

NOTE The FC-VI mapping of FC-VI Connections to FC-FS Exchanges is dynamic to allow the number of FC-VI Connections to be independent from the number of Exchanges. The maximum number of FC-VI Connections is 2^{32}-1 per port, limited by the FCVI_HANDLE field in the FC-VI Device_Header.

A VI Message may be transferred with a FC-VI Send Message Transfer, a FC-VI Write Message Transfer or a FC-VI Read Message Transfer operation.

The use of Continuously Increasing SEQ_CNT, Continuously Increasing Relative Offset, and total Message length facilitate the detection of lost frames at the Message responder within an Exchange. If the value of SEQ_CNT repeats within an Exchange, Continuously Increasing Relative Offset may be used to guarantee frame uniqueness within an Exchange and facilitate the detection of frame synonyms (identical SEQ_ID and SEQ_CNT values for frames within an Exchange). The value of Relative Offset is guaranteed to never repeat within an Exchange. See 5.5.

The following subclauses detail the different cases for VI Messages mapping into one FC Exchange. These subclauses only define error free operation for FC-VI operation. See Clause 8 for the definition of FC-VI error detection and recovery.

5.2.2 FC-VI send message transfer operation

A VI Message may be transferred using one or more FCVI_SEND_RQST IUs. The VI Message may be sent in Unreliable Delivery or Reliable Delivery mode, as shown in Figure 2, or Reliable Reception mode, as shown in Figure 3. The FC-VI Provider may send one or multiple FCVI_SEND_RQST IUs to transmit the VI Message. No Response shall be issued for Unreliable Delivery or Reliable Delivery modes. A re-

sponse consisting of one FCVI_SEND_RESP IU is required for Reliable Reception mode. Each figure below represents one FC Exchange, and each arrow represents one FC Sequence or IU.

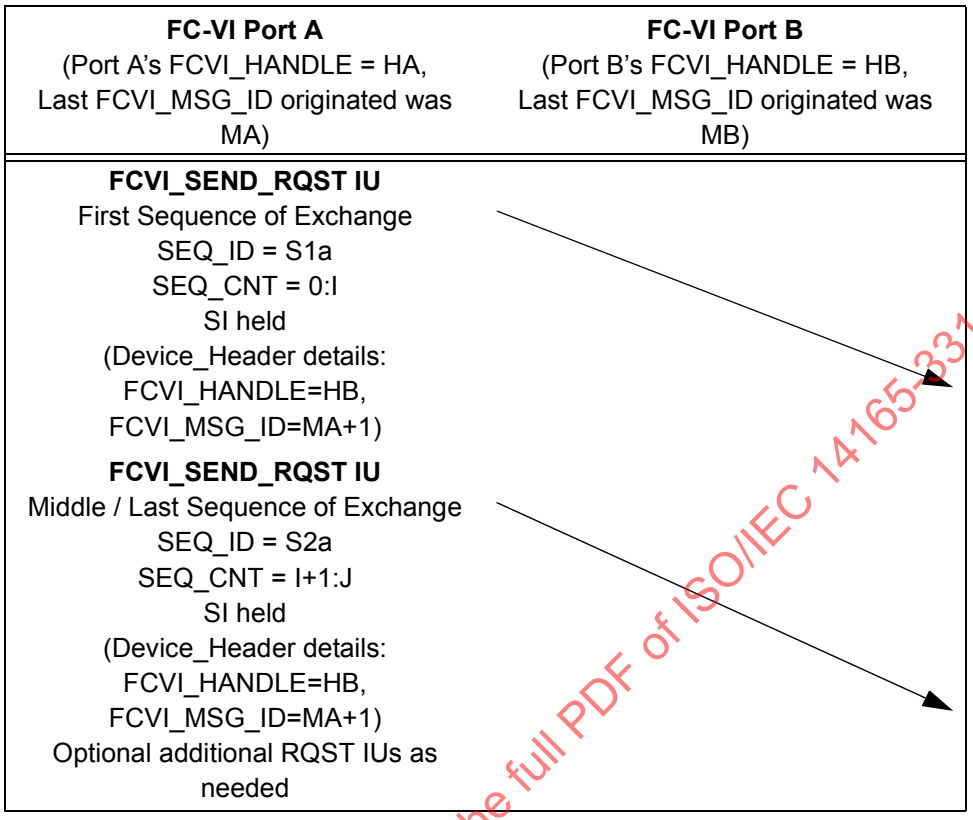


Figure 2 – FC-VI send for Unreliable Delivery or Reliable Delivery

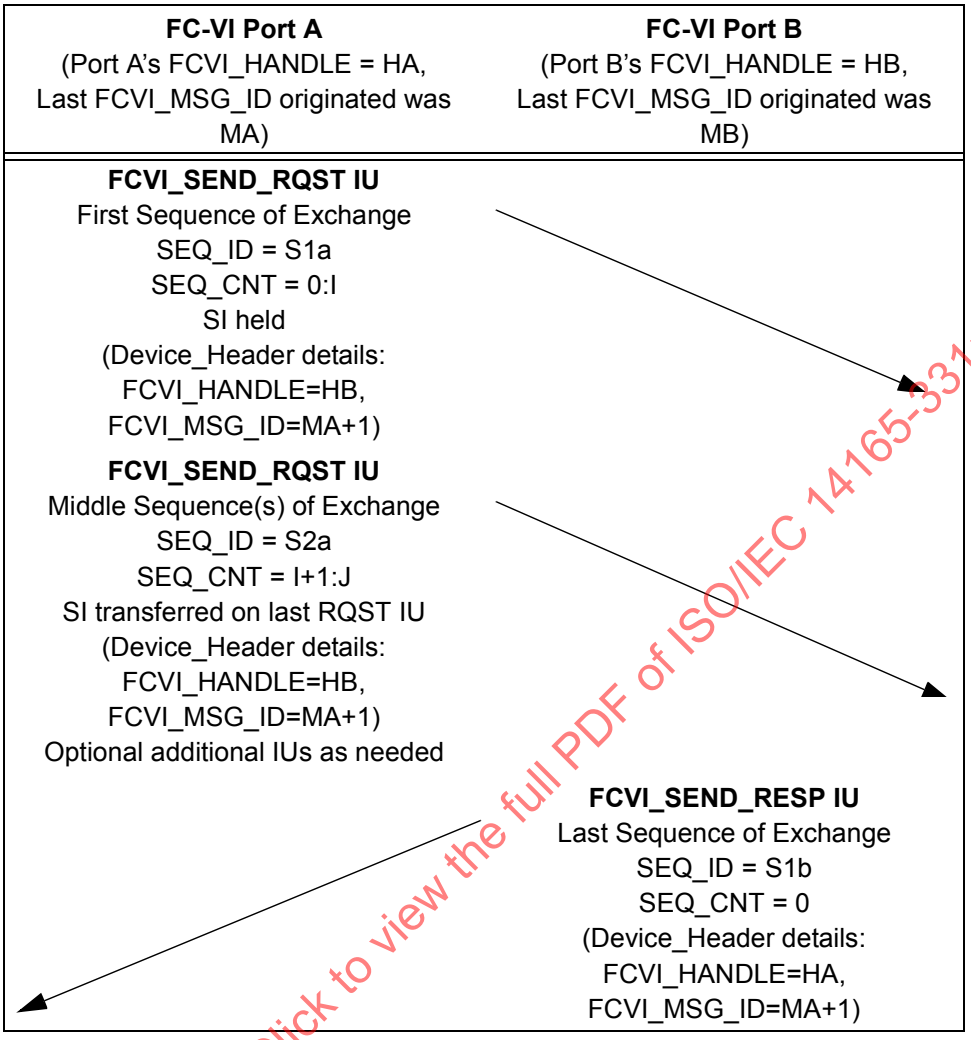


Figure 3 – FC-VI send for Reliable Reception

5.2.3 FC-VI RDMA write message transfer operation

A VI Message may be transferred using one or more FCVI_WRITE_RQST IUs. The VI Message may be sent in Unreliable Delivery or Reliable Delivery mode, as shown in Figure 4, or sent in Reliable Reception mode, as shown in Figure 5. The FC-VI Provider may send one or multiple FCVI_WRITE_RQST IUs to deliver the VI Message. No response shall be issued for Unreliable Delivery or Reliable Delivery modes. A

response consisting of one FCVI_WRITE_RESP IU is required for Reliable Reception mode. Each figure below represents one FC Exchange, and each arrow represents one FC Sequence.

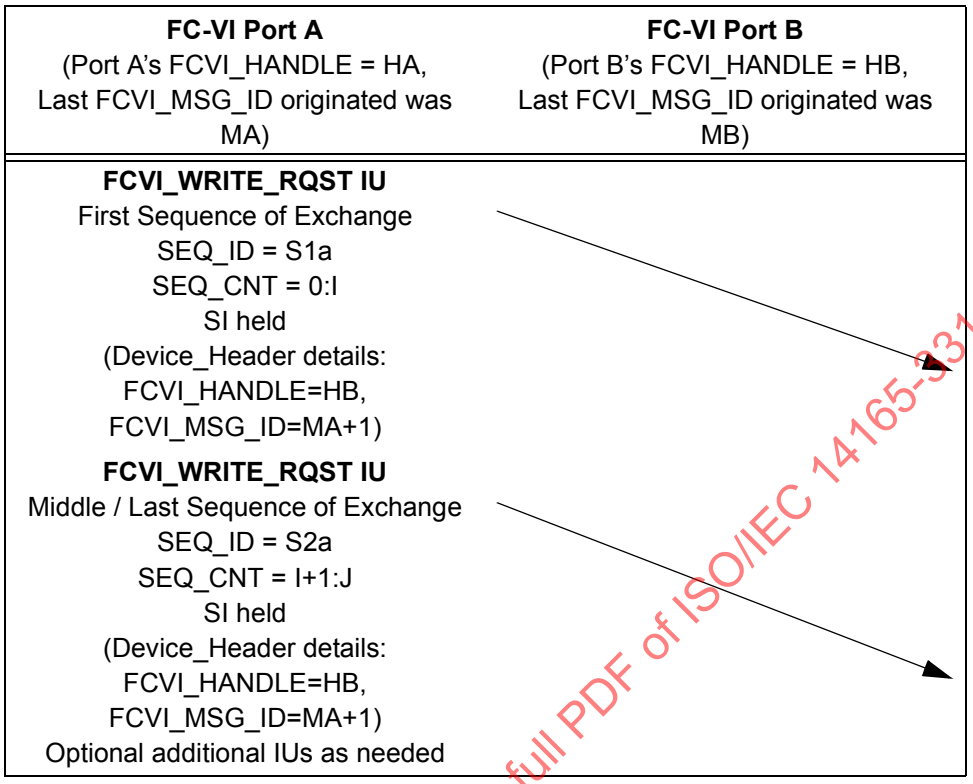


Figure 4 – FC-VI RDMA write for Unreliable Delivery or Reliable Delivery

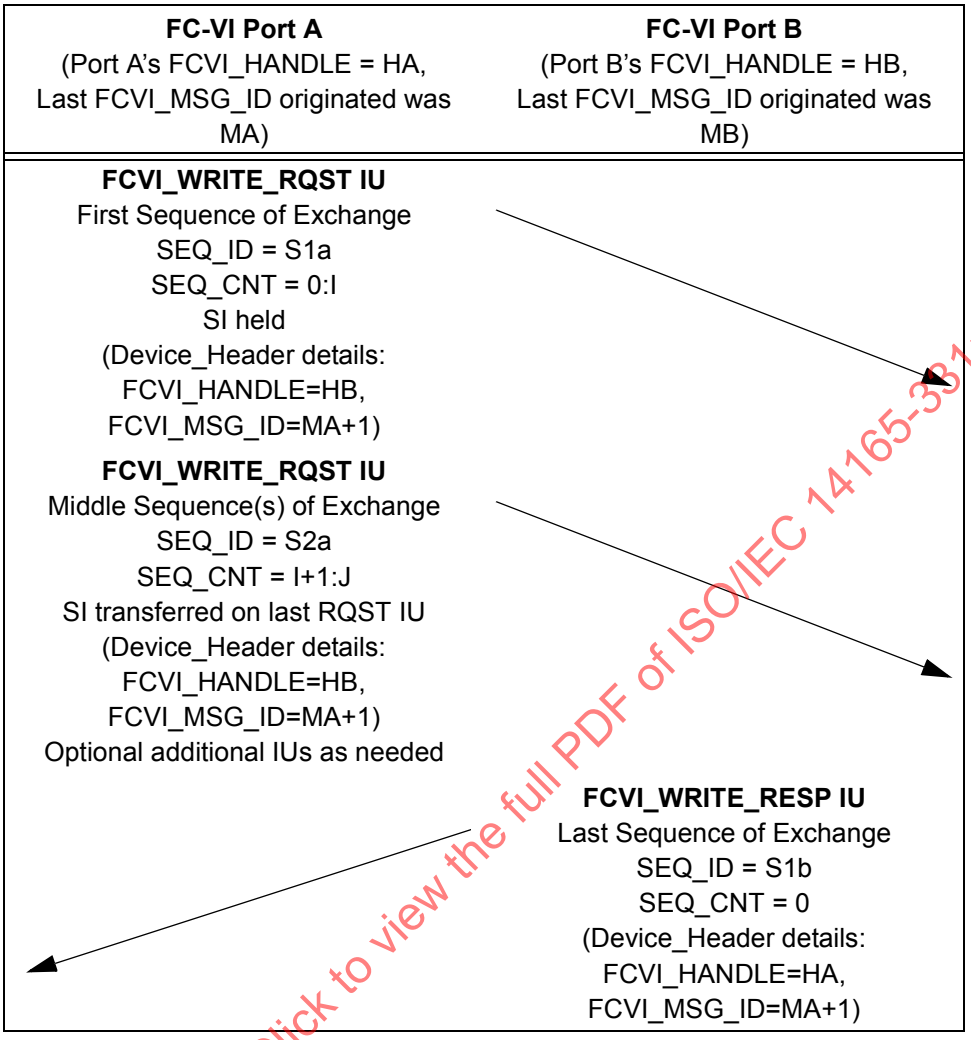


Figure 5 – FC-VI RDMA write for Reliable Reception

5.2.4 FC-VI RDMA read message transfer operation

A VI Message may be transferred with one FCVI_READ_RQST IU and one or more FCVI_READ_RESP IUs. The VI Message my be transferred in Reliable Delivery or Reliable Reception mode, as shown in Figure 6. The FC-VI Provider may send one or multiple FCVI_READ_RESP IUs. A response is always required. Each figure below represents one FC Exchange, and each arrow represents one FC Sequence.

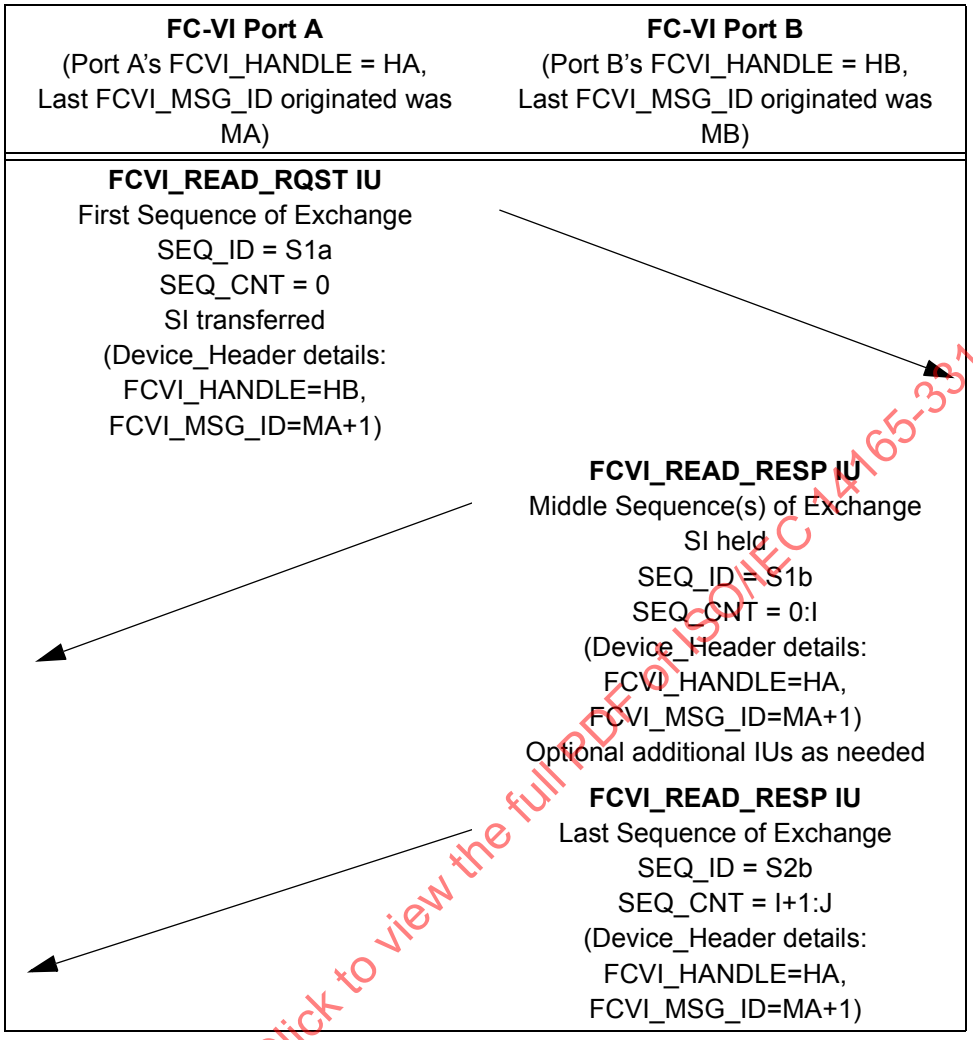


Figure 6 – FC-VI RDMA read for Reliable Reception and Reliable Delivery

5.2.5 IU reception at an FC-VI edpoint

A FC-VI Provider that supports RDMA Reads or Reliable Reception shall be capable of managing two potentially concurrent receive streams for each Local FC-VI Endpoint - a requestor stream and a responder stream. A requestor stream consists of solicited RDMA Read responses and solicited Reliable Reception responses for Messages originated by the Local FC-VI Endpoint. A responder stream consists of unsolicited Sends, RDMA Read requests, and RDMA Writes originated by the Remote FC-VI Endpoint.

An example of concurrent requestor and responder receive streams is shown in Figure 7. The Local Endpoint at FC-VI Port X has originated a RDMA Read Message (Message MA) and is receiving the RDMA Read response data from FC-VI Port Y. At the same time, the Remote Endpoint in FC-VI Port Y has originated a RDMA Write Message (Message MB) to the Local Endpoint in FC-VI Port X. Since the FC-VI Providers in FC-VI Ports X and Y assign FCVI_HANDLEs independently, it is possible that MA may be equal to MB.

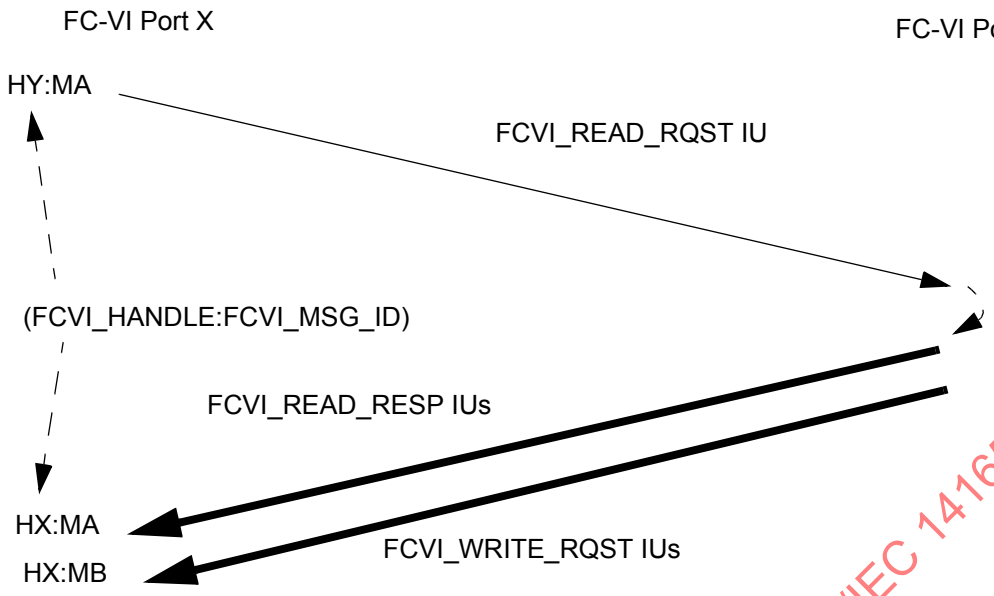


Figure 7 – Concurrent Receive Streams at a FC-VI Endpoint

Thus, the Local Endpoint in FC-VI Port X needs another mechanism other than the tuple {FCVI_HANDLE:FCVI_MSG_ID} to demultiplex the two receive streams to the correct FC-VI Endpoint context. The FC-VI Provider may use the Fully Qualified Message ID (FQMID), which is defined by the tuple {FCVI_HANDLE, FCVI_MSG_ID, Exchange Context (F_CTL:23)}, to route received IUs to the correct FC-VI Endpoint context. The Exchange Context allows the FC-VI Provider to differentiate between solicited responses and unsolicited requests targeted for the same Endpoint. An alternative is to use the Exchange ID and not the FCVI_HANDLE to demultiplex multiple requestor streams (RDMA Read responses, Read responses and Write responses) to the correct FC-VI Endpoint at the Exchange originator.

5.3 FC-VI connection setup operation

5.3.1 FC-VI client-server and peer-peer connection setup

For a Peer-to-Peer Connection Setup, each Peer is a connection originator and a connection responder. For Client-Server, only the Client is the connection originator, while the Server is the connection responder.

Each Connect Request and Connect Response is a separate FC-VI Connection IU. Sequence Initiative is always passed for each connection IU. The FCVI_CONNECT_RQST is the first Sequence of the Exchange, the FCVI_CONNECT_RESP1 and FCVI_CONNECT_RESP2 IUs are the middle Sequences of the Exchange, and FCVI_CONNECT_RESP3 IU is the last Sequence of the Exchange.

This subclause and the following subclauses detail all of the different cases for FC-VI Connections being established or removed. These subclauses only define error free operation for FC-VI operation. See Clause 8 for the definition of FC-VI error detection and recovery.

5.3.2 FC-VI client-server connection setup

Figure 8 illustrates FC-VI Connection Setup for Client-Server. In Figure 8 the Client is Port A while the Server is Port B. The FCVI_FLAGS = 0 in the FCVI_CONNECT_RESP1 IU indicates that the Server has accepted the Client's connection request. The FCVI_FLAGS = 0 in the FCVI_CONNECT_RESP2 indicates that the Client acknowledges the Server's connect accept. The FCVI_FLAGS = 0 in the FCVI_CONNECT_RESP3 indicates that the Server acknowledges the Client's acknowledgment.

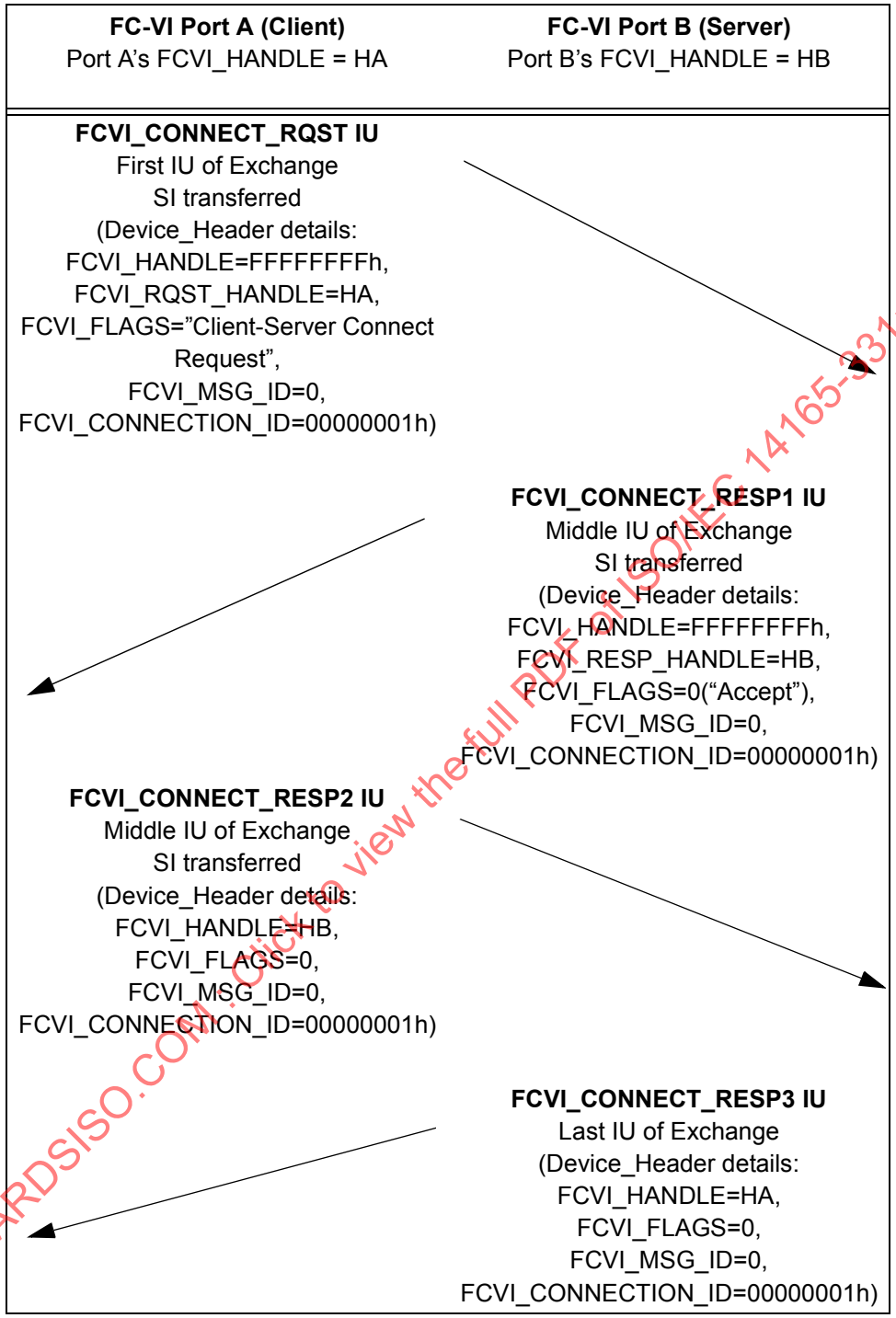


Figure 8 – FC-VI client-server connection setup

The Connect Request originator shall set the FCVI_HANDLE field in the Device_Header of the FCVI_CONNECT_RQST IU equal to the FFFFFFFFh, which indicates the responder handle is not yet assigned. The Connect Request originator shall set the FCVI_RQST_HANDLE field in the Device_Header of the FCVI_CONNECT_RQST IU equal to the FCVI_HANDLE of the Local FC-VI Endpoint. The Connect Request responder shall set the FCVI_HANDLE in the FCVI_CONNECT_RESP3 IU equal to this value received in the FCVI_RQST_HANDLE field in the Device_Header of the FCVI_CONNECT_RQST IU.

The Connect Request responder shall set the FCVI_HANDLE field in the Device_Header of the FCVI_CONNECT_RESP1 IU equal to the FFFFFFFFh, which indicates the requestor handle is not yet assigned. The Connect Request responder shall set the FCVI_RESP_HANDLE field in the Device_Header of the FCVI_CONNECT_RESP1 IU equal to the FCVI_HANDLE of the Local FC-VI Endpoint. The Connect Request originator shall set the FCVI_HANDLE in the FCVI_CONNECT_RESP2 IU equal to this value received in the FCVI_RESP_HANDLE field in the Device_Header of the FCVI_CONNECT_RESP1 IU.

The FCVI_CONNECTION_ID shall be used by the Connect Request originator to assign a unique identifier to all open Connection Setup operations. The FCVI_MSG_ID shall be set to zero for all Connection Setup IUs.

5.3.3 FC-VI Peer-to-Peer Connection Establishment

Figure 9 illustrates the FC-VI Connection Setup for Peer-to-Peer when one Peer issues a Connect Request before the other Peer has begun the Connection Setup. In Figure 9, Peer 1 is Port A, while Peer 2 is Port B. The VI Application in Peer 2 has not yet issued a Peer Connect Request when the FCVI_CONNECT_RQST IU is received from Peer 1. Peer 2 replies with a CONN_STS set to one and a Reason Code of “No Waiting Remote Connectionpoint”. At some later point Peer 2 issues a matching FCVI_CONNECT_RQST IU and a FC-VI Connection is established between Port A and Port B.

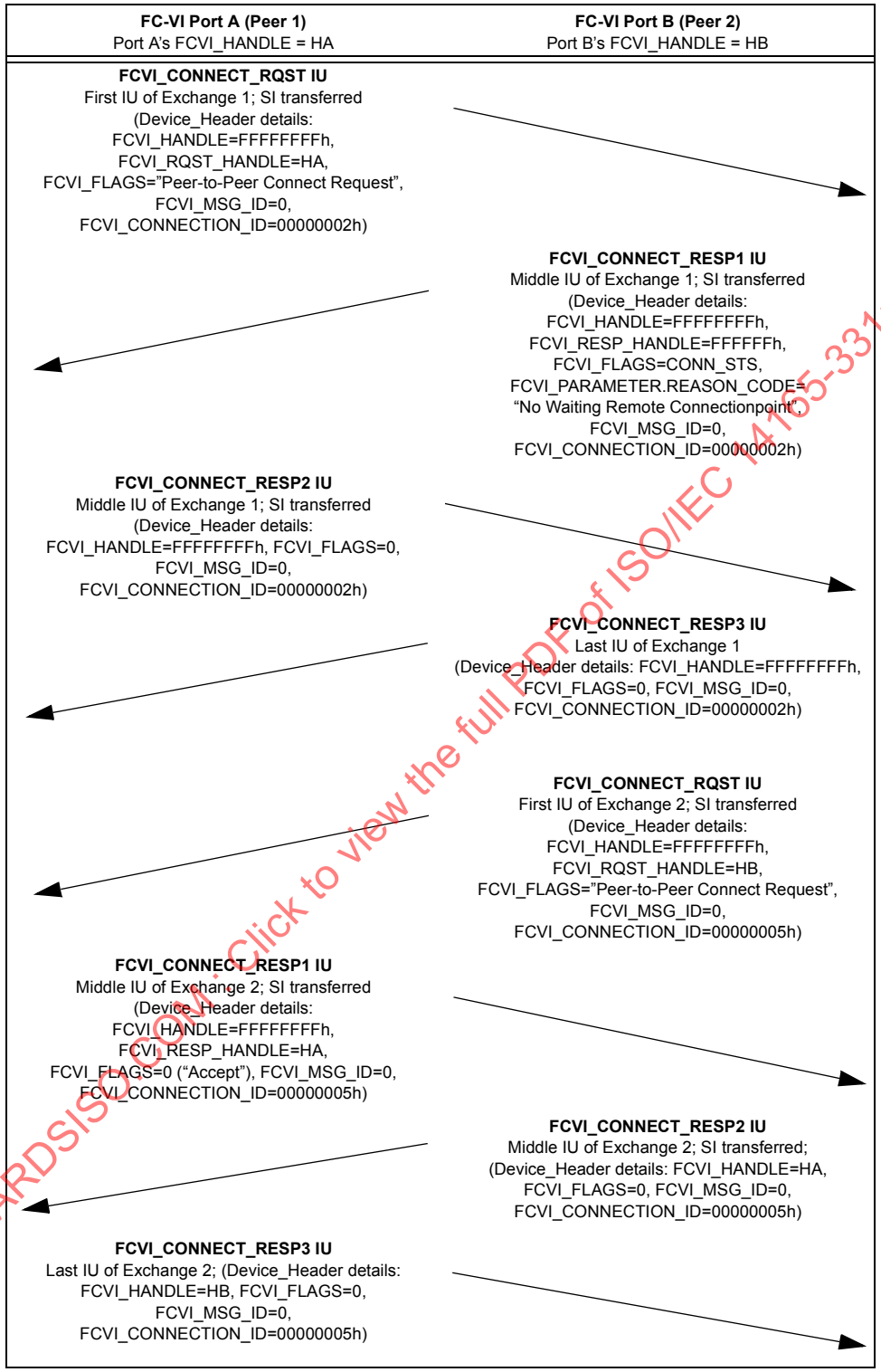


Figure 9 – Peer-to-peer connection setup

For Exchange 1, the Connect Request responder shall set the FCVI_RESP_HANDLE field in the Device_Header of the FCVI_CONNECT_RESP1 IU (FCVI_CONNECTION_ID = 00000002h) equal to FFFFFFFFh, since there is no waiting Peer at the Connect Request responder. The Connect Request originator shall set the FCVI_HANDLE in the Device_Header in the FCVI_CONNECT_RESP2 IU in Exchange 1 equal to FFFFFFFFh, the value of FCVI_RESP_HANDLE. The Connect Request responder shall set the FCVI_HANDLE in the Device_Header in the FCVI_CONNECT_RESP3 IU in Exchange 1 equal to FFFFFFFFh. The unassigned FCVI_HANDLES in FCVI_CONNECT_RESP2 and FCVI_CONNECT_RESP3 indicate the Connection Setup did not establish a connection.

For Exchange 2, the Connect Request responder shall set the FCVI_RESP_HANDLE field in the Device_Header of the FCVI_CONNECT_RESP1 IU (FCVI_CONNECTION_ID = 00000005h) equal to FFFFFFFFh. The Connect Request originator shall set the FCVI_HANDLE in the Device_Header in the FCVI_CONNECT_RESP2 IU in Exchange 2 equal to “HA”h, the value of FCVI_RESP_HANDLE, since there now is a waiting Peer at the Connect Request responder. The Connect Request responder shall set the FCVI_HANDLE in the Device_Header in the FCVI_CONNECT_RESP3 IU in Exchange 2 equal to “HB”h, the value of FCVI_RQST_HANDLE.

5.3.4 FC_VI concurrent peer-to-peer connection setup

Figure 10 illustrates FC-VI Connection Setup for Peer-to-Peer when each Peer issues a matching FCVI_CONNECT_RQST IU before a FCVI_CONNECT_RESP1 IU is received. This situation is defined as Concurrent Matching Peer-to-Peer Requests.

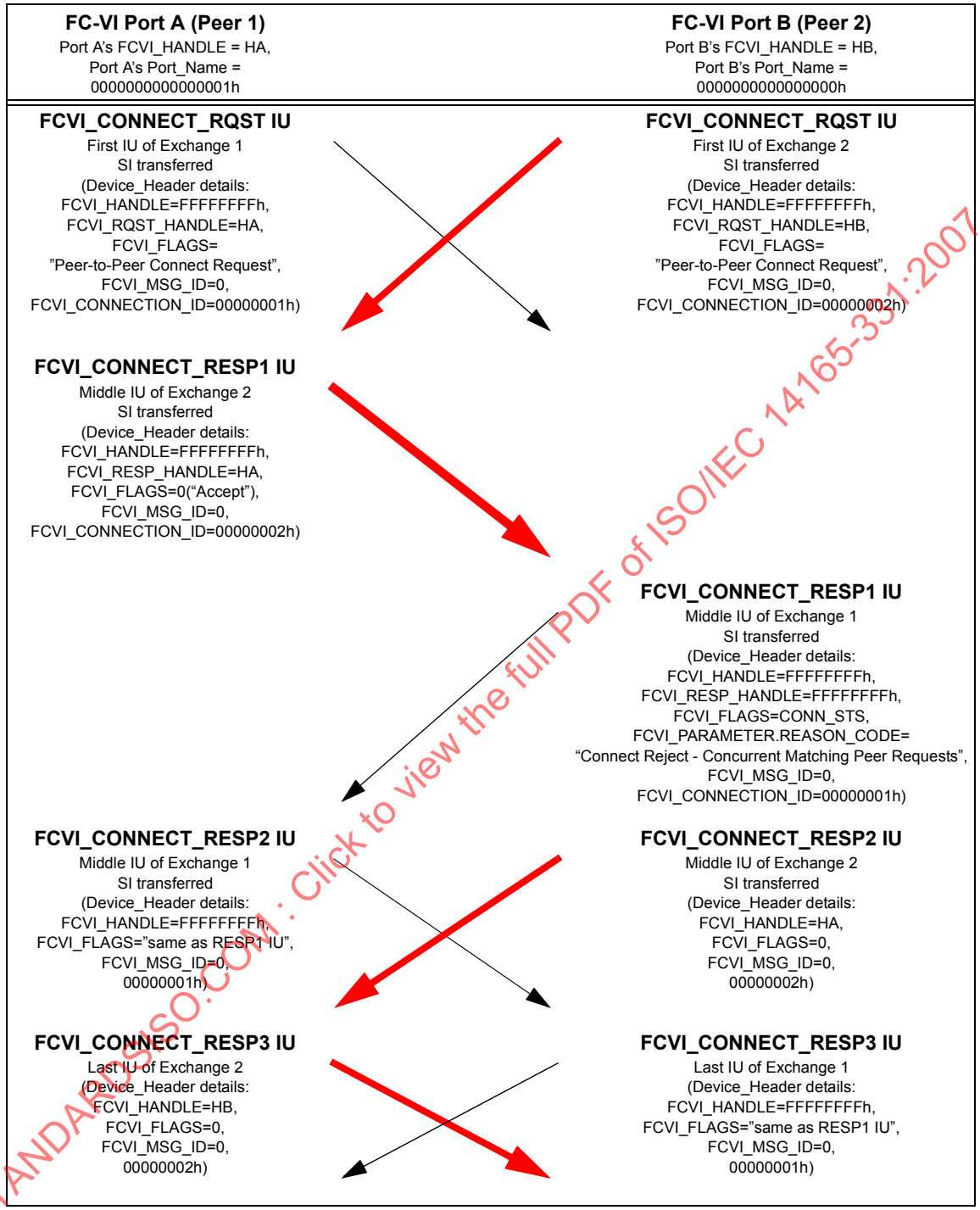


Figure 10 – Peer-to-peer connection setup, concurrent matching peer requests

For Concurrent Matching Peer-to-Peer Requests, an arbitration protocol shall be used to guarantee that only one Peer accepts a matching Connect Request and successfully completes the Connection Setup. A comparison between the Port_Names of the connection originators is used for the arbitration protocol.

The Peer with the numerically lower Port_Name shall win the arbitration and rejects the Peer's Connect Request. The Peer with the numerically higher Port_Name shall lose the arbitration and accept the Peer's Connect Request. Thus, the Peer with the numerically lower Port_Name successfully completes the Connection Setup. The following rules define the arbitration protocol for Concurrent Matching Peer-to-Peer Requests:

a) If my Port_Name is greater than my Peer's Port_Name, send a FCVI_CONNECT_RESP1 of "Accept" (CONN_STS = 0 in FCVI_FLAGS).

b) Else, if my Port_Name is less than the Peer's Port_Name, wait for the Peer's FCVI_CONNECT_RESP1.

   1) If the Peer's FCVI_CONNECT_RESP1 is "No Waiting Remote Connectionpoint" or "No Discriminator Match", send a FCVI_CONNECT_RESP1 of "Connect Accept".

   2) Else, if the Peer's FCVI_CONNECT_RESP1 is "Connect Accept", send a FCVI_CONNECT_RESP1 with CONN_STS equal to one with a Reason Code of "Connect Reject - Concurrent Matching Peer Requests".

   3) Else, if the Peer's FCVI_CONNECT_RESP1 is "Connect Reject - Concurrent Matching Peer Requests", send a FCVI_CONNECT_RESP1 with CONN_STS equal to one with a Reason Code of "Connect Reject - Protocol Error".

   4) Else, send a FCVI_CONNECT_RESP1 with CONN_STS equal to one with a Reason Code of "Connect Reject".

c) Else, if my Port_Name equal to Peer's Port_Name, send a FCVI_CONNECT_RESP1 with CONN_STS equal to one with a Reason Code of "Connect Reject - Protocol Error".

To support loopback of Peer-to-Peer Connection Setup, a FC-VI Provider shall recognize that the Connect Request originator and the Connect Request responder are the same Port and simulate the Port_Name comparison with an implementation dependent mechanism (i.e., the ability to arbitrate between concurrent matching requests).

Table 2 defines the correct FCVI_CONNECT_RESP1 from the Peer (Peer B) with the lower Port_Name for Concurrent Matching Peer-to-Peer Requests for each possible FCVI_CONNECT_RESP1 from the Remote Peer with the higher Port_Name (Peer A). See Annex A for examples of Concurrent Matching Peer-to-Peer Requests.

Figure 2 is an example of Concurrent Matching Peer-to-Peer Requests. Peer 1 is Port A, while Peer 2 is Port B. The thicker arrows correspond to IUs for Exchange 2, while the thinner arrows correspond to Exchange 1. In the case illustrated, Peer 1's Port_Name is numerically greater than Peer 2's Port_Name. Peer 1 accepts the connection request while Peer 2 rejects it. Thus, Peer 2 shall successfully complete the connection.

5.3.5 FC-VI Disconnect Operation

Figure 11 shows the FC-VI protocol to perform a FC-VI Disconnect. A VI Application decides when it is appropriate to disconnect, according to the VI Architecture specifications. A FC-VI Disconnect operation may be initiated from either FC-VI Port by sending a FCVI_DISCONNECT_RQST IU. The other FC-VI Port shall respond with a FCVI_DISCONNECT_RESP IU. The FC-VI Provider that controls the FC-VI Port may also issue a Disconnect in certain cases.

