InfiniBand™ Host Channel Adapter Verb Implementer's Guide

March 23, 2003

Revision 1.3

Send Comments To
ashok.raj@intel.com

DISCLAIMERS

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1. Purpose and Scope

The InfiniBand™ Architecture provides a high performance, low latency and reliable means for communication among hosts and I/O units attached to a switched, high-speed fabric. In such a system, the *Host Channel Adapters* (HCA's) provide InfiniBand fabric connection to hosts. This specification describes methods and the mechanisms to manage HCA adapter drivers. This document is only a guideline on implementing Verbs Drivers and does not specifically target any particular HCA adapter or Vendor implementations.

1.2. Intended Audience

This specification is intended to guide the development of Software interface for the Verbs Interface specification. Verbs specifically address the hardware and software interface that abstracts the hardware component. This allows both *Independent Hardware Vendor* (IHV) community and *Operating System Vendor* (OSV) community, to write software interfaces in a consistent way that does not vary between hardware vendors or operating systems. This document is intended for HCA driver writers to implement Verb Interfaces as described in the SourceForge open source implementation for Linux™.

The reader is expected to be familiar with the InfiniBand Architecture. Deeper knowledge of the *InfiniBand Architecture Specification Volume 1, Release 1.1,* and associated annexes are necessary to understand some sections of this document. A HCA vendor must pay particular attention to the compliance statements in Chapters 10, Chapter 11 and Chapter 17 in order to implement verbs behavior in a consistent manner.

The reader is also expected to be familiar with operating system concepts that would enhance the understanding and proper mapping of the functions as appropriate to a specific OS. In most cases as applicable we will provide references to Linux Operating system interfaces as an example.

1.3. Document Organization

This document is structures as follows.

- Chapter 2 provides a brief overview of InfiniBand Architecture, and explains the role of HCA's in this architecture.
- Chapter 3 provides a brief overview of the software architecture, and explains the different roles that each module supports.
- Chapter 4 describes the available verbs as specified in the InfiniBand Architecture, Volume 1.0.a of the specifications, their relationships.
- Chapter 5 describes initialization and shutdown procedures.
- Chapter 6 describes the transport resource management, which covers QP, CQ, AV, PD, RDD allocation and management.
- Chapter 7 describes methods for managing VPTT and registration resources.
- Chapter 8 explains the work request processing, and completion processing methods.
- Chapter 9 describes the interrupt handling and event processing.
- Chapter 10 describes methods of supporting mandatory agents such as *Subnet Management Agent* (SMA) in the *Channel Interface* (CI).

1.4. Conventions

This document uses the following conventions:

Acronyms: The first time an acronym is used; it is enclosed in parentheses and preceded by the phrase it represents in italics. An example is a *Three Letter Acronym* (TLA).

Data Format: All multi-byte data fields defined by the InfiniBand™ Architecture are in big-endian byte order. All values are decimal unless preceded by "0x" signifying that the value is in hexadecimal notation, or "0b" signifying that the value is in binary notation.

References: Reference documents and their associated shortened reference tags (e.g., *[IBA Vol1]*) are specified in Section 1.5

Cross-references: Cross-references are used liberally through out this document to assist in online reviewing. Most references to figures, tables, sections and acronyms are provided with cross-references to take the reader to the interest area quickly. Cross-references are indicated by blue underlined text.

Byte Ordering: The data structures shared with software are specified in increasing byte order is first from right to left, and than from top to bottom. However, in figures in which arrays of data structures are depicted, the increasing byte order is first right to left, and than from bottom to top.

Discrepancies: In writing of this document we made every effort to provide accurate and consistent information. However, when discrepancies exist between figures and text, then the figures must be considered as correct.

1.5. Reference Documents

[IBA Vol1]	*InfiniBand Architecture Specification, Volume 1, Release 1.X*
[IBA Vol2]	*InfiniBand Architecture Specification, Volume 2, Release 1.X*

For more details and to obtain a copy of most recent documents, please visit *http://infiniband.sourceforge.net*

The source can be obtained from *http://infiniband.bkbits.net*

1.6. Commonly Used Acronyms

AETH	ACK Extended Transport Header
B_Key	Baseboard Management Key
BTH	Base Transport Header
CI	Channel Interface
CQ	Completion Queue
CQE	Completion Queue Element
CRC	Cyclic Redundancy Check
DETH	Datagram Extended Transport Header
DGID	Destination Globally Unique Identifier
DLID	Destination Local Identifier
EEC	End-to-End Context
EQ	Event Queue
EQE	Event Queue Element
GID	Global Identifier

GMP	General Management Packet
GRH	Global Route Header
GSI	General Service Interface
DMA	Direct Memory Access
SMP	Sub-net Management Packet
GMP	General Services Management Packet
GUID	Globally Unique Identifier
HCA	Host Channel Adapter
IBA	InfiniBand Architecture
ICRC	Invariant CRC
IHV	Independent Hardware Vendor
IPv6	Internet Protocol, version 6
IOC	I/O Controller
L_Key	Local Key
LID	Local Identifier
LMC	LID Mask Control
LRH	Local Route Header
M_Key	Management Key
MAD	Management Datagram
MSN	Message Sequence Number
MTU	Maximum Transfer Unit
P_Key	Partition Key
PD	Protection Domain
PSN	Packet Sequence Number
Q_Key	Queue Key
QoS	Quality of Service
QP	Queue Pair
R_Key	Remote Key
RC	Reliable Connection
RD	Reliable Datagram
RDETH	Reliable Datagram Extended Transport Header
RDMA	Remote Direct Memory Access
SGID	Source Global Identifier
SLID	Source Local Identifier
SL	Service Level
SM	Subnet Manager
SMA	Subnet Management Agent
SMP	Subnet Management Packet
TCA	Target Channel Adapter
UC	Unreliable Connection
UD	Unreliable Datagram
VCRC	Variant CRC
VL	Virtual Lane
WC	Work Completion
WQ	Work Queue
WQE	Work Queue Element
WQP	Work Queue Pair
WR	Work Request

1.7. Definitions and Commonly Used Terms

Address Vector
A collection of address and path information specifying a remote port and the parameters to be used when communicating with it.