Table 2 – Peer B actions based on connect responses from peer A

Response from Peer A (higher Port_Name)	Peer B (lower Port_Name) Replies with:
Connect Accept	Connect Reject - Concurrent Matching Peer Requests
No Discriminator Match	Connect Accept
No Waiting Remote Connectionpoint	Connect Accept
Connect Reject - Concurrent Matching Peer Requests	Connect Reject - Protocol Error
Connect Reject - Transport Error	Connect Reject
Connect Reject - Protocol Error	Connect Reject
Any other response	Connect Reject

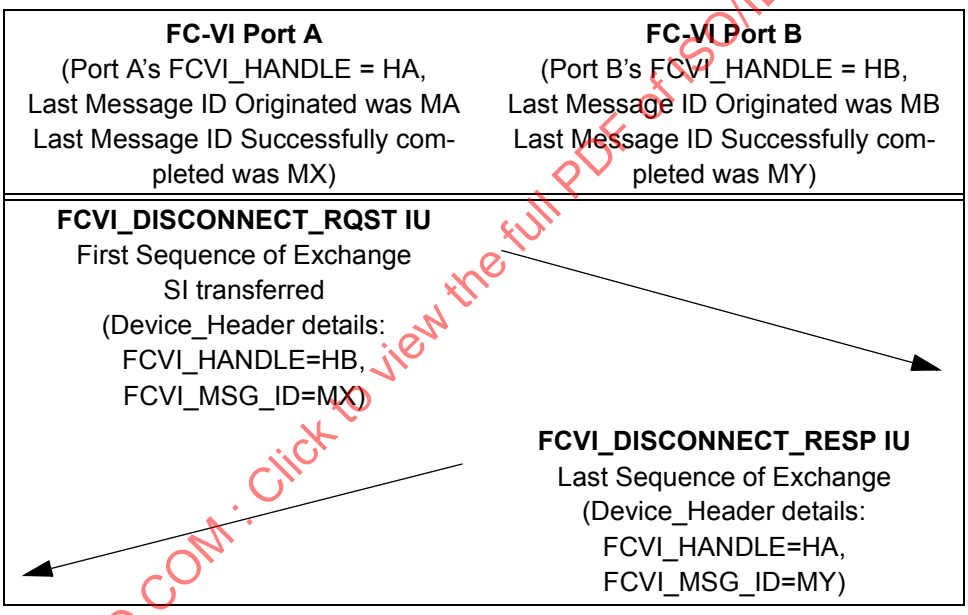


Figure 11 – FC-VI disconnect operation

5.4 Exchange ID reuse

FC-VI requires Exchange termination on the last IU, either request or response, of the Message. Each Message transfer starts a new Exchange. Since FCVI_MSG_ID is guaranteed to be unique for each Message, a Message Originator may immediately reuse an Exchange ID of a previously terminated Exchange.

5.5 Sequence ID reuse

As required by ISO/IEC 14165-251 (FC-FS), all FC-VI Information Units shall be represented by one FC Sequence. The Sequence ID (SEQ_ID) may be reused by the Sequence Initiator as soon as the Sequence is closed.

NOTE FC-FS requires the use of at least two different SEQ_IDs for Sequence streaming.

In Class 3, the Sequence Initiator shall consider a Sequence closed when the last frame of the Sequence is sent. In Class 2, the Sequence Initiator shall consider a Sequence closed when all ACKs for all frames in the Sequence have been received. For either class 2 or class 3 a Sequence ID may be reused when the Sequence ID has been recovered after a Sequence error.

5.6 Frame synonym detection

FC-VI frames shall use a Continuously Increasing Sequence Count for each Exchange to assist detecting potentially missing frames. Sequence Count shall not repeat within one IU. An FC-VI Port shall set Word 1, bit 17 - SEQ_CNT (S) - to one in the N_Port Common Service Parameters when originating a PLOGI. An FC-VI Port shall set Word 1, bit17 in the ACC to a PLOGI.

If the SEQ_CNT (S) bit is not set in the PLOGI or ACC, an FC-VI Port shall reject any Connect Request from the Port that sent the PLOGI or ACC with the (S) bit not set by transmitting a FCVI_CONNECT_RESP1 with CONN_STS set to one and a Reason Code of “Invalid Service Parm”. In addition, an FC_VI Port shall not originate a FC-VI Connection Setup with a Port that sent a PLOGI or ACC with the (S) bit not set.

To facilitate frame synonym detection at the Sequence Recipient, FC-VI requires that the Sequence Initiator shall use Continuously Increasing Relative Offset. All FC-VI compliant N_Ports shall indicate support for Continuously Increasing Relative Offset by setting the Continuously Increasing (C) bit to one in the Common Service Parameters of both the PLOGI and ACC (Word 1, bit 30). The Relative Offset of the first byte of the VI Message payload in the first frame of the next Sequence to transmit shall be +1 greater than the maximum relative offset of any byte of the VI Message payload in the previous Sequence.

If the Continuously Increasing (C) bit is not set the PLOGI or ACC, an FC-VI Port shall reject any Connect Request from the Port that sent the PLOGI or ACC with the (C) bit not set by transmitting a FCVI_CONNECT_RESP1 with CONN_STS set to one and a Reason Code of “Invalid Service Parm”. In addition, an FC_VI Port shall not originate a FC-VI Connection Setup with a Port that sent a PLOGI or ACC with the (C) bit not set.

Figure 12 shows an example of FC-2 header operation during an FC-VI Exchange. Figure 12 assumes a uniform frame size.

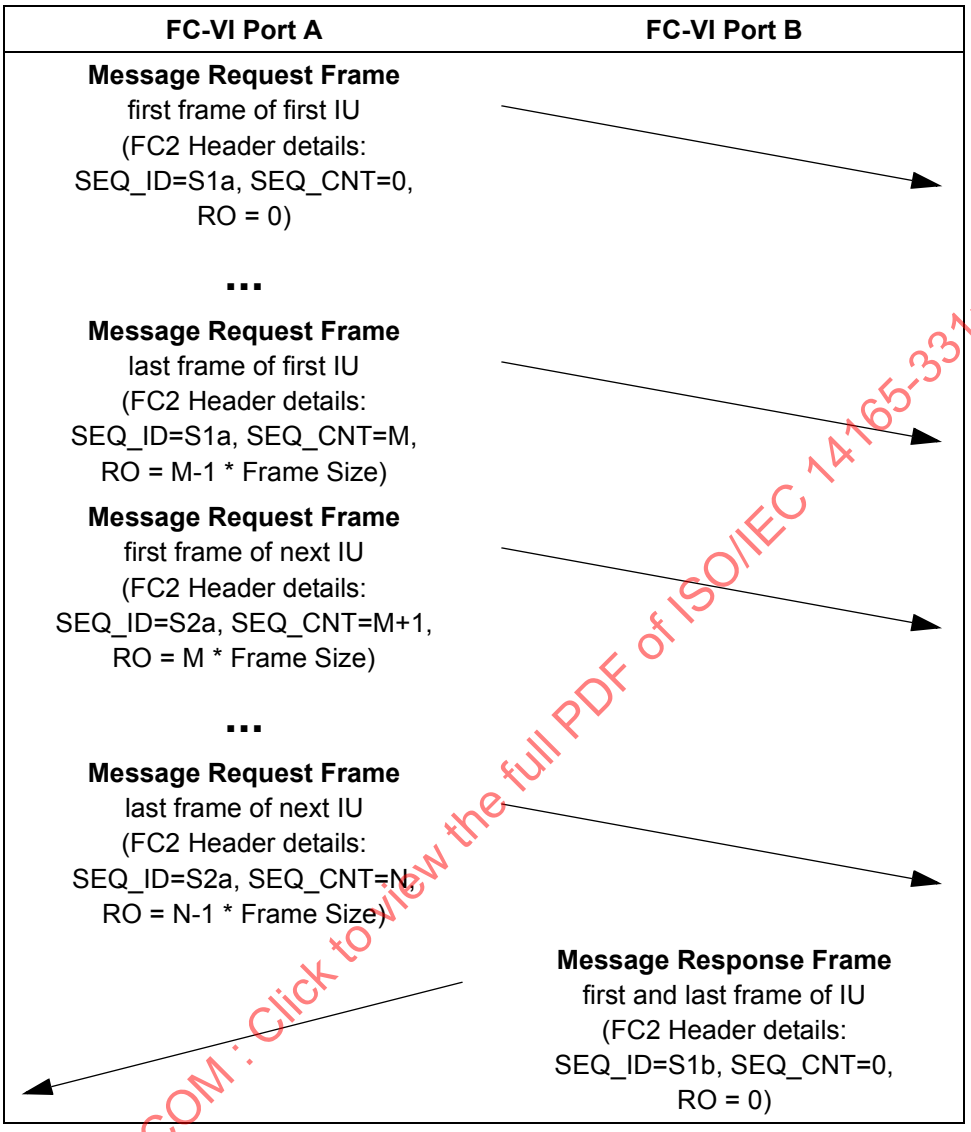


Figure 12 – FC-FS header for send operation

5.7 VI message length

VI Messages may be a maximum length of 2^{32} -1 bytes, which is the maximum VI Message size specified by the VI Architecture.

5.8 FC-FS header usage for FC-VI

5.8.1 FC-FS header usage

For FC-VI, fields in the frame header use the standard FC-FS definitions. The following explanations of the FC header fields provide information about the use of those fields to implement FC-VI functionality. If a field is not defined below, then its usage is the same as defined by FC-FS.

5.8.2 CS_CTL field

FC-VI Providers that choose to make use of the PREF bit defined in the CS_CTL field of the frame header (see FC-FS) shall conform to the following policy: Use of the Preference Function shall be on a per FC-VI Connection basis. If a particular FC-VI Connection is to make use of the Preference Function then every Fibre Channel frame for a VI Message sent on that FC-VI Connection shall have the PREF bit set to 1 in the CS_CTL field. If a particular FC-VI Connection is not making use of the Preference Function then each and every Fibre Channel frame for a VI Message sent on that FC-VI Connection shall have the PREF bit set to 0 in the CS_CTL field. Each FC-VI Endpoint of an FC-VI Connection may independently make use of the Preference Function. It is not required that both FC-VI Endpoints simultaneously set the PREF bit to one. The Preference Function shall only be used for FC-VI Message Transfer operations.

5.8.3 TYPE field

The TYPE field shall be 58h for all frames of any FC-VI IU.

5.8.4 F_CTL field

For Message Requests that do not require a Message Response, the Last Sequence bit shall be set to one in the last Message Request IU. For Message Requests that do require a Message Response, the Sequence Initiative bit shall be set to one in the last Message Request IU. The Last Sequence bit shall be set to one in the last Message Response IU.

For FC-VI Connection Setup IUs, the Sequence Initiative bit shall be set to one in a FCVI_CONNECT_RQST IU, a FCVI_CONNECT_RESP1 IU and a FCVI_CONNECT_RESP2 IU. The Last Sequence bit shall be set to one in a FCVI_CONNECT_RESP3 IU. The Sequence Initiative bit shall be set to one in a FCVI_DISCONNECT_RQST IU. The Last Sequence bit shall be set to one in a FCVI_DISCONNECT_RESP IU.

5.8.5 DF_CTL field

The DF_CTL field indicates any optional headers that may be present. FC-VI IUs require either a 16 or 32 byte Device_Header. The Device_Header is present in all frames of any FC-VI IU. The size of the Device_Header associated with each IU is defined in Table 1, in the DH Size column.

5.8.6 SEQ_CNT field

The SEQ_CNT field indicates the frame order within the Sequence as defined by FC-FS. FC-VI requires that the SEQ_CNT shall be continuously increasing within an Exchange, independent of whether the Sequence is streamed or non-streamed. The Sequence Initiator shall not reuse the same SEQ_CNT within a single Sequence.

See 5.5.

5.8.7 Parameter field

The Parameter field shall indicate the Relative Offset of the first byte of each frame's payload, referenced to a starting value of 0. FC-VI requires that the Relative Offset shall be continuously increasing within an Exchange. For RDMA operations, the Relative Offset value of each frame shall be added to the FCVI_RMT_VA field in the Device_Header to determine the starting address for the VI Application buffer read or write.

See 5.5.

5.9 FC-VI device_header

5.9.1 FC-VI device_header description

Every FC-VI frame shall contain a Device_Header. For FCVI_SEND_RESP and FCVI_WRITE_RESP IUs, a 16 byte Device_Header is used, as shown in Table 3. For all other IUs, a 32 byte Device_Header is used, as shown in Table 4.

Table 3 – 16-byte FC-VI device_header

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to00 Byte 3
0	FCVI_HANDLE			
1	FCVI_OPCODE	FCVI_FLAGS	Reserved	
2	FCVI_MSG_ID			
3	FCVI_PARAMETER			

Table 4 – 32-byte FC-VI device_header

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	FCVI_HANDLE			
1	FCVI_OPCODE	FCVI_FLAGS	Reserved	
2	FCVI_MSG_ID			
3	FCVI_PARAMETER			
4	(MSB) FCVI_RMT_VA			
5				(LSB)
6	FCVI_RMT_VA_HANDLE			
7	FCVI_TOT_LEN / FCVI_CONNECTION_ID			

5.9.2 FCVI_HANDLE field

The FCVI_HANDLE field is used by a FC-VI Port to uniquely identify the FC-VI Endpoint for the incoming FC-VI IU. Each FC-VI Port shall put the correct FCVI_HANDLE value into every IU transmitted on an FC-

VI Connection. The FCVI_HANDLE value for each FC-VI Endpoint of an FC-VI Connection is established during FC-VI Connection Setup. Each FC-VI Port chooses its own FCVI_HANDLE value for the other FC-VI Port to use. Values from 00000000h to FFFFFFFEh are valid FCVI_HANDLEs.

The value of FFFFFFFFh is used to indicate a not yet assigned FCVI_HANDLE. It shall be used for the first two IUs sent during FC-VI Connection Setup, FCVI_CONNECT_RQST and FCVI_CONNECT_RESP1. It shall also be used for a FCVI_DISCONNECT_RQST sent during FC-VI Connection Setup when the FCVI_HANDLE is not yet for the Remote Endpoint of the connection.

5.9.3 FCVI_OPCODE field

The FCVI_OPCODE field completely specifies the FC-VI IU type and format. Table 1 indicates the FCVI_OPCODE assigned to the various FC-VI IUs. There are four different IU categories defined. The FC-VI Connect Request/Response IUs are used for creating and destroying FC-VI Connections between FC-VI Ports. The VI Message Request/Response IUs are used for Message Transfer only after the FC-VI Connection is established.

5.9.4 FCVI_FLAGS field

5.9.4.1 FCVI_FLAGS field description

The FCVI_FLAGS field contains control flags for the FC-VI operation. Reserved bits shall always be zero. Table 5 define the FCVI_FLAGS settings for Message Request IUs; Table 6 defines the FCVI_FLAGS settings for Message Response IUs; Table 7 defines the FCVI_FLAGS settings for Connect Request IUs; Table 9 defines the FCVI_FLAGS settings for Connect Response IUs; Table 10 define the FCVI_FLAGS settings for Disconnect IUs.

5.9.4.2 FCVI_FLAGS for message request IUs

The FCVI_FLAGS bit positions and definitions for all Message Request IUs are shown in Table 5.

Table 5 – FCVI_FLAGS Bit definitions for message request IUs

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	Reserved	0
3	Reserved	0
2	Reserved	0
1	Reserved	0
0	IMM_DATA	FCVI_PARAMETER field is Immediate Data when one

If IMM_DATA (bit zero) in FCVI_FLAGS is equal to one, the FCVI_PARAMETER field in the FC-VI Device_Header (Word 3) shall contain Immediate Data. If IMM_DATA in FCVI_FLAGS is equal to zero, the FCVI_PARAMETER field in the FC-VI Device_Header (Word 3) is reserved. IMM_DATA may only be set for a FCVI_SEND_RQST or a FCVI_WRITE_RQST IU.

5.9.4.3 FCVI_FLAGS for message response IUs

The bit positions and definitions for all Message Response IUs are shown in Table 6.

Table 6 – FCVI_FLAGS Bit definitions for message response IUs

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	Reserved	0
3	TRANS_ERR	Transport Error when one.
2	PROT_ERR	RDMA Protection Error when one.
1	DESC_ERR	Remote Descriptor Error when one.
0	RESP_ERR	Message Transfer successful when zero, errored when one.

For Message Response IUs, the RESP_ERR bit is set when the VI Message responder indicates a failure of the Message Transfer. The reason for the failure is indicated by TRANS_ERR, PROT_ERR and DESC_ERR. The definitions of the TRANS_ERR, PROT_ERR and DESC_ERR bits are defined in 5.9.7.3.1, Descriptor Error Reason Codes.

5.9.4.4 FCVI_FLAGS for connect request IUs

The bit positions and definitions for all Connect Request IUs are shown in Table 7.

Table 7 – FCVI_FLAGS Bit definitions for connect request IUs

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	RETRY	Retried Connection Setup when one
3	FCVI_CONN_INFO	If one, the Connect Request contains 256 bytes of FC-VI Provider Connect Info If zero, the 256 bytes of FC-VI Provider Connect Info is not present
2	CONN_MODE2	Connection Mode bit 2
1	CONN_MODE1	Connection Mode bit 1
0	CONN_MODE0	Connection Mode bit 0

For Connect Request IUs, the Connection Mode bits 2:0 shall define the connection mode requested. The defined values are shown in Table 8.

If FCVI_CONN_INFO is set to one, the payload of the FCVI_CONNECT_RQST IU contains 256 bytes of connection information from the Local FC-VI Provider in the FCVI_CONNECT_INFO field in the

FCVI_CONNECT_RQST payload. If FCVI_CONN_INFO is set to zero, the 256 bytes of connection information are not present in the FCVI_CONNECT_RQST IU.

Connection information may be passed between FC-VI Providers in the FCVI_CONN_INFO field. There is currently no defined VI Architecture interface to allow VI Applications to specify connection information during Connection Setup.

If RETRY is set to one, the FCVI_CONNECT_RQST IU is part of a retried Connection Setup for a particular value of FCVI_CONNECTION_ID. If RETRY is set to zero, the FCVI_CONNECT_RQST IU part of an initial Connection Setup for a particular value of FCVI_CONNECTION_ID. The Connect Request originator shall set the RETRY bit in the FCVI_CONNECT_RQST IU when a Connection Setup is being retried.

Table 8 – FC-VI connection mode definition

CONN_MODE[2:0]	Connection Mode
111	0
110	0
101	0
100	0
011	0
010	Peer-to-Peer Connect Request
001	Client-Server Connect Request
000	0

In Table 8, for Connect Request IUs a Connection Mode of 001b indicates a Client-Server Connect Request, while a Connection Mode of 010b indicates a Peer-to-Peer Connect Request.

5.9.4.5 FCVI_FLAGS for connect response IUs

The bit positions and definitions for all Connect Response IUs are shown in Table 9.

Table 9 – FCVI_FLAGS Bit definitions for connect response IUs

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	Reserved	0
3	Reserved	0
2	RETRY	Retried Connection Setup when one
1	FCVI_CONN_INFO	If one, the Connect Response contains 256 bytes of FC-VI Provider Connect Info If set to zero, the 256 bytes of FC-VI Provider Connect Info is not present
0	CONN_STS	Connection Setup status is present when one.

The CONN_STS bit is set to one when a Connection Setup status is available. It may be set in the FCVI_CONNECT_RESP1 IU, FCVI_CONNECT_RESP2 IU or the FCVI_CONNECT_RESP3 IU. Thus, CONN_STS may be set for all Connection IUs except FCVI_CONNECT_RQST. If CONN_STS is set, the FCVI_PARAMETER field shall indicate the additional status information. See Table 11 and Table 12 for the definition of Connection Setup status.

For a FCVI_CONNECT_RESP1 IU, if FCVI_CONN_INFO is set to one, the payload of the FCVI_CONNECT_RESP1 IU contains 256 bytes of connection information from the Remote FC-VI Provider in the FCVI_CONNECT_INFO field. If FCVI_CONN_INFO is set to zero, the 256 bytes of connection information are not present in the FCVI_CONNECT_RESP IU.

If RETRY is set to one, the Connect Response IU is part of a retried Connection Setup for a particular value of FCVI_CONNECTION_ID. If RETRY is set to zero, the Connect Response IU part of an initial Connection Setup for a particular value of FCVI_CONNECTION_ID. RETRY shall be set to one for all IUs sent on a retried Connection Setup.

The Connect Setup responder shall set the RETRY bit in the FCVI_CONNECT_RESP1 and FCVI_CONNECT_RESP3 IUs equal to the RETRY bit it received in the FCVI_CONNECT_RQST IU within an Exchange. The Connect Setup requestor shall set the RETRY bit in the FCVI_CONNECT_RESP2 IU equal to the RETRY bit it sent in the FCVI_CONNECT_RQST IU within an Exchange.

5.9.4.6 FCVI_FLAGS for disconnect IUs

The bit positions and definitions for all Disconnect IUs is shown in Table 10.

Table 10 – FCVI_FLAGS Bit definitions for disconnect IUs

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	Reserved	0
3	Reserved	0
2	CONN_SETUP_ABORT	If one, a Connection Setup is being aborted If zero, an established connection is being aborted.
1	VI_APP_DISCON	If one, the VI Application has issued a Disconnect request If zero, the FC-VI Provider has issued a Disconnect request
0	CONN_STS	Connection Setup status is present when one.

The CONN_STS bit is set to one when additional connection status information is available. It may be set in the FCVI_DISCONNECT_RQST IU or the FCVI_DISCONNECT_RESP IU. If CONN_STS is set, the FCVI_PARAMETER field shall indicate the additional status information. See Table 11 and Table 12 for the definition of connection status. If the CONN_STS bit is set to zero, no additional connection status information is available.

For a FCVI_DISCONNECT_RQST IU, if VI_APP_DISCON (bit one in FCVI_FLAGS) is set to one, a VI Application has issued the Disconnect Request. For a FCVI_DISCONNECT_RQST IU, if VI_APP_DISCON is set to zero, the FC-VI Provider has issued the Disconnect Request. In a FCVI_DISCONNECT_RESP IU the VI_APP_DISCON shall be set equal the VI_APP_DISCON in the FCVI_DISCONNECT_RQST IU for this Exchange.

If the CONN_SETUP_ABORT bit is set to one, the Disconnect request origniator is aborting a Connection Setup. If the CONN_SETUP_ABORT bit is set to zero, the Disconnect request origniator is aborting an established connection. For a FCVI_DISCONNECT_RESP IU, CONN_SETUP_ABORT shall be the same as that received in the FCVI_DISCONNECT_RQST IU for this Exchange.

5.9.5 Reserved fields

All bits in any Reserved field shall be set to zero by the originator. The responder shall ignore all bit values in any Reserved field.

5.9.6 FCVI_MSG_ID field

The FCVI_MSG_ID field is used to uniquely identify a VI Message within a FC-VI Endpoint on a FC-VI Connection. The FCVI_MSG_ID field facilitates the Message responder detecting lost Messages. Each Message originator maintains a FCVI_MSG_ID count for Messages sent on each of its open FC-VI Connections. The FCVI_MSG_ID shall be assigned by the originator for all Request IUs. The originator shall use a value of 00000001h on the first Message of the VI. Subsequent Messages shall increment the FCVI_MSG_ID by one. When the FCVI_MSG_ID reaches FFFFFFFFh, the originator shall wrap the FCVI_MSG_ID to 0000000h on the next Message. The responder shall return the originator's FCVI_MSG_ID value on all corresponding Response IUs for that Message.

To prevent Message frame synonyms, the FC-VI Provider shall not wrap FCVI_MSG_ID within R_A_TOV.

5.9.7 FCVI_PARAMETER field

5.9.7.1 FCVI_PARAMETER field format

The format of the FCVI_PARAMETER field is specified by the FCVI_FLAGS field.

For VI Message Request IUs (except FCVI_READ_RQST), if the FCVI_FLAGS field has the IMM_DATA bit asserted, then the FCVI_PARAMETER field contains Immediate Data from the Descriptor. Otherwise, the FCVI_PARAMETER field shall be zero for Message Request IUs.

For Message Response IUs, the Parameter field shall be set to zero.

For Connect Request IUs, the Parameter field shall be set to zero.

For Connect Response and Disconnect IUs, if the CONN_STS bit is set in the FCVI_FLAGS field, the FCVI_PARAMETER field contains additional connection status information as shown in Table 11.

Table 11 – FCVI_PARAMETER field for connect response and disconnect IUs

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	Reserved	Reason Code	Reason Explana-tion	Vendor Unique

The Reason Codes are defined in Table 12. A Reason Explanation of 00h indicates no additional explanation for that Reason Code; all other values are reserved.

Table 12 – Reason code for CONN_STS

Code Point	Reason Code
00h	Reserved
01h	No Discriminator Match
02h	Connect Timeout
03h	No Waiting Remote Connectionpoint
04h	Connect Reject
05h	Connect Reject - Concurrent Matching Peer Requests
06h to 1Fh	Reserved
20h	Connect Reject - Invalid Service Parm
21h	Connect Reject - Protocol Error
22h	Connect Reject - Transport Error
23h to 3Fh	Reserved
40h	Transport Error
41h	RDMA Protection Error
42h	Remote Descriptor Error
43h	Remote RDMA Write Protection Error
44h	Remote RDMA Write Data Error
45h	Remote RDMA Write Packet Abort
46h	Remote RDMA Transport Error
47h	Remote RDMA Read Protection Error
48h	Protocol Error
49h	Connection Setup Timeout
4Ah	Connection Does Not Exist
4Bh	Disconnect Accept
4Ch to 5Fh	Reserved
60h to EFh	Reserved
F0h to FFh	Vendor Unique Error

Reason Codes 01h to 1Fh are non-error response codes for Connect Response IUs (FCVI_CONNECT_RESP1, FCVI_CONNECT_RESP2 and FCVI_CONNECT_RESP3). Reason Codes 20h to 3Fh are error response codes for Connect Response IUs. Reason Codes 40h to 5Fh are valid for Disconnect IUs (FCVI_DISCONNECT_RQST IUs or FCVI_DISCONNECT_RESP IUs). Reason Codes

60h to 6Fh are reserved. Reason Codes 60h to 6Fh are Vendor Unique. The Vendor Unique field may be used by FC-VI Providers to specify additional Reason Codes.

5.9.7.2 Connect response reason codes

5.9.7.2.1 Connect response non-error reason codes

The following Reason Codes are valid for all non-error status for Connect Response IUs.

No Discriminator Match: There is no Remote Connectionpoint that matches the Discriminator in the Connect Request.

Connect Timeout: The VI Application at the Connect Request originator has timed out waiting for a Connect Response.

No Waiting Remote Connectionpoint: There is no Connectionpoint waiting at the Remote FC-VI Provider.

Connect Reject: The Remote Endpoint has rejected the Connect Request.

Connect Reject - Concurrent Matching Peer Requests: The Remote FC-VI Provider has detected a Concurrent Matching Peer Request and has rejected the Connect Request.

Reserved: Reserved for future Connect Response non-error codes.

5.9.7.2.2 Connect response error reason codes

The following Reason Codes are valid for all error status for Connect Response IUs.

Connect Reject - Invalid Service Param: The Remote FC-VI Provider has rejected the Connect Request due to invalid PLOGI service parameters (see 5.5).

Connect Reject - Protocol Error: The Remote FC-VI Provider has detected a protocol error and has rejected the Connect Request.

Connect Reject - Transport Error: The Remote FC-VI Provider has detected a transport error and has rejected the Connect Request.

Reserved: Reserved for future Connect Response error codes.

5.9.7.3 Message response / disconnect reason codes

5.9.7.3.1 Descriptor error reason codes

The VI Architecture (see 2.3.1 in VI-ARCH), defines the following Descriptor error codes. The FC-VI Provider may directly map the following FC-VI error codes to the corresponding bits in the Status Field of the Descriptor Control Segment (see 5.2 in VI-DG).

The following Reason Codes are valid for all Message Response IUs and all Disconnect IUs.

Transport Error: The Local or Remote FC-VI Provider has detected a Fibre Channel transport error.

RDMA Protection Error: A protection error for an RDMA operation was detected by the Remote FC-VI Provider.

Remote Descriptor Error: The Remote FC-VI Provider has detected a Descriptor error. Possible errors include no receive Descriptor posted, or length, format or protection error on Descriptor access.

5.9.7.3.2 Remote FC-VI port non-descriptor errors

The VI Architecture defines non-Descriptor error codes (except for protocol error) as seen by the Local VI Application when reported in an asynchronous error notification (See 3.8.1 in VI-DG). FC-VI allows the Disconnect originator to report the originator's non-Descriptor error codes in the FCVI_DISCONNECT_RQST IU, or the Disconnect responder to report the responder's non-Descriptor error codes in the FCVI_DISCONNECT_RESP IU.

NOTE The VI Architecture does not provide an interface to report these Remote errors to the Local VI Application in either the Descriptor or an asynchronous error notification. The Local FC-VI Provider may record these errors for reporting to a VI management application, or it may make the information available to a VI Application through a proprietary interface.

The following Reason Codes are valid for all Disconnect IUs.

Remote RDMA Write Protection Error: The Remote FC-VI Provider detected a protection error on an incoming RDMA Write operation.

Remote RDMA Write Data Error: The Remote FC-VI Provider detected a data corruption error on an incoming RDMA Write operation.

Remote RDMA Write Packet Abort Error: The Remote FC-VI Provider detected a partial data packet on an incoming RDMA Write operation.

Remote RDMA Transport Error: The Remote FC-VI Provider detected a Fibre Channel transport error on an incoming RDMA operation.

Remote RDMA Read Protection Error: The Remote FC-VI Provider detected a protection error on an incoming RDMA Read operation.

Protocol Error: The Remote FC-VI Provider detected a protocol error.

Connection Setup Timeout: The Connect Request originator has detected a timeout waiting for a FCVI_CONNECT_RESP3 IU.

Connection Does Not Exist: The connection or Connection Setup does not exist at the Disconnect responder.

Disconnect Accept: The Disconnect responder is accepting the Disconnect request and is aborting the connection or the Connection Setup.

Reserved: Reserved for future Message Response or Disconnect error codes.

5.9.7.3.3 Reserved for future expansion

Reserved: Reserved for future connection status or error codes.

5.9.7.3.4 Vendor unique reason codes

Vendor Unique: Vendor unique error codes.

5.9.8 FCVI_RMT_VA field

The FCVI_RMT_VA field specifies the base virtual address for the Remote memory region for the FCVI_RDMA_READ_RQST IU, the FCVI_RDMA_WRITE_RQST IU or the FCVI_RDMA_READ_RESP IU. It is reserved for all other IUs. The FCVI_RMT_VA is a 64 bit quantity. The MSB (Most Significant Byte) appears in Word 4, Byte 0, while the LSB (Least Significant Byte) appears in Word 5, Byte 3. The value of FCVI_RMT_VA is identical for all IUs for a Message.

5.9.9 FCVI_RMT_VA_HANDLE field

The FCVI_RMT_VA_HANDLE field specifies the handle for the Remote memory region for the FCVI_RDMA_READ_RQST IU, the FCVI_RDMA_WRITE_RQST IU or the FCVI_RDMA_READ_RESP IU. It is reserved for all other IUs.

5.9.10 FCVI_TOT_LEN field / FCVI_CONNECTION_ID field

The FCVI_TOT_LEN field specifies the VI Message total data length for all FC-VI Message Request IUs. It indicates the total Message length for FCVI_SEND_RQST IUs, FCVI_WRITE_RQST IUs, and FCVI_READ_RQST IUs.

The FCVI_CONNECTION_ID field is used for all Connection IUs. It shall be used by the Connect Request originator to assign a unique identifier to all open Connection Setup operations.

For all Connect Response IUs, the FCVI_CONNECTION_ID field shall be set equal to the FCVI_CONNECTION_ID field from the FCVI_CONNECT_RQST for that Exchange.

If a Connection Setup is aborted, the FCVI_CONNECTION_ID field in the FCVI_DISCONNECT_RQST shall be set equal to the FCVI_CONNECTION_ID field from the FCVI_CONNECT_RQST. The FCVI_CONNECTION_ID in the Disconnect response shall be set equal to the FCVI_CONNECTION_ID field from the FCVI_DISCONNECT_RQST.

A value of FFFFFFFFh is reserved for an unassigned or unknown FCVI_CONNECTION_ID.

6 FC-VI Information Unit (IU) formats

6.1 FC-VI IU overview

The following subclauses describes the Device_Header and payload contents of the IUs transferred during an FC-VI operation. The Information Units used by FC-VI Ports and their characteristics are described in 5.1 of this document.

6.2 FCVI_SEND_RQST IU

6.2.1 FCVI_SEND_RQST IU description

The FCVI_SEND_RQST IU shall transfer all or part of a VI Message to the Remote FC-VI Endpoint. One or more FCVI_SEND_RQST IUs may be sent per Exchange. Refer to Figure 2 or Figure 3 for a definition of the complete Exchange. The Payload of each frame in this IU carries VI Application data. For Unreliable Delivery and Reliable Delivery, the last FCVI_SEND_RQST IU shall terminate the Exchange. For Reliable Reception, the last FCVI_SEND_RQST IU shall transfer Sequence Initiative.

6.2.2 FCVI_SEND_RQST Device_Header information

The FCVI_SEND_RQST IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 5. Sending VI Immediate Data is optional with this IU. If Immediate Data is sent, the IMM_DATA bit in the FCVI_FLAGS field shall be one, and the FCVI_PARAMETER field shall contain the Immediate Data value. Otherwise, the FCVI_PARAMETER and FCVI_FLAGS fields shall be zero.

The FCVI_MSG_ID field shall be incremented by one from the previously sent Message on this connection.

The FCVI_RMT_VA and FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_TOT_LEN field is set to the total length of the VI Message. The FC-VI Provider shall set FCVI_TOT_LEN equal to the Length field in the Control Segment of the Descriptor (see 4.3 in VI-DG).

6.3 FCVI_SEND_RESP IU

6.3.1 FCVI_SEND_RESP IU description

The FCVI_SEND_RESP IU shall be sent in response to a FCVI_SEND_RQST for Reliable Reception. At most one FCVI_SEND_RESP IU shall be sent per VI Message. This IU is not allowed for Unreliable Delivery or Reliable Delivery. Refer to Figure 3 for a definition of the complete Exchange. The Payload of this IU shall be empty. This IU shall contain only one frame. The FCVI_SEND_RESP IU shall terminate the Exchange.

6.3.2 FCVI_SEND_RESP Device_Header information

The FCVI_SEND_RESP IU shall contain a 16 byte FC-VI Device_Header as shown in Table 3.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined by Table 6.

The FCVI_PARAMETER field shall be zero.

6.4 FCVI_WRITE_RQST IU

6.4.1 FCVI_WRITE_RQST IU overview

The FCVI_WRITE_RQST IU shall write all or part of a VI Message to the Remote FC-VI Endpoint. The VI Message data is written to the Remote virtual address generated by adding the FCVI_RMT_VA field in the Device_Header and the Relative Offset field in the frame header. One or more FCVI_WRITE_RQST IUs may be sent per Exchange. Refer to Figure 4 or Figure 5 for a definition of the complete Exchange. The Payload of each frame in this IU carries VI Application data. For Unreliable Delivery and Reliable Delivery, the last FCVI_WRITE_RQST IU shall terminate the Exchange. For Reliable Reception, the last FCVI_WRITE_RQST IU shall transfer Sequence Initiative.

6.4.2 FCVI_WRITE_RQST IU Device_Header information

The FCVI_WRITE_RQST IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 5. Sending VI Immediate Data is optional with this IU. If Immediate Data is sent, then the IMM_DATA bit in the FCVI_FLAGS field shall be one, and the FCVI_PARAMETER field shall contain the Immediate Data value. Otherwise, the FCVI_PARAMETER and FCVI_FLAGS fields shall be zero.

The FCVI_MSG_ID field shall be incremented by one from the previously sent Message on the connection.

The FCVI_RMT_VA field shall be set to the base address of the Remote virtual memory region where the VI Message shall be written to. This value shall be the same for all Request IUs in the Message.

The FCVI_RMT_VA_HANDLE is the handle for the Remote virtual memory region where the VI Message shall be written to. This value shall be the same for all request IUs in the Message.

The FCVI_TOT_LEN field is set to the total length of the VI Message. The FC-VI Provider shall set FCVI_TOT_LEN equal to the Length field in the Control Segment of the Descriptor (see 4.3 in VI-DG). This value shall be the same for all request IUs in the Message.

6.5 FCVI_WRITE_RESP IU

6.5.1 FCVI_WRITE_RESP IU description

The FCVI_WRITE_RESP IU shall be sent in response to a FCVI_WRITE_RQST for Reliable Reception. It is not allowed for Unreliable Delivery or Reliable Delivery. At most one FCVI_WRITE_RESP IU shall be sent per VI Message. Refer to Figure 5 for a definition of the complete Exchange. The Payload of this IU shall be empty. This IU shall contain only one frame. The FCVI_WRITE_RESP IU shall terminate the Exchange.

6.5.2 FCVI_WRITE_RESP IU Device_Header information

The FCVI_WRITE_RESP IU shall contain a 16 byte FC-VI Device_Header as shown in Table 3.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 6.

The FCVI_MSG_ID field shall be the same as that received in the FCVI_WRITE_RQST IU.

The FCVI_PARAMETER field shall be zero.

6.6 FCVI_READ_RQST IU

6.6.1 FCVI_READ_RQST IU description

The FCVI_READ_RQST IU shall request the read of a VI Message from the Remote FC-VI Endpoint This IU is allowed only for Reliable Delivery or Reliable Reception. The VI Message data is read from the Remote memory region beginning at the virtual address specified in the FCVI_RMT_VA field of the Device_Header. The Payload of this IU shall be empty. This IU shall contain only one frame. Sequence Initiative shall be passed at the completion of this IU. Refer to Figure 6 for a definition of the complete Exchange.

6.6.2 FCVI_READ_RQST IU Device_Header information

The FCVI_READ_RQST IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 5. Sending VI Immediate Data is not allowed with this IU. The IMM_DATA bit in the FCVI_FLAGS field shall be zero, and the FCVI_PARAMETER field shall be zero.

The FCVI_MSG_ID field shall be incremented by one from the previously sent Message on the connection.

The FCVI_RMT_VA field shall be set to the base address of the Remote virtual memory region where the VI Message shall be read from.

The FCVI_RMT_VA_HANDLE is the handle for the Remote virtual memory region where the VI Message shall be read from.

The FCVI_TOT_LEN field is set to the total length of the VI Message. The FC-VI Provider shall set FCVI_TOT_LEN equal to the Length field in the Control Segment of the Descriptor (see 4.3 in VI-DG).

6.7 FCVI_READ_RESP IU

6.7.1 FCVI_READ_RESP IU description

The FCVI_READ_RESP IU shall be sent in response to a FCVI_READ_RQST for Reliable Delivery and Reliable Reception. It shall transfer all or part of a VI Message to the Remote FC-VI Endpoint. The VI Message data is read from the Remote virtual address generated by adding the FCVI_RMT_VA field in the Device_Header and the Relative Offset field in the frame header. One or more FCVI_READ_RESP IUs may be sent per Exchange. The last FCVI_READ_RESP IU shall terminate the Exchange. Refer to Figure 6 for a definition of the complete Exchange. The Payload of each frame in this IU carries VI Application data.

6.7.2 FCVI_READ_RESP IU Device_Header information

The FCVI_READ_RESP IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined by Table 6.

The MSG_ID field shall be the same as that received in the FCVI_READ_RQST IU.

The FCVI_PARAMETER field shall be zero.

The FCVI_RMT_VA field shall be set to the base address of the Remote virtual memory region where the VI Message is read from. This value shall be the same as the FCVI_RMT_VA field in the FCVI_READ_RQST IU. This value shall be the same for all Response IUs in the Message.

The FCVI_RMT_VA_HANDLE is the handle for the Remote virtual memory region where the VI Message is read from. This value shall be the same as the FCVI_RMT_VA_HANDLE field in the FCVI_READ_RQST IU. This value shall be the same for all Response IUs in the Message.

The FCVI_TOT_LEN field is set to the FCVI_TOT_LEN received in the FCVI_READ_RQST IU. This value shall be the same for all Response IUs in the Message.

6.8 FCVI_CONNECT_RQST IU

6.8.1 FCVI_CONNECT_RQST IU description

The FCVI_CONNECT_RQST IU shall request that an FC-VI Connection be established. Refer to Figure 8, Figure 9, or Figure 10 for a definition of the complete Exchange. The Payload for the FCVI_CONNECT_RQST IU is shown in Table 13. Sequence Initiative shall be passed at the completion of this IU. The payload length for the FCVI_CONNECT_RQST IU is 596 bytes if the FCVI_CONNECT_INFO is present, or 340 bytes if the FCVI_CONNECT_INFO is not present.

6.8.2 FCVI_CONNECT_RQST Device_Header information

The FCVI_CONNECT_RQST IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE is unassigned.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field are defined in Table 7.

The FCVI_MSG_ID field shall be set to zero.

The FCVI_PARAMETER field shall be zero.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_CONNECTION_ID shall be set to a unique value by the Connect Request originator.

6.8.3 FCVI_CONNECT_RQST Payload Information

Table 13 shows the payload for the FCVI_CONNECT_RQST IU.

Table 13 – FCVI_CONNECT_RQST IU payload format

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	Reserved			
1	Reserved		FCVI_REVISION	
2	FCVI_RQST_HANDLE			
3	MSB			
4 to 38	FCVI_LOC_ADDR			
39				LSB
40	MSB			
41 to 75	FCVI_REM_ADDR			
76				LSB
77	MSB			
78 to 83	FCVI_LOC_ATTRS			
84				LSB
85	MSB			
86 to 147	FCVI_CONNECT_INFO (optional)			
148				LSB

The FCVI_REVISION field shall be set to 0001h to indicate the first revision of this standard.

The FCVI_RQST_HANDLE field shall be chosen by the Connect Request originator of this IU. The Connect Request responder shall use this same FCVI_RQST_HANDLE in the FCVI_HANDLE field when transferring VI Messages on this FC-VI Connection.

The FCVI_LOC_ADDR field is the FCVI_NET_ADDRESS for the FC-VI Connectionpoint of the Connect Request originator. Table 15 defines the format of this field.

The FCVI_REM_ADDR field is the FCVI_NET_ADDRESS for the FC-VI Connectionpoint of the Connect Request responder. Table 15 defines the format of this field.

The FCVI_LOC_ATTRS field is the VI attributes for the FC-VI Endpoint of the Connect Request originator. Table 16 defines the format of this field.

If the FCVI_CONN_INFO is set to one in the FCVI_FLAGS field of the Connect Request, the FCVI_CONNECT_INFO field contains 256 bytes of optional Local FC-VI Provider supplied connection information passed from the Connect Request originator to the Connect Request responder. If the FCVI_CONN_INFO is set to zero, the FCVI_CONNECT_INFO field is not present.

6.9 FCVI_CONNECT_RESP1 IU

6.9.1 FCVI_CONNECT_RESP1 IU description

The FCVI_CONNECT_RESP1 IU shall be sent in response to a FCVI_CONNECT_RQST IU. Refer to Figure 8, Figure 9, or Figure 10 for a definition of the complete Exchange. The Payload for the FCVI_CONNECT_RESP1 IU is shown in Table 14. Sequence Initiative shall be passed at the completion of this IU. The payload length for the FCVI_CONNECT_RESP1 IU is 596 bytes if the FCVI_CONNECT_INFO is present, or 340 bytes if the FCVI_CONNECT_INFO is not present. The format of the FCVI_CONNECT_RESP1 IU is identical to the format of the FCVI_CONNECT_RQST IU.

6.9.2 FCVI_CONNECT_RESP1 Device_Header information

The FCVI_CONNECT_RESP1 IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE field is unassigned.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field shall be defined in Table 9.

The FCVI_MSG_ID field shall be set to zero.

If the FCVI_FLAGS field is zero, then the FCVI_PARAMETER field shall be zero. If the FCVI_FLAGS field is not zero (i.e., the CONN_STS bit is set), then the FCVI_PARAMETER field shall contain connection status information as shown in Table 11 and Table 12.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_CONNECTION_ID shall be the same as that received in the FCVI_CONNECT_RQST IU for this Exchange.

6.9.3 FCVI_CONNECT_RESP1 Payload Information

Table 14 – FCVI_CONNECT_RESP1 IU Payload Format

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	Reserved			
1	Reserved		FCVI_REVISION	
2	FCVI_RESP_HANDLE			
3	MSB	FCVI_LOC_ADDR		
4-38				
39				LSB
40	MSB	FCVI_REM_ADDR		
41 to 75				
76				LSB
77	MSB	FCVI_LOC_ATTRS		
78 to 83				
84				LSB
85	MSB	FCVI_CONNECT_INFO (optional)		
86 to 147				
148				LSB

The FCVI_REVISION field shall be set to 0001h to indicate the first revision of this standard.

The FCVI_RESP_HANDLE field shall be chosen by the Connect Request responder, the originator of this IU. The Connect Request originator shall use the value specified in the FCVI_RESP_HANDLE field for the FCVI_HANDLE field in all Message request IUs sent to the Connect Request responder on this FC-VI Connection.

The FCVI_LOC_ADDR field is the FCVI_NET_ADDRESS for the FC-VI Connectionpoint of the Connect Request responder. Table 15 defines the format of this field.

The FCVI_REM_ADDR field is the FCVI_NET_ADDRESS for the FC-VI Connectionpoint of the Connect Request originator. Table 15 defines the format of this field.

The FCVI_LOC_ATTRS field is the VI attributes for the FC-VI Endpoint of the Connect Request responder. Table 16 defines the format of this field.

If the FCVI_CONN_INFO is set to one in the FCVI_FLAGS field of the FCVI_CONNECT_RESP1, the FCVI_CONNECT_INFO field contains 256 bytes of optional FC-VI Provider supplied connection information passed from the Connect Request responder to the Connect Request originator. If the FCVI_CONN_INFO is set to zero, the FCVI_CONNECT_INFO field is not present.

6.10 FCVI_CONNECT_RESP2 IU

6.10.1 FCVI_CONNECT_RESP2 IU description

The FCVI_CONNECT_RESP2 IU shall be sent in response to a FCVI_CONNECT_RESP1 IU. Refer to Figure 8, Figure 9, or Figure 10 for a definition of the complete Exchange. The payload for the FCVI_CONNECT_RESP2 IU shall be empty. Sequence Initiative shall be passed at the completion of this IU. This IU shall contain only one frame.

6.10.2 FCVI_CONNECT_RESP2 Device_Header information

The FCVI_CONNECT_RESP2 IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE field shall be set to the FCVI_RESP_HANDLE received in the payload of the FCVI_CONNECT_RESP1 IU.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 9.

The FCVI_MSG_ID field shall be set to zero.

If the FCVI_FLAGS field is zero, then the FCVI_PARAMETER field shall be zero. If the FCVI_FLAGS field is not zero (i.e., the CONN_STS bit is set), then the FCVI_PARAMETER field shall contain connection status information as shown in Table 11 and Table 12.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_CONNECTION_ID shall be the same as that sent in the FCVI_CONNECT_RQST IU for this Exchange.

6.11 FCVI_CONNECT_RESP3 IU

6.11.1 FCVI_CONNECT_RESP3 IU description

The FCVI_CONNECT_RESP3 IU shall be sent in response to a FCVI_CONNECT_RESP2 IU. Refer to Figure 8, Figure 9, or Figure 10 for a definition of the complete Exchange. The payload for the FCVI_CONNECT_RESP3 IU shall be empty. This IU shall terminate the Exchange. This IU shall contain only one frame.

6.11.2 FCVI_CONNECT_RESP3 Device_Header information

The FCVI_CONNECT_RESP3 IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE field shall be set to the FCVI_RQST_HANDLE received in the payload of the FCVI_CONNECT_RQST IU.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 9.

The FCVI_MSG_ID field shall be set to zero.

If the FCVI_FLAGS field is zero, then the FCVI_PARAMETER field shall be zero. If the FCVI_FLAGS field is not zero (i.e., the CONN_STS bit is set), then the FCVI_PARAMETER field shall contain connection status information as shown in Table 11 and Table 12.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_CONNECTION_ID shall be the same as that received in the FCVI_CONNECT_RQST IU for this Exchange.

6.12 FCVI_DISCONNECT_RQST IU

6.12.1 FCVI_DISCONNECT_RQST IU description

The FCVI_DISCONNECT_RQST IU shall request an FC-VI Connection be removed or a Connection Set-up be aborted. Refer to Figure 11 for a definition of the complete Exchange. The Payload for the FCVI_DISCONNECT_RQST IU shall be empty. Sequence Initiative shall be passed at the completion of this IU. This IU shall contain only one frame.

6.12.2 FCVI_DISCONNECT_RQST Device_Header information

The FCVI_DISCONNECT_RQST IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup. If a Connection Setup is being aborted and the FCVI_HANDLE of the Remote FC-VI Endpoint is unknown, the FCVI_HANDLE shall be set to FFFFFFFFh.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 9. If the CONN_STS bit is set to one, the FCVI_PARAMETER field contains additional connection status information from the Disconnect responder as defined in Table 11 and Table 12. If the CONN_STS bit is set to zero, the FCVI_PARAMETER is set to zero. If the Disconnect originator is aborting a Connection Setup, the CONN_SETUP_ABORT shall be set to one. If the Disconnect originator is aborting an established FC-VI Connection, the CONN_SETUP_ABORT shall be set to zero. If the VI Application issues the Disconnect request, the VI_APP_DISCON shall be set to one. If the VI Provider issues the Disconnect request, the VI_APP_DISCON shall be set to zero.

If the Disconnect originator is aborting a Connection Setup, the FCVI_MSG_ID field shall be set to zero. If the Disconnect originator is aborting an established connection, the FCVI_MSG_ID field shall be set to the FCVI_MSG_ID of the last Message successfully completed by the Disconnect originator.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

If the Disconnect originator is aborting a Connection Setup, the FCVI_CONNECTION_ID shall be the same as that sent or received in the FCVI_CONNECT_RQST IU for this Exchange. If the Disconnect originator is aborting an established connection, the FCVI_CONNECTION_ID shall be zero.

6.13 FCVI_DISCONNECT_RESP IU

6.13.1 FCVI_DISCONNECT_RESP IU description

The FCVI_DISCONNECT_RESP IU shall be sent in response to a FCVI_DISCONNECT_RQST IU. Refer to Figure 11 for a definition of the complete Exchange. The Payload for the FCVI_DISCONNECT_RESP IU shall be empty. This IU shall contain only one frame, and the Exchange is terminated at the completion of this IU.

6.13.2 FCVI_DISCONNECT_RESP Device_Header information

The FCVI_DISCONNECT_RESP IU shall contain a 32 byte FC-VI Device_Header as shown in Table 4.

The FCVI_HANDLE shall be set to the FCVI_HANDLE communicated by the Remote FC-VI Endpoint during FC-VI Connection Setup. If a Connection Setup is being aborted and the FCVI_HANDLE of the Remote FC-VI Endpoint is unknown, the FCVI_HANDLE shall be set to FFFFFFFFh.

The FCVI_OPCODE field is defined in Table 1.

The FCVI_FLAGS field is defined in Table 9. The CONN_SETUP_ABORT and VI_APP_DISCON flags shall be the same as that received in the FCVI_DISCONNECT_RQST IU for this Exchange. If the CONN_STS bit is set to one, the FCVI_PARAMETER field contains additional connection status information from the Disconnect responder as defined in Table 11 and Table 12. If the CONN_STS bit is set to zero, the FCVI_PARAMETER is set to zero.

If a Connection Setup is being aborted (CONN_SETUP_ABORT is set to one), the FCVI_MSG_ID field shall be set to zero. If an established connection is being aborted (CONN_SETUP_ABORT is set to zero), the FCVI_MSG_ID field shall be set to the FCVI_MSG_ID of the last Message successfully completed by the Disconnect responder.

The FCVI_RMT_VA shall be set to zero

The FCVI_RMT_VA_HANDLE shall be set to zero.

The FCVI_CONNECTION_ID shall be the same as that received in the FCVI_DISCONNECT_RQST IU for this Exchange. This field may only be non-zero when a Connection Setup is being aborted.

7 FC-VI Addressing and naming

7.1 FC-VI Addressing and naming overview

This Clause defines address assignment for FC-VI Ports and the translation between VI Architecture addresses and FC-VI address constructs.

7.2 FCVI_NET_ADDRESS format

The VI Architecture does not define the format of a VI Address. Instead, the VI Architecture defines an abstract VI Address structure for a VI NIC, which contains a Host Address and a Discriminator. This VI Address structure is called a VIP_NET_ADDRESS and is defined by VI Architecture as follows:

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

The VIP_NET_ADDRESS is visible to a VI Application and it is used to establish a VI connection with a Remote VI Endpoint. The VI Architecture assumes that the VI Provider maps a VIP_NET_ADDRESS to a transport specific addressing structure. FC-VI maps a VIP_NET_ADDRESS to the FCVI_NET_ADDRESS format.

The Host Address for FC-VI shall be an IPv6 address (see RFC2373). The VI Architecture defines an interface to a generic name service to map from Host Names to Host Addresses. Since the Host Address is an IP address, FC-VI facilitates the use of DNS to map between Host Names and IP addresses. Thus, FC-VI implicitly assumes that each FC-VI Host or Node has the capability to access DNS, typically by an Ethernet connection running TCP/IP.

The specific format of the Discriminator portion of the VI Address is outside the scope of this standard, as it is intended to be application specific. However, the VI Architecture specifies that the Discriminator portion may be of variable width from 0 to MaxDiscriminatorLen bytes as indicated in the VI NIC Attributes. FC-VI defines a 128 byte field in FCVI_NET_ADDRESS field for the Discriminator. An FC-VI Provider shall indicate a MaxDiscriminatorLen in the VI NIC Attributes of no more than 128 bytes.

The actual byte length allocated within the Connect Request and response payload for the FCVI_NET_ADDRESS shall be 2 bytes (Reserved) + 1 byte (HostAddressLen) + 1 byte (DiscriminatorLen) + 16 bytes (IP Addr) + 128 bytes (Discriminator) = 148 bytes. For FC-VI, the value of HostAddressLen shall be 16 bytes. The value of DiscriminatorLen may be any value between 16 and 128 bytes. The Discriminator is always allocated as a 128 byte field in FCVI_NET_ADDRESS.

Table 15 illustrates the format used within FC-VI Connection IUs to convey a FCVI_NET_ADDRESS. The HOST_ADD field shall be the Host Address in IPv6 format. The MSB (Most Significant Byte) of the HOST_ADD is contained in Byte 0, Word 1. The LSB (Least Significant Byte) of the HOST_ADD is contained in Byte 3, Word 4. The HOST_ADD_LEN shall be equal to 10h. The DISCRIM field shall be the Discriminator, whose actual length is specified by the DISCRIM_LEN field. The maximum allowed value of DISCRIM_LEN is 80h, while the minimum value is 10h. The MSB (Most Significant Byte) of the DISCRIM is contained in Byte 0, Word 5. The LSB (Least Significant Byte) of the DISCRIM is contained in Byte 3, Word 36.

Table 15 – FCVI_NET_ADDRESS Format

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	Reserved		HOST_ADD_LEN	DISCRIM_LEN
1	MSB			
2 to 3	HOST_ADD			
4				LSB
5	MSB			
6 to 35	DISCRIM			
36				LSB

7.3 FCVI_ATTRIBUTES format

Table 16 illustrates the format used within FC-VI Connection IUs to convey FCVI_ATTRIBUTES. For a Connect Request, the FCVI_ATTRIBUTES convey the VI Attributes of the connection originator. For a FCVI_CONNECT_RESP1, the FCVI_ATTRIBUTES convey the VI Attributes of the connection responder.

Table 16 – FCVI_ATTRIBUTES format

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	Reserved		FCVI_RELIABILITY_LVL	FCVI_ATTR_FLAGS
1	FCVI_MAX_TRANS_SIZE			
2	MSB			
3 to 5	FCVI_QOS			
6				LSB

FCVI_RELIABILITY_LVL: The Reliability Level of the VI. If set to 01h, the Reliability Level is Unreliable Delivery. If set to 02h, the Reliability Level is Reliable Delivery. If set to 03h, the Reliability Level is Reliable Reception. All other values are reserved.

FCVI_ATTR_FLAGS: The VI Attributes flags are defined in Table 17.

FCVI_MAX_TRANS_SIZE: The maximum Message size in bytes for this VI.

FCVI_QOS: The VI QOS parameters are defined in Table 18.

Table 17 – Format of FCVI_ATTR_FLAGS in FCVI_ATTRIBUTES

Bit Position	Bit Name	Bit Definition
7	Reserved	0
6	Reserved	0
5	Reserved	0
4	Reserved	0
3	Reserved	0
2	Reserved	0
1	EnRdmaWr	VI enabled for RDMA Write when one
0	EnRdmaRd	VI enabled for RDMA Read when one

Table 17 defines the format of the FCVI_ATTR_FLAGS field in the FCVI_ARRTIBUTES field (see Table 16). If EnRDMAWr is set to one, the VI is enabled to receive RDMA Write requests. If EnRDMAWr is set to zero, the VI is not enabled to receive RDMA Write requests. If EnRDMARd is set to one, the VI is enabled to receive RDMA Read requests. If EnRDMARd is set to zero, the VI is not enabled to receive RDMA Read requests.

Table 18 – FCVI_QOS format

Bits Word	31 to 24 Byte 0	23 to 16 Byte 1	15 to 08 Byte 2	07 to 00 Byte 3
0	FCVI_PREF	Reserved		
1	FCVI_MAX_BANDWIDTH			
2	FCVI_MIN_BANDWIDTH			
3	FCVI_MAX_DELAY			
4	FCVI_PIPELINE_DEPTH			

Table 18 defines the format for FCVI_QOS field in FCVI_ATTRIBUTES field (see Table 16).

FCVI_PREF: If FCVI_PREF is set to 80h, the FC-VI Provider is requesting that all frames sent on this VI use the FC-FS Preference Function. When FCVI_PREF is set to 80h, the FC-VI Provider shall always send a Message frame with the CS_CTL field specifying the Preference Function for this VI. If FCVI_PREF is set to 00h, the FC-VI Provider is requesting that all frames sent on this VI shall not use the FC-FS Preference Function. When FCVI_PREF is set to 00h, the FC-VI Provider shall never send a Message frame with the CS_CTL field specifying the Preference Function for this VI. All other values are reserved.

FCVI_MAX_BANDWIDTH: Specifies the maximum bandwidth for the VI in bytes per second. If non-zero, the FC-VI Provider is guaranteeing the maximum bandwidth between the VI Endpoints of a connection shall be no more than FCVI_MAX_BANDWIDTH bytes per second. If zero, the FC-VI Provider is not guaranteeing the maximum bandwidth between the VI Endpoints of a connection. For this version of the FC-VI standard, this field shall always be zero.

FCVI_MIN_BANDWIDTH: Specifies the minimum bandwidth for the VI in bytes per second. If non-zero, the FC-VI Provider is requiring the minimum bandwidth between the VI Endpoints of a connection shall be at least FCVI_MIN_BANDWIDTH bytes per second. If zero, the FC-VI Provider is not guaranteeing the minimum bandwidth between the VI Endpoints of a connection. For this version of the FC-VI standard, this field shall always be zero.

FCVI_MAX_DELAY: Specifies the maximum delay for the VI in microseconds. If non-zero, the FC-VI Provider is requiring the maximum delay between the VI Endpoints of a connection be no more than FCVI_MAX_DELAY microseconds. If zero, the FC-VI Provider is not guaranteeing the maximum delay between the VI Endpoints of a connection. For this version of the FC-VI standard, this field shall always be zero.

FCVI_PIPELINE_DEPTH: Specifies the maximum number of Messages that an originator may send without a confirmation of delivery indication from the Message responder. The pipeline depth may represent credits, command queue slots, unacknowledged messages or some other FC-VI Provider specific construct. If FCVI_PIPELINE_DEPTH is set to zero, then only one Message may be open at any one time in this VI. If FCVI_PIPELINE_DEPTH is set to some positive integer N, then at most N + 1 messages may be open at any one time on this VI.

A Message is open when the FC-VI Provider begins processing the Message Descriptor. The Message is closed when the Message Descriptor is completed by the FC-VI Provider.

7.4 FC-VI address resolution

A VI Application uses a Host Address to specify which VI Host it wishes to connect to during Connection Setup. FC-VI defines the Host Address as an IP address. The FCVI_CONNECT_RQST IU requires the N_Port Identifier for the destination FC-VI Port before a Connect Request may be originated.

FC-VI requires all FC-VI Ports to support FARP to map between IP addresses and N_Port Identifiers. All compliant implementations of FC-VI Ports shall be capable of initiating FARP-REQ to request an IP address to N_Port ID mapping for an FC-VI Port. All compliant implementations of FC-VI Ports shall recognize and respond to a FARP-REQ with a FARP-REPLY if the IP address in the FARP-REQ matches the IP address of the responding FC-VI Port. Alternative methods for FC-VI address resolution are allowed, as long as FARP is supported.

If a FC-VI Port is attached to a Fabric that supports a Name Server, the Name Server should be used in lieu of FARP to resolve IP address to N_Port Identifier mappings.

7.5 FARP ELS

When issuing a FARP-REQ, an FC-VI Port shall

1) Set the Match Address Code Points equal to 00000100b (Match on IP address of responder).

2) Set the Responder Action equal to 02h, requesting the responder originate a FARP-REPLY to the FARP-REQ originator if the Match is successful.

3) Set the Requesting N_Port Identifier equal to the N_Port Identifier of the FARP-REQ originator.

4) Set the Responding N_Port Identifier field equal to zero.

5) Set the Requesting N_Port Port_Name equal to the Port_Name of the FARP-REQ originator.

6) Set the Responding N_Port Port_Name equal to zero.

7) Set the Requesting N_Port Node_Name equal to the Node_Name of the FARP-REQ originator.

8) Set the Responding N_Port Node_Name equal to zero.

9) Set the Requesting N_Port IP address equal to the IP address of the FARP-REQ originator.

10) Set the Responding N_Port IP address equal to the IP address of the FC-VI Port the FARP-REQ originator wishes to establish a FC-VI Connection with.

When receiving a FARP-REQ, an FC-VI Port shall perform the following three tests:

11) Compare for equality its FC-VI Port IP address to the Responding N_Port IP address in the FARP request payload

12) Test if the Match on IP address of responder code point, 00000100b, is set in the FARP-REQ Match Address Code Points

13) Test if the Responder Action in the FARP-REQ to is set to 02h, requesting the responder to originate a FARP-REPLY with the Requesting N_Port Identifier if the Match is successful.

If any test fails, the FC-VI Provider at the FARP-REQ recipient shall take no action.

NOTE Other ULPs behind the Port, such as FC-IP, may take action on the FARP-REQ if the FC-VI Provider does not act on the FARP-REQ.

If all three tests pass, the FC-VI Provider at the FARP-REQ recipient shall originate a FARP-REPLY to the FARP-REQ originator by formatting the FARP-REPLY payload as follows:

14) Set the Match Address Code Points equal to the Match Address Code Points in the FARP-REQ (00000100b).

15) Set the Requesting N_Port Identifier equal to the N_Port Identifier of the FARP-REQ originator.

16) Set the Responder Action equal to the Responder Action in the FARP-REQ (02h).

17) Set the Responding N_Port Identifier equal to its N_Port Identifier.

18) Set the Requesting N_Port Port_Name equal to the Port_Name of the FARP-REQ originator.

19) Set the Responding N_Port Node_Name equal to the Port_Name of the FARP-REPLY originator.

20) Set the Requesting N_Port Node_Name equal to the Node_Name of the FARP-REQ originator.

21) Set the Responding N_Port Node_Name equal to the Node_Name of the FARP-REPLY originator.

22) Set the Requesting N_Port IP address equal to the IP address of the FARP-REQ originator.

23) Set the Responding N_Port IP address equal to the IP address of the FARP-REPLY originator.

7.6 Name server queries

An FC-VI Port that is attached to a Fabric which supports the Name Server may map N_Port Identifiers to IP addresses by issuing a GID_IPP Request to the Name Server. The Name Server is defined in FC-GS-2 and FC-GS-3. The originator of the GID_IPP Name Server Query supplies the IP address for which the N Port Identifier is sought.

The Name Server may return a list of N_Port Identifiers associated with one IP address. The FC-VI Provider which receives more than one N_Port Identifier to a GID_IPP request may use an implementation dependent method to select one of the N_Port Identifiers to use as the destination for a Connection Setup request.

7.7 Validation of host address to N_Port Identifier mappings

7.7.1 Address mapping overview

At all times, the <Host Address, N_Port Identifier> mapping shall be valid before use by an FC-VI Provider. There are many events that can invalidate this mapping. After a FC link interruption occurs, the N_Port Identifiers of all other FC-VI Ports that have previously completed PLOGI (N_Port Login) with an FC-VI Port may have changed, and its own N_Port Identifier may have changed. Because of this, validation of the address mapping is required after a LIP in a loop topology or after NOS/OLS in a point-to-point or Fabric topology. In addition, mappings may have changed due to remote link events in a Fabric topology. N_Port Identifiers shall not change as a result of Link Reset (LR), thus validation is not required.

If a logout is sent to or received from a Remote FC-VI Port, all FC-VI Connections shall be terminated with that Remote FC-VI Port.

7.7.2 Point-to-point topology

In a point-to-point topology, validation is not required after LR. NOS/OLS causes implicit Logout of the other FC-VI Port. After a NOS/OLS, each FC-VI Port shall perform a PLOGI before any communication with the other FC-VI Port may occur.

7.7.3 Private loop topology

After a LIP in a private loop topology, an FC-VI Port shall not transmit any FC-VI IU's to an FC-VI Port until the <Host Address, N_Port Identifier> mappings of the logged in FC-VI Port has been validated. The validation shall consist of completing an ADISC (sending or receiving ADISC ACC) to all logged in FC-VI Ports

after a LIP. If the three addresses returned in the ACC to the ADISC - N_Port Identifier, Port_Name, Node_Name - exactly match the values of the Remote FC-VI Port prior to the LIP, then any active Exchanges may continue with that FC-VI Port. If any of the three addresses have changed, then the Remote FC-VI Port must be explicitly logged out. If a FC-VI Port's N_Port Identifier changes after a LIP, then all <Host Address, N_Port Identifier> mappings shall be invalidated and an explicit LOGO shall be sent to each logged in FC-VI Port.

7.7.4 Public loop topology

In a public loop topology, a FC-VI Port shall wait for the receipt of a FAN ELS after a LIP. If the Port_Name and Node_Name of the Fabric FL_Port contained in the FAN ELS response exactly match the values before the LIP, and if the AL_PA obtained by the FC-VI Port is the same as the one before the LIP, the FC-VI Port may resume all Exchanges. If not, then FLOGI (Fabric Login) shall be performed with the Fabric, all <Host Address, N_Port Identifier> mappings shall be invalidated, and all logged in FC-VI Ports shall be explicitly logged out. A public loop device shall perform private loop authentication to any nodes on the local loop which have an Area + Domain Address = 00-00-XXh by issuing an ADISC to each private FC-VI Port and performing address validation as specified for a private loop topology in 7.7.3.

7.7.5 Fabric topology

In a point-to-point topology, no validation is required after LR (link reset). After NOS/OLS, a FC-VI Port shall perform FLOGI. If, after FLOGI, the N_Port Identifier of the FC-VI Port, the Port_Name of the Fabric, and the Node_Name of the Fabric are the same as before the NOS/OLS, then the FC-VI Port may resume all Exchanges. If not, then FLOGI (Fabric Login) shall be performed with the Fabric, all <Host Address, N_Port Identifier> mappings shall be invalidated, and all logged in FC-VI Ports shall be explicitly logged out.

A Fabric attached FC-VI Port may elect to register for Registered State Change Notification as defined in FC-FS to receive notification for the loss of connectivity (link, switch or node failures) to other Fabric attached FC-VI Ports.

8 FC-VI Error detection and recovery

8.1 FC-VI error detection and recovery overview

This Clause defines the rules for FC-VI error detection and recovery. Two sets of rules are defined. One set is defined for Message Transfer operations, while a second set is defined for Connection Setup operations. The rules support acknowledged (class 2) and unacknowledged (class 3) classes of service, as well as In-Order and Out-of-Order Fabrics.

Certain errors may cause the VI Endpoint to transition to the Error state. Once in the Error state, the FC-VI Endpoint stops all inbound and outbound traffic. However, a VI Application decides when it is appropriate to disconnect the VI Connection, according to the VI Architecture specifications.

A VipDisconnect is used by the VI Application to transition the Local VI Endpoint back to the Idle state and to signal the Disconnect to the Remote VI Endpoint. A VipDisconnect causes the FC-VI Provider to transmit a FC-VI Disconnect IU. A VipDisconnect may be issued by either the Local or Remote VI Application at any time for a given connection.

8.2 FC-VI endpoint states

The VI Architecture defines the valid states for a VI Endpoint. FC-VI Endpoints adopt these same states and adds a fifth state - Pending Connect Retry. These states are defined below.

a) Idle - FC-VI Endpoint state after a VI is created. Idle state transitions to Pending Connect state when a Connect Request is transmitted.

b) Pending Connect - FC-VI Endpoint state after a Connect Request is transmitted. If the Connection Setup succeeds, transition to the Connected state. If the VI Application times out the Connect Request or the Connect Response indicates a non-error condition (see 5.9.7.2.1, Connect Response Non-error Reason Codes), transition to the Idle state. If the Connection Setup fails due to a transport error, transition to the Pending Connect Retry state. Otherwise, transition to the Error state.

c) Pending Connect Retry - FC-VI Endpoint state for retrying failed Connect Requests. If Connection Setup succeeds, transition to Connected state. If Connection Setup does not succeed, transition to the Idle state. Otherwise, transition to the Error state.

NOTE The Connect Response status was No Discriminator Match, No Waiting Remote Connectionpoint, or Connect Reject.

d) Connected - Connect Request completed successfully. Transition to Idle state after successful Disconnect. Upon error, transition to the Error state.

e) Error - A Disconnect is required to transition back to the Idle state.

8.3 FCVI_ULP_TIMEOUT definition

FC-VI Providers shall wait for a Message Response for up to FCVI_ULP_TIMEOUT.

FC-VI Providers that are originating a Connect Request shall wait for a FCVI_CONNECT_RESP3 IU for the same Exchange for up to FCVI_ULP_TIMEOUT.

FC-VI Providers that have received a FCVI_CONNECT_RQST IU shall wait for a FCVI_CONNECT_RESP2 IU for the same Exchange for up to two times FCVI_ULP_TIMEOUT.

FC-VI Providers that are originating a FCVI_DISCONNECT_RESP IU shall wait for a FCVI_DISCONNECT_RESP IU for up to FCVI_ULP_TIMEOUT.

The value of FCVI_ULP_TIMEOUT shall be set to R_A_TOV.

8.4 Message transfer error detection and recovery rules

8.4.1 Message error detection

The following rules support error detection for Unreliable Delivery, Reliable Delivery, and Reliable Reception for Sends, RDMA Writes and RDMA Reads. The detected error is a Message Error. For Reliable Delivery and Reliable Reception, an FC-VI Provider shall deliver all Messages in order to the VI Application.

a) If a Message Response is required, the Message originator shall detect a Message Error if FCVI_ULP_TIMEOUT expires for the Message Response.

b) For In-Order Fabrics and Unreliable Delivery, the Message responder may detect a Message Error if any frames are received out-of-order.

c) For In-Order Fabrics and either Reliable Delivery or Reliable Reception, the Message responder shall detect a Message Error if any frames or Messages are received out-of-order.

d) For Out-Of-Order Fabrics, the Message responder shall detect a Message Error when a missing frame error is detected. See FC-FS.

Note 1 If the prior sequence within a Message was received without error, a Message responder may wait an implementation dependent amount of time to receive the next Sequence, since Fibre Channel does not require the Sequence Initiator to transmit a subsequent Sequence within any defined time period. If the next Sequence is not received before this implementation dependent amount of time has expired a Message Error shall be detected.

e) For Out-Of-Order Fabrics, a Message responder shall detect a Message Error for missing Messages.

Note 2 A Message responder should detect a missing Message by implementation dependent timeout methods.

8.4.2 Message transfer error recovery

The following rules are used to perform error recovery on a Local FC-VI Endpoint.

a) If the Message originator detected a FCVI_ULP_TIMEOUT on a Message response, it shall abort the Exchange associated with the Message using ABTS.

b) If the Message originator receives an ACK with Abort Sequence Condition bits set to 01b (Abort Sequence, Perform ABTS) in F_CTL, it shall abort the Exchange associated with the Message using ABTS.

c) For Unreliable Delivery, if a Message Error is detected, the FC-VI Provider may transition the Local FC-VI Endpoint to the Error state.

d) For Unreliable Delivery, if a Message Error is detected by a Message responder and the Local FC-VI Endpoint is not transitioned to the Error state, the FC-VI Provider shall discard any frames for the FCVI_MSG_ID corresponding to the Message Error for R_A_TOV.

e) For Reliable Delivery and Reliable Reception, if a Message Error is detected, the FC-VI Provider shall transition the Local FC-VI Endpoint to the Error state.

f) If the Local FC-VI Endpoint has transitioned to the Error state, the FC-VI Provider shall discard all Message frames received with the FCVI_HANDLE of the FC-VI Connection for R_A_TOV. An FCVI_HANDLE shall not be reused for a new FC-VI Connection unless a Disconnect has been successfully completed (i.e., a Disconnect response has been received) and R_A_TOV has elapsed.

g) If an implicit or explicit Port logout is sent or received from an FC-VI Port, the FC-VI Provider may reuse any FCVI_HANDLEs and clear any Message Errors associated with the logged out FC-VI Port.

h) For class 2 service, if a Message Error is detected, the Message responder shall transmit an ACK with the Abort Sequence Condition bits set to 01b (Abort Sequence, Perform ABTS) in F_CTL.

8.5 Connection setup error detection and recovery rules

8.5.1 Connection setup error handling overview

The following rules support error detection and recovery for FC-VI Connection Setup. See 5.3 for the FC-VI Connection Setup protocol description. Detected errors include a Connection Setup Exchange Error and a Connection Setup Timeout Error.

8.5.2 Connection setup error detection

a) If a Connect Request originator or responder detects any Sequence errors on an Exchange corresponding to a Connection Setup, a Connection Setup Exchange Error shall be detected.

b) If a Connect Request originator does not receive a FCVI_CONNECT_RESP3 within FCVI_ULP_TIMEOUT from transmitting a FCVI_CONNECT_RQST within the same Exchange, a Connection Setup Timeout Error shall be detected.

c) If a Connect Request responder does not receive a FCVI_CONNECT_RESP2 within two times FCVI_ULP_TIMEOUT from transmitting a FCVI_CONNECT_RESP1 in the same Exchange, a Connection Setup Timeout Error shall be detected.

8.5.3 Connection setup error recovery

a) If a Connection Setup Exchange Error or a Connection Setup Timeout Error is detected by the Connect Request originator, it shall abort the Exchange using ABTS and discard all frames received with the corresponding FCVI_CONNECTION_ID for R_A_TOV.

   1) If the Local FC-VI Endpoint is in the Pending Connect Retry state, the Endpoint shall be transitioned to the Error state.

   2) If the Local FC-VI Endpoint is in the Pending Connect state, the Endpoint shall be transitioned to the Pending Connect Retry state and the Connection Setup shall be retried.

3) If the Local FC-VI Endpoint is in the Connect state, the Endpoint shall be transitioned to the Error state.

NOTE If a FCVI_CONNECT_RESP2 or a a FCVI_CONNECT_RESP3 is lost, the Connect Request originator may be in the Connect state. The connection cannot be retried transparent to the VI Application at the Connect Request originator (see rule a)3) in 8.5.3).

b) An FCVI_CONNECTION_ID for a failed FC-VI Connection Setup shall not be reused by a Connect Request originator unless a Disconnect has been successfully completed (i.e., a Disconnect response has been sent or received) and R_A_TOV has elapsed since the last use of the FCVI_CONNECTION_ID in a Connect Request Message.

c) If an implicit or explicit Port logout is sent or received from an FC-VI Port, an FC-VI Provider may reuse any FCVI_CONNECTION_ID associated with the logged out FC-VI Port.

d) If a Connection Setup Exchange Error or a Connection Setup Timeout Error is detected by the Connect Request responder, the Local FC-VI Endpoint shall be transitioned to the Error state.

8.5.4 Connection setup originator retry rules

The Connect Request originator shall perform a single retry for a failed Connection Setup request under the conditions defined in this subclause. If the retry fails, the Local FC-VI Endpoint is transitioned to the Error state. The following rules support Connect Request retries.

a) A Connect Request originator shall retry a previous Connection Setup if the Local FC-VI Endpoint is in the Pending Connect state by issuing a new FCVI_CONNECT_RQST on a new Exchange with the original FCVI_HANDLE, a new FCVI_CONNECTION_ID, and the RETRY bit set to one in the FCVI_FLAGS field. The RETRY bit must be set in all the IUs sent for this FCVI_CONNECTION_ID. The Endpoint state shall be transitioned to the Pending Connect Retry state.

b) A Connect Request responder that receives a FCVI_CONNECT_RQST or a FCVI_CONNECT_RESP2 with the RETRY bit set shall respond in accordance with the protocol in 5.3, except that the RETRY bit shall be set in all correspondingly transmitted IUs.

8.6 Disconnect operation error detection and recovery rules

8.6.1 Disconnect operation error handling overview

The following rules support error detection and recovery for the FC-VI Disconnect operation. See 5.3.5 for the FC-VI Disconnect operation protocol description. Detected errors include a Disconnect Operation Exchange Error and a Disconnect Operation Timeout Error.

8.6.2 Disconnect operation error detection

a) If a Disconnect Request originator or responder detects any Sequence errors on an Exchange corresponding to a Disconnection operation, a Disconnect Operation Exchange Error shall be detected.

b) If a Disconnect Request originator does not receive a FCVI_DISCONNECT_RESP for a FCVI_DISCONNECT_RQST in the same Exchange within FCVI_ULP_TIMEOUT a Disconnect Operation Timeout Error shall be detected.

8.6.3 Disconnect operation error recovery rules

a) If a Disconnect Operation Exchange or Timeout Error is detected by a Disconnect Request originator, it shall abort the Exchange as specified in FC-FS, the FC-VI Endpoint shall be transitioned to the Error state and complete the VipDisconnect with a return code of VIP_NOT_REACHABLE.

b) If a Disconnect Operation Exchange Error is detected by a Disconnect Request responder, the Disconnect Request responder shall ignore the Disconnect Request.

Annex A
(normative)

Concurrent matching peer requests example

A.1 Overview

This annex shows Peer-to-Peer Connection Setup examples for Concurrent Matching Peer-to-Peer Requests. A Concurrent Matching Peer-to-Peer Request occurs when a Peer has issued a FCVI_CONNECT_RQST IU and receives a FCVI_CONNECT_RQST IU from another Peer and the Discriminators and IP addresses in the two Connect Requests match (the Peers are attempting to connect with each other). To prevent both Peers from completing the connection, the Peer with the lower Port_Name completes the connection while the Peer with the higher Port_Name has its Connect Request rejected.

Table A.1 is a version of Table 2 annotated with a Case # column. An example is used to illustrate each row or Case in the table. The examples assume an Out-of-Order Fabric to illustrate that the FC-VI Connection Setup protocol operates correctly for Out-of-Order Fabrics.

Table A.1 – Peer B actions based on connect responses from peer A

Case #	Response from Peer A (higher Port_Name)	Peer B (lower Port_Name) Replies With:
1	Connect Accept	Connect Reject - Concurrent Matching Peer Requests
2	No Discriminator Match	Connect Accept
3	No Waiting Remote Connectionpoint	Connect Accept
4	Connect Reject - Concurrent Matching Peer Requests	Connect Reject - Protocol Error
5	Connect Reject - Transport Error	Connect Reject
6	Connect Reject - Protocol Error	Connect Reject
7	Any other response	Connect Reject - Protocol Error

For all examples shown in this annex, Peer B on the left of the figure has a lower Port_Name than Peer A on the right side of the figure. All examples eliminate FCVI_CONNECT_RESP2 and FCVI_CONNECT_RESP3 for clarity.

A.2 Case 1

Figure A.1 shows an example of Case 1 in Table A.1.

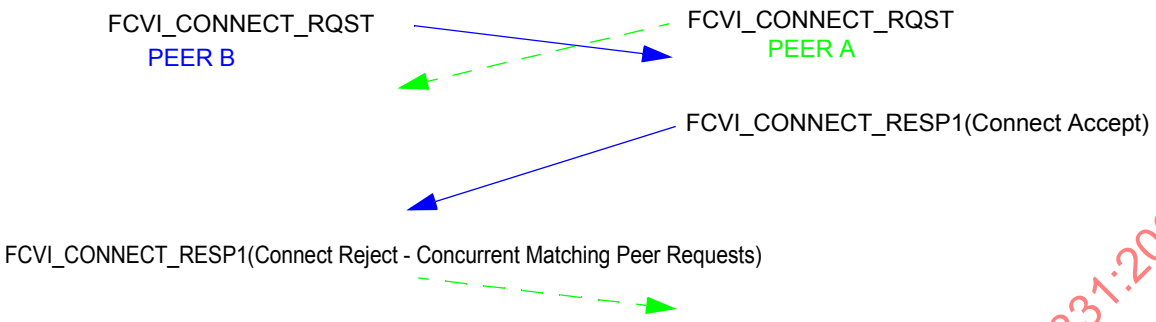


Figure A.1 – Case 1

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of "Connect Accept" indicates Peer A has accepted Peer B's Connect Request, since Peer A's Port_Name is greater than Peer B's Port_Name.

Peer B then rejects Peer A's Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS Reason Code of "Connect Reject - Concurrent Matching Peer Requests". Peer B successfully completes the connection to Peer A.

A.3 Case 2

Figure A.2 shows an example of Case 2 in Table A.1

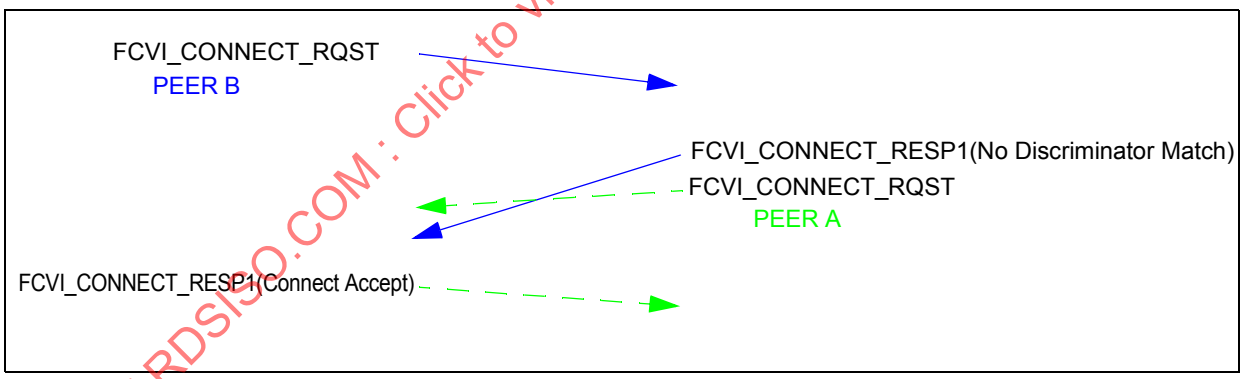


Figure A.2 – Case 2

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of "No Discriminator Match" signifies that Peer A's FCVI_CONNECT_RESP1 was transmitted before Peer A's FCVI_CONNECT_RQST was transmitted, but they were reordered in the Fabric. There is a Peer waiting, but on a different Connectionpoint set up from another FCVI_CONNECT_RQST. Peer A's reordered FCVI_CONNECT_RQST is for the matching Connectionpoint.

Peer B accepts Peer A's Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS set to zero, which signifies a "Connect Accept". Peer A successfully completes the connection to Peer B. Otherwise, if Peer B rejects Peer A, neither side completes the Connection Setup.

A.4 Case 3

Figure A.3 shows an example of Case 3 in Table A.1.

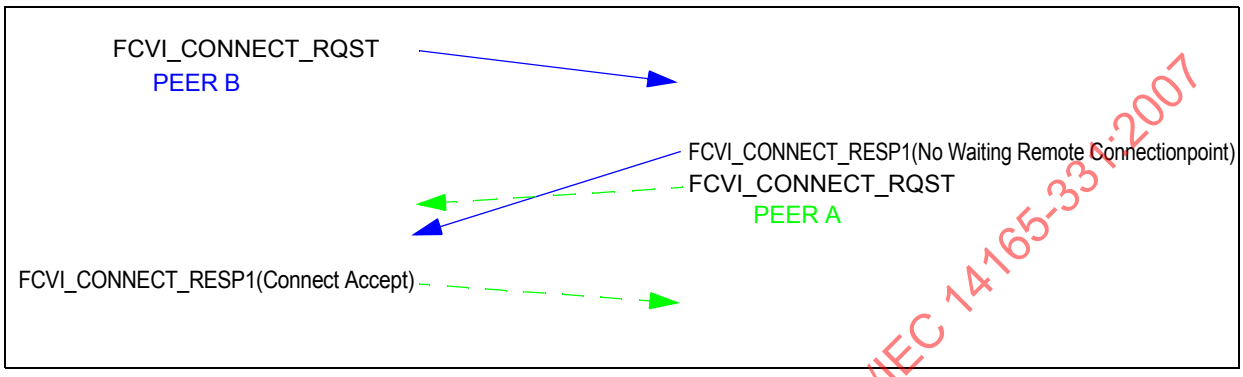


Figure A.3 – Case 3

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of "No Waiting Remote Connectionpoint" signifies that Peer A's FCVI_CONNECT_RESP1 was transmitted before Peer A's FCVI_CONNECT_RQST was transmitted, but they were reordered in the Fabric. Peer A attempts a matching connection with Peer B just after it transmitted the "No Waiting Remote Connectionpoint".
Peer B accepts Peer A's Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS set to zero, which signifies a "Connect Accept". Peer A successfully completes the connection to Peer B. Otherwise, if Peer B rejects Peer A, neither side completes the Connection Setup.

Case 3 is very similar to Case 2.

A.5 Case 4

Figure A.4 shows an example of Case 4 in Table A.1.

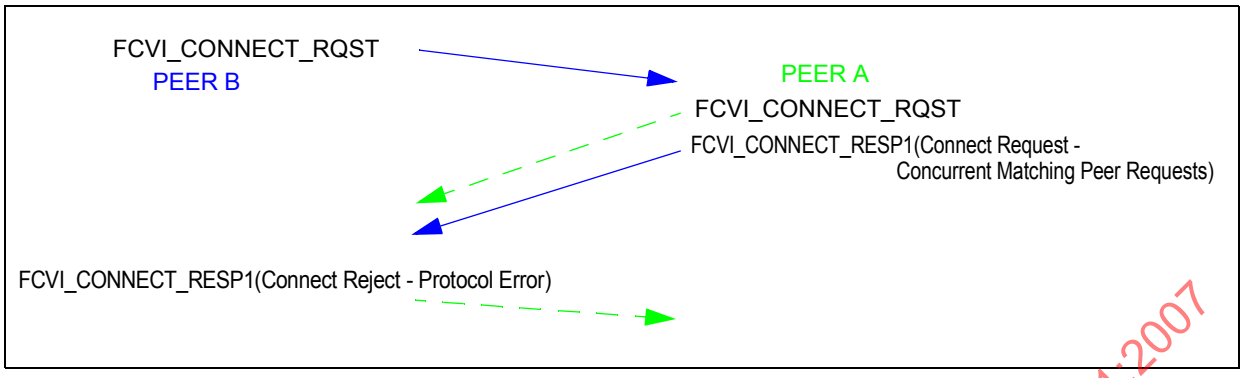


Figure A.4 – Case 4

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of "Connect Reject - Concurrent Matching Peer Requests" signifies that Peer A believes it has the lower Port_Name, which is clearly a protocol error.

Peer B rejects Peer A's Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS Reason Code "Connect Reject". Neither side completes the Connection Setup.

A.6 Case 5

Figure A.5 shows an example of Case 5 in Table A.1.

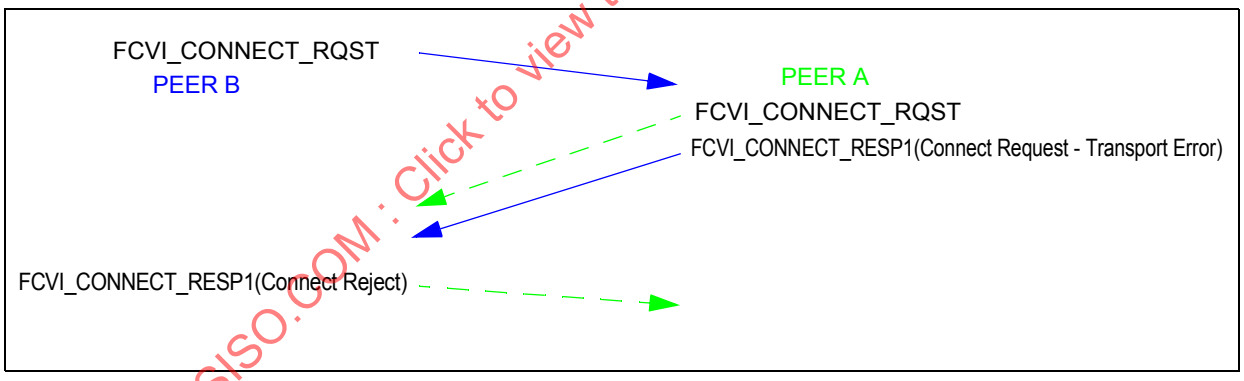


Figure A.5 – Case 5

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of "Connect Reject - Transport Error" signifies that Peer A has detected a transport error with the Connection Setup.

Peer B rejects Peer A's Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS Reason Code "Connect Reject". Neither side completes the Connection Setup.

A.7 Case 6

Figure A.6 shows an example of Case 6 in Table A.1.

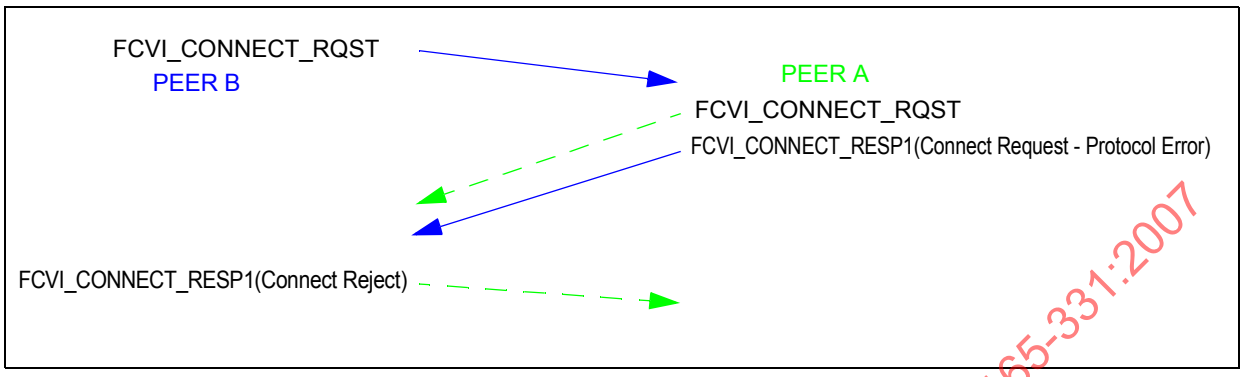


Figure A.6 – Case 6

Peer B receives a matching FCVI_CONNECT_RQST before it receives a FCVI_CONNECT_RESP1 to its own FCVI_CONNECT_RQST. Peer B waits for the FCVI_CONNECT_RESP1 from Peer A. The response of “Connect Reject - Protocol Error” signifies that Peer A is rejecting Peer B’s Connection Setup request due to a protocol error.

Peer B rejects Peer A’s Connect Request by transmitting a FCVI_CONNECT_RESP1 with a CONN_STS Reason Code “Connect Reject”. Neither side completes the Connection Setup.

Annex B
(informative)

FC-VI message transfer error handling examples

B.1 Overview

This Annex shows an example of an implementation for FC-VI error detection and recovery. R_A_TOV is used to time the arrival of missing frames or Messages. Other implementations are possible.

B.2 Message transfer error handling operation

B.2.1 Message transfer error handling operation overview

The following rules support error detection and recovery for Sends, RDMA Writes and RDMA Reads. This set of rules supports acknowledged (class 2) and unacknowledged (class 3) service, as well as In-Order Fabrics and Out-of-Order Fabrics.

B.2.2 Message transfer error definitions

Define “M” as a Message identified by the FCVI_MSG_ID component of a Fully Qualified Message ID (FQMID). For a Message responder, Message M is open if at least one Message payload frame has been received and if any associated Descriptor has not been completed. In addition, Message M is considered open at the responder if no frames have arrived but M is less than Mrcv (see definition for Mrcv below). For a Message originator, Message M is open if at least one Message frame has been sent and if any associated Descriptor has not been completed.

Define "Mexp" as the next expected FCVI_MSG_ID (modulo FCVI_MSG_ID) for a FQMID. There is only one value of Mexp per FCVI_HANDLE.

Define "Mrcv" as the FCVI_MSG_ID of the last received frame for a FQMID.

Define "SCexp" as the next expected SEQ_CNT value for each open Message M for a FQMID. For a new Message for which no frames have arrived, SCexp = 0. SCexp is incremented (modulo SEQ_CNT) after each frame is processed. For an In-Order Fabric, there is only one value for SCexp per FCVI_HANDLE, since only one Message may be open per VI at the Message responder. For an Out-of-Order Fabric, there may be many open Messages at the Message responder, where each open Message has its own value for SCexp.

Define "SCrcv" as the SEQ_CNT of the last received frame for a FQMID.

Define “Merr” as a Message assembly error for Message M identified by a FCVI_MSG_ID for a FQMID. Merr is set only for Unreliable Delivery.

Define “Herr” as a connection error for FCVI_HANDLE H for a FQMID. An Herr is set for FCVI_HANDLE H when an error is reported for Reliable Delivery or Reliable Reception. Certain catastrophic errors, such as a Protocol Error, can cause Herr to be set for Unreliable Delivery. An R_A_TOV timer is started when Herr is set.

B.2.3 Error Detection and Recovery Rule Processing

Rules 1 through 6 are processed whenever a Device_Data frame is received. Rules 7 and 8e are processed when a timeout occurs, which is asynchronous to frame reception. If no rule evaluates to true (i.e., is processed), the received frame is processed normally.

B.2.4 Message responder and message originator error recovery actions

Rule 1 is processed by both a Message Originator and a Message Responder for Device_Data frames when an unrecoverable error has been detected and a Device_Data frame is received. Rule 2 may also be processed if Rule 1 is not in effect (i.e., Herr is not set for this FCVI-HANDLE). Rules 1 and 2 are used to discard either all IUs for a connection (rule 1) or to discard all IUs for a particular Message (rule 2). Rule 3 is only processed when Merr is first set for a FCVI_MESSAGE_ID. Any mechanism used to recover from errors are beyond the scope of the error detection and recovery method illustrated in this clause.

A Message Originator uses rules 1 and 2 to process received Message Response frames (FCVI_SEND_RESP, FCVI_WRITE_RESP, FCVI_READ_RESP). A Message Responder uses rules 1 and 2 to process received Message Request frames (FCVI_SEND_RQST, FCVI_WRITE_RQST, FCVI_READ_RQST).

If rule 1 is in effect, Herr can only be cleared after R_A_TOV if the Connection has been closed with an acknowledged Disconnect (either a Disconnect response has been sent or received). Otherwise, Herr will remain indefinitely set for the FCVI_HANDLE (until logout or node reset).

Rule 1: If Herr is set, then

a) Discard any received frames for FCVI_HANDLE H. If class 2 and the Message Responder, then transmit an ACK with the Abort Sequence Condition bits set to 01b (Abort Sequence, Perform ABTS) in F_CTL.

b) If a Disconnect has been successfully completed for this connection (i.e., a Disconnect response has been received) and R_A_TOV has elapsed, clear Herr.

c) Else if a LOGO, either implicit or explicit, has been issued to or received from the Remote FC-VI Port, clear Herr.

Rule 2: If Merr is set, then

a) Discard any received frames for Message M. If class 2 and the Message Responder, then transmit an ACK with the Abort Sequence Condition bits set to 01b (Abort Sequence, Perform ABTS) in F_CTL.

b) If a LOGO, either implicit or explicit, has been issued to or received from the Remote FC-VI Port, clear Merr

c) Else clear Merr after R_A_TOV

Rule 3 is processed only by the Message Responder when Merr is first set.

Rule 3a: If any frames for M have been received prior to setting Merr for a Send or a RDMA Write with Immediate Data, then complete the Descriptor for M in error.

Rule 3b: If no Descriptor is completed in error for Merr, an FC-VI Provider may optionally log the error in an implementation dependent manner.

B.2.5 Message responder error detection actions

Rules 4 through 7 are used to detect Message errors at the Message Responder. No ordering is implied by the rule numbering. Rules 4, 5 and 6 are designed to maintain Message ordering within a VI connection by tracking the expected Message ID (Mexp) for the next Device_Data frame. Rule 4 (Mrcv > Mexp), rule 5 (Mrcv < Mexp), and rule 6 (Mrcv = Mexp) are mutually exclusive - only one rule will be processed.

Some implementations that compare frame header fields (S_ID, SEQ_ID, SEQ_CNT, Relative Offset) to determine if the received frame is the next expected frame within the current Sequence being assembled may use this frame header comparison method as an alternative to rule 6 (Mrcv = Mexp).

Rule 4a: If Mrcv > Mexp, if In-Order Fabric, and if the Reliability Level is Unreliable Delivery, then

a) for each Message M, Mexp <= M < Mrcv, set Merr.

b) If SCrcv != 0, then

   1) set Merr for Message Mrcv

   2) set Mexp = Mrcv + 1

   3) set SCexp = 0.

c) If SCrcv = 0, then

   1) set Mexp = Mrcv

   2) set SCexp = SCrcv + 1

   3) process the received frame.

Rule 4b: If Mrcv > Mexp, if In-Order Fabric, and if the Reliability Level is Reliable Delivery or Reliable Reception, then

a) transition the VI to the Error state

b) set Herr

Rule 4c: If Mrcv > Mexp and if Out-of-Order Fabric, then

a) for each Message M, Mexp < M < Mrcv, start an R_A_TOV timer for all potentially missing frames for M,

b) for Message Mexp, start an R_A_TOV timer for all frames greater than or equal to SCexp,

c) for Message Mrcv, start an R_A_TOV timer for all frames less than SCrcv,

d) set SCexp = SC + 1,

e) set Mexp = Mrcv.

Rule 5a: If Mrcv < Mexp, if In-Order Fabric, and if Merr is not set, then

a) transition the VI to the Error state,

b) set Herr (this is a Protocol Error).

NOTE Since the Merr rule (Rule 2) has precedence, a frame where Mrcv is less than Mexp should not occur for In-Order Fabric, i.e., a missing frame or Message was not detected.

Rule 5b: If Mrcv < Mexp and if Out-of-Order Fabric, process the frame.

Rule 6a: If Mrcv = Mexp, a Sequence error is detected and the Reliability Level is Unreliable Delivery, then

a) set Merr for Message Mexp and process Rule 7,

b) set Mexp = Mrcv + 1,

c) set SCexp = 0.

Rule 6b: If Mrcv = Mexp, a Sequence error is detected and the Reliability Level is Reliable Delivery or Reliable Reception, then

a) transition the VI to the Error state,

b) set Herr (this is a Protocol Error).

Rule 6c: If Mrcv = Mexp, if In-Order Fabric, if SCrcv != SCexp and SCexp != 0, and if the Reliability Level is Unreliable Delivery, then

a) set Merr for Message Mexp,

b) set Mexp = Mrcv + 1,

c) set SCexp = 0.

Rule 6d: If Mrcv = Mexp, if In-Order Fabric, if SCrcv != SCexp, and if the Reliability Level is Reliable Delivery or Reliable Reception, then

a) transition the VI to the Error state,

b) set Herr.

Rule 6e: If Mrcv = Mexp, if Out-of-Order Fabric, and if SCrcv > SCexp, then

a) start an R_A_TOV timer for all potentially missing frames greater than or equal to SCexp and less than SCrcv in Message Mexp,

b) process the received frame,

c) set SCexp = SCrcv + 1.

Rule 7a: If an R_A_TOV timeout occurs for Message M and the Reliability Level is Unreliable Delivery, then

a) set Merr for Message Mexp.

Rule 7b: If an R_A_TOV timeout occurs for Message M and the Reliability Level is Reliable Delivery or Reliable Reception, then

a) transition the VI to the Error state,

b) set Herr.

B.2.6 Message originator Class 2 error detection actions

B.2.6.1 Message originator Class 2 error detection overview

Rules 8a to 8d are processed whenever a Link_Control frame is received at the Message Originator in class 2. No ordering is implied by the rule numbering. If no rule evaluates to true, the Link_Control frame is processed normally. Rule 8e is processed whenever an E_D_TOV timeout occurs, which may be asynchronous to Link_Control frame reception.

Rule 8a: If an F_BSY or P_BSY is received in response to a data frame and if an In-Order Fabric, then

a) abort the Exchange by transmitting an ABTS,

b) if Unreliable Delivery, set Merr,

c) else (for Reliable Delivery or Reliable Reception),

   1) transition the VI to the Error state,

   2) set Herr.

Rule 8b: If an F_BSY or P_BSY is received in response to a data frame and if Out-of-Order Fabric, retransmit the data frame as described in FC-FS.

Rule 8c: If an F_RJT or P_RJT is received in response to a data frame, then

a) abort the Exchange by transmitting an ABTS,

b) if Unreliable, set Merr,

c) else (for Reliable Delivery or Reliable Reception)

   1) transition the VI to the Error state,

   2) set Herr.

Rule 8d: If an ACK is received with the Abort Sequence Condition bits in F_CTL set to 01'b (Abort Sequence, Perform ABTS), then

a) abort the Exchange by transmitting an ABTS,

b) if Unreliable, set Merr,

c) else (for Reliable Delivery or Reliable Reception)

   1) transition the VI to the Error state,

   2) set Herr.

Rule 8e: If an E_D_TOV timeout occurs waiting for an ACK, then

a) abort the Exchange by transmitting an ABTS,

b) if Unreliable, set Merr,

c) else (for Reliable Delivery or Reliable Reception)

   1) transition the VI to the Error state,

   2) set Herr.

B.2.6.2 Message response timeout at message originator

The Message Originator waits for up to FCVI_ULP_TIMEOUT for a Message Response and then detects a Message error.

Rule 9: If a Message Response IU (FCVI_SEND_RESP, FCVI_WRITE_RESP, FCVI_READ_RESP) IU is not received within FCVI_ULP_TIMEOUT after Sequence Initiative has been passed on the last Message Request IU for this Message, then

a) abort the Exchange by transmitting an ABTS,

b) transition the VI to the Error state,

c) set Herr.

B.3 Message transfer error detection and recovery examples

B.3.1 Error examples overview

This subclause illustrates examples of Message transfer error detection and recovery for the rules illustrated in B.2.

B.3.2 Mrcv > Mexp error example

B.3.2.1 Mrcv > Mexp example description

Figure B.1 shows an example of a stream of Messages being received on one FCVI_HANDLE labeled H. The receiver last received frame SEQ_CNT 26 (SCexp-1) for Message 5 (Mexp). The next expected Message frame is SEQ_CNT 27 for Message 5.

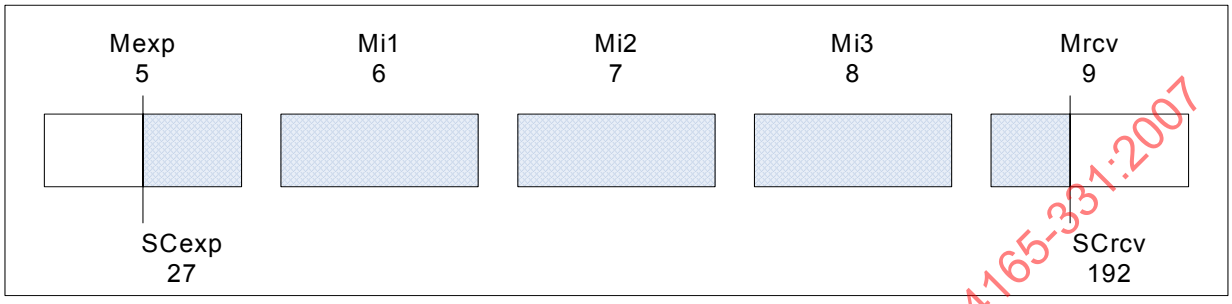


Figure B.1 – Mrcv > Mexp

When the next frame arrives, Mrcv is equal to 9, which is greater than Mexp (5), and SCrcv is equal to 192.

B.3.2.2 Mrcv > Mexp: In-order fabric and unreliable

Assume Figure B.1 an In-Order Fabric, class 3, a Reliability Level of Unreliable, and Message data is transmitted with a Send operation. From B.2.5, Rule 4a applies.

Merr is set for Messages 5, 6, 7, and 8. Since SCrcv = 192, which is not equal to zero, Merr is also set for Message 9. The received frame is discarded. Mexp is advanced to Mrcv + 1 or 10. SCexp is set to zero, since the next expected non-errored frame is frame 0 for Message 10.

Since Merr has been set, Rule 3 from B.2.5 is processed. Since frames have been received for Message 5 with a Send operation, the Descriptor for Message 5 is completed in error with a status of VIP_STATUS_TRANSPORT_ERROR. The Connection is allowed to persist, since the Reliability Level is Unreliable.

If no Descriptors were completed in error (for example, assume in the above example that Message 5 had completed and Mexp = 6, SCexp = 0), the FC-VI Provider may optionally log the error in an implementation dependent manner. There is no defined interface in the VI Architecture to signal an asynchronous error condition (VipErrorCallback) for a transport error (i.e., there is no error code of VIP_ERROR_TRANSPORT) to the VI Application.

Since Merr is set, rule 2 from B.2.5 is now in effect. Any subsequent frames that arrive for Messages 5 to 9 are discarded for a period of R_A_TOV. The receiver starts looking for frame 0 (i.e., SEQ_CNT = 0) of Message 10.

B.3.2.3 Mrcv > Mexp: In-order fabric and reliable delivery

Assume Figure B.1, an In-Order Fabric, class 3 and a Reliability Level of Reliable Delivery. The results would be the same if the Reliability Level was Reliable Reception. From B.2.5, Rule 4b applies.

The VI mapped to FCVI_HANDLE H is transitioned to the Error state. Herr is set for FCVI_HANDLE H. The received frame is discarded.

Since the VI is in the Error state, the VI Application should eventually transmit a VipDisconnect and tear down the connection. The Local FC-VI Provider may indicate the cause of the Disconnect to the Remote FC-VI Provider by setting CONN_STS to one with a Reason Code of “Transport Error” in the FCVI_DISCONNECT_RQST IU if it is the Disconnect originator or FCVI_DISCONNECT_RESP IU if it is the Disconnect responder.

Since Herr is set, rule 1 from B.2.4 now applies. All subsequent received frames for H are discarded for a period of R_A_TOV. If a logout is issued to or received from the Remote FC-VI Port, Herr is cleared and H may be reused. Otherwise, Herr is cleared after R_A_TOV and a Disconnect has been successfully completed. If a Disconnect is not successfully completed (a Disconnect response was neither sent nor received), Herr will remain set until a logout is sent or received from the Remote FC-VI Port or the Local node is reset or power cycled.

B.3.2.4 Mrcv > Mexp: Out-of-order fabric

Assume Figure B.1, an Out-of-Order Fabric, class 3, and any Reliability Level. From B.2.5, Rule 4c applies.

An R_A_TOV timer is started for all frames for Messages 6, 7 and 8. An R_A_TOV timer is started for all frames greater than or equal to SEQ_ID 27 (SCexp) for Message 5. An R_A_TOV timer is started for all frames less than SEQ_CNT 192 (SCrcv) for Message 9. SCexp is set equal to 193 (SCrcv + 1) and Mexp is set equal to 9 (Mrcv).

The series of frames from SEQ_CNT 27, Message 5 (SCexp, Mexp) to SEQ_ID 191, Message 9 (SCrcv - 1, Mrcv) creates an “epoch”, for which all frames in the epoch must arrive within R_A_TOV.

If any frame in the epoch arrives before R_A_TOV, Rule 4c applies and the received frame is processed.

If R_A_TOV expires and not all frames in the epoch have arrived, then Rule 7a applies for Unreliable Delivery and Rule 7b applies for Reliable Delivery or Reliable Reception.

If any frame in the epoch arrives after an R_A_TOV for a potentially missing frame, then Rule 1 (Herr is set) applies for Reliable Delivery and Reliable Reception and Rule 2 (Merr is set) applies for Unreliable Delivery.

B.3.3 Mrcv = Mexp error example

B.3.3.1 Mrcv = Mexp example description

Figure B.2 shows an example of frames being received for a single Message. The receiver last received frame SEQ_CNT 26 (SCexp-1) for Message 5 (Mexp) for FCVI_HANDLE H. The next expected Message frame is SEQ_CNT 27 for Message 5.

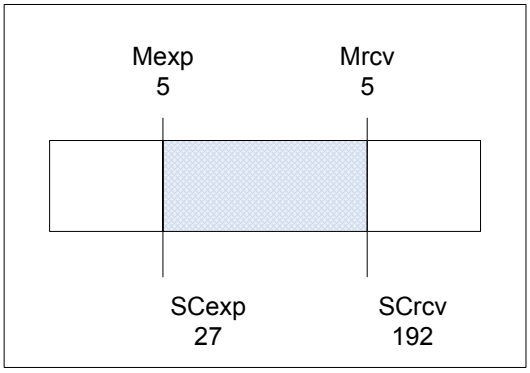


Figure B.2 – Mrcv = Mexp

The next frame to arrive is SCrcv = 192, Mrcv = 5.

B.3.3.2 Mrcv = Mexp: In-order fabric and unreliable

Assume Figure B.2, an In-Order Fabric, class 3, a Reliability Level of Unreliable, and Message data is transmitted with a Send operation. From B.2.5, Rule 6a or 6c is processed.

Merr is set for Message 5. The received frame is discarded. Mexp is set to 6 (Mexp + 1) and SCexp is set to zero, since next expected non-errored frame is frame 0 for Message 6. Rule 7a is processed and the Descriptor for Message M is completed in error.

Rule 2 in B.2.5 is now in effect for Message 5.

B.3.3.3 Mrcv = Mexp: Out-of-order fabric

Assume Figure B.B.2, an Out-of-Order Fabric, class 3, and any Reliability Level. From B.2.5, Rule 6e applies.

An R_A_TOV timer is started for all frames greater than or equal to SEQ_ID 27 (SCexp) and less than SEQ_CNT 192 (SCrcv) for Message 5. SCexp is set equal to 193 (SCrcv + 1) and Mexp is equal to 5 (Mrcv). The received frame is processed.

The series of frames from SEQ_CNT 27, Message 5 (SCexp, Mexp) to SEQ_ID 191, Message 5 (SCrev - 1, Mrcv) creates an “epoch”, for which all frames in the epoch must arrive within R_A_TOV.

If any frame in the epoch arrives before R_A_TOV, the frame is processed.

If R_A_TOV expires, Rule 7 applies. For Unreliable, Rule 7a applies and Merr is set for Message 5. Since Merr has been set, Rule 3a applies and the Descriptor for Message 5 is completed in error with a status of VIP_STATUS_TRANSPORT_ERROR. The Connection is allowed to persist, since the Reliability Level is Unreliable. Rule 2 is now in effect for Message M.

If R_A_TOV expires and the Reliability Level is Reliable Delivery or Reliable Reception, Rule 7b applies. The VI is transitioned to the Error state and Herr is set for FCVI_HANDLE H. Rule 1 is now in effect for FCVI_HANDLE H.

Annex C
(informative)

Connection setup error handling examples Overview

The following rules support error detection and recovery for FC-VI Connection Setup. This set of rules support acknowledged (class 2) and unacknowledged (class 3) service, as well as In-Order Fabrics and Out-of-Order Fabrics.

C.1 Connection setup error handling definitions

Define “CSerr” as a Connection Setup error for FCVI_CONNECTION_ID “ID”. A CSerr is set for FCVI_CONNECTION_ID ID when a Connection Setup does not successfully complete due to the following errors (see Table 12):

a) Connect Reject - Protocol Error

b) Connect Reject - Transport Error

c) Connection Setup Timeout

d) Protocol Error

e) Transport Error

C.2 Connect request originator and connect request responder rules

Rule 1: If CSerr is set, then the Local FC-VI Provider

a) discards any received frames for FCVI_CONNECTION_ID ID.

b) If a Disconnect has been successfully completed (i.e., a Disconnect response has been received) and R_A_TOV has elapsed, clears CSerr,

c) or if a LOGO, either implicit or explicit, has been issued to or received from the Remote FC-VI Port, clears CSerr.

C.3 Connect request originator rules

Note 1The Originator does not time the arrival of FCVI_CONNECT_RESP1. A lost FCVI_CONNECT_RESP1 will result in a timeout on the arrival of FCVI_CONNECT_RESP3.

Rule 2: If a FCVI_CONNECT_RESP3 is not received for a FCVI_CONNECT_RQST in the same Exchange within FCVI_ULP_TIMEOUT, and the Local FC-VI Endpoint is in the Idle state, the Local FC-VI Provider

a) transmits an ABTS for the Exchange ID used in FCVI_CONNECT_RQST IU,

b) waits for the BA_ACC to ABTS for up to R_A_TOV. If the BA_ACC does not arrive within the time-out period, the FC-VI Provider may perform second level error recovery as specified in FC-FS.

c) The FC-VI Provider may log a “Transport Error” in an implementation dependent manner.

Note 2 There is no supported interface defined in the VI Architecture to indicate a transport error associated with a VI Application timed out Connection Setup.

The Local FC-VI Endpoint may have transitioned from the Pending Connect to the Idle state for any of the following reasons.

a) The Local VI Application may have timed out waiting for a Connect Response.

b) One of the following replies from the Remote Endpoint was received in the Connect Response:

1) no waiting Remote Connectionpoint;

2) no Discriminator match:

3) connect reject.

The Local Endpoint may have transitioned to the Idle state from either the Pending Connect state or the Pending Connect retry state.

Rule 3: If a FCVI_CONNECT_RESP3 is not received for a FCVI_CONNECT_RQST in the same Exchange within FCVI_ULP_TIMEOUT, and the Local FC-VI Endpoint is in the Pending Connect state, the Local FC-VI Provider

a) transmits an ABTS for the Exchange ID used in FCVI_CONNECT_RQST IU,

b) waits for the BA_ACC to ABTS for up to R_A_TOV. If the BA_ACC does not arrive within the time-out period, the FC-VI Provider may perform second level error recovery as specified in FC-FS,

c) retries the Connection Setup by transmitting the Connect Request in a new Exchange with the same FCVI_HANDLE, a new FCVI_CONNECTION_ID and the RETRY bit set to one in the FCVI_FLAGS field in the Device_Header of the Connect Request,

d) transitions the Local FC-VI Endpoint to the Pending Connect Retry state.

Rule 4: If a FCVI_CONNECT_RESP3 is not received for a FCVI_CONNECT_RQST in the same Exchange within FCVI_ULP_TIMEOUT, and the Local FC-VI Endpoint is in the Pending Connect retry state, the Local FC-VI Provider

a) transitions the VI to the Error state,

b) aborts the Exchange ID used in FCVI_CONNECT_RQST IU as defined in FC-FS,

c) sets CSerr for the FCVI_CONNECTION_ID sent in the Device_Header of the FCVI_CONNECT_RQST IU,

d) completes any Descriptors queued on the VI with an error of “Descriptor Flushed”,

e) returns the status (return code) of VIP_NOT_REACHABLE to the VI Application that issued the VipConnectRequest or the VipConnectPeerRequest.

Rule 5: If a FCVI_CONNECT_RESP3 is not received for a FCVI_CONNECT_RQST in the same Exchange within FCVI_ULP_TIMEOUT, and the Local FC-VI Endpoint is in the Connected state, the Local FC-VI Provider

a) transitions the VI to the Error state,

b) aborts the Exchange ID used in FCVI_CONNECT_RQST IU as defined in FC-FS,

c) sets CSerr for the FCVI_CONNECTION_ID sent in the Device_Header of the FCVI_CONNECT_RQST IU,

d) completes any Descriptors queued on the VI with an error of “Descriptor Flushed”,

e) returns the status (return code) of VIP_ERROR_CONN_LOST to the VI Application that issued the VipConnectRequest or the VipConnectPeerRequest.

C.4 Connect request responder rules

Retrying a Peer-to-Peer Connection Setup is more complicated than retrying a Client-Server Connection Setup. As an example, the following situation may occur:

1) peer 1 issues a FCVI_CONNECT_RQST IU and a FCVI_CONNECT_RESP1 IU never arrives (either IU is lost),

2) peer 2 issues a Connect Request which to Peer 1 looks like a concurrent matching request,

3) peer 1 wins (higher Port Name) and accepts the Connect Request,

4) peer 1 times out, aborts the Exchange, and tears down the connection.

The problem is tearing down the connection and starting over for a retry would be visible to the VI Application at either Endpoint. Besides, the connection is already set up. The solution is to require a Peer to ignore the timeout if a valid Peer-to-Peer connection exists. Rule 6a implements the solution.

Rule 6: If a FCVI_CONNECT_RESP2 is not received for a FCVI_CONNECT_RESP1 in the same Exchange within two times FCVI_ULP_TIMEOUT, the FC-VI Provider

a) if the Connection Setup is Peer-to-Peer and the Local VI Endpoint is in the Connected state, then

1) complete the Connection Setup without terminating the connection,

2) optionally log a FCVI_ULP_TIMEOUT error;

b) else

1) transition the VI to the Error state,

2) set CSerr for the FCVI_CONNECTION_ID received in the Device_Header of the FCVI_CONNECT_RQST IU,

3) complete any Descriptors queued on the VI with an error of “Descriptor Flushed.

C.4.1 Connect request responder retry rules

Rule 8: If the FC-VI Provider receives a FCVI_CONNECT_RQST IU, the RETRY bit is set in the FCVI_FLAGS field, and a Connection Setup is open for the FCVI_HANDLE indicated in the FCVI_RQST_HANDLE in the payload of the FCVI_CONNECT_RQST, the FC-VI Provider

a) transmits the FCVI_CONNECT_RESP1, which is identical to any previously transmitted Connect Response IU for this FCVI_HANDLE except:

1) the RETRY bit in the FCVI_FLAGS field is set,

2) the Exchange ID is set to the Exchange ID of the retried FCVI_CONNECT_RQST,

3) the FCVI_CONNECTION_ID is set to the FCVI_CONNECTION_ID of the retried FCVI_CONNECT_RQST,

4) set a Retry Flag for the FCVI_HANDLE indicated in FCVI_RQST_HANDLE,

5) set CSerr for any previous FCVI_CONNECTION_ID for this FCVI_RQST_HANDLE.

Rule 9: If the FC-VI Provider receives a FCVI_CONNECT_RQST IU, the RETRY bit is set in the FCVI_FLAGS field, and the Local Endpoint is in the Connected state, the FC-VI Provider then

a) Transmits the FCVI_CONNECT_RESP1, which is identical to any previously transmitted Connect Response IU for this FCVI_HANDLE except:

1) the RETRY bit in the FCVI_FLAGS field is set,

2) the Exchange ID is set to the Exchange ID of the retried FCVI_CONNECT_RQST,

3) the FCVI_CONNECTION_ID is set to the FCVI_CONNECTION_ID of the retried FCVI_CONNECT_RQST,

4) set a Retry Flag for the FCVI_HANDLE indicated in FCVI_RQST_HANDLE.

Note 3 Since the Connection Setup completed successfully, the CONN_STS in the FCVI_FLAGS in the FCVI_CONNECT_RESP1 is set to zero (“Connect Accept”).

Rule 10: If the FC-VI Provider receives a FCVI_CONNECT_RESP2 IU, the RETRY bit is set in the FCVI_FLAGS field, and the Local Endpoint is in the Connected state, the FC-VI Provider then

a) transmits the FCVI_CONNECT_RESP3, which is identical to any previously transmitted Connect Response IU for this FCVI_HANDLE except:

1) the RETRY bit in the FCVI_FLAGS field is set,

2) the Exchange ID is set to the Exchange ID of the retried FCVI_CONNECT_RESP2,

3) the FCVI_CONNECTION_ID is set to the FCVI_CONNECTION_ID of the retried FCVI_CONNECT_RESP2,

Note 4 clear the Retry Flag for the FCVI_HANDLE indicated in FCVI_CONNECT_RESP2.

Note 5 Since the Connection Setup completed successfully, the CONN_STS in the FCVI_FLAGS in the FCVI_CONNECT_RESP1 is set to zero ("Connect Accept").

Rule 11: If the FC-VI Provider receives a FCVI_CONNECT_RQST IU, the RETRY bit is set in the FCVI_FLAGS field, a Connection Setup is not open for the FCVI_HANDLE indicated in the FCVI_RQST_HANDLE in the payload of the FCVI_CONNECT_RQST, and a Connection does not exist for the same FCVI_HANDLE, the FC-VI Provider responds as if it were a new Connect Request, with the exception that the RETRY bit in the FCVI_FLAGS field is set for all Connect Responses for this Exchange.

Rule 12: If the FC-VI Provider is within a retried Connection Setup (Retry Flag set) for a particular FCVI_HANDLE and it receives a FCVI_CONNECT_RESP2 IU for the same FCVI_HANDLE with the correct FCVI_CONNECTION ID and Exchange ID, the FC-VI Provider responds with the appropriate FCVI_CONNECT_RESP3 IU and complete the Connection Setup at the Local Connectionpoint.

C.5 Error detection and recovery examples for connection setup

C.5.1 Overview

Figure C.1 shows a typical Client-Server FC-VI Connection Setup operation. For all figures in Clause C.5, solid vertical lines indicate when VI Application calls to the VI Provider complete relative to FC-VI IU transmission or reception. Dashed vertical lines indicate the duration of timer values specified in either the VI Application call or by the FC-VI Provider.

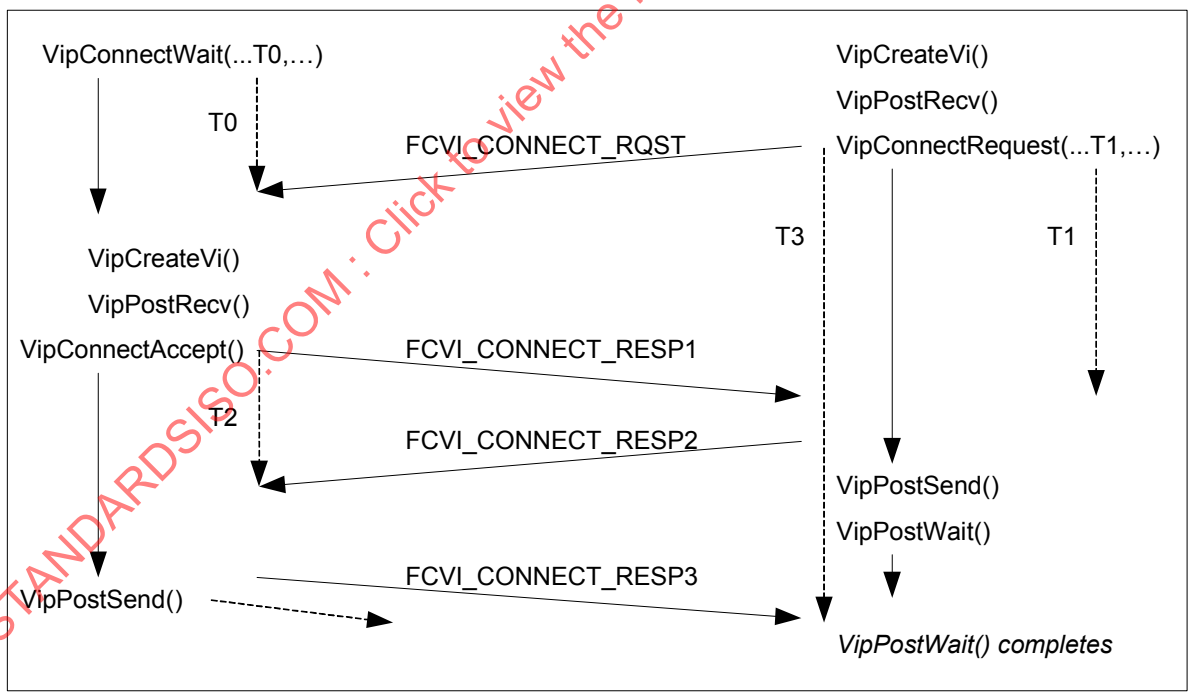


Figure C.1 – Client-server connection setup

In the example shown in Figure C.1, the Client on the right side of the figure creates a VI by issuing a VipCreateVI call to the VI Provider. The Client then posts a receive buffer with VipPostRecv and attempts to establish a connection by issuing a VipConnectRequest call. The Client specifies how long it is willing to wait for the Connection Setup to complete by setting the Timeout parameter of VipConnectRequest to "T1". VipConnectRequest completes at the Client when

a) FCVI_CONNECT_RESP2 IU is sent,

b) Timer T1 expires,

c) Connection Setup has been retried without success.

The Client may issue VipPostSend call(s) after VipConnectRequest successfully completes. In this example, the Client waits for the completion of the VipPostSend by executing a VipPostWait call.

The Client FC-VI Provider should not complete any VipSendWait (or VipSendDone) calls until a FCVI_CONNECT_RESP3 is received or T3 (see definition in next subclause) expires and the Connection Setup fails. Since the Client goes to the Connected state (when FCVI_CONNECT_RESP2 is sent) before the Server (when FCVI_CONNECT_RESP3 is sent), the motivation is to throttle the Client from sending a significant number of Messages before the Server can send a Message.

In Figure C.1, the Server on the left side of the figure issues a VipConnectWait call to the VI Provider. The Server VI Application specifies how long it is willing to wait for a matching Connect Request by setting the Timeout parameter of VipConnectWait to "T0". VipConnectWait completes at the Server when a FCVI_CONNECT_RQST IU is received with a matching Host Address and Discriminator (see Clause 6.7 in VI-ARCH) or T0 expires. In this particular example, the Server VI Application accepts the Connect Request and creates a VI with matching attributes with the VipCreateVI call, then queues a receive buffer with a VipPostRecv call. The Server then issues a VipConnectAccept call to accept the Connect Request. VipConnectAccept returns when a FCVI_CONNECT_RESP3 IU is sent or T2 (see definition in next subclause) expires. The Server may then issue VipPostSend calls if VipConnectAccept successfully completes.

A Peer-to-Peer Connection Setup is similar to the Client side of a Client-Server Connection Setup. One difference is VipConnectPeerRequest is non-blocking and returns immediately (i.e., before the FCVI_CONNECT_RESP1 is received). The VI Application uses VipConnectDone or VipConnectWait to determine when the initial connection phase completes. In the above example, VipConnectRequest would be replaced with the pair of calls VipConnectPeerRequest, VipConnectPeerWait. Another difference is that there is no ability for the Peer application to accept or reject an incoming Connect Request.

C.5.2 FC-VI connection setup timers

On the Server, "T0" is the Timeout parameter specified by a VI Application in the VipConnectWait call. If T0 expires, FCVI_CONNECT_RESP1 IU originated by the Server should not indicate a CONN_STS Reason Code of "Connect Timeout", since the Server does not need to maintain state of all VipConnectWait calls that have ever timed out. A CONN_STS Reason Code of "No Waiting Remote Connectionpoint" is the correct response if the Server times out the VipConnectWait call.

On the Server, timer "T2" is set to two times FCVI_ULP_TIMEOUT and is specified by the Server FC-VI Provider. It is set when a FCVI_CONNECT_RESP1 is sent and cleared when a FCVI_CONNECT_RESP2 IU is received. If T2 expires, the Server FC-VI Providers completes the VipConnectAccept call with a return code of VIP_TIMEOUT, transition the VI to the Error state, and gener-

ates an asynchronous error notification with an Error Code of VIP_ERROR_CONN_LOST. Note that in the VI Architecture there is no mechanism to return a “transport error” for the VipConnectAccept call. The only solution is to return VIP_TIMEOUT (which may also mean the Client stopped waiting for a reply to its Connect Request) and then generates an asynchronous error notification of VIP_ERROR_CONN_LOST.

On the Client, “T1” is the Timeout parameter specified by a VI Application in the VipConnectRequest call. If T1 expires before FCVI_CONNECT_RESP1 IU arrives, FCVI_CONNECT_RESP2 IU indicates a CONN_STS Reason Code of “Connect Timeout”. VipConnectAccept then completes with return code of VIP_TIMEOUT on the Server.

On the Client, timer “T3” is set to FCVI_ULP_TIMEOUT and is specified by the Client FC-VI Provider. It is set when FCVI_CONNECT_RQST IU is sent and cleared when FCVI_CONNECT_RESP3 IU is received. Since T1 is settable by the VI Application, its value may be greater than, less than or equal to T3. If T3 expires before T1 (the Client is still waiting for a Connect Response) and the Client is still waiting on a Connect Response (a reject or no waiting Connectionpoint was not received), the Connect Request is retried once by the FC-VI Provider. If T1 expires before T3 (the Client is no longer waiting for a Connect Response), the FC-VI Provider attempts to complete the Connection Setup handshake by waiting up to T3 seconds for the FCVI_CONNECT_RESP3 IU.

C.5.3 VipConnectRequest completion

The Client FC-VI Provider executes the following steps as a single atomic operation when completing VipConnectRequest:

1. Clear T1
2. transmit a FCVI_CONNECT_RESP2
3. complete VipConnectRequest

C.5.4 VipConnectAccept completion

The Server FC-VI Provider executes the following steps as a single atomic operation when completing VipConnectAccept:

1. Clear T2
2. transmit a FCVI_CONNECT_RESP3
3. complete VipConnectAccept

C.5.5 Enabling message transmission and reception

The Client is enabled for Message reception after a FCVI_CONNECT_RESP1 IU is received and before a FCVI_CONNECT_RESP2 IU is sent.

The Server is enabled for Message reception after a FCVI_CONNECT_RQST IU is received and before a FCVI_CONNECT_RESP1 IU is sent.

The Client is enabled for Message transmission after FCVI_CONNECT_RESP2 IU is sent and VipConnectRequest completes. Note that Message data may be sent by the Client before the FCVI_CONNECT_RESP3 IU is received. The Client FC-VI Provider does not complete any Descriptors until the Local Endpoint has transitioned to the Connected, Idle or Error states.

If T3 expires when the Local Endpoint is in the Pending Connect retry state, all posted Descriptors are completed in error. If T1 expires, the Local Endpoint transitions to the Idle state and all posted Descriptors are completed with an error of “Descriptor Flushed”.

The Server is enabled for Message transmission after a FCVI_CONNECT_RESP3 IU is sent and VipConnectAccept completes.

C.5.6 Client timeout of VipConnectRequest

Figure C.2 illustrates the Client timing out the Connect Request before it completes (T1 < T3). In this case, the FCVI_CONNECT_RESP2 IU indicates a CONN_STS with a Reason Code of “Connect Timeout”. The VipConnectAccept call on the Server completes with a return code of VIP_TIMEOUT. Neither Client or Server see a completed connection and the Local Endpoint transitions to the Idle state.

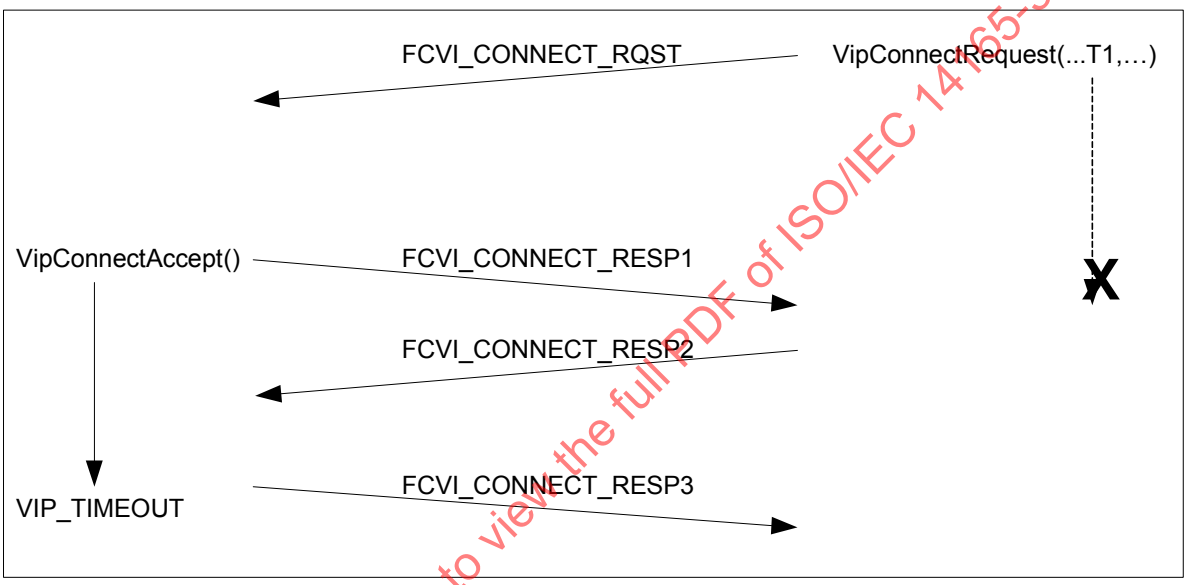


Figure C.2 – Client timeout of VipConnectRequest

C.5.7 Lost FCVI_CONNECT_RQST IU

C.5.7.1 Lost FCVI_CONNECT_RQST IU example

Figure C.3 shows an example of a lost FCVI_CONNECT_RQST IU.

The Client FC-VI Provider transmits a FCVI_CONNECT_RQST IU as a result of the Client VI Application issuing a VipConnectRequest call to the VI Provider.

Assume that T1 (VIP_TIMEOUT parameter in VipConnectRequest) < T3 (FCVI_ULP_TIMEOUT). The FCVI_CONNECT_RQST IU is lost and the Client VI Application times out waiting for the Connect Request to complete. VipConnectRequest completes with a return code of VIP_TIMEOUT. The Client FC-VI Provider transitions the Local Endpoint from the Connect Pending to the Idle state.

The Client FC-VI Provider waits for T3 seconds for the FCVI_CONNECT_RESP3 IU to arrive and times out. The Client FC-VI Provider then transmits an ABTS to recover the Exchange ID used for the Connection Setup. The Server FC-VI Provider has no Exchange or Connection Setup context, since it never received the FCVI_CONNECT_RQST IU. The Server FC-VI Provider replies by transmitting a BA_ACC (see “Special case - new Exchange” in 23.6.1.1 of FC-FS). The Client FC-VI Provider may optionally log a transport error.

The Client FC-VI Provider will not attempt a retry, since the Local VI Application is responsible for issuing a new Connect Request.

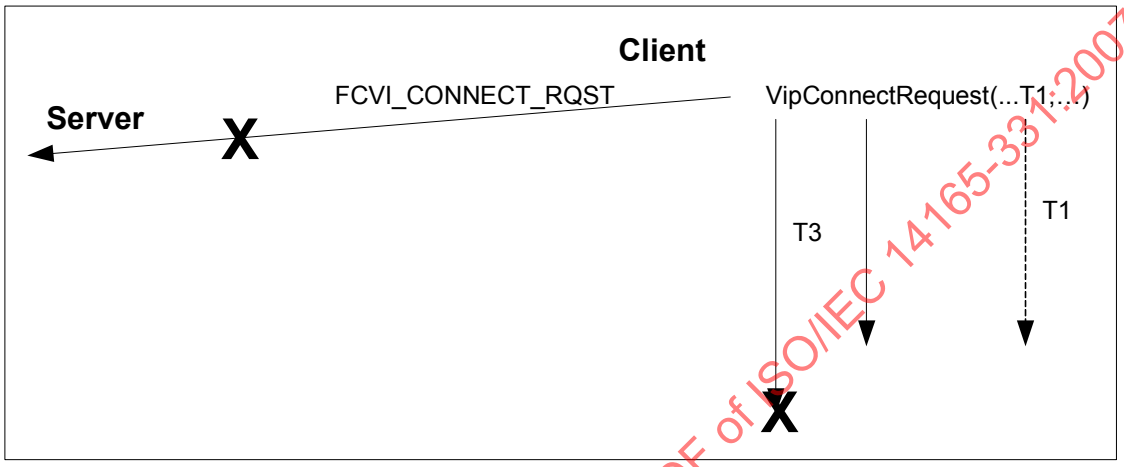


Figure C.3 – Lost FCVI_CONNECT_RQST IU

C.5.7.2 Retried connection setup

If T1 has not expired and T3 expires, the Client FC-VI Provider retries the Connection Setup and retransmits the FCVI_CONNECT_RQST IU. The RETRY bit is set to one in all Connection Setup IUs.

If the retried Connection Setup fails (T3 expires waiting for the retried FCVI_CONNECT_RESP3) and T1 has not expired, the Client FC-VI Provider transitions the Local Endpoint from the Connect Pending retry state to the Error state, clears the T1 timer and complete the VipConnectRequest with a return code of VIP_NOT_REACHABLE. The Client FC-VI Provider transmits an ABTS to recover the Exchange ID used for the retried Connection Setup.

If the Client VI Application issues a VipDisconnect call to the VI Provider, the Client FC-VI Provider transmits a FCVI_DISCONNECT_RQST IU with the CONN_SETUP_ABORT flag set, the VIP_APP_DISCON flag set and the CONN_STS set in the FCVI_FLAGS field.

NOTE If the VI Application never issues a Disconnect, the VI remains in the Error state until the VI Application terminates or a VipCloseNIC call is issued.

The Reason Code for the error is set to “Connection Setup Timeout”. The FCVI_HANDLE in the Device_Header of the FCVI_DISCONNECT_RQST IU is set equal to the FCVI_RQST_HANDLE sent in the FCVI_CONNECT_RQST IU. The FCVI_CONNECTION_ID in the Device_Header is set equal to the

FCVI_CONNECTION_ID in the Device_Header of the FCVI_CONNECT_RQST IU of the retried Connection Setup.

The Server FC-VI Provider responds by transmitting a FCVI_DISCONNECT_RESP IU with the CONN_SETUP_ABORT and VIP_APP_DISCON flags set equal to those in the FCVI_CONNECT_RQST IU. The CONN_STS set in the FCVI_FLAGS field. The Reason Code for the error is set to “Connection Does Not Exist”. The FCVI_HANDLE is set to FFFFFFFFh. The FCVI_CONNECTION_ID in the Device_Header of the FCVI_DISCONNECT_RESP IU is set equal to the FCVI_CONNECTION_ID in the Device_Header of the FCVI_DISCONNECT_RQST IU received on the same Exchange.

If the FCVI_DISCONNECT_RESP IU is received within FCVI_ULP_TIMEOUT seconds, the VipDisconnect call completes with a return code of VIP_SUCCESS. If the FCVI_DISCONNECT_RESP IU is not received within FCVI_ULP_TIMEOUT seconds, the Client FC-VI Provider may retry the Disconnect. If the retried FCVI_DISCONNECT_RESP IU is not received, the VipDisconnect call completes with a return code of VIP_NOT_REACHABLE and the Local Endpoint is not transitioned from the Error state.

Whenever a FCVI_ULP_TIMEOUT occurs waiting for a FCVI_DISCONNECT_RESP IU, the Client FC-VI Provider recovers the Exchange ID by transmitting an ABTS.

C.5.8 Lost FCVI_CONNECT_RESP1 IU

Figure C.4 shows an example of a lost FCVI_CONNECT_RESP1 IU.

The Client FC-VI Provider transmits a FCVI_CONNECT_RQST IU as a result of the Client VI Application issuing a VipConnectRequest call to the VI Provider. Assume that T1 (VIP_TIMEOUT parameter in VipConnectRequest) < T3 (FCVI_ULP_TIMEOUT). The Server FC-VI Provider responds by issuing a FCVI_CONNECT_RESP1 IU.

The FCVI_CONNECT_RESP1 IU is lost and the Client VI Application times out waiting T1 seconds for the Connect Request to complete. VipConnectRequest completes with a return code of VIP_TIMEOUT. The Client FC-VI Provider transitions the Local Endpoint from the Connect Pending to the Idle state.

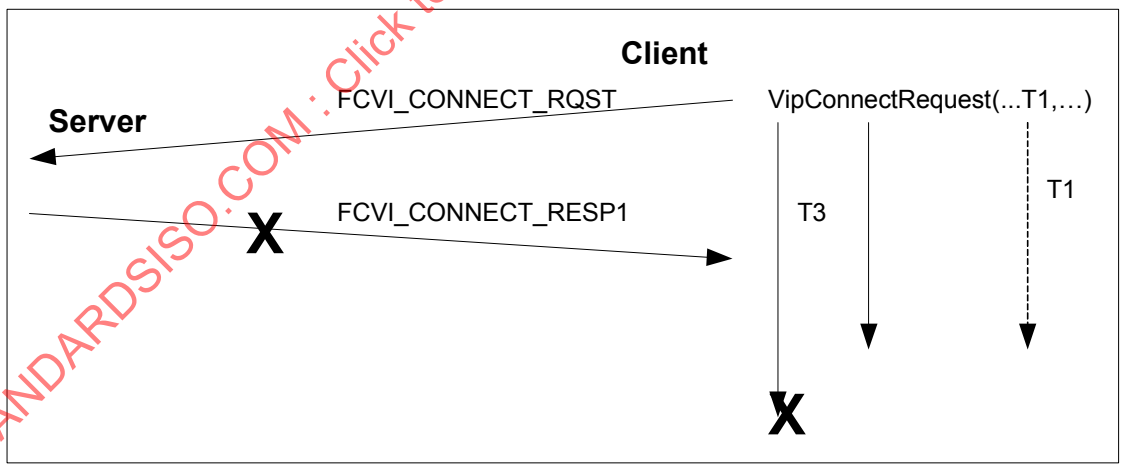


Figure C.4 – Lost FCVI_CONNECT_RESP1 IU

Since the Client cannot tell the difference between a lost FCVI_CONNECT_RQST and a lost FCVI_CONNECT_RESP1, the behavior is identical to the lost FCVI_CONNECT_RQST example described in **the previous section**, with the exception that the Server FC-VI Provider has Connection Setup context and will reply to a Disconnect request with a Reason Code other than “Connection Does Not Exist”.

C.5.9 Lost FCVI_CONNECT_RESP2 IU

C.5.9.1 Lost FCVI_CONNECT_RESP2 IU example

Figure C.5 shows an example of a lost FCVI_CONNECT_RESP2 IU.

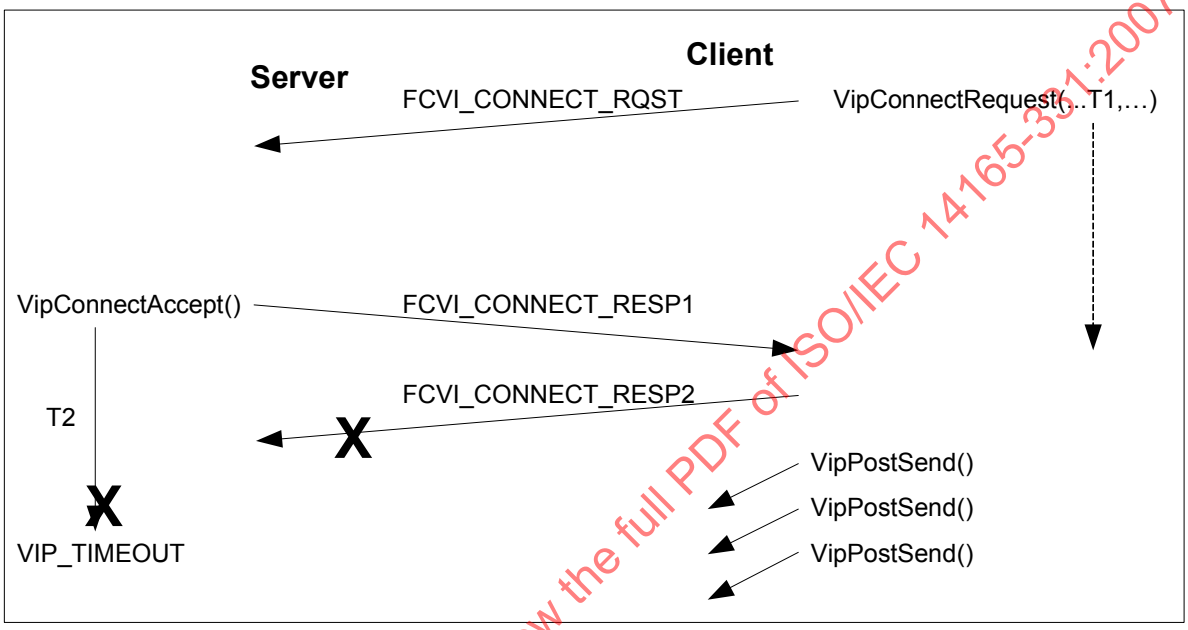


Figure C.5 – Lost FCVI_CONNECT_RESP2

The Client FC-VI Provider transmits the FCVI_CONNECT_RESP2 and completes VipConnectRequest.

Assume the FCVI_CONNECT_RESP1 indicates a “Connect Accept”. VipConnectRequest returns with a Return Code of VIP_SUCCESS. The Client Endpoint transitions to the Connected state. The Client VI Application may then transmit Messages by issuing VipPostSend calls.

The Client FC-VI Provider waits for T3 seconds for the FCVI_CONNECT_RESP3 IU to arrive and times out. The Client FC-VI Provider then transmits an ABTS to recover the Exchange ID used for the Connection Setup. The Client Endpoint is transitioned to the Error state. An asynchronous error notification of VIP_ERROR_CONN_LOST is generated by the Client FC-VI Provider. A Herr is set for the Local FCVI_HANDLE and frames are discarded for up to R_A_TOV.

If the Client VI Application issues a VipDisconnect call to the VI Provider, the Client FC-VI Provider transmits a FCVI_DISCONNECT_RQST IU with the CONN_SETUP_ABORT flag clear, the VIP_APP_DISCON flag set and the CONN_STS set in the FCVI_FLAGS field. The Reason Code for the error is set to “Transport Error”. The FCVI_HANDLE in the Device_Header of the FCVI_DISCONNECT_RQST IU is set equal to the FCVI_HANDLE for the connection. The FCVI_CONNECTION_ID in the Device_Header is set equal to FFFFFFFFh.

The Server FC-VI Provider responds by transmitting a FCVI_DISCONNECT_RESP IU with the CONN_SETUP_ABORT and VIP_APP_DISCON flags set equal to those received in the FCVI_DISCONNECT_RQST IU. The FCVI_CONNECTION_ID in the Device_Header of the FCVI_DISCONNECT_RESP IU is set equal to the FCVI_CONNECTION_ID in the Device_Header of the FCVI_DISCONNECT_RQST IU received on the same Exchange. A Herr is set for the Local FCVI_HANDLE sent as the FCVI_RESP_HANDLE and frames are discarded for up to R_A_TOV.

If the FCVI_DISCONNECT_RESP IU is received by the Client within FCVI_ULP_TIMEOUT seconds, the VipDisconnect call completes with a return code of VIP_SUCCESS. If the FCVI_DISCONNECT_RESP IU is not received within FCVI_ULP_TIMEOUT seconds, the Client FC-VI Provider may retry the Disconnect. If the retried FCVI_DISCONNECT_RESP IU is not received, the VipDisconnect call completes with a return code of VIP_NOT_REACHABLE and the state of the Client Endpoint is not changed from the Error state. Whenever a FCVI_ULP_TIMEOUT occurs waiting for a FCVI_DISCONNECT_RESP IU, the Client FC-VI Provider recovers the Exchange ID by transmitting an ABTS.

C.5.9.2 Server timing out connection setup

If FCVI_DISCONNECT_RQST IU arrives before T2 expires at the Server, T2 is cleared and the Server Endpoint is transitioned to the Error state. Since the Client was in the Connected state, the Disconnect request will indicate that a connection is being aborted (CONN_SETUP_ABORT flag clear). A Herr is set for the Local FCVI_HANDLE sent as the FCVI_RESP_HANDLE and frames are discarded for up to R_A_TOV.

If T2 expires before FCVI_DISCONNECT_RQST IU arrives, the Server Endpoint will be transitioned to the Error state and CSerr will be set for the FCVI_CONNECTION_ID received in the FCVI_CONNECT_RQST IU. Frames will be discarded for FCVI_CONNECTION_ID for up to R_A_TOV.

The Server VI Application may issue a VipDisconnect, possibly before the Client VI Application. Each Endpoint must issue a VipDisconnect to transition the Local Endpoint from the Error state to the Idle state.

C.5.10 Lost FCVI_CONNECT_RESP3 IU

C.5.10.1 Lost FCVI_CONNECT_RESP3 IU example

Figure C.6 shows an example of a lost FCVI_CONNECT_RESP3 IU.

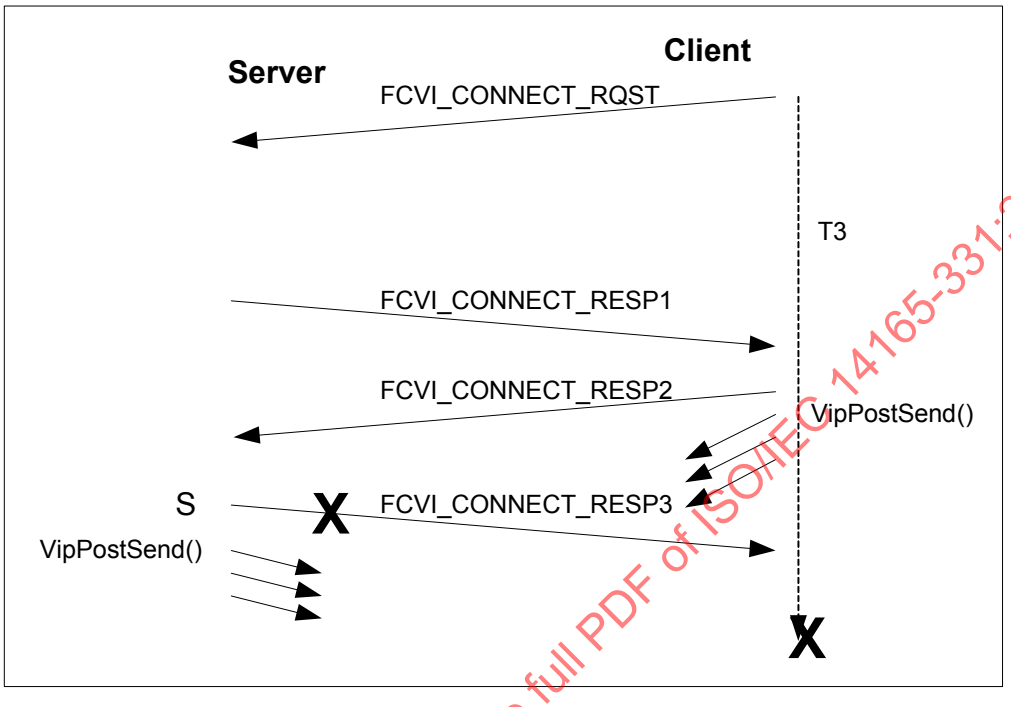


Figure C.6 – Lost FCVI_CONNECT_RESP3 IU

The Server FC-VI Provider transmits a FCVI_CONNECT_RESP3 IU and the VipConnectAccept completes with a return code of VIP_SUCCESS. The Server VI Application may then issue one or more VipPostSend calls to transfer Messages to the Client. Since the FCVI_CONNECT_RESP3 IU has been sent, the Server FC-VI Provider may successfully complete Send Descriptors for Unreliable Delivery or Reliable Delivery. The Client FC-VI Provider does not issue any Message Response IUs until Connection Setup has successfully completed (FCVI_CONNECT_RESP3 arrives).

The Client may transmit Message data as soon as the FCVI_CONNECT_RESP2 IU is sent. However, the Descriptors may not be completed until the FCVI_CONNECT_RESP3 IU arrives. The Client may receive Message Response IUs from the Server.

When T3 expires at the Client, the Client Endpoint will be transitioned to the Error state and an Herr will be set for the Client FCVI_HANDLE. When the Client VI Application issues a VipDisconnect call, the receipt of a FCVI_DISCONNECT_RQST IU will transition the Server Endpoint from the Connected to the Error state. The Server VI Application must then issue its own VipDisconnect to transition the Server Endpoint from the Error state to the Idle state.

Annex D
(informative)

Disconnect operation error handling examples

D.1 Disconnect operation example description

The following rules support error detection and recovery for FC-VI Disconnect operation. This set of rules support acknowledged (class 2) and unacknowledged (class 3) service, as well as In-Order Fabrics and Out-of-Order Fabrics.

Rule 1: If a VI Application issues a VipDisconnect call and an established connection is being aborted, the FC-VI Provider

a) transmits a FCVI_DISCONNECT_RQST IU with FCVI_APP_DISCON set to one and the CONN_SETUP_ABORT flag set to zero. If the FC-VI Provider can provide information as to the cause of the connection abort, it may set CONN_STS to one and set the Reason Code appropriately (see Table 13). The FCVI_HANDLE in the Device_Header is set equal to the FCVI_HANDLE of the Remote Endpoint. The FCVI_MSG_ID in the Device_Header is set equal to the highest Message ID successfully completed by the Disconnect request originator. The FCVI_CONNECTION_ID in the Device_Header is set to a value of FFFFFFFFh.

b) waits for the FCVI_DISCONNECT_RESP IU up to FCVI_ULP_TIMEOUT. If a FCVI_DISCONNECT_RESP IU is not received within the timeout period, the FC-VI Provider transmits an ABTS for the Exchange used for the FCVI_DISCONNECT_RQST IU. The FC-VI Provider may retry the FCVI_DISCONNECT_RQST IU.

c) If a FCVI_DISCONNECT_RESP IU is never received, leave the Local Endpoint in the Error state and complete the VipDisconnect with a return code of VIP_NOT_REACHABLE. Else, transition the Local Endpoint to the Idle state and complete the VipDisconnect with a return code of VIP_SUCCESS.

Rule 2: If a VI Application issues a VipDisconnect call as the result of a timeout during a Connection Set-up (FCVI_CONNECT_RESP1 was never received and the Client did not time out the Connect Request), the FC-VI Provider

a) transmits a FCVI_DISCONNECT_RQST IU with FCVI_APP_DISCON set to one, the CONN_SETUP_ABORT flag set to one, and CONN_STS set to one in the FCVI_FLAGS field with a Reason Code of “Connection Setup Timeout”. Set the FCVI_HANDLE equal to FFFFFFFFh. Set the FCVI_CONNECTION_ID in the Device_Header set to the FCVI_CONNECTION_ID sent in the FCVI_CONNECT_RQST IU.

b) waits for the FCVI_DISCONNECT_RESP IU up to FCVI_ULP_TIMEOUT. If a FCVI_DISCONNECT_RESP IU is not received within the timeout period, the FC-VI Provider transmits an ABTS for the Exchange used for the FCVI_DISCONNECT_RESP IU. The FC-VI Provider may retry the FCVI_DISCONNECT_RQST IU.

c) Transition the VI to the Idle state if a FCVI_DISCONNECT_RESP IU is received.

Rule 3: If an FC-VI Provider receives a FCVI_DISCONNECT_RQST and the FCVI_HANDLE is not defined and the FCVI_CONNECTION_ID is unknown, the FC-VI Provider

a) transmits a FCVI_DISCONNECT_RESP with the FCVI_HANDLE set to FFFFFFFFh in the Device_Header, the FCVI_CONNECTION_ID set to FFFFFFFFh in the Device_Header, the FCVI_APP_DISCON and CONN_SETUP_ABORT flags set equal to the value received in the FCVI_DISCONNECT_RQST, and the CONN_STS set to one in the FCVI_FLAGS field with a Reason Code of “Connection Does Not Exist”.

Rule 6: If an FC-VI Provider receives a FCVI_DISCONNECT_RQST, the FCVI_HANDLE is defined, the FCVI_CONNECTION_ID is unknown, and the CONN_SETUP_ABORT flag is set, the FC-VI Provider

a) transmits a FCVI_DISCONNECT_RESP with the FCVI_HANDLE set to the FCVI_HANDLE of the Disconnect request originator, the FCVI_APP_DISCON and CONN_SETUP_ABORT flags set equal to the value received in the FCVI_DISCONNECT_RQST, and the CONN_STS set to one in the FCVI_FLAGS field with a Reason Code of “Protocol Error”.

D.2 FC-VI disconnect operation example

Figure D.1 illustrates a typical Disconnect of an established Connection between the two Endpoints of a FC-VI Connection. Assume the left Endpoint is the Local Endpoint and the right Endpoint is the Remote Endpoint. Assume that an error has occurred and the Local Endpoint has transitioned to the Error state.

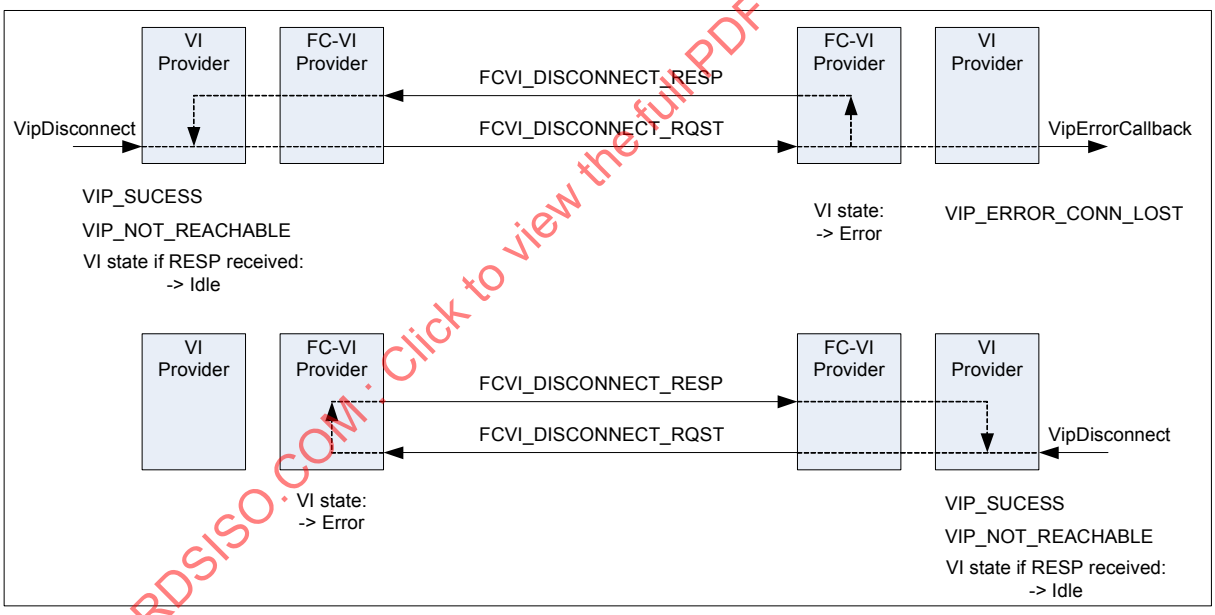


Figure D.1 – FC-VI disconnect operation

In the top of Figure D.1, the Local VI Application issues a VipDisconnect to the VI Provider, which causes the FC-VI Provider to transmit a FCVI_DISCONNECT_RQST IU.

The receipt of the FCVI_DISCONNECT_RQST IU by the Remote FC-VI Provider causes a VipErrorCallback with an error code of VIP_ERROR_CONN_LOST to be issued to the Remote VI Application. The Remote Endpoint is transitioned to the Error state.

The Remote FC-VI Provider transmits a FCVI_DISCONNECT_RESP IU. The VipDisconnect completes at the Local Endpoint with a return code of VIP_SUCCESS after receipt of the FCVI_DISCONNECT_RESP IU. If a FCVI_DISCONNECT_RESP never arrives, the VipDisconnect completes with a return code of VIP_NOT_REACHABLE. The Local Endpoint remains in the Error State if the FCVI_DISCONNECT_RESP IU never arrives.

At some point the Remote VI Application should issue a VipDisconnect to transition its Endpoint to the Idle state, as shown in the bottom of Figure D.1. This causes the Remote FC-VI Provider to transmit a FCVI_DISCONNECT_RQST IU. The Local FC-VI Provider replies with a FCVI_DISCONNECT_RESP IU. Since the Local Endpoint is already in the Idle state, no VipErrorCallback is issued to the Local VI Application. The receipt of a FCVI_DISCONNECT_RESP IU transitions the Local Endpoint to the Idle state.

Under different circumstances, both the Local and Remote Endpoints may issue a VipDisconnect at about the same time. Each Endpoint would already be in the Error State when the FCVI_DISCONNECT_RESP IU arrives, and no VipErrorCallback is issued on either Endpoint.

Annex E
(informative)

Message streaming for reliable reception

Similar to Unreliable Delivery and Reliable Delivery, Messages may be streamed for Reliable Reception. A Message is considered streamed when subsequent Messages have begun processing[2] before all prior Messages have been completed on a VI.

The failure semantics of Reliable Reception, as defined by the VI Architecture, dictate that if a failure occurs on a particular Message, no subsequent Descriptors will be successfully completed for that VI. This allows subsequent Descriptors to be processed by the FC-VI Provider, but they may not be completed until all previous Descriptors have been completed for that VI. If a failure occurs on a given Descriptor, all subsequent Descriptors will be completed in error and the VI connection will be terminated for that VI.

In a hypothetical error-free transport, if an error occurs in processing the Message at the responder, both the originator Descriptor and responder Descriptor for Message M will complete in error. All Descriptors for Messages less than M will have completed successfully on both the originator and responder, while all Message Descriptors greater than or equal to M will complete in error on both the originator and responder. Thus, the Message originator and responder remain in lock step with respect to Descriptor completions. This facilitates error recovery in a distributed environment such as SANs, since the originator knows exactly where the responder has failed and that no subsequent steps (i.e., Descriptors) is completed.

However, transport errors such as lost or corrupt frames may result in the Message responder successfully completing more Descriptors than the Message originator. For example, if a FCVI_RESP IU is lost, the responder may have successfully completed a Descriptor while the originator will complete the Descriptor in error for the same Message. If Messages are streamed to some depth N (i.e., N open Messages), then the responder may complete N more Descriptors than the originator for a VI if multiple transport errors occur.

A VI Application may limit the streaming depth by setting the FCVI_PIPELINE_DEPTH to a value N that limits the number of open Messages for a VI to no more than N Messages. The FCVI_PIPELINE_DEPTH is an attribute in the FCVI_QOS structure in the FCVI_ATTRIBUTES (see Table 18). This guarantees that, even in the presence of multiple transport errors, the responder can complete at most N more Descriptors than the originator.

Both the Disconnect request and the Disconnect response contain the FCVI_MSG_ID of the last Message successfully completed by the Remote Endpoint. A VI Application may use this value to determine at what point the Remote VI Application stopped processing a chain of Descriptors on a VI that terminated due to error. How a VI Application access this information is FC-VI Provider specific. There is currently no defined interface in the VI Architecture to accesses this information.

2. For example, some Message data has been transferred.

Annex F
(informative)

Enabling Message transmission in the FC-VI NIC

This annex describes a mechanism to enable Message transmission in the FC-VI NIC during Connection Setup without the direct involvement of the FC-VI Connection Manager in the critical path.

The Connection Manager is typically implemented as a software module within the Kernel Agent. In the implementation model used in this Annex, the Connection Manager receives and processes all FC-VI Connection Setup IUs. On the Client, Message completion is enabled by the Connection Manager after it has received a FCVI_CONNECT_RESP3 IU, while Message reception is enabled before the FCVI_CONNECT_RESP2 IU is sent (see Figure C.1). Since the Server may send Messages immediately after a FCVI_CONNECT_RESP3 IU is sent, Client receives must be enabled before the FCVI_CONNECT_RESP3 IU is received at the Client. There is a race between the Client enabling Message reception and the Server sending Messages. The only way to avoid this race is to enable receives at the Client before the “Clear-To-Send” (CTS) indication is given to the Server, so receives must be enabled before the Client transmits the FCVI_CONNECT_RESP2 IU (the “CTS”).

However, this creates a potential problem if the FCVI_CONNECT_RESP3 IU never arrives. If this occurs, the Server may transmit Messages for up to FCVI_ULP_TIMEOUT before the Client times out on the Connection Setup and aborts the partial connection. The root of the problem is the Server sees the connection as established while the Client is waiting for the Connection Setup to complete.

Any connection protocol is subject to one Endpoint believing the connection is established while the other is waiting for the connection to complete if the last Connection Setup IU is lost. The Endpoint that transmitted the last IU believes the connection is established and is clear to send, while the Endpoint that is waiting for receipt of the last IU does not observe the connection as established and is prohibited from completing sends.

The mechanism proposed in this subclause significantly reduces the time during which a partially established connection may operate. For the Client, the FCVI_CONNECT_RESP3 IU is used as the Clear-To-Send (CTS) signal by the FC-VI NIC. The FCVI_CONNECT_RESP3 IU is received by the FC-VI NIC as the first “Message” on the VI. However, it is routed to the receive queue for the Connection Manager instead of the receive queue for the VI. The first real received Message on the VI is routed to the correct receive context for the VI. The FCVI_CONNECT_RESP3 IU has a FCVI_MSG_ID of zero and the first Message has a FCVI_MSG_ID of one. For an In-Order Fabric, the FC-VI NIC immediately detects a lost Message if the FCVI_CONNECT_RESP3 IU is lost. For an Out-of-Order Fabric, the FC-VI NIC detects a lost Message within R_A_TOV.

The successful receipt of CTS informs the Client FC-VI NIC that it may complete any Descriptors that have been processed. On the Server side, the FCVI_CONNECT_RESP2 IU cannot effectively be used as a CTS signal by the Server FC-VI NIC. The Server Connection Manager must create and transmit a FCVI_CONNECT_RESP3 IU and complete the VipConnectAccept before the Server transmits any Messages, so it must be involved in the receive path for FCVI_CONNECT_RESP3 IU.

Virtual Interface Architecture Specification

Draft Revision 1.0
December 4, 1997

THIS IS AN INTERIM, DRAFT REVISION. MANY CHANGES MAY OCCUR BETWEEN THIS REVISION AND A FINAL REVISION. ANY DESIGNS OR IMPLEMENTATIONS BASED ON THIS REVISION ARE DONE AT THE DESIGNERS OR IMPLEMENTERS OWN RISK.

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

Table of Contents

1. Introduction

1.1. Overview

This document describes an architecture for the interface between high performance network hardware and computer systems. The goal of this architecture is to improve the performance of distributed applications by reducing the latency associated with critical message passing operations. This goal is attained by substantially reducing the system software processing required to exchange messages compared to traditional network interface architectures. This design is called the Virtual Interface (VI) Architecture.

This document is divided into several parts. They are arranged as follows:

Chapter 1 – Introduction.

This chapter describes the architecture's purpose and scope. It also lays a foundation with definitions of some terms.

Chapter 2 – VI Architecture Overview.

This chapter identifies the main components and key features of the VI Architecture.

Chapters 3 to 8

These chapters describe the semantics, behavior and features of the VI Architecture in detail.

Appendix A

As an aid to hardware implementers, this section contains examples of how the VI Architecture might be interfaced to operating system communication facilities.

Appendix B

This section describes an example VI Descriptor format.

Appendix C

This section describes an example design for a VI enabled network interface controller and a functional description of a VI Kernel Agent.

1.2. Purpose

Distributed applications require the rapid and reliable exchange of information across a network to synchronize operations and/or to share data. The performance and scalability of these applications depend upon an efficient communication facility.

Traditional network architectures do not provide the performance required by these applications, largely due to the host-processing overhead of kernel-based transport stacks. This processing overhead has a negative performance impact in several ways:

- Bandwidth - the overhead limits the actual bandwidth that a given network can deliver. Network hardware bandwidths are increasing by orders of magnitude, while software overhead in available networking stacks remains relatively constant.
- Latency and Synchronization - efficient synchronization is a major scalability factor for distributed and network-based applications. The overhead directly contributes to end-to-end latency of messages used for synchronization.
- Host processing load - the overhead consumes CPU cycles that could be used for other processing.

These problems are addressed in the VI Architecture by moving the network interface much closer to the application, increasing its functionality, and better matching its features to application requirements. The result is a substantial reduction in processing overhead along the communication paths that are critical to performance.

1.3. Architectural Scope

This specification defines an architecture for an interface between high performance network hardware and computer systems. The following items are within the scope of the specification:

- Logical and physical components that comprise the interface.
- Semantics/behavior seen by consumers and providers of the interface.
- Operations required of the interface components. The critical data movement operations are covered in detail. Supporting operations, such as connection establishment, are covered in more general terms.
- Example software interface to illustrate how software could interact with the VI hardware.
- Example design for network hardware that supports the interface architecture. Implementers are not required to utilize this design.

The following items are outside the scope of the specification:

- Specification of the implementation details of the VI Architecture components. This information is operating system and network hardware specific. Implementations may vary as long as the specified behavior is maintained.
- The programming interface used by applications to access hardware that supports the VI Architecture. This interface is operating system specific. It is expected that operating system vendors will issue companion documents that specify mappings of their programming interfaces onto the VI Architecture.
- Absolute values for performance and reliability. General guidelines based on the strength of the architecture are specified, allowing for variance from one implementation to another.

1.4. Document Scope

This document serves two audiences: network hardware vendors and operating system vendors. This document is not intended for applications programmers that use services built on top of the VI Architecture.

1.5. Terminology

1.5.1. 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.

1.5.2. 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.

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

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.

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.

1.5.3. 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.

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.

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.

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.

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

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 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 Card that complies with the VI Architecture Specification.

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. VI Architecture Overview

In the traditional network architecture, the operating system (OS) virtualizes the network hardware into a set of logical communication endpoints available to network consumers. The OS multiplexes access to the hardware among these endpoints. In most cases, the operating system also implements protocols that make communication between connected endpoints reliable. This model permits the interface between the network hardware and the operating system to be very simple. The drawback of this organization is that all communication operations require a call or trap into the operating system kernel, which can be quite expensive to execute. The de-multiplexing process and reliability protocols also tend to be computationally expensive.

The VI Architecture eliminates the system-processing overhead of the traditional model by providing each consumer process with a protected, directly accessible interface to the network hardware - a Virtual Interface. Each VI represents a communication endpoint. VI endpoint pairs can be logically connected to support bi-directional, point-to-point data transfer. A process may own multiple VIs exported by one or more network adapters. A network adapter performs the endpoint virtualization directly and subsumes the tasks of multiplexing, de-multiplexing, and data transfer scheduling normally performed by an OS kernel and device driver. An adapter may completely ensure the reliability of communication between connected VIs. Alternately, this task may be shared with transport protocol software loaded into the application process, at the discretion of the hardware vendor.

2.1. VI Architecture Components

The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider is composed of a physical network adapter and a software Kernel Agent. The VI Consumer is generally composed of an application program and an operating system communication facility. The organization of these components is illustrated in Figure 1.

VI Architectural Model

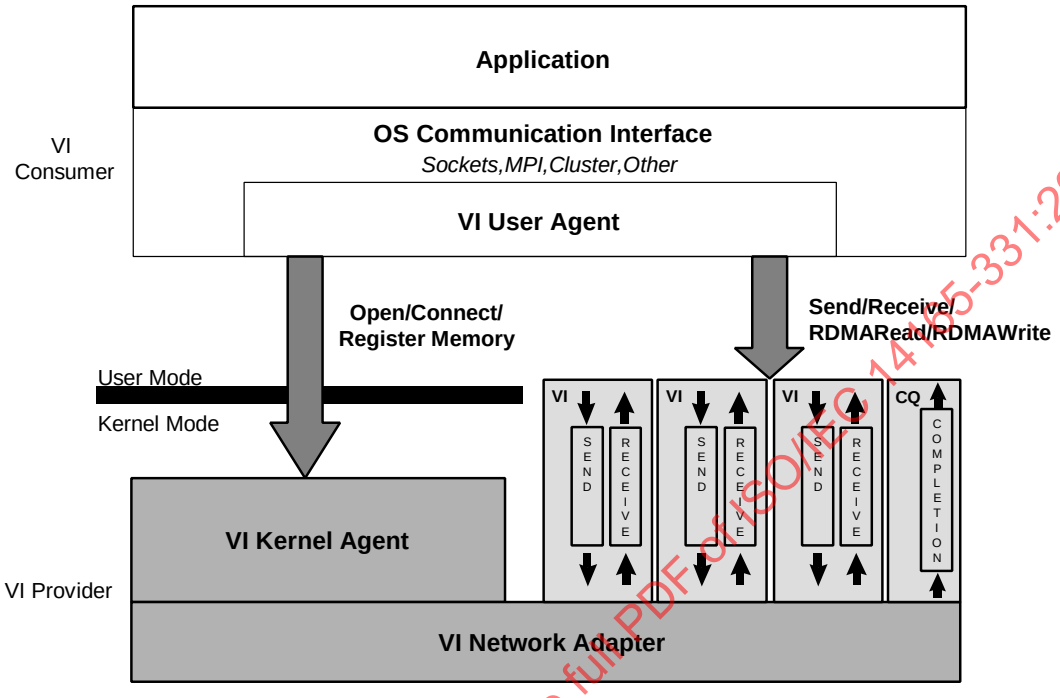


Figure 1: The VI Architectural Model

2.1.1. Virtual Interfaces

A Virtual Interface is the mechanism that allows a VI Consumer to directly access a VI Provider to perform data transfer operations. Figure 2 illustrates a Virtual Interface.

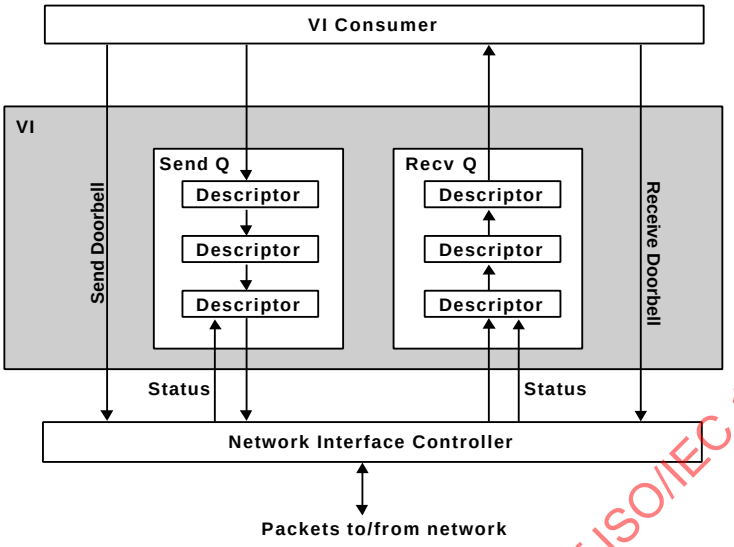


Figure 2: A Virtual Interface

A VI consists of a pair of Work Queues: a send queue and a receive queue. VI Consumers post requests, in the form of Descriptors, on the Work Queues to send or receive data. A Descriptor is a memory structure that contains all of the information that the VI Provider needs to process the request, such as pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them with a status value when completed. VI Consumers remove completed Descriptors from the Work Queues and use them for subsequent requests. Each Work Queue has an associated Doorbell that is used to notify the VI network adapter that a new Descriptor has been posted to a Work Queue. The Doorbell is directly implemented by the adapter and requires no OS intervention to operate.

A Completion Queue allows a VI Consumer to coalesce notification of Descriptor completions from the Work Queues of multiple VIs in a single location. Completion queues are discussed in more detail in section 2.4.

2.1.2. VI Provider

The VI Provider is the set of hardware and software components responsible for instantiating a Virtual Interface. The VI Provider consists of a network interface controller (NIC) and a Kernel Agent.

The VI NIC implements the Virtual Interfaces and Completion Queues and directly performs data transfer functions. The specific design of a VI NIC is not mandated in this document, but reference material for the implementation of a VI NIC is included in an Appendix to facilitate implementers.

The Kernel Agent is a privileged part of the operating system, usually a driver supplied by the VI NIC vendor, that performs the setup and resource management functions needed to maintain a Virtual Interface between VI Consumers and VI NICs. These functions include the creation/destruction of VIs, VI connection setup/teardown, interrupt management and/or processing, management of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms such as

system calls. Kernel Agents interact with VI NICs through standard operating system device management mechanisms.

2.1.3. VI Consumer

The VI Consumer represents the user of a Virtual Interface. While an application program is the ultimate consumer of communication services, applications access these services through standard operating system programming interfaces such as Sockets or MPI. The OS facility is generally implemented as a library that is loaded into the application process.

The OS facility makes system calls to the Kernel Agent to create a VI on the local system and connect it to a VI on a remote system. Once a connection is established, the OS facility posts the application's send and receive requests directly to the local VI. The data transfer mechanism will be discussed further in section 2.3.

The OS communication facility often loads a library that abstracts the details of the underlying communication provider, in this case the VI and Kernel Agent. This component is shown as the VI User Agent in Figure 1. It is supplied by the VI Hardware vendor, and conforms to an interface defined by the OS communication facility. Appendix A illustrates an example User Agent interface that exposes all of the capabilities of the VI Architecture.

2.2. Memory Registration

Most computer system designs require that the memory pages used to hold messages are locked down and that their virtual addresses be translated into physical locations before a NIC can access them. The pages are unlocked when the transfer is complete. Traditional network transports perform these operations on every data transfer request. This processing contributes significant overhead to the data transfer operation. The VI Architecture requires the VI Consumer to identify memory used for a data transfer prior to submitting the request. Only memory that has been registered with the VI Provider can be used for data transfers. This memory registration process allows the VI Consumer to reuse registered memory buffers, thereby avoiding duplication of locking and translation operations. Memory registration also takes this processing overhead out of the performance-critical data transfer path.

Memory registration enables the VI Provider to transfer data directly between the buffers of a VI Consumer and the network without copying any data to or from intermediate buffers. Traditional network transports often copy data between user buffers and intermediate kernel buffers. Data copies and buffer management are a large component of overhead in communication and consume memory bandwidth.

Memory registration consists of locking the pages of a virtually contiguous memory region into physical memory and providing the virtual to physical translations to the VI NIC. The VI Consumer gets an opaque handle for each memory region registered. The VI Consumer can reference all registered memory by its virtual address and its associated handle.

2.3. Data Transfer Models

There are two types of data transfer facilities provided by the Virtual Interface Architecture. These data transfer models are 1) a traditional Send/Receive messaging model, and 2) the Remote Direct Memory Access (RDMA) model.

2.3.1. Send/Receive

The Send/Receive model of the VI Architecture follows a well known and well understood model of transferring data between two endpoints. In this model, the VI Consumer on the local node always specifies the location of the data. On the sending side, the sending process specifies the memory regions that contain the data to be sent. On the receiving side, the receiving process specifies the memory regions where the data will be placed. Given a single connection, there is a

one to one correspondence between send Descriptors on the transmitting side and receive Descriptors on the receiving side.

The VI Consumer at the receiving end pre-posts a Descriptor to the receive queue of a VI. The VI Consumer at the sending end can then post the message to the corresponding VI's send queue. The Send/Receive model of data transfer requires that the VI Consumers be notified of Descriptor completion at both ends of the transfer, for synchronization purposes.

VI Consumers are responsible for managing flow control on a connection. The VI Consumer on the receiving side must post a Receive Descriptor of sufficient size before the sender's data arrives. If the Receive Descriptor at the head of the queue is not large enough to handle the incoming message, or the Receive Queue is empty, an error will occur. The connection may be broken if it is intended to be reliable. See section 2.5.

The VI Architecture differs from some existing models in that all Send/Receive operations complete asynchronously.

2.3.2. Remote Direct Memory Access (RDMA)

In the RDMA Model, the initiator of the data transfer specifies both the source buffer and the destination buffer of the data transfer. There are two types of RDMA operations, RDMA Write and RDMA Read.

For the RDMA Write operation, the VI Consumer specifies the source of the data transfer in one of its local registered memory regions, and the destination of the data transfer within a remote memory region that has been registered on the remote system. The source of an RDMA Write can be specified as a gather list of buffers, while the destination must be a single, virtually contiguous region. The RDMA Write operation implies that prior to the data transfer, the VI Consumer at the remote end has informed the initiator of the RDMA Write of the location of the destination buffer, and that the buffer itself is enabled for RDMA Write operations. The remote location of the data is specified by its virtual address and its associated memory handle.

For the RDMA Read operation, the VI Consumer specifies the source of the data transfer at the remote end, and the destination of the data transfer within a locally registered memory region. The source of an RDMA Read operation must be a single, virtually contiguous buffer, while the destination of the transfer can be specified as a scatter list of buffers. The RDMA Read operation implies that prior to the data transfer, the VI Consumer at the remote end has informed the initiator of the RDMA Read of the location of the source buffer, and that the buffer itself is enabled for RDMA Read operations. The remote location of the data is specified by its virtual address and its associated memory handle.

No Descriptors on the remote node's receive queue are consumed by RDMA operations. No notification is given to the remote node that the request has completed. The exception to this rule is that if Immediate Data is specified by the initiator of an RDMA Write request it will consume a Descriptor on the remote end when the data transfer is complete, thus allowing for synchronization. The VI Consumer on the receiving side must post a Receive Descriptor to receive the Immediate Data, before the sender executes the RDMA Write. If no Descriptor is posted, an error will occur and the connection may be broken. See Section 2.5. Immediate Data is not allowed on RDMA Reads.

RDMA Write is a required feature of the VI Architecture. VI Provider support for RDMA Read is optional. A VI Provider should supply a mechanism by which a VI Consumer can determine if the Provider supports RDMA Read operations.

2.4. Completion Queues

Notification of completed requests can be directed to a Completion Queue on a per-VI Work Queue basis. This association is established when a VI is created. Once a VI Work Queue is associated with a Completion Queue, all completion synchronization must take place on that Completion Queue.

As with VI Work Queues, notification status can be placed into the Completion Queue by the VI NIC without an interrupt, and a VI Consumer can synchronize on a completion without a kernel transition.

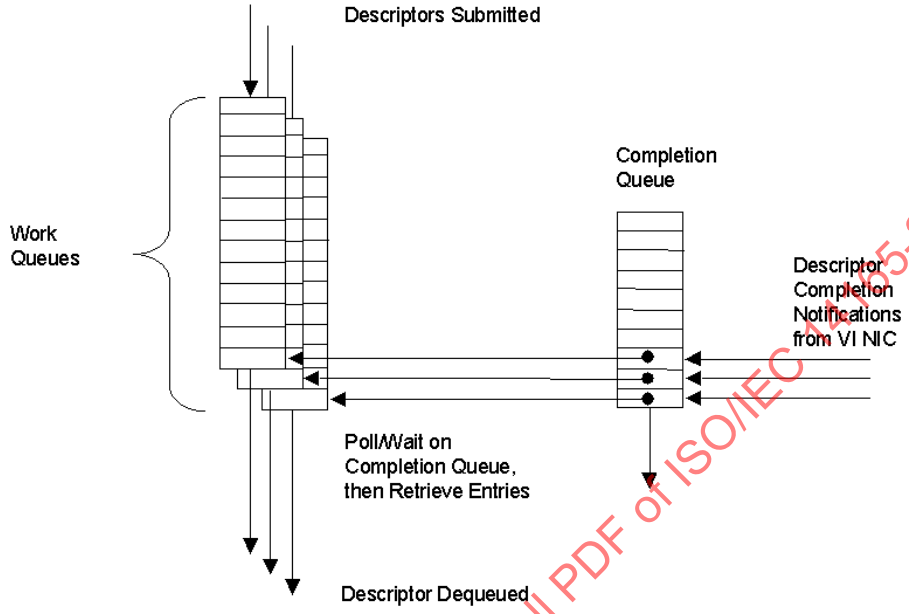


Figure 3: VI Architecture Completion Queue Model

2.5. Reliability Levels

The VI Architecture supports three levels of communication reliability at the NIC level: Unreliable Delivery, Reliable Delivery and Reliable Reception. All VI NICs are required to support the Unreliable Delivery level. Support for Reliable Delivery and Reliable Reception is optional. Support for one, or both, of the higher reliability levels is strongly recommended because it enables lighter weight Consumer software. The reliability level is an attribute of a VI. Only VIs with the same reliability level can be connected. Table 1 summarizes the properties of the three levels of reliability.

Property/Level of Reliability	Unreliable	Reliable Delivery	Reliable Reception
Corrupt data detected	Yes	Yes	Yes
Data delivered at most once	Yes	Yes	Yes
Data delivered exactly once	No	Yes	Yes
Data order guaranteed	No	Yes	Yes
Data loss detected	No	Yes	Yes
Connection broken on error	No	Yes	Yes
RDMA Read Support	No	Optional	Optional
RDMA Write Support	Yes	Yes	Yes
State of Send/RDMA Write when request completed	In-flight	In-flight	Completed on remote end also
State of in-flight Send/RDMA Write when error occurs	Unknown	Unknown	First one unknown, others not delivered

Table 1: Reliability Guarantees

2.5.1. Unreliable Delivery

Unreliable Delivery VIs guarantee that a Send or RDMA Write is delivered at most once to the receiving VI. In other words, a Send or RDMA Write request will cause at most one Receive Descriptor to be consumed.

Unreliable VIs guarantee that corrupted transfers are always detected on the receiving side. Specifically, if an incoming Send or RDMA Write consumes a Descriptor on the receive queue, and the transferred data is corrupt, the associated error bit must be set in the status field of the Receive Descriptor.

Sends and RDMA Writes may be lost on an Unreliable VI. VI Providers are required neither to detect this condition nor to retransmit the lost data. Requests are not guaranteed to be delivered to the receiver in the order submitted by the sender; however, the sending VI Provider must adhere to the Descriptor processing ordering rules. See section 6.3.1.

RDMA Writes should only be used on Unreliable connections in situations where late delivery or loss of the data can be tolerated. A video frame buffer is one example. Because the VI Provider has direct access to the VI Consumer's memory, a delayed Write may occur at an arbitrary time. The ordering of the Write with respect to Send requests also cannot be guaranteed. As a result, it is difficult for a VI Consumer to achieve reliable RDMA Write service through standard schemes such as retransmission.

The connection between two Unreliable VIs is not broken when a request processing error is detected. The error is simply reported to the VI Consumer. The VI does not transition to the *Error* state.

RDMA Reads are not supported on an Unreliable VI. Posting an RDMA Read request on an Unreliable VI will result in a Descriptor format error.

Because Unreliable Delivery VIs will often be used for latency sensitive operations, such as cluster membership heartbeats, they must not unduly delay transmission of messages. In other words, a Reliable Delivery VI or Reliable Reception VI cannot act as an Unreliable Delivery VI if it introduces substantial retransmission delays. An Unreliable Delivery VI that allowed very large transfers to starve small transfers would also be unacceptable.

2.5.2. Reliable Delivery

A Reliable Delivery VI 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. Transport errors are considered catastrophic and should be extremely rare for VI Providers offering this level of service. The VI Provider will deliver an error to the VI Consumer if a transfer is lost, corrupted, or delivered out of order. An error will also be delivered if an RDMA Write with Immediate Data or a Send is lost because the Receive Queue is empty or the Descriptor at the head of the queue is not of sufficient size to contain the data. Upon detection of any error, the VI transitions to the *Error* state, the connection is broken, and all posted Descriptors are completed with an error status.

A Send or RDMA Write Descriptor is completed with a successful status once the associated data has been successfully transmitted on the network. An RDMA Read Descriptor is completed with a successful status once the requested data has been written to the target buffers on the initiator's system.

Errors that occur on the initiating system, such as Descriptor format errors or local memory protection errors, cause a Descriptor to be completed with an unsuccessful status. One or more error bits will be set in the Descriptor's Status field.

Errors that occur after a Descriptor has been completed with a successful status, such as a transport error, hardware error, lost packet, reception error, or sequencing error, are delivered to the VI Consumer asynchronously through a mechanism supplied by the VI Provider. Errors may be reported at either the sending side or the receiving side. One or more additional Descriptors may complete before a non-local error is reported for a previously completed Descriptor. The status of the transfers initiated by these Descriptors is unknown.

2.5.3. Reliable Reception

Reliable Reception VIs behave like Reliable Delivery VIs with the following differences: A Descriptor is completed with a successful status only when the data have been delivered into the target memory location. If an error occurs that prevents the successful in-order, intact, exactly once delivery of the data into the target memory, the error is reported through the Descriptor status. The Provider guarantees that, when an error occurs, no subsequent Descriptors are processed after the Descriptor that caused the error. From the point of view of a VI Consumer, the VI Provider guarantees strict ordering, regardless of the actual behavior between the local and remote end-points.

The asynchronous error handling mechanism may still be invoked for errors such as disconnection and hardware errors. Errors may be reported to the remote Consumer as well as to the local one.

As with Reliable Delivery connections, any error causes the connection to be broken, the VI to transition to the *Error* state, and all posted Descriptors to be completed in error. Transport errors are considered catastrophic and should be extremely rare for VI Providers offering this level of service.

2.6. System Area Networks

The VI Architecture is designed to enable applications to communicate over a System Area Network (SAN). A SAN is a type of network that provides high bandwidth, low latency communication. SANs have very low error rates. SANs are often made highly available through the use of redundant interconnect fabrics. SAN performance more closely resembles that of a memory subsystem than a traditional network, such as a LAN.

A SAN is usually used to interconnect nodes within a distributed computer system, such as a cluster. These systems are members of a common administrative domain and are usually within close physical proximity. A SAN is assumed to be physically secure.

A SAN must exhibit fair arbitration and forward progress for every connection. A SAN must scale under load. A SAN may use media typically associated with a LAN or WAN, but this is not a requirement. A SAN does not replace traditional LANs and WANs, but instead exhibits properties that make it a unique network type.

3. Managing VI Components

This section discusses how the components of a Virtual Interface are created, destroyed, and managed.

3.1. Accessing a VI NIC

A VI Consumer gains access to the Kernel Agent of a VI Provider using standard operating system mechanisms. Normally, this involves opening a handle to the Kernel Agent that represents the target VI NIC. The VI Consumer uses this handle to perform general management operations such as registering Memory Regions, creating Completion Queues and creating VIs. This mechanism would also be used to retrieve information about the VI NIC, such as the reliability levels it supports and its transfer size limits.

VI hardware resources cannot be shared across multiple VI NICs, even if they are managed by the same Kernel Agent. Hardware resources may include Completion Queues, mapped memory and other resources that are associated with an instance of the hardware.

A Kernel Agent must use standard operating system mechanisms to detect when a VI Consumer process exits so that it can cleanup any resources used by the process. The Kernel Agent must keep track of all resources associated with a VI Consumer's use of a VI NIC.

3.2. Memory Management

3.2.1. Registering and De-registering Memory

The VI Architecture requires that memory used for data transfers, both buffers and Descriptors, be registered with the VI Provider. The memory registration process defines one or more virtually contiguous physical pages as a Memory Region. A VI Consumer registers a Memory Region with the Kernel Agent, which returns a Memory Handle that, along with its virtual address, uniquely identifies the registered region. The VI Consumer must qualify any virtual address used in an operation on a VI with the corresponding Memory Handle. A VI Consumer must de-register a Memory Region when the region is no longer in use.

When a Memory Region is registered, every page within the region is locked down in physical memory. This guarantees to the VI NIC that the memory region is physically resident (not paged out) and that the virtual to physical translation remains fixed when the NIC is processing requests that refer to that region. The VI Kernel Agent manages the VI NIC's Page Table. The Page Table contains the mapping and protection information for registered Memory Regions.

The VI Consumer is allowed to specify arbitrary alignment and lengths of memory regions to be registered, but the translation and the memory attributes of the region are applied to each complete page within that memory region. The VI Provider is only required to ensure that the memory location being referenced is in a valid page of a registered memory region.

Memory is registered on a per process basis. Memory registered by a thread within a process is accessible by any thread within that process. One process cannot use another process's Memory Handle to access the same memory region. An attempt to do so will result in a memory protection error.

Registration may fail due to the NIC's inability to find a Page Table entry large enough to register the memory region. No memory is registered in this case. Registration must either fully succeed or fail, atomically.

The same virtual address range may be registered multiple times, resulting in multiple Memory Handles. This is true on a single NIC, as well as across multiple NICs.

Posted Descriptors that are contained in or reference memory that is de-registered will result in a protection violation. This error will be generated at the time that the VI Provider attempts the memory reference.

3.2.2. Memory Protection

Only the two VI Consumer processes associated with a pair of connected VIs are allowed to exchange memory contents. The processes can also limit each other's access to specific regions of registered memory. This is accomplished by two mechanisms that complement Memory Handles: Memory Protection Tags and Memory Protection Attributes.

3.2.2.1. Memory Protection Tags

Memory Protection Tags are unique IDs that are associated both with VIs and with Memory Regions. The VI Provider creates and destroys Memory Protection Tags on behalf of the VI Consumer. Each Memory Protection Tag must be unique within a VI Provider. The VI Consumer assigns a Memory Protection Tag to a Memory Region when the region is registered and to a VI when the VI is created. The tag can be replaced later on with a different tag by changing the attributes of a Memory Region, and/or of the VI. The VI Provider must ensure that the Protection Tag that is associated with a VI or registered memory region is valid only for that VI Consumer. A memory access is only allowed by a VI NIC if the Memory Protection Tag of the VI and of the Memory Region involved are identical. Accesses that violate this rule result in a memory protection error and no data is transferred. The validation of Protection Tags applies to registered memory regions that are used to hold Descriptors, as well as memory regions that hold data. A VI Provider must be able to supply at least one Protection Tag for each VI instance that it supports.

A VI Consumer that is not concerned with protection would use the same Memory Protection Tag for all VIs and all Memory Regions. Another VI Consumer might need to keep different remote systems from accessing memory used by each other. That VI Consumer would use one Memory Protection Tag for each client, and associate the tag with those VIs and Memory Regions that the client may use. More sophisticated sharing relationships are made possible by registering the same memory region multiple times.

3.2.2.2. Memory Protection Attributes

Memory Protection Attributes are associated with Memory Regions and VIs. They are used to control RDMA Read and RDMA Write access to a given Memory Region.
The Memory Protection Attributes are RDMA Read Enable and RDMA Write Enable. These permissions are set for Memory Regions and VIs when they are created. These permissions can be modified later on by changing the attributes of the Memory Region, and/or of the VI.

If Memory Protection Attributes between a VI and a Memory Region do not match, the attribute offering the most protection will be honored. For instance, if a VI has RDMA Read Enabled, but the Memory Region does not, the result is that RDMA Reads on that VI from that Memory Region will fail.

Memory Protection Attributes are enforced at the remote end of the connection that is referred to by the Address Segment of the Descriptor. They do not apply at the end posting the RDMA request to the send queue.

An RDMA operation that violates the permission settings results in a memory protection error and no data is transferred.

3.3. Creating and Destroying VIs

A VI is created by a VI Provider at the request of a VI Consumer. A VI consists of a pair of Work Queues and a pair of Doorbells, one for each Work Queue. Work Queues are structures that are allocated from a VI Consumer process' virtual memory. The VI Provider maps and locks this memory and informs the VI NIC of its location. A Doorbell is hardware resource located on the VI

NIC. The Kernel Agent maps this resource into the virtual address space of a VI Consumer Process using standard operating system facilities. The VI Provider supplies the VI Consumer with the information needed to directly access these structures when a VI is created. If these resources cannot be allocated and mapped, an error will result and the VI will not be created.

There is no connection established upon creation of a VI. No data will flow until the VI is connected to another VI. See section 4 for more information on connecting VIs.

A VI Consumer should instruct a VI Provider to destroy a VI that is no longer in use. A VI cannot be destroyed if any packets remain on its Work Queues. A VI may only be destroyed if it is in the *Idle* state. See Section 5 for a discussion of VI states. The Work Queue pair and Doorbell are de-allocated when the associated VI is destroyed.

In order to avoid consuming large parts of a VI Consumer's virtual address space, it is recommended that the VI Provider map multiple Doorbells into a single page if a VI Consumer opens multiple VIs. Doorbells that belong to different processes must be mapped in different pages.

3.4. Creating and Destroying Completion Queues

A Completion Queue can be used to direct notification of Descriptor completions from multiple Work Queues to a single location. The Work Queues associated with a Completion Queue may span multiple VIs on the same VI NIC. A Completion Queue must be created before any of its associated VI Work Queues are created. A Completion Queue is created by a VI Provider at the request of a VI Consumer. Each VI Work Queue is optionally associated with a Completion Queue when the VI is created. Work queues on the same VI may be associated with a different Completion Queue, if desired.

The maximum number of Descriptors that can be outstanding at any given time in a Completion Queue is defined by the VI Consumer when the Completion Queue is created. The VI Consumer is responsible for ensuring that this number is large enough to prevent overflow of the queue. The VI NIC must be able to support Completion Queues with at least 1024 entries.

In order to create a Completion Queue, the VI Provider allocates memory for the queue in the VI Consumer's virtual address space. It then maps and locks this memory and informs the VI NIC of its location. If enough memory cannot be allocated, or it cannot be mapped and locked, an error will result and the Completion Queue will not be created.

Completion Queues may be resized dynamically through the VI Provider. It is important to understand that while this operation is taking place, all IO to the Completion Queue may cease, depending on the VI Provider's implementation of this function. Incoming requests should still be satisfied, and no incoming data should be rejected unless there is an insufficient number of Descriptors.

A VI Consumer should instruct a VI Provider to destroy a Completion Queue that is no longer in use. A Completion Queue cannot be destroyed until all VIs associated with it have been destroyed. VI Providers are responsible for destroying any Completion Queues still associated with a process when the process is destroyed by the operating system.

4. VI Connection and Disconnection

The VI Architecture provides connection-oriented data transfer service. When a VI is initially created, it is not associated with any other VI. A VI must be connected with another VI through a deliberate process in order to transfer data. When data transfer is completed, the associated VIs must be disconnected.

4.1. VI Connection

A VI Consumer issues requests to its VI Provider in order to connect its VI to a remote VI. VI Providers must implement robust and reliable connection protocols. In particular, VI Providers must prevent interference with current connections and the creation of stale or duplicate connections by delayed or duplicate packets from extinct connections.

The endpoint association model is a client-server model. The server side waits for incoming connection requests and then either accepts them or rejects them based on the attributes associated with the remote VI. A state diagram depicting this process is shown in Figure 4. A functional definition of the process is as follows.

The server VI Consumer issues a ConnectWait request to its VI Provider. This request contains the address discrimination values that are acceptable to the VI Consumer. A VI Consumer should be able to accept a connection from any remote endpoint or a specific remote endpoint, based on the discriminator supplied. The request also contains a data structure used to receive information about the remote VI that is requesting a connection. The request may indicate a timeout value.

Sometime after the server VI Consumer begins waiting for a connection, the client VI Consumer issues a ConnectRequest request to its VI Provider. This request specifies the local VI that is to be connected, an address structure that indicates the remote VI to which to connect, and a timeout value. It also specifies a data structure used to receive information about the corresponding server VI, if the connect operation completes successfully.

The client's ConnectRequest request results in one of two actions. If the specified remote VI does not exist, is not reachable, is in the wrong state, or its discriminator doesn't match, then the VI Provider will return an error to the VI Consumer's request. If the specified remote VI is available then the server VI Consumer's ConnectWait request completes, and information about the client VI is returned to the server VI Consumer. A unique identifier for the incoming connection request is also returned.

The server VI Consumer then decides whether to accept this incoming request or to reject it. If the server intends to accept the connection, it must prepare a VI for the connection. The server VI Consumer may either choose a VI from a pool that it has previously created or it may create a new VI with attributes it considers appropriate for this connection request. The reliability level of the new VI must match that of the remote VI. The VI Providers must also agree on the MTU to be used on the connection.

If the server VI Consumer intends to accept the connection, it issues a ConnectAccept request to the VI Provider, specifying the incoming connection ID as well as the local VI to be used. If the local VI's reliability attributes match those required by the remote VI, the connection is established and the VI State transitions according to the State Diagram in Section 5. If the local VI's attributes do not meet the requirements, then the ConnectAccept will complete in error; however, the incoming connection request remains valid. The server VI Consumer must either issue a valid ConnectAccept request, or reject the connection.

If the server intends to reject the connection, it issues a ConnectReject request to the VI Provider, specifying the incoming connection ID.

If the connection request was rejected, the client VI Consumer's ConnectRequest request returns with a status indicating that fact.

If the connection request was accepted, the client VI Consumer's ConnectRequest request returns successfully. The VI transitions states according to the State Diagram in Section 5.

Figure 4 illustrates the VI connection process.

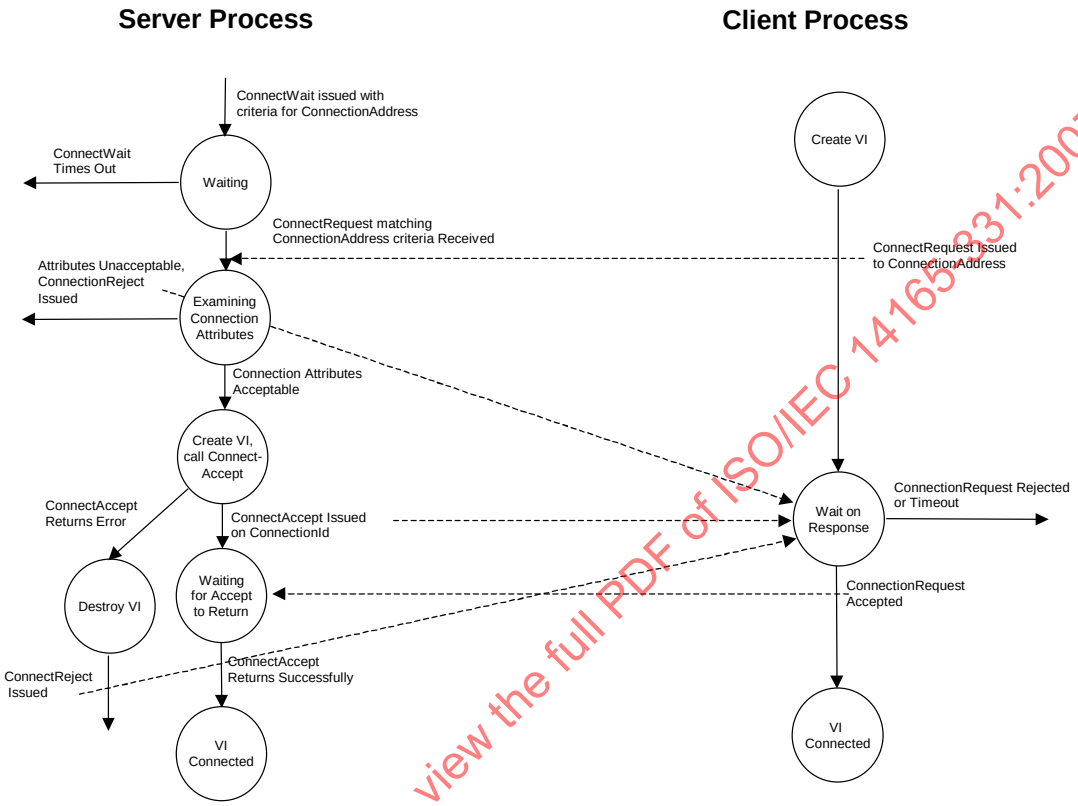


Figure 4: VI Architecture Endpoint Connection Process

The VI connection model does not attempt to apply either authorization or authentication of a VI connection. It is recommended that connected VI Consumers perform an authentication process, especially before providing RDMA access to registered memory.

The VI connection model must allow a connection to be established between two endpoints on the same node.

4.2. VI Disconnection

A VI Consumer issues a Disconnect request to a VI Provider in order to disconnect a connected VI. The Disconnect request unilaterally aborts the connection. 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.

Implementers must ensure that stale connections cannot be reused.

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.

When a VI Consumer issues a Disconnect request for a VI, the VI will transition to a new state according to the change of state rules listed in Section 5.

4.3. VI Address Format

Each VI Provider must define an address format that uniquely identifies all possible VIs on a SAN. A VI Consumer must be aware of the address format used by a VI Provider. The address format must allow VI discrimination across systems as well as on the same node. The format must also permit discrimination of connection requests from only a specific VI or from any VI. The VI Address Format does not require support for multicast or broadcast capability.

5. VI States

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.

Figure 5 depicts the VI states and the transitions between them.

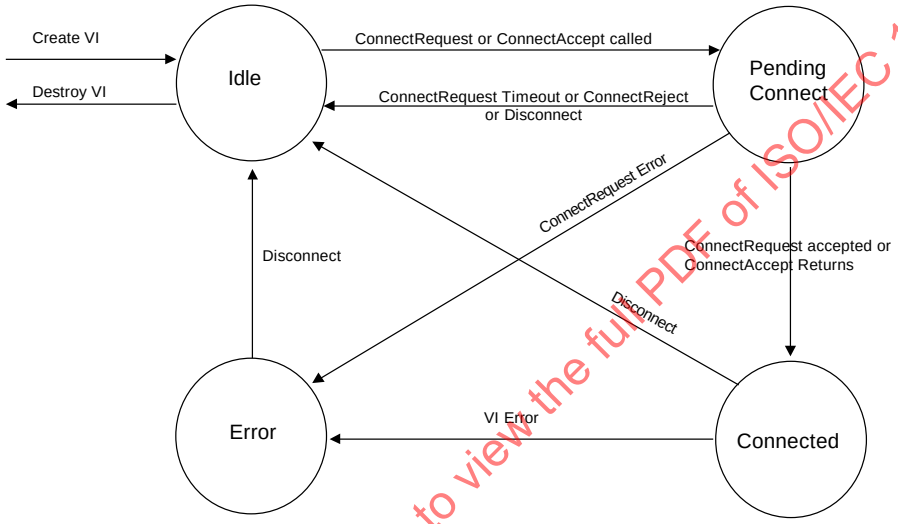


Figure 5: VI State Diagram

5.1. Idle State

A VI is in the *Idle* state when it is created. A VI can only be destroyed when in this state. There must be no Descriptors on the VI's Work Queues in order to destroy it.

Submitting a ConnectRequest request to the VI Provider will transition the VI to the *Pending Connect* state. If the connection request does not complete within the timeout period specified, the VI will return to the *Idle* state.

When a ConnectAccept request is made, the VI transitions into the *Pending Connect* state. If the request fails or times out, the VI returns to the *Idle* state.

Descriptors may be posted to the Receive Queue for a VI while the VI is in the *Idle* state, but the Descriptors will not be processed until the VI transitions to the *Connected* state or the VI Consumer issues a Disconnect request.

Descriptors posted to the Send Queue of an *Idle* VI are immediately completed in error.

Errors that occur while a VI is in the *Idle* state, which would normally result in the invocation of the asynchronous error handling mechanism, are reported when the VI Consumer attempts to connect the VI.

5.2. Pending Connect State

The *Pending Connect* state indicates that a connection request has been submitted to the VI Provider but that the connection has not yet been established.

This state is entered when a ConnectRequest operation is submitted to the VI Provider while the VI is in the *Idle* state.

This state is entered when a ConnectAccept operation has been submitted to the VI Provider for a VI in the *Idle* state. The VI stays in this state until the connection acceptance processing has completed.

A confirmed and successful response to a ConnectRequest request will transition the VI into the *Connected* state.

A successful ConnectAccept request will transition the VI into the *Connected* state.

A timeout or rejection of the Connect request transitions the VI into the *Idle* state.

A Disconnect request issued for a VI in the *Pending Connect* state results in the VI transitioning into the *Idle* state.

Transport or hardware errors generated from the VI NIC will transition the VI into the *Error* state.

Descriptors may be posted to the Receive Queue of a VI in the *Pending Connect* state, but they will not be processed until the VI transitions to the *Connected* state or the VI Consumer issues a Disconnect request.

Descriptors posted to the Send Queue of a VI in the *Pending Connect* state are completed in error.

There cannot be any inbound or outbound traffic on a VI in this state, because the VI is not connected.

5.3. Connected State

The *Connected* state is the state of normal processing. This is the state in which data flows across the connection.

A VI transitions into this state from the *Pending Connect* state upon a confirmed and successful response to a ConnectRequest request.

A VI transitions into this state from the *Pending Connect* state upon successful completion of a ConnectAccept request.

A Disconnect request submitted to the VI Provider by the local VI Consumer transitions a VI to the *Idle* state.

Hardware errors will result in the VI transitioning into the *Error* state. Other errors that occur may also result in the VI transitioning into the *Error* state. This behavior is dependent upon the reliability level of the connection, as discussed in Section 2.5.

Descriptors posted to the Send or Receive Queue of a VI in the *Connected* state are processed normally.

All inbound and outbound traffic is processed normally.

5.4. Error State

The *Error* state is entered as the result of an error during normal processing, or an event generated by the VI NIC.

For VIs with Reliable Delivery or Reliable Reception guarantees, certain classes of errors that occur while the VI is in the *Connected* state will result in transitions into the *Error* state. This is discussed in Section 2.5.

A Disconnect request will transition the VI into the *Idle* state. Descriptors pending or posted to either the Receive Queue or the Send Queue when the VI is in this state will result in the Descriptor being completed in error.

Inbound traffic sent to this VI is refused. There is no outbound traffic, since requests posted to the Send Queue are completed in error. Any outbound traffic left on a queue when the VI transitions into this state is aborted and the corresponding Descriptors are completed in error.

6. Descriptor Processing Model

Data transfer requests are represented by Descriptors. Descriptors contain all the information needed to process a request, such as the type of transfer to make, the status of the transfer, queue information, immediate data and a scatter-gather style buffer pointer list.

Figure 6 illustrates the VI Architecture Descriptor processing models, namely the Work Queue Model and the Completion Queue Model. They differ only in how the VI Consumer is notified of Descriptor completion. In both cases, Descriptors are enqueued and dequeued from a VI's Work Queue. In the Work Queue Model, the VI Consumer polls for completions on a particular Work Queue by examining the status of the Descriptor at the head of the queue. When the head Descriptor is completed, the VI Consumer dequeues it. In the Completion Queue Model, the VI Consumer polls for completions on a set of Work Queues by examining the head of the Completion Queue. When a Descriptor completes, its identity is written to the Completion Queue. Once the VI Consumer receives notification of a Descriptor completion from the Completion Queue, the VI Consumer must dequeue the Descriptor from the appropriate Work Queue. A VI Provider should supply a mechanism by which VI Consumers can wait for Descriptors to complete on a VI Work Queue or wait for a notification to be posted to a Completion Queue.

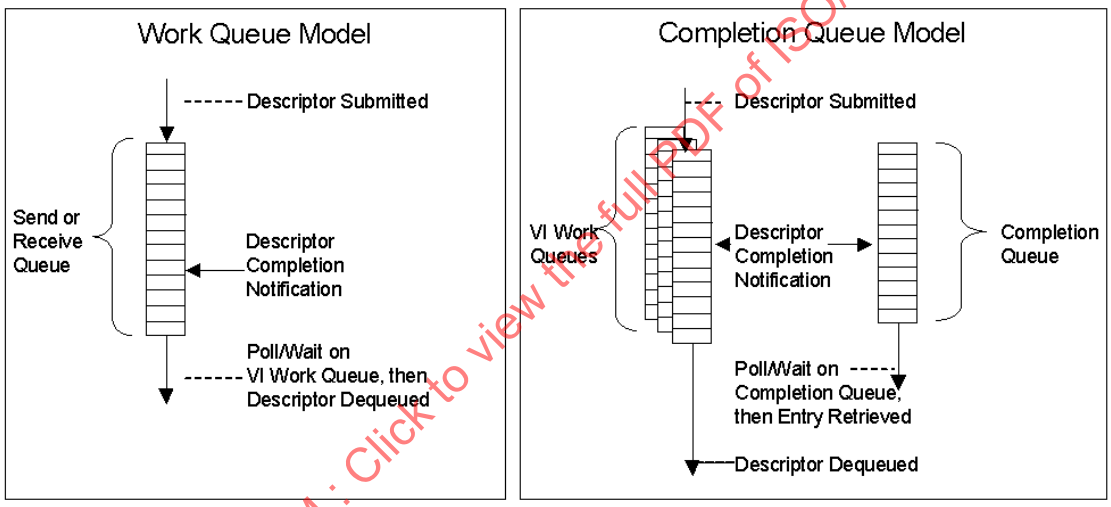


Figure 6: VI Architecture Descriptor Processing Models

6.1. Forming Descriptors

There are two types of Descriptors, each one serving two different functions. There are Send/Receive type Descriptors and there are RDMA type Descriptors. The two functions are outbound data transfer and inbound data transfer.

A Descriptor occupies a variable-length, virtually contiguous region of a process's virtual address space. Descriptors must be aligned on a 64-byte boundary. Descriptors must reside in memory regions that have been registered by the VI Consumer with the VI Provider. Descriptors may cross physical page boundaries, as long as a single Descriptor resides within a single memory region. Any time a Descriptor is used in a request to the VI Provider, it must be accompanied by the memory handle associated with the region of memory where the Descriptor lies.

Descriptor allocation is managed by the VI Consumer.

Posting a Descriptor to a VI Work Queue transfers ownership of the Descriptor to the VI Provider. Modification of a Descriptor by the VI Consumer while it is posted results in unpredictable behavior. Ownership of the Descriptor is returned to the VI Consumer when the VI Provider

completes the Descriptor. A VI Provider completes a Descriptor by writing a completion status value into the Descriptor. The VI Provider must also write an entry to the associated Completion Queue, if one is in use.

Descriptors are composed of segments. There are three types of segments: control, address and data. 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, but only for RDMA operations. This segment contains remote buffer information for RDMA Read and RDMA Write operations.

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

6.1.1. Data Considerations

6.1.1.1. Scatter-Gather Considerations

The Descriptor format allows the VI Consumer to specify a scatter-gather list in each Descriptor so that data can be sent from or placed in the desired location(s) directly by the hardware. The VI Consumer may construct a scatter-gather list of any length up to and including the NIC's segment count limit. The Scatter-Gather List length can be zero so that a data transfer of only Immediate Data, or even no data, can take place. Zero length Scatter-Gather elements are also acceptable. The VI NIC must support a minimum segment count limit of at least 252 Data Segments.

6.1.1.2. Size Considerations

Data transfers can be of any length, not exceeding the Maximum Transmission Unit (MTU) of any VI Provider. The sum of the lengths of the Scatter-gather elements is the total data length.

The minimum allowable MTU is 32KB. This guarantees the VI Consumer that any VI Provider can transmit at least 32KB in a single Descriptor. Some vendors may choose to support an MTU larger than 32KB. Each VI Provider should supply a mechanism by which a VI Consumer can determine the MTU supported by the Provider.

6.2. Posting Descriptors

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. The format and operation of the Doorbell is specific to each VI Provider.

Send and Receive requests are posted to their respective Work Queues. Remote DMA requests, both Write and Read, are always posted to the Send Queue.

Receive Descriptors may be posted before a connection is established, but they will not complete, except in error cases, until the VI is connected and data is received. Pre-posting Descriptors can prevent error conditions, specifically data being dropped or rejected due to lack of receive Descriptors.

Send and RDMA Descriptors that are submitted to a VI before a connection is established will be completed in error. They cannot be processed until the receiving endpoint is defined, and thus are considered an error.

Posting a Descriptor does not block the processing of already posted Descriptors.

6.3. Processing Descriptors

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. Immediate data is transferred at this time as well.

If an error is encountered during processing of the Descriptor, the VI NIC is expected to take the appropriate action, according to the type of error encountered and the reliability level of the Connection.

6.3.1. Ordering Rules and Barriers

The order of processing of Descriptors posted to the VI Work Queues must be consistent across all VI Architecture implementations. These rules apply to the processing on a single queue. Ordering across multiple Work Queues is undefined, but is expected to provide fairness.

6.3.1.1. Ordering Rules

Receive and Send queues are FIFO queues. Descriptors are enqueued and dequeued in order. The VI Provider does not reorder queues.

Receive queues are strict FIFO queues. Once enqueued, Descriptors are processed, completed, and dequeued in FIFO order.

Send queues are FIFO queues. Once enqueued, Descriptors begin processing in order, but may complete out of order. Descriptors are always dequeued in FIFO order regardless of completion order. Send and RDMA Write Descriptors are completed in strict FIFO order. RDMA Read Descriptors may complete out of order with respect to Send and RDMA Write Descriptors. RDMA Reads complete in FIFO order with respect to one another.

Unlike Sends and RDMA Writes, RDMA Reads require a round trip path. For performance reasons, it is not optimal to specify that Descriptor processing is stopped on a given queue when an RDMA Read is issued until after the data are returned. This would make queue processing on Reliable Delivery connections dependent on remote node processing capability. Therefore, Sends and RDMA Writes may begin processing before RDMA Reads have completed.

A Send that bypasses an RDMA Read may write data into the receiving memory regions at any time prior to the completion of the RDMA Read. The contents of the pended receive buffer memory are unpredictable until the associated remote Receive queue Descriptor is completed successfully. On a Reliable Delivery connection, the bypassing Send may consume a remote Receive queue Descriptor and be completed successfully prior to the completion of the RDMA Read. The remote VI Consumer may receive the bypassing Send successfully prior to the completion of the RDMA Read and/or in the event of an error on the RDMA Read. On a Reliable Reception connection, the receiving VI Provider must not successfully complete the bypassing Send until after all RDMA Read data have been transmitted without error. In the event of an error on the RDMA Read, no remote Receive queue Descriptor is consumed.

An RDMA Write that bypasses an RDMA Read may write data into the remote memory region only on a Reliable Delivery connection. In the event of an error on the RDMA Read, the bypassing RDMA Write may have completed successfully and have side effects. It is the responsibility of the issuing VI Consumer to dequeue all commands and take appropriate action. No such side effects are permitted on a Reliable Reception connection. The remote VI Provider must not modify the contents of the remote memory region or, in the case of immediate data, consume a Receive queue Descriptor until after the RDMA Read has completed. In the event of an error on the RDMA Read, the RDMA Write must have no remote effect.

Sends and RDMA Writes always complete in the order in which they are queued.

- Sends do not pass sends
- Sends do not pass RDMA Writes
- RDMA Writes do not pass sends
- RDMA Writes do not pass RDMA Writes

RDMA Reads do not necessarily complete before Sends or RDMA Writes queued after them begin processing.

- RDMA Reads do not pass sends
- RDMA Reads do not pass RDMA Writes
- RDMA Reads do not pass RDMA Reads
- Sends can pass RDMA Reads
- RDMA Writes can pass RDMA Reads

If strict order is required, the VI Consumer will have to use a processing barrier. This would be true in the case of a VI Consumer using a Reliable Delivery connection desiring that a Send following an RDMA Read notify the connecting VI Consumer that the buffer is now free to reuse.

The above rules provide ordering guarantees to VI consumers. VI implementers are permitted to implement more strict ordering rules than those exposed above.

6.3.1.2. Barriers

Barriers are a way to indicate to the VI NIC that all Descriptors before the designated Descriptor must complete before processing can continue.

By setting the Queue Fence bit in a Descriptor, the VI Consumer can ensure that all Descriptors posted on the Send Queue before that Descriptor have completed before processing begins on that Descriptor. This provides the VI Consumer with a synchronization mechanism to guarantee the contents of registered memory.

6.3.2. Address Translation and Memory Access

A VI NIC uses its Page Tables, combined with Memory Handles and Virtual Addresses in order to perform address translation. This information is combined with the Memory Protection Tag to ensure not only that the address is valid, but that the memory access is permitted by this VI and/or by incoming RDMA requests. All addresses of RDMA requests must be validated according to the incoming Handle and Virtual Address as well as the Memory Protection Tag and Memory Protection Attributes associated with the Handle and the VI before data transfer can take place.

6.4. Completing Descriptors

When data transfer has completed, the Descriptor must be completed. This is achieved in two phases. In the first phase, the VI Provider updates the Control Segment contents and, if the queue is linked to a Completion Queue, an entry is generated in the Completion Queue. In the second phase, the VI Consumer dequeues the Descriptor.

6.4.1. Completing Descriptors by the VI Provider

When the VI NIC has completed processing a Descriptor, it must update the information in the Control Segment of the Descriptor with completion codes, length fields and possibly immediate data.

When the completion indicator is set in the Descriptor, the VI NIC is signaling that it has finished updating all information in the Descriptor and the Descriptor has been removed from the VI NIC's internal processing queues. Note that the Status must be the last item updated to ensure that the Descriptor is not pulled prematurely. The Done bit in the Status is used for synchronization.

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 VI Consumer is responsible for ensuring that a Completion Queue is large enough to hold a Completion Queue entry for each Descriptor that may complete at any given time. The VI Provider is not required to report an error if the Completion Queue Overruns. If a Completion Queue overruns, completion entries will be lost.

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 into the *Error* state as per the discussions in Sections 2.5 and 5. The Receive Descriptor is marked in error with a length error.

If an error occurred, the Descriptor is marked with the appropriate error status and the VI will transition to a new state according to the discussions in Sections 2.5 and 5.

6.4.1.1. Kernel Agent Interactions

The Kernel Agent is invoked during normal processing only if the VI NIC generates an interrupt. The VI NIC always generates an interrupt on asynchronous errors. An interrupt is also generated when a VI Consumer thread is blocked in the VI Provider, waiting for a Descriptor at the head of a queue to complete.

If the VI Queue or Completion Queue is enabled for interrupts and the VI NIC has completed processing a Descriptor for that VI Queue or Completion Queue, then the VI NIC will generate an interrupt. This action allows the Kernel Agent to unblock any VI Consumer threads waiting for completions.

If the interrupt is due to an error that is valid for the reliability level of this VI, an asynchronous error handling mechanism is invoked to deliver the error to the VI Consumer.

6.4.2. Completing Descriptors by the VI Consumer

The VI Consumer may synchronize on completed Descriptors in the following ways:

- Poll a VI Work Queue
- Wait on a VI Work Queue
- Poll a Completion Queue
- Wait on a Completion Queue

A VI Consumer polls on a Work Queue by reading the Status field of the Descriptor at the head of the queue. When the Done bit is set, the Consumer may dequeue the Descriptor. A VI Consumer polls on a Completion Queue by examining the head of the Queue. When a Descriptor is completed, information describing it will be written to the Completion Queue. The VI Consumer uses this information to find the appropriate Work Queue and dequeue the head Descriptor.

A VI Consumer can wait for completions on a Work Queue or a Completion Queue through a mechanism supplied by the Kernel Agent. This mechanism involves informing the NIC that an interrupt should be generated for the next completion on the specified Work Queue or Completion Queue. The Kernel Agent fields this interrupt and unblocks the appropriate thread. The thread then processes the completion as if it had polled for it.

7. Error Handling

Most errors are reported synchronously to the VI Consumer, either at a time an operation is attempted, or when a Descriptor completes. Some errors can not be reported synchronously.

Asynchronous errors include those that cause queue processing to hang, those that occur after a Descriptor has been completed, and errors requiring immediate intervention such as a network cable disconnection. A VI Provider should supply a mechanism by which asynchronous errors may be delivered to a VI Consumer. Some errors may also be logged according to standard operating system conventions.

The asynchronous errors reported by a VI Provider vary according to the reliability level of the connection, as defined in the following sections. Certain types of errors will be reported asynchronously regardless of the reliability level of the connection. These include hardware-related issues that require immediate notification.

7.1. Error Handling for Unreliable Connections

The asynchronous error handling mechanism for Unreliable Connections is only invoked in the case of a catastrophic hardware error. This includes queue hangs, VI NIC errors, or link loss.

This class of connection assumes that the VI Consumer will implement any appropriate protocol logic deemed necessary to ensure packet arrival, so no attempt is made to determine transport level issues or to report them to the VI Consumer.

7.2. Error Handling for Reliable Delivery Connections

The asynchronous error handling mechanism for Reliable Delivery Connections is invoked in the case of catastrophic transport or hardware errors. This includes queue hangs, VI NIC errors or link loss, and catastrophic transport oriented errors such as dropped or missed packets.

Upon any error, the Connection is placed into the *Error* state and will behave according to the rules discussed in Section 5.4. The error handling mechanism is also invoked. The VI Consumer should attempt to clean up and take whatever action it deems necessary, according to the conditions of the error.

For Reliable Delivery VIs, the asynchronous error handling mechanism will be invoked for transport errors after *one* or *more* posted Descriptors packets have been completed by the VI NIC.

7.3. Error Handling for Reliable Reception Connections

The asynchronous error handling mechanism for Reliable Reception Connections is invoked in the case of catastrophic transport or hardware errors. This includes queue hangs, VI NIC errors or link loss, and transport oriented errors such as dropped or missed packets.

Upon any error, the Connection is placed into the *Error* state and will behave according to the rules discussed in Section 5.4. The error handling mechanism is also invoked. The VI Consumer should attempt to clean up and take whatever action it deemed necessary, according to the conditions of the error.

For Reliable Reception Connection, the asynchronous error handling mechanism will be invoked for transport errors after *no more than one*, and possibly zero, posted Descriptors have been completed by the VI NIC.

8. Guidelines

VI Consumers should be aware of the trade-offs in time and resources when using the VI Architecture. The VI Architecture optimizes only the data transfer path. Some VI Provider operations are inherently more time consuming than data transfers. These operations are:

- Buffer Registration requires a kernel transition, locking down of the physical pages in memory and manipulation of the VI NIC page tables. De-registration is also time consuming. It is recommended that memory regions are registered infrequently by the VI Consumer and that space within the memory region is judiciously managed.
- Creation of a VI requires a kernel transition, allocation of resources in the kernel and in the VI NIC, and interaction with the VI NIC so that a Doorbell can be created and mapped into the VI Consumer's address space.
- Connection of a VI to another VI implies execution of a protocol between endpoints and consists of several operations by the VI Consumer and VI Provider.
- Transfer of data that is poorly aligned could result in longer transfer times.

Resources available on any given VI NIC are finite.

- The number of VIs that a VI NIC supports is finite.
- There may be side effects in over-using the registration/de-registration process such as fragmentation of the VI NIC's page tables.
- Registered memory consumes non-pageable memory on the host node.
- The number of Completion Queues that a VI NIC supports is finite.
- A VI Consumer will have to perform segmentation and reassembly to transfer data larger than the MTU supported by the VI Provider.

8.1. Scalability

VI Architecture implementers should be especially concerned with scalability considerations. The interconnect mechanisms, which include the VI NIC, the physical media and any switch or connection facilitator, need to scale with the number of instances of the application as well as the number of applications running within a node. The throughput of the network must scale as well. Fairness and scalability must be ensured at all levels to adequately grow the number of active connections.

A usage model of (1+M)(N-1) VIs per node is suggested as a minimum number of VIs supported by an implementation, where N is the number of nodes in a cluster and M is the number of distinct processes consuming VIs on each node. This allows for one VI in kernel mode, and for each application to consume one VI to establish a connection to every other node in the cluster.

A sample usage of this formula is a 16 node cluster with three distributed applications running across all nodes. This would result in (1+3)(16-1) or 100 VIs open in a given machine. This is the recommended minimum number of VIs that a VI NIC should consider supporting, with 1024 VIs being a more reasonable number to allow for growth and application flexibility.

9. Appendix A

9.1. Example VI User Agent Overview

This section describes an example functional interface to the VI Architecture. It is meant as an aid to hardware implementers, to illustrate a reasonably complete software embodiment of the VI Architecture. The material is presented in the form of groups of related routines, followed by definitions of data structures, constants and error codes.

9.2. Hardware Connection

9.2.1. 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.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_ERROR_RESOURCE – An error was detected due to insufficient resources.

VIP_INVALID_PARAMETER – One of the parameters were invalid.

9.2.2. 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.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – Caller specified an invalid NIC handle.

9.3. Endpoint Creation and Destruction

9.3.1. 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 Attributes 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 – 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.

9.3.2. 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_ERROR_RESOURCE – The VI was not in the Idle state or there are still Descriptors posted on the work queues.

9.4. Connection Management

9.4.1. 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.

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 that is requesting a connection.

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

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

Description

VipConnectWait is used to look for incoming connection requests.

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 LocalAddress, the caller is returned the remote address 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*.

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 connection operation failed due to a resource limit.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

9.4.2. 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.

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.

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.

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.

9.4.3. 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. Notification is sent to the remote end that the associated connection request was rejected.

Returns

VIP_SUCCESS – The operation completed successfully.

VIP_INVALID_PARAMETER – The ConnHandle parameter was invalid.

9.4.4. 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.

RemoteAddr: The remote network address.

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. The user specifies a local and remote network address for the connection.

If a connection is successfully established, the specified 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, 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 timeout error.

Returns

VIP_SUCCESS – The connection was successfully established.

VIP_TIMEOUT – The connection operation timed out.

VIP_ERROR_RESOURCE – The connection operation failed due to a resource limit.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

VIP_REJECTED – The connection was rejected by the remote end.

9.4.5. 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 posted Descriptors are marked completed because of disconnection, and the local endpoint transitions to the Idle state.

Returns

VIP_SUCCESS – The disconnect was successful.

VIP_INVALID_PARAMETER – The ViHandle parameter was invalid.

9.5. Memory protection and registration

9.5.1. VipCreatePtag

Synopsis

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

Parameters

NicHandle: The NIC handle associated with the protection tag.

ProtectionTag: 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 a resource limit.

VIP_INVALID_PARAMETER – One of the parameters was invalid.

9.5.2. VipDestroyPtag

Synopsis

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

Parameters

NicHandle: The NIC handle associated with the protection tag.

ProtectionTag: 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.

VIP_INVALID_PARAMETER – One of the input parameters was invalid.

9.5.3. 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 a resource limit.

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.

9.5.4. 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 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.

9.6. Data transfer and completion operations

9.6.1. 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.

9.6.2. 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. Otherwise an error is returned, and the contents of DescriptorPtr are invalid.

Returns

VIP_SUCCESS – A completed Descriptor was returned.

VIP_NOT_DONE – No completed Descriptor was found.

VIP_INVALID_PARAMETER – The VI handle was invalid.

9.6.3. 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

VipSendWait 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.

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.

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.

9.6.4. 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.

9.6.5. 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. Otherwise an error is returned and the contents of DescriptorPtr are invalid.

Returns

VIP_SUCCESS – A completed receive Descriptor was returned.

VIP_NOT_DONE – No completed Descriptor was found.

VIP_INVALID_PARAMETER – The VI handle was invalid.

9.6.6. 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.

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 Descriptor was found on the receive queue.

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.

9.6.7. 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 TRUE, indicates that the completion was associated with the receive queue of the VI. If 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.

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.

9.6.8. 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 TRUE, indicates that the completion was associated with the receive queue of the VI. If 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*.

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_INVALID_PARAMETER.

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.

9.6.9. 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 the 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 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.

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.

Returns

VIP_SUCCESS – The routine was successfully completed.

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

VIP_ERROR_RESOURCE – The send queue of the VI is associated with a Completion Queue.

9.6.10. 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 the 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 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.

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.

Returns

VIP_SUCCESS – The routine was successfully completed.

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

VIP_ERROR_RESOURCE – The receive queue of the VI is associated with a Completion Queue.

9.6.11. 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: If TRUE, indicates that the completion was associated with the receive queue of the VI. If 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 the 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 is invoked with the ViHandle and RecvQueue 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.

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.

9.7. Completion Queue Management

9.7.1. 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 how many completion entries that the queue must contain. If successful, it returns a handle to the newly created Completion Queue.

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.

9.7.2. 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.

9.7.3. 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 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.

9.8. Querying Information

9.8.1. VipQueryNic

Synopsis

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

Parameters

`NicHandle:` The handle of a VI NIC.

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

Description

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

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The NIC handle was invalid.

9.8.2. VipSetViAttributes

Synopsis

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

Parameters

ViHandle: The handle of a VI instance.

Attributes: 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.

Returns

VIP_SUCCESS – Operation completed successfully.

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.

9.8.3. VipQueryVi

Synopsis

```
VIP_RETURN
    VipQueryVi(
        IN    VIP_VI_HANDLE        ViHandle,
        OUT   VIP_VI_STATE         *State,
        OUT   VIP_VI_ATTRIBUTES    *Attributes
    )
```

Parameters

ViHandle: The handle of a VI instance.

State: The current state of the VI.

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

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.

9.8.4. 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.

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.

9.8.5. 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.

9.8.6. VipQuerySystemManagementInfo

Synopsis

```
VIP_RETURN
        VipQuerySystemManagementInfo(
                IN      VIP_NIC_HANDLE      NicHandle,
                IN      VIP_ULONG           InfoType,
                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.

Returns

VIP_SUCCESS – Operation completed successfully.

VIP_INVALID_PARAMETER – The NIC handle was invalid.

9.9. Error handling

9.9.1. 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 according to operating system conventions.

If an error handling function has been specified via the *VipErrorCallback* function, the default error handling function can be restored by calling it with an Handler parameter of NULL.

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 – The virtual address and memory handle of the Descriptor was not valid when attempting to post a Descriptor.
- 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 the remote end of an 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 the remote end of an 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 the remote end of an 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 Read Protection Error – A protection error was detected on the remote end of an 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.

Returns

VIP_SUCCESS – Operation completed successfully.

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

9.10. Data Structures and Values

9.10.1. 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.
- 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.

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_RETURN
```

9.10.2. 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_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_IMMEDIATE               0x0004
#define     VIP_CONTROL_QFENCE                  0x0008
```

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
```

9.10.3. 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.
- Resource code – Allows the application to tell if the error was due to a NIC problem, VI problem, queue problem or Descriptor problem.
- VI handle – If non-NULL, refers to the VI instance 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.
- Descriptor pointer – If non-NULL, refers to the Descriptor related to the error.
- 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_CODE
```

9.10.4. 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.
- Hardware Version – 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.
- 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.
- 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 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 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.

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_NIC_ATTRIBUTES
```

9.10.5. 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 – As input parameter, it is the requested maximum transfer size for this connection. As output parameter, it is the granted Maximum Transfer Size for the connection. The Transfer Size specifies the amount of payload data that can be transferred in a single VI packet.
- QoS – As input parameter, it is the requested quality of service for the connection. As output parameter, it is the granted quality of service for the connection.
- Ptag – The protection tag to be associated with the VI.
- EnableRdmaWrite – If TRUE, accept RDMA Write operations on this VI from the remote end of a connection.
- EnableRdmaRead – If 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
```

The possible values for VIP_RELIABILITY_LEVEL are:

```
typedef enum {
        VIP_SERVICE_UNRELIABLE,
        VIP_SERVICE_RELIABLE_DELIVERY,
        VIP_SERVICE_RELIABLE_RECEPTION
} VIP_RELIABILITY_LEVEL
```

9.10.6. 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 TRUE, allow RDMA Write operations into this registered memory region.
- EnableRdmaRead – If 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
```

9.10.7. 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
```

9.10.8. 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.

10. Appendix B

10.1. Example Descriptor Format Overview

This Appendix describes an example 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.

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 7. The format of an RDMA Descriptor is shown in Figure 8.

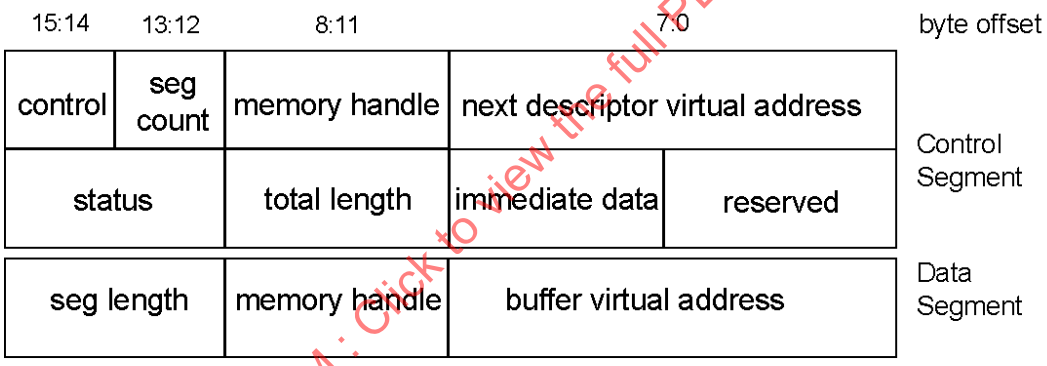


Figure 7: Send and Receive Descriptor Format

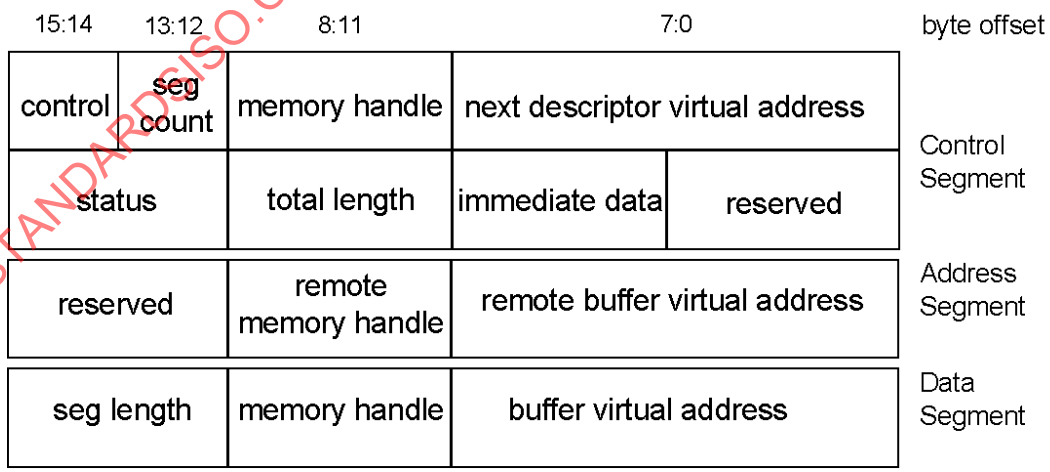


Figure 8: RDMA Descriptor Format

10.2. 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 Consumer fills in this field in the Descriptor that is currently the tail of a queue to add a new Descriptor to the queue.

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

This field is the matching memory handle for the Next Descriptor Virtual Address. A VI Consumer 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 Consumer 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 Consumer 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. Once the Descriptor is completed, this bit is used to indicate that the Immediate Data contains valid data that was sent from the connected VI.

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. This feature is discussed in section 6.3.1.2.

Control field bits 15:4: Reserved

These bits are reserved for future use. They must be set to zero by the VI Consumer 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 Consumer 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 Consumer in the case of Send and RDMA Write operation and is returned to the VI Consumer 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 Consumer sets this field when formatting the Descriptor. For send Descriptors, 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.

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 Consumer 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 Consumer to easily check for successful completion or for completion in error. Bits 16:31 contain flags to provide additional information to the VI Consumer. The VI Consumer must set this field to zero before posting a Descriptor.

The format for the Status Field is shown below.

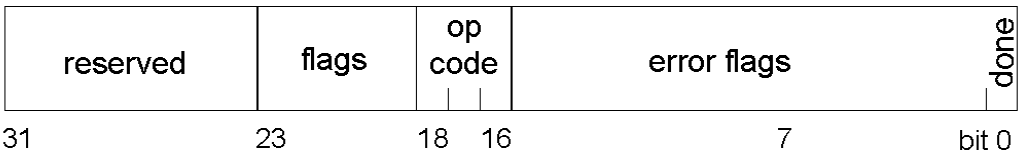


Figure 9: Status Field Format

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, as discussed in Section 2.5.

Status Field, bit 1: Local Format Error

This field is set if the locally posted Descriptor has a format error. This includes errors such as invalid operation codes, reserved fields set by the software and invalid VI Identifiers. 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 field 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, or insufficient rights for the requested operation.

This bit is valid on all Descriptor and Connection types.

Status Field, bit 3: Local Length Error

This field is set if the sum of the locally posted Descriptor's Data Segment lengths exceed the VI NIC's MTU on a Descriptor posted to the send queue. It will also be set if the total of the locally posted Descriptor's data segment lengths does not match the control segment length field of a Descriptor posted to the send queue.

This bit will be set if the total of the locally posted Descriptor's data segment lengths is too small to receive the incoming packet for Descriptors posted to the Receive Queue.

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. If this bit is set, the VI has transitioned to the *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.

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.

10.3. 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.

10.4. 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 Consumer 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 MTU of the VI NIC 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 Consumer.

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.

11. Appendix C

11.1. Example Hardware Model Overview

This Appendix describes an example hardware model for a VI NIC. Also included is a discussion of the associated VI Kernel Agent. The reader can use this chapter to help solidify the architectural concepts previously discussed. Implementers may wish to use the example model as a starting point for their own design.

11.2. Example VI NIC

This section describes an example hardware model for a VI NIC.

The VI Architecture relies on a significant portion of the functionality to be implemented in VI NIC hardware to achieve the lowest communication latency. The following diagram shows the example VI NIC hardware model:

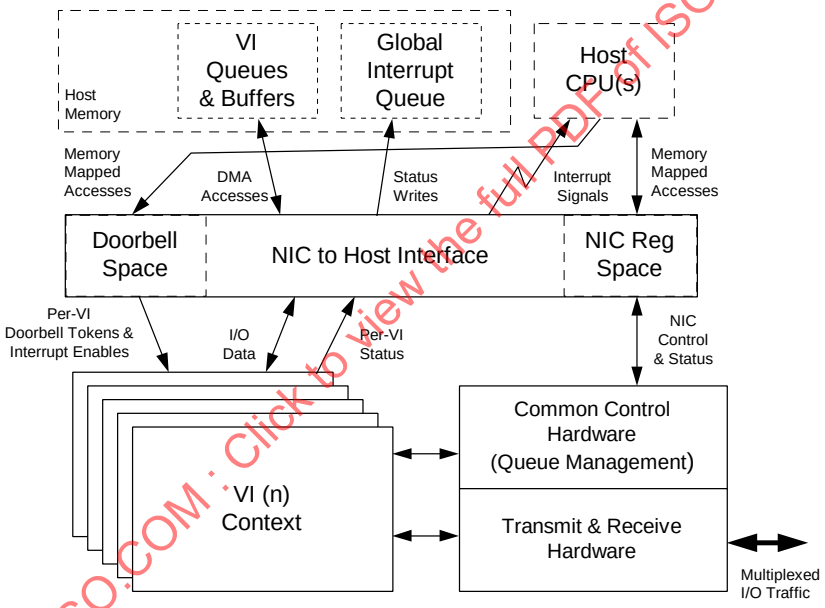


Figure 10: VI NIC Hardware Model

This hardware controls a set of Virtual Interfaces and schedules (multiplexes) between them in an order determined by a transmit scheduling mechanism and an input data stream.

It virtualizes the hardware interfaces and associates each with a VI by storing context for each VI and each direction of transfer. When the common hardware executes a time slice for a given VI and direction, it uses the corresponding context to control the operation of the common hardware.

It multiplexes data traffic from host memory through the NIC out to the network in an order determined by a transmit scheduler and keeps the transmit hardware fed from host memory accordingly.

It executes virtualized receive interfaces and feeds receive data to host memory in an order determined by the order of packets received from the network.

It manages the network side of all VI queues and directly accesses queues and buffers in host memory through DMA transactions.

It recognizes transactions to Doorbell pages within the host's physical address space and in response, tracks the number of Descriptors posted on each VI queue.

It translates virtual address information received in Descriptors, Doorbell tokens, and RDMA pseudo addresses to physical addresses and ensures that the owner of the associated VI also owns the physical memory addressed.

It provides asynchronous notification of significant events to the host through DMA writes of per-VI interrupt status words to a global interrupt queue in host memory. In addition, when necessary, it uses an interrupt signal to invoke execution of the interrupt handler of its associated VI Kernel Agent.

It enables the host to access registers and memories on the NIC via programmed I/O transactions by kernel level driver software to provide overall control and initialization of the hardware resources.

11.2.1. Hardware Interface

11.2.1.1. Address Translation

Memory is registered with the VI NIC for two reasons:

1) to allow the NIC to perform virtual to physical address translation
2) to allow the NIC to perform protection checking.

Consumers are able to use virtual addresses to refer to VI Descriptors and communication buffers. The VI NIC is able to translate from virtual to physical addresses through the use of its Translation and Protection Table (TPT). The TPT of the example NIC resides on the NIC in order to assure fast, non-contentious access and because it is accessed during performance critical data movement. The fields of each TPT entry are:

a) a physical page address
b) a protection tag
c) an RDMA Write Enable Bit
d) an RDMA Read Enable Bit
e) a Memory Write Enable Bit

The size of the TPT is configurable. There is one entry in the TPT for each page that can be registered by the user. A memory region of N contiguous virtual pages consumes N contiguous entries in the TPT.

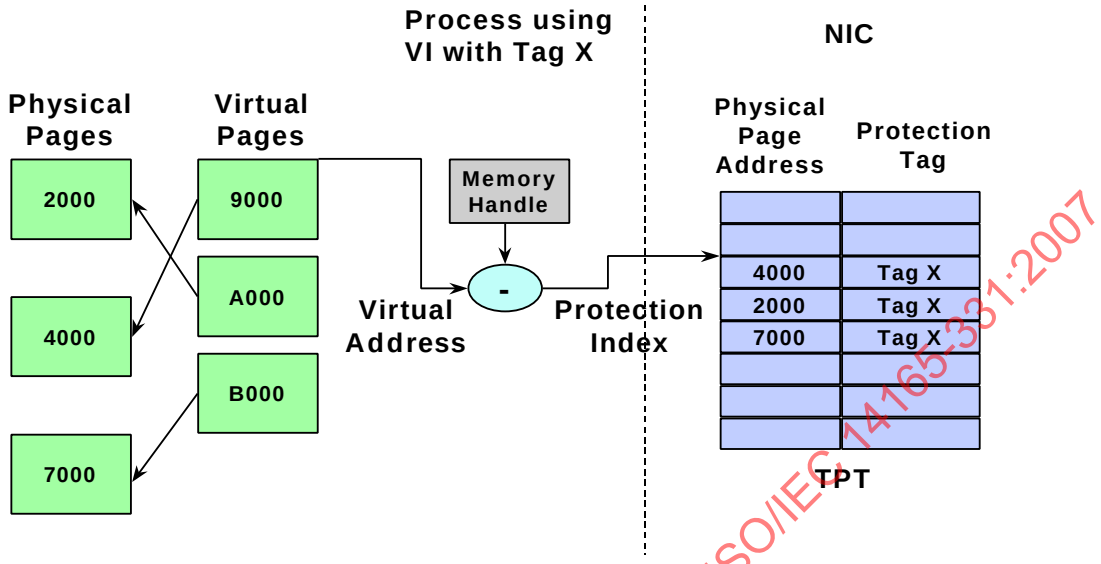


Figure 11: Translation and Protection Table

When a memory region is registered with the NIC, the Kernel Agent allocates a contiguous set of entries from the TPT and initializes them with the corresponding physical page addresses and protection tag specified by the process that registered the memory region. The protection tag specified by the process when it creates a VI is stored in the context memory of the VI. The NIC has access to the protection tag in both of these areas, allowing it to compare these values to detect invalid accesses

Page sizes larger than 4KB are supported and page size may differ among nodes of the SAN. For page sizes larger than 4KB the low order bits of the memory handle are sacrificed to make room for an offset field larger than 12 bits. This reduces the number of pages that can be mapped, but the amount of memory that can be registered is constant regardless of the page size

On a node with a page size larger than 4KB the 32-bit memory handle is returned from the register memory operation with the corresponding low order bits set to 0. The example hardware model has a TPT with 2^{32} entries. Only 2^{44} bytes of the virtual address space may be mapped by the TPT at any time; i.e., the number of bits used for the offset and the significant number of bits in the handle are a total of 44 bits.

A pseudo address is an internal construct that supports remote-DMA operations. The pseudo address is transmitted from the RDMA initiator (in a control field of the RDMA request) to specify the remote address to read from or write to. Pseudo addresses are formed from virtual addresses by multiplying the corresponding protection index (see Figure 11) by the page size and adding the page offset.

Because of the scheme described above for support of page sizes larger than 4KB, the calculation of a pseudo address can be done without knowing the page size of the remote node.

11.2.1.2. Doorbells

A Doorbell is a window in memory that allows a process to inform the NIC that a new Descriptor is available on a Work Queue. There is a Doorbell for every Work Queue. With respect to a process, a Doorbell is a location in its virtual address space. With respect to the NIC, a Doorbell

is a memory mapped control register. The Kernel Agent provides Doorbell mappings for all processes and Work Queues.

To ring a Doorbell software writes a Doorbell token to a Doorbell register. Upon receiving the Doorbell token the NIC increments a counter of outstanding requests on the associated Work Queue. Whenever the NIC completes processing of a Descriptor it decrements the counter of outstanding requests. The counter allows the NIC to determine whether there is work pending on a Work Queue.

The format of the Doorbell token is shown in Table 2 below. The Doorbell is 64 bits long to support up to 64 bit addressing. If the host supports atomic 64 bit writes then the entire Doorbell can be written at once. If the host only supports 32 bit atomic writes then only the low-order 32 bits of the Doorbell can be written and addressing is limited to 32 bits. When the NIC sees a 32-bit write to the Doorbell it assumes the high-order 32 bits of the token are zero.

Table 2: Doorbell Token Format

Bits	Use	Description
63:44	Unused	
43:X	Protection Index	The protection index corresponding to the registered virtual address of the Descriptor for the current data movement operation request.
X-1:6	Descriptor Offset	The offset, in 64-byte increments, into the physical page where the newly posted Descriptor starts. The physical page size for the host system determines how many bits are in this field.
5:0	Reserved	Reserved for future use.

Where X is the number of bits in the offset portion of a virtual address; e.g., 12 for a 4KB page size

Descriptors are posted by en-queuing them on the tail of a send or receive Work Queue and writing a token to the VI's corresponding send or receive Doorbell. Doorbell tokens are formed using the following calculation:

Doorbell Token = Virtual Address of Descriptor - (Handle << 12)

11.2.1.3. Marking Completion

When the NIC finishes processing a Descriptor it writes the status and length back to the Descriptor. Included in the status is the 'done' bit, which passes control back to the VI Consumer from the VI NIC.

If there is a Completion Queue associated with the VI then information identifying the VI and queue (i.e., send or receive) on which the operation completed is written to the next Completion Queue entry and the Completion Queue pointer is incremented.

11.2.1.4. NIC Context

11.2.1.4.1. Per-VI NIC Context

For each VI the NIC keeps the following context:

- Protection tag associated with the VI
- An RDMA Read enable bit
- An RDMA Write enable bit
- The Maximum Transfer Size for this VI

For each send and receive Work Queue on each VI the NIC keeps the following context:

- Count of outstanding operations on the queue
- Address of next Descriptor on the queue
- Reference to Completion Queue associated with the queue (NULL if none)
- Indicator as to whether interrupts are enabled for the queue (NA if a Completion Queue is associated with the VI)

11.2.1.4.2. Per-Completion Queue NIC Context

For each Completion Queue the NIC keeps the following context:

- Address of start of Completion Queue
- Number of entries in Completion Queue
- Address of next entry in Completion Queue
- Indicator of whether interrupts are enabled for this Completion Queue

11.2.1.5. Packet Format

A network packet consists of a header, a data payload, and a payload CRC. One network packet is generated for each Descriptor placed on a send Work Queue.

Note: The example hardware model assumes a single network frame per packet. A more complex implementation may break a packet into a series of cells to prevent a single large packet from tying up the network for long periods.

11.2.1.5.1. Packet Header

Each network packet contains a header consisting of the following fields:

- The VI for which the packet is intended
- An opcode specifying the operation to perform; i.e., SEND, RDMA Write, RDMA Read-Request, or RDMA Read-Reply
- Immediate data (NA for RDMA Read)
- Byte count; this is the length of the payload for send and RDMA/Write, and the number of bytes to read in the case of RDMA Read
- A pseudo address of the region to read or write (RDMA operations only)
- CRC covering the header

11.2.1.5.2. Data Payload

Concatenation of all the data buffers specified in the data segment of the send or RDMA/Write Descriptor. The data payload is of length 0 for RDMA Read-Request.

11.2.1.5.3. Payload CRC

CRC covering the data payload, it is ignored in the case of RDMA Read-Request.

11.2.2. NIC Hardware Functions

11.2.2.1. Transmit Hardware Functions

The transmit hardware for a VI NIC reads a Descriptor from host memory, generates the CRC-32 values for the header and trailer, assembles a complete physical layer frame and transmits the frame to the network. Each time a frame is transmitted the NIC writes completion status to the corresponding send Descriptor. If a Completion Queue is associated with the send Work Queue completion status is also written to the next Completion Queue entry. If interrupts are enabled for the send Work Queue the NIC issues an interrupt.

A round-robin transmit scheduling mechanism is used to ensure timely servicing of all active send queues and provide fairness in arbitration between VIs. This functionality must be implemented in hardware to provide efficient multiplexing between a large number of VIs.

11.2.2.2. Receive Hardware Functions

VI receive queues are serviced according to the sequence that frames are received from the VI fabric. The NIC attempts to always have the Descriptor segments for each active VI pre-fetched and ready for execution when the corresponding frame is received; note, however, that not all incoming frames consume a Descriptor.

The receive hardware reads the frame header from the network and checks the header CRC. Further processing of incoming frames depends on the opcode contained in the header:

Send: If the frame header control field indicates there is immediate data it is copied to the corresponding receive Descriptor's immediate data field. The NIC then sets up DMA operations to copy data from the frame payload into the memory regions specified by the receive Descriptor's data segments. When processing of a frame completes the NIC writes completion status to the receive Descriptor. If a Completion Queue is associated with the receive Work Queue then completion status is also written to the next Completion Queue entry. If interrupts are enabled for the receive Work Queue then the NIC issues an interrupt.

RDMA/Write: The NIC sets up DMA operations to copy data from the frame payload into the memory region specified by the RDMA pseudo address field of the frame header. If the frame's control field header indicates there is no immediate data associated with this RDMA/Write then no receive Descriptor is consumed, no completion status is written, and no interrupts are generated.

If the frame header control field indicates that there is immediate data it is copied to the corresponding receive Descriptor's immediate data field and the NIC writes completion status to the receive Descriptor. If a Completion Queue is associated with the receive Work Queue completion status is also written to the next Completion Queue entry. If interrupts are enabled for the receive Work Queue then the NIC issues an interrupt.

Read-Request: A Read-Request opcode indicates the initiation of an RDMA Read operation. The reference implementation does not specify RDMA Read operation.

Read-Reply: A Read-Reply opcode indicates the reply to an RDMA Read operation. The reference implementation does not specify RDMA Read operation.

11.2.2.3. Interrupt Support

The example VI hardware model provides a global interrupt queue. The NIC writes entries to the queue in response to events generated at both the VI level and the NIC level. Entries in the global interrupt queue contain an interrupt code and identify whether the interrupt was for a particular VI and Work Queue, Completion Queue or is related to the NIC as a whole.

The VI architecture requires support for two kinds of asynchronous events: completions and errors. The flow diagram in Figure 12 shows the hardware and software model for handling these events. Note that Figure 12 assumes that the thread entering the wait path has mutually exclusive

access to the VI work queue. Only those errors that cannot be reported through Descriptor status are reported asynchronously.

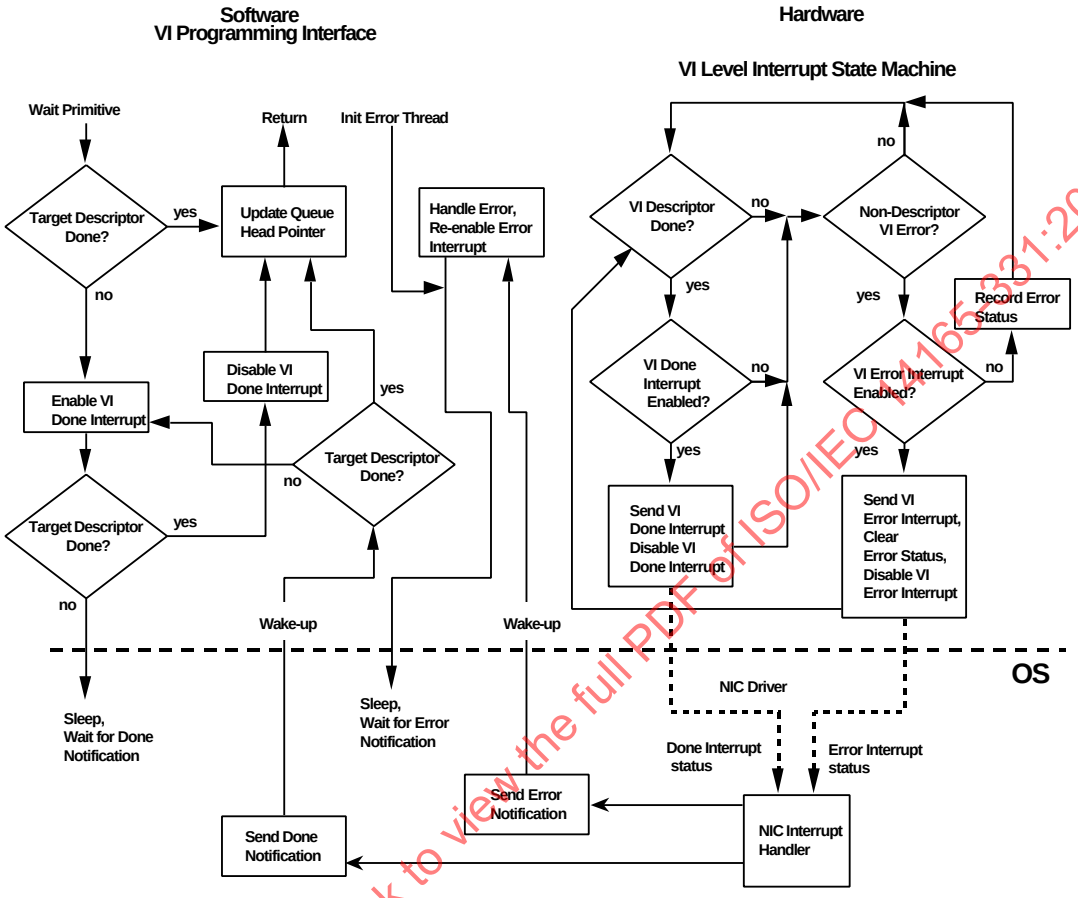


Figure 12: Asynchronous Event Model

11.3. Kernel Agent Example

This section is a functional description of a VI Kernel Agent associated with the example VI NIC hardware.

The VI Kernel Agent implements the set of software services inside the target operating system. It is implemented as a kernel-mode driver containing control and resource management functions. They reside in the kernel to provide centralized control and trusted operation using the facilities of the operating system itself.

Functionally, the services are:

- Network Device Control and Management
- Virtual Interface Resource Management
- Completion Queue Resource Management
- Host Memory Management
- Connection Management

- Asynchronous Error Delivery

11.3.1. NIC Initialization

The Kernel Agent is responsible for initializing its internal state, as well as the NIC hardware at the time that it is first invoked. Normally, the Kernel Agent will perform initialization at system initialization time, or at the time that it is loaded and initially entered by its host operating system.

11.3.2. Interrupt Processing

The Kernel Agent is responsible for the handling of all NIC device interrupts. When the Kernel Agent initializes, it must register itself with the OS in order to receive interrupts from its associated VI NIC. When interrupts are generated by the NIC, it must dispatch them to the proper thread that is waiting for the event.

11.3.3. Memory Registration

The memory registration function is implemented in the Kernel Agent. Kernel mode privilege is required in order to translate the caller's virtual address to physical addresses, to lock the pages into physical memory, and to access the protection entries of the VI NIC.

Refer to the example function *VipRegisterMem* in Section 9.5.3.

11.3.4. Memory De-registration

Memory de-registration is implemented in the Kernel Agent. Kernel mode privilege is required to unlock the caller's pages from physical memory and to invalidate the protection entries of the VI NIC.

Refer to the example function *VipDeregisterMem* in Section 9.5.4.

11.3.5. Setting and Querying Memory Attributes

The memory attributes for a registered memory region are managed in the Kernel Agent. It maintains the state of each memory region and sets the appropriate flags in the protection entries of the VI NIC.

The memory attributes that are visible to the VI Consumer are the Protection Tag, the RDMA Write Enable and the RDMA Read Enable. The example Kernel Agent also maintains an additional bit, the Memory Write Enable bit. The Memory Write Enable allows the Kernel Agent to check for pages that are marked as read-only by the Virtual Memory system, and protect them from being modified by the VI NIC.

Refer to the example functions *VipSetMemAttributes* and *VipQueryMem* in Section 9.

11.3.6. VI Creation

Creates a new Virtual Interface instance, allocates its resources and sets the initial state. It returns a VI identifier to the calling process along with the information needed in order to manipulate that VI.

Refer to the example function *VipCreateVi* in Section 9.3.1.

11.3.7. VI Destruction

Tears down a Virtual Interface, and frees any associated resources in kernel memory, as well as on the VI NIC. If the specified VI is not in the idle state, or if all of the Descriptors have not been de-queued from its work queues, the VI will not be destroyed.

Refer to the example function *VipDestroyVi* in Section 9.3.2.

11.3.8. Setting and Querying VI Attributes

The Kernel Agent manages the attributes of a VI. It maintains the state of each VI instance and sets the appropriate state in the VI NIC. The initial attributes of the VI are set when the VI is created. VI attributes can subsequently be changed. If the VI is in a state such that changing the attributes would cause non-deterministic behavior, the request to change the attributes fails. Querying the attributes of a VI simply passes the current values back to the caller.

Refer to the example functions *VipSetViAttributes* and *VipQueryVi* in Section 9.

11.3.9. Protection Tag Creation

Creates a new protection tag for a specified NIC instance.

Protection tags must be unique identifiers for a particular instance of a NIC. Protection tags are subsequently associated with VI endpoints, and with registered memory regions. The protection tag needs a reference count to ensure that they cannot be destroyed while associated with an active VI, or with a registered memory region.

Refer to the example function *VipCreatePtag* in Section 9.5.1.

11.3.10. Protection Tag Destruction

Destroys a previously created protection tag. A protection tag should not be destroyed unless it has no current associations to VI instances or registered memory regions.

Refer to the example function *VipDestroyPtag* in Section 9.5.2.

11.3.11. Connection Management

The Kernel Agent implements all connection management operations.

Refer to the example functions *VipConnectWait, VipConnectRequest, VipConnectAccept and VipConnectReject* in Section 9.4.

11.3.12. Block on Send and Receive

For the example VI NIC, the blocking semantics for send are implemented directly in the Kernel Agent. The block on send function blocks the calling thread until a done interrupt is generated by the NIC for the associated VI Work Queue.

11.3.13. Create Completion Queue

Creates a new Completion Queue instance, allocates its resources and sets the initial state. It returns a Completion Queue identifier to the calling process along with the information needed in order to manipulate that Completion Queue.

Refer to the example function *VipCreateCQ*.

11.3.14. Resize Completion Queue

The kernel agent implements the resizing of Completion Queues. This operation allows the VI Consumer to dynamically grow or shrink the number of entries that the Completion Queue can potentially hold. The Kernel Agent is responsible for allocating the resources and registering them with the NIC in support of the Completion Queue structure.

For the hardware example, three steps are performed by this function:

1. A new Completion Queue is created.

2. The NIC is informed of the new Completion Queue parameters, such as the queue address and size; once it knows of the new Completion Queue the NIC places all subsequent completion status in the new queue.

3. At the point that software next checks the original previous Completion Queue and finds no completed entry, software starts using the new queue, and can destroy the previous queue. This solution requires that software must be able to atomically change the Completion Queue address and length parameters stored in the NIC.

Refer to the example function *VipResizeCQ* in Section 9.7.3.

11.3.15. Block on Completion Queue

In this reference, the blocking semantics for Completion Queues is implemented directly in the Kernel Agent. The block on Completion Queue function blocks the calling thread until a done interrupt is generated by the NIC for the associated VI.

11.3.16. Destroy Completion Queue

Destroys a Completion Queue instance and de-allocates its resources.

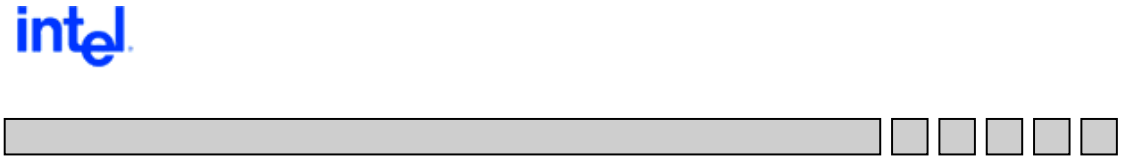
Refer to the example function *VipDestroyCQ* in Section 9.7.2.

11.3.17. Error Callback

It is the responsibility of the Kernel Agent to associate completion events with Consumer specified error handler functions, and to deliver all asynchronous errors to the Consumer.

11.3.18. Resource cleanup

When a process exits, a mechanism must be provided to allow the Kernel Agent to de-allocate all of its resources associated with that VI Provider. Normal OS mechanisms must allow the Kernel Agent to be notified when a process exits so that it can do resource cleanup. The Kernel Agent keeps track of all per-process resources associated with a Consumer's access to a NIC.



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.

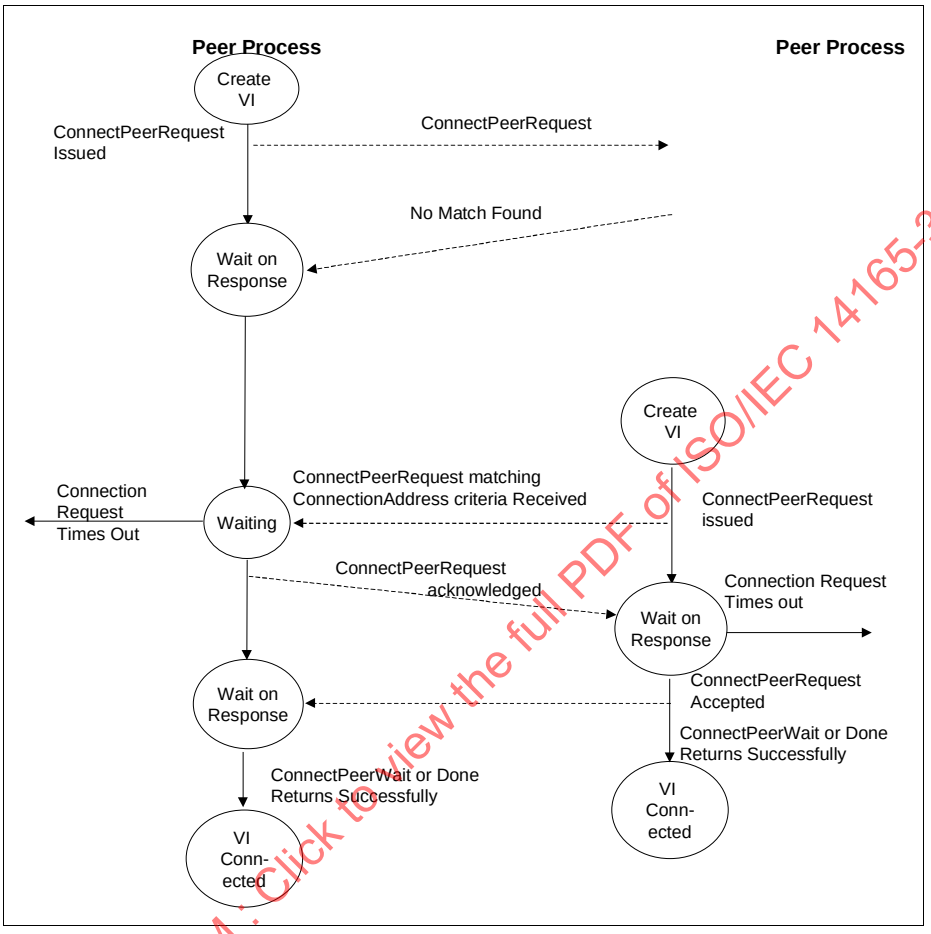


Figure 1: Peer-to-Peer Connection Model Timing Diagram

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.

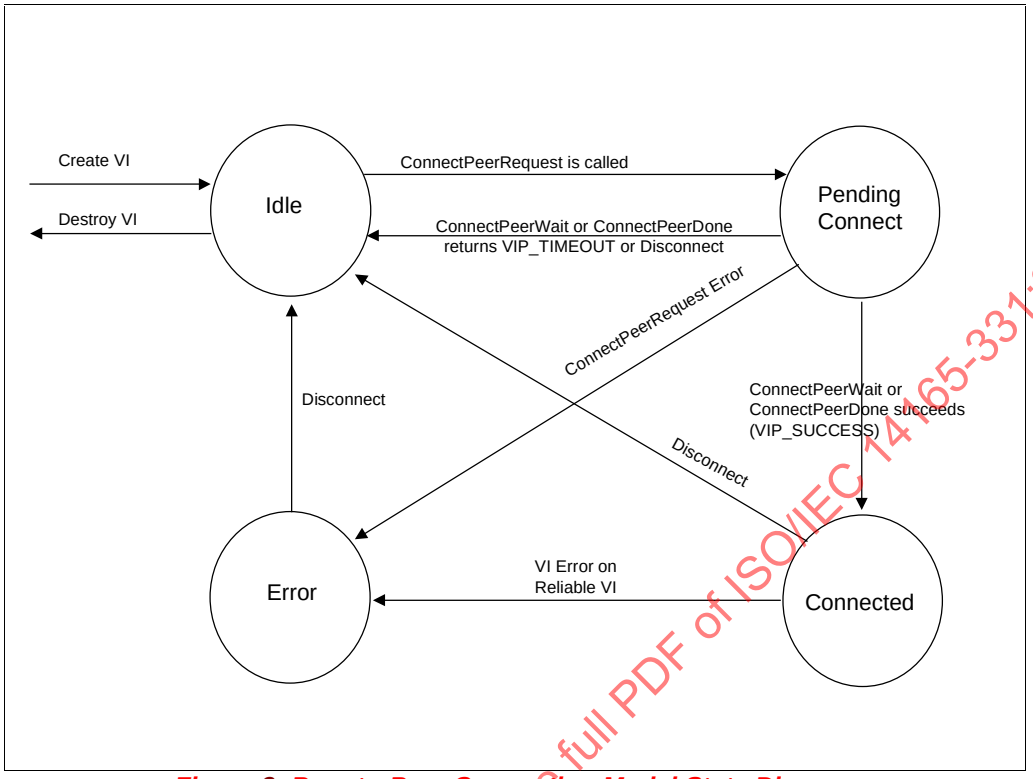


Figure 2: Peer-to-Peer Connection Model State Diagram

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_QUENCE                  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.

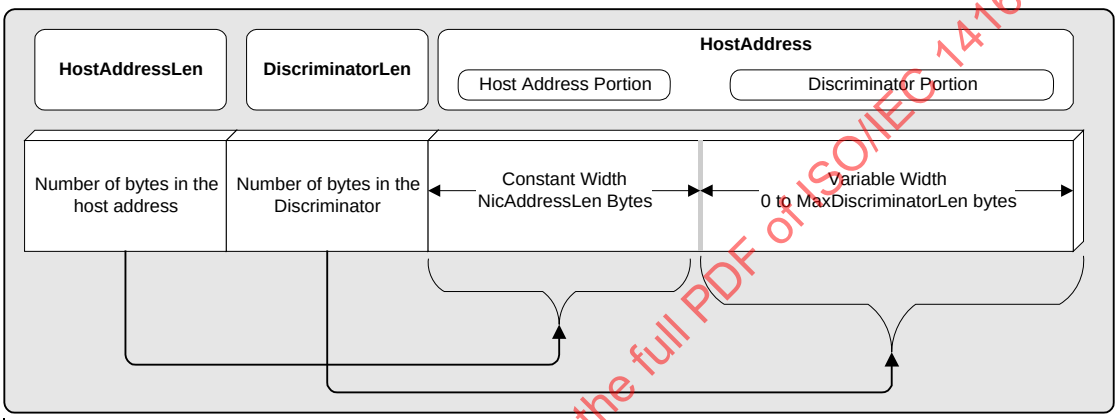


Figure 3: VIP_NET_ADDRESS memory layout

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.

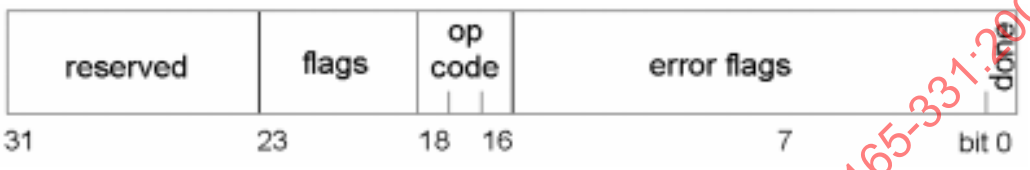


Figure 6: Status Field Format

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.

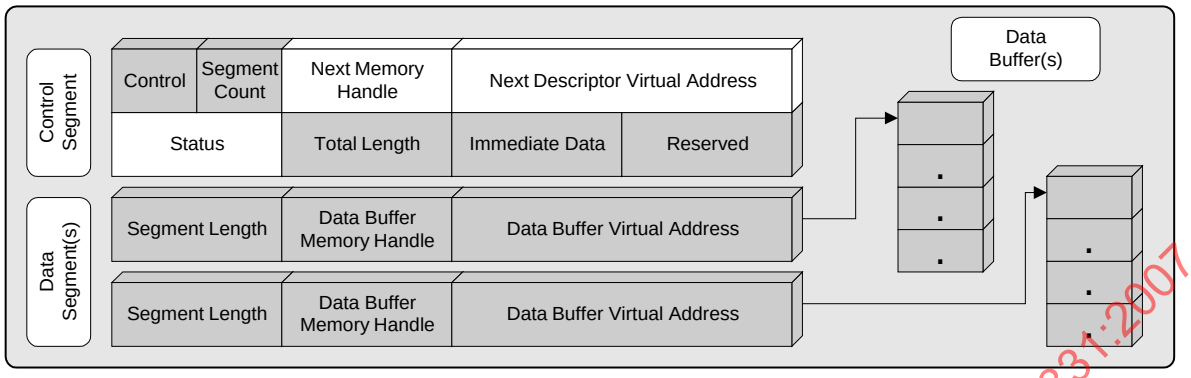


Figure 7: Descriptor fields updated by VI Application

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.

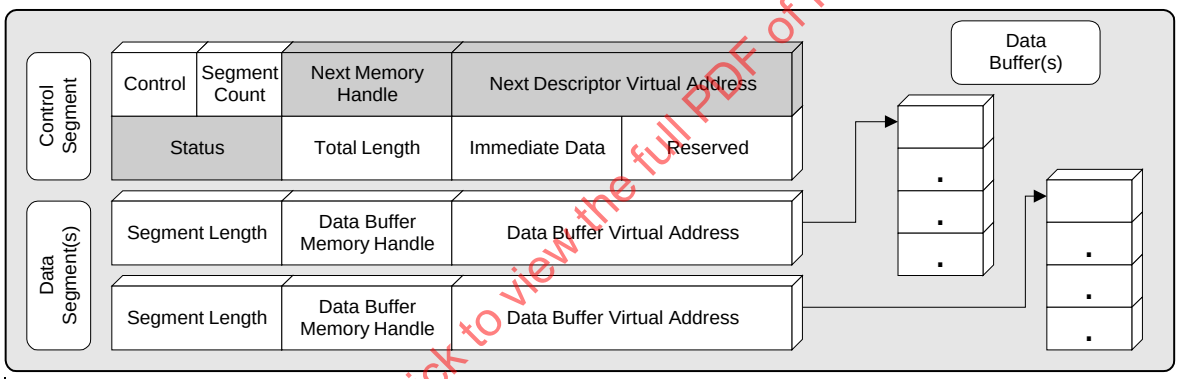


Figure 8: Descriptor fields updated by VIPL

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.

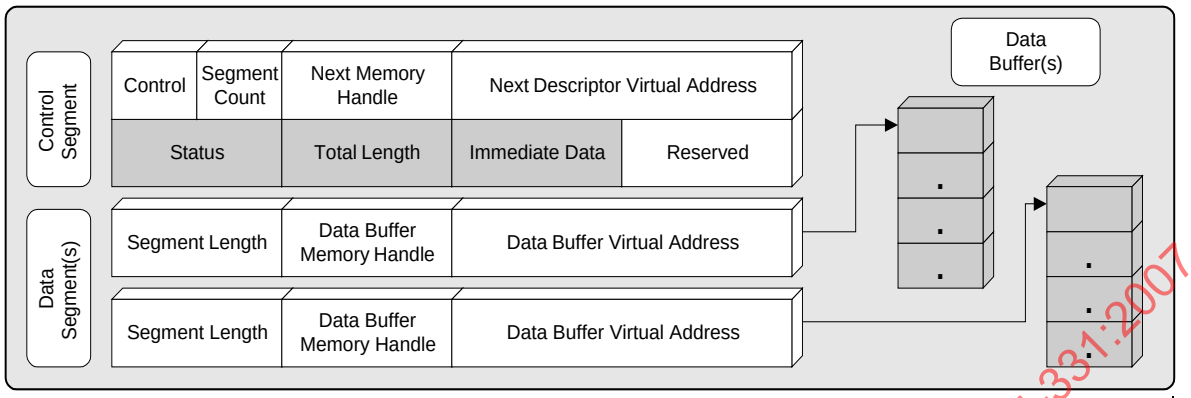


Figure 9: Descriptor fields updated by VI NIC on completion

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).

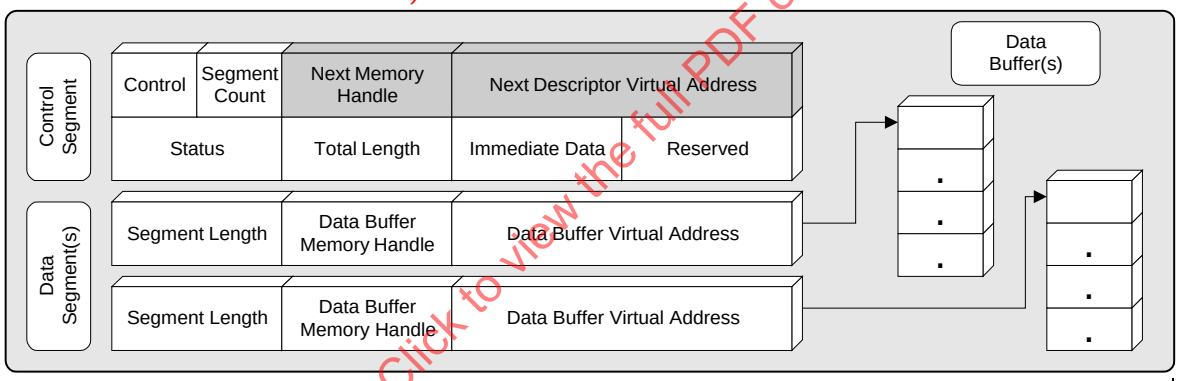


Figure 10: Descriptor fields updated by VIPL on completion

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.

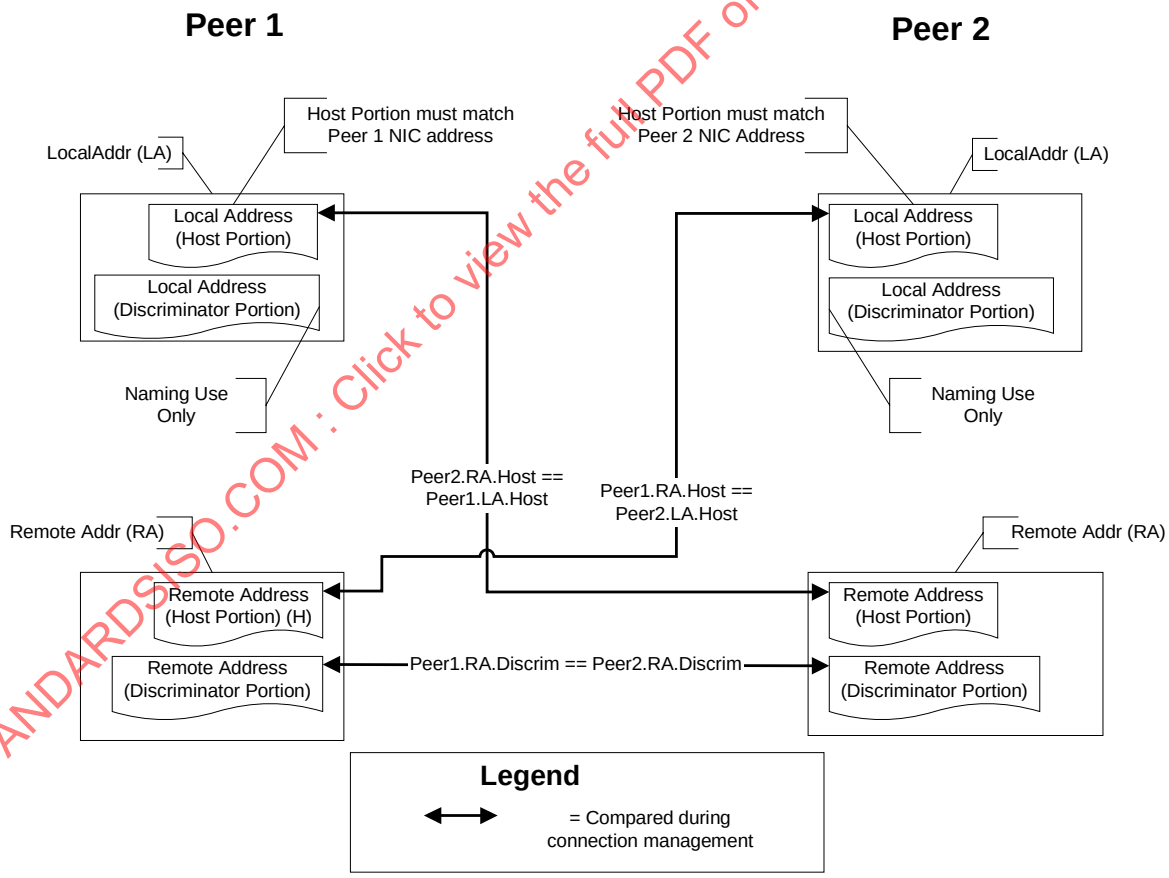


Figure 11: Address Comparison and Data Exchange for Peer-to-Peer

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)

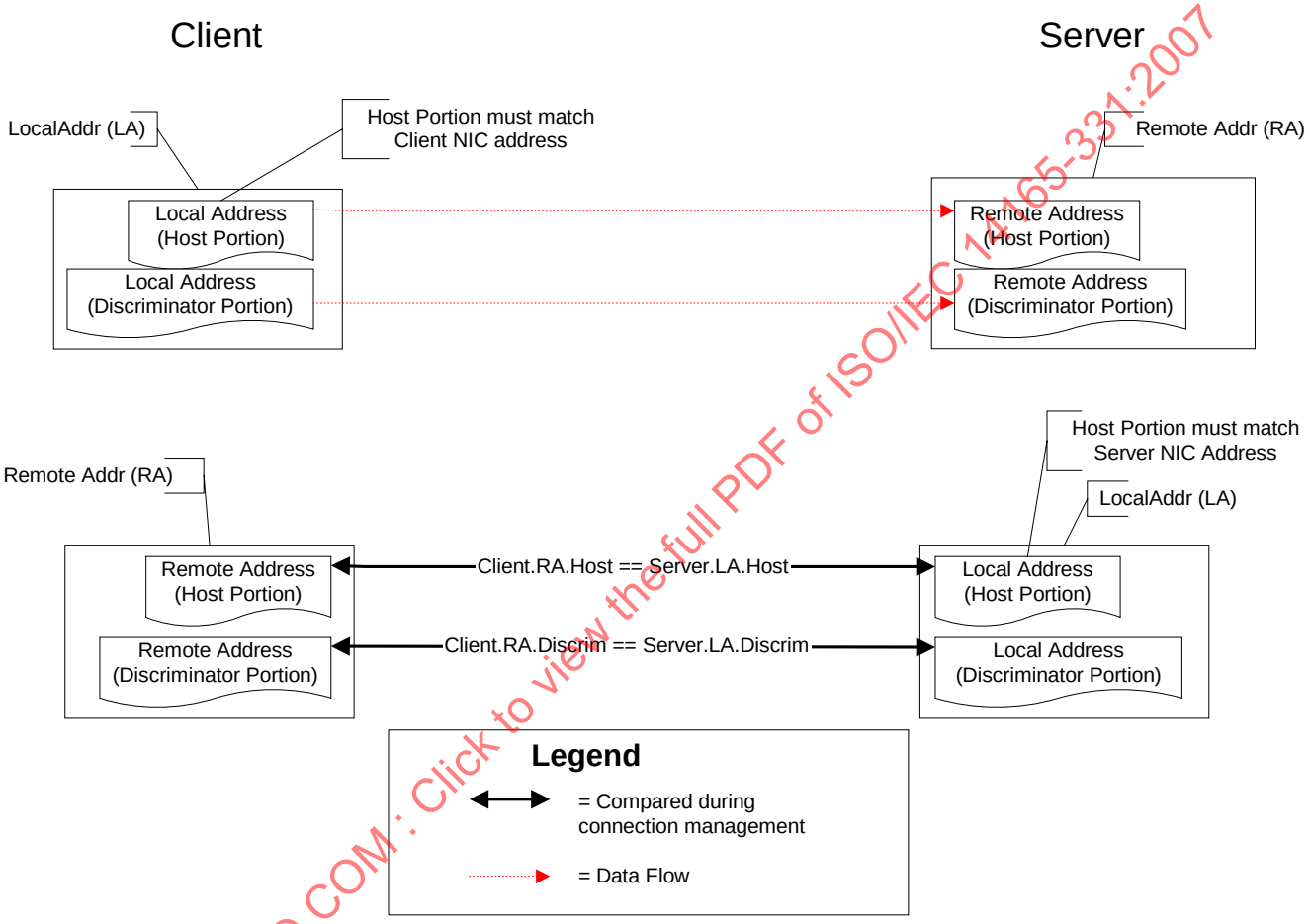


Figure 12: Address Comparison and Data Exchange for Client/Server

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.

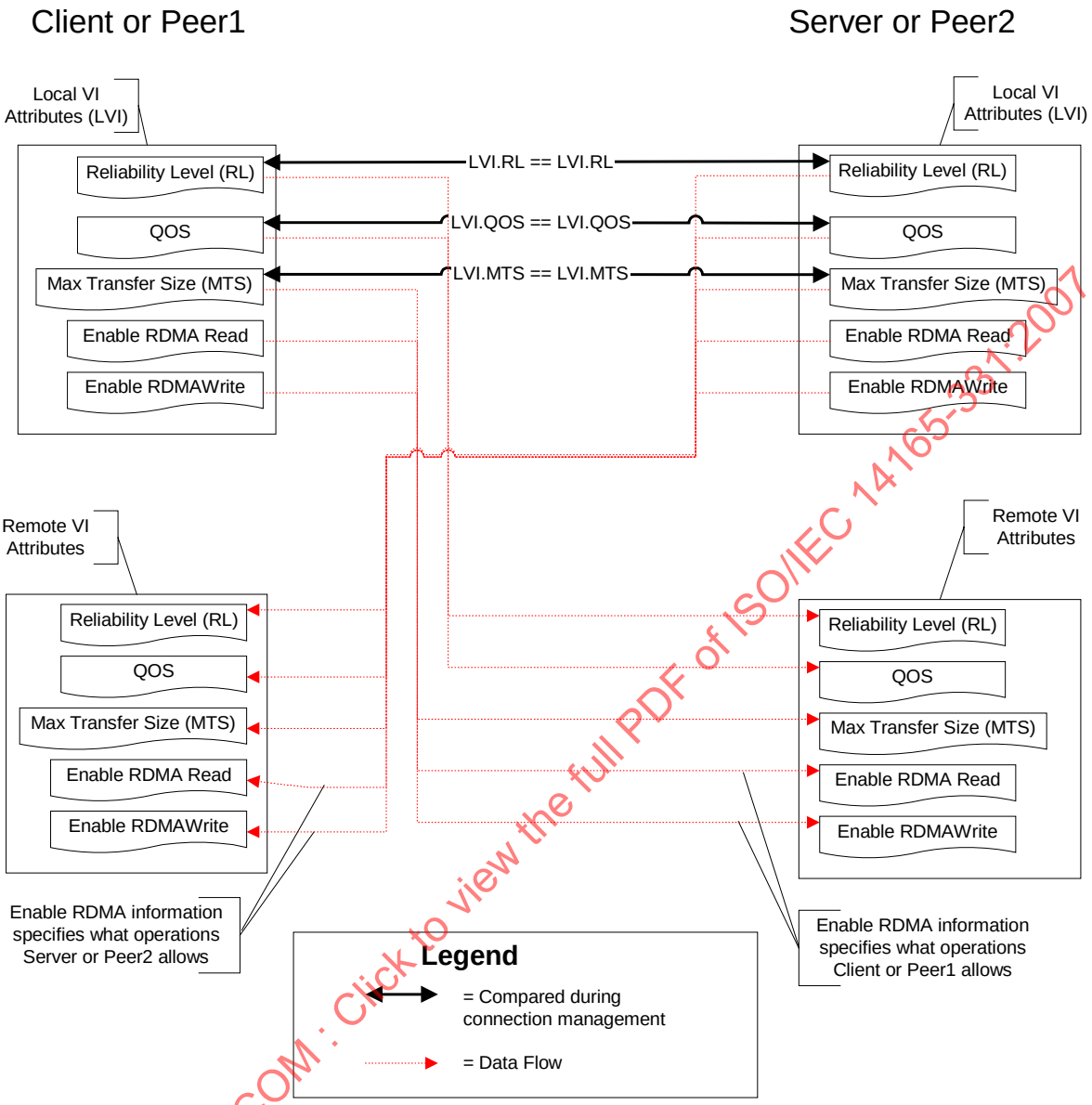


Figure 13: VI Attribute Comparison and Exchange

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.

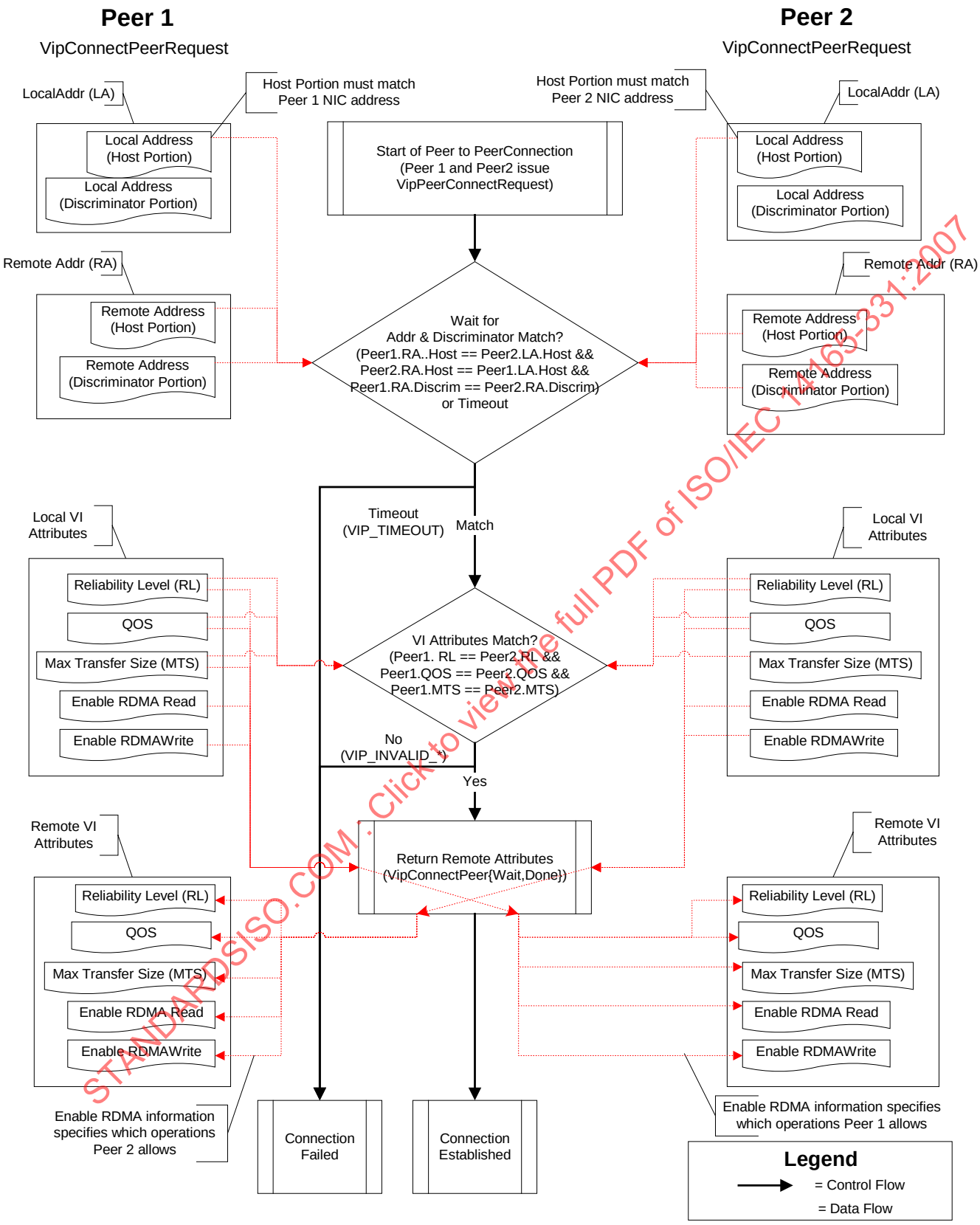


Figure 14: Peer-to-Peer Combined

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.