Automatic Path Migration
The process in which a Channel Adapter, on a per-Queue Pair basis, signals another CA to cause Path Migration to a preset alternate Path. Automatic Path Migration uses a bit in a request or response packet (MigReq) to signal the other channel adapter to migrate to the predefined alternate path.

Base LID
The numerically lowest Local Identifier that refers to a Port. The Path Bits of a Base LID are always zero.

Binding
The act of associating a virtual address range in a specified Memory Region with a Memory Window.

Channel
The abstraction represented by a queue pair on each of two InfiniBand nodes such that a message placed on the send queue of one node arrives at the receive queue of the other.

Channel Adapter
Device that terminates a link and executes transport-level functions. One of Host Channel Adapter or Target Channel Adapter.

Channel Interface
The presentation of the channel to the Verbs Consumer as implemented through the combination of the Host Channel Adapter, associated firmware, and device driver software.

Completion Error
Permanent interface or processing error reported through completion status.

Completion Queue
A queue containing one or more Completion Queue Entries. Completion Queues are internal to the Channel Interface, and are not visible to verb consumers.

Completion Queue Entry
The Channel Interface-internal representation of a Work Completion.

Connection
An association between a pair of entities (e.g., processes) over one or more Channels.

Data Payload
The data, not including any control or header information, carried in one packet.

Data Segment
A tuple in a Work Request that specifies a virtually contiguous buffer for Host Channel Adapter access. Each Data Segment consists of a Virtual Address, an associated Local Key or Remote Key, and a length.

End to End Context
The endpoint of a Reliable Datagram channel.

End to End Flow Control
A mechanism to prevent a sender from transmitting messages during periods when receive buffers are not posted at the recipient.

Fabric
The collection of Links, Switches, and Routers that connects a set of Channel Adapters.

General Service Interface
An interface providing management services (e.g., connection, performance, diagnostics) other than subnet management.

Global Identifier
A 128-bit identifier used to identify a port on a channel adapter, a port on a router, or a multicast group. GID's are valid 128-bit IPv6 addresses (per RFC 2373) with additional properties / restrictions defined within IBA to facilitate efficient discovery, communication, and routing.

Global Route Header
Routing header present in InfiniBand TM Architecture packets targeted to destinations outside the sender's local subnet.

Globally Unique Identifier
A number that uniquely identifies a device or component.

Host
One or more Host Channel Adapters governed by a single memory/CPU complex.

Host Channel Adapter
A Channel Adapter that supports the Verbs interface.

Immediate Data
Data contained in a Work Queue Element that is sent along with the payload to the remote Channel Adapter and placed in a Receive Work Completion.

Invariant CRC
A CRC covering the fields in a packet that do not change from the source to the destination.

I/O Controller
One of the two architectural divisions of an I/O Unit. An I/O controller (IOC) provides I/O services, while a Target Channel Adapter provides transport services.

I/O Unit
An I/O unit (IOU) provides I/O service(s). An I/O unit consists of one or more I/O Controllers attached to the fabric through a single Target Channel Adapter.

IPv6 Address
A 128-bit address constructed in accordance with IETF RFC 2460 for IPv6.

LID Mask Control
A per-port value assigned by the Subnet Manager. The value of the LMC specifies the number of Path Bits in the Local Identifier.

Link
A full duplex transmission path between any two fabric elements, such as Channel Adapters or Switches.

Local Identifier
An address assigned to a port by the Subnet Manager, unique within the subnet, used for directing packets within the subnet. The Source and Destination LIDs are present in the Local Route Header. A Local Identifier is formed by the sum of the Base LID and the value of the Path Bits.

Local Key
An opaque object, created by a verb, referring to a Memory Registration, used with a Virtual Address to describe authorization for the HCA hardware to access local memory. It may also be used by the HCA hardware to identify the appropriate page tables for use in translating virtual to physical addresses.

Local Route Header
Routing header present in all InfiniBand Architecture packets, used for routing through switches within a subnet.

Management Datagram
Refers to the contents of an Unreliable Datagram packet used for communication among HCAs, switches, routers, and TCA's to manage the fabric. InfiniBand Architecture describes the format of a number of these management commands.

Management Key
A construct that is contained in IBA management datagrams to authenticate the sender to the receiver.

Memory Region
A virtually contiguous area of arbitrary size within a Consumer's address space that has been registered, enabling HCA local access and optional remote access.

Memory Registration
The act of registering a host Memory Registration for use by a consumer. The memory registration operation returns a Memory Region Handle. The process provides this with any reference to a virtual address within the memory region.

Memory Window
An allocated resource that enables remote access after being bound to a specified area within an existing Memory Registration. Each Memory Window has an associated Window Handle, set of access privileges, and current R_Key.

Message
A transfer of information between two or more Channel Adapters that consists of one or more packets.

Multicast
A facility by which a packet sent to a single address may be delivered to multiple ports.

Multicast Identifier
An identifier for a set of ports making up a Multicast Group, typically belonging to different Channel Adapters. On a subnet, Multicast Identifiers share the address space of Local Identifiers.

Multicast Group
A collection of Channel Adapter ports that receive Multicast packets sent to a single address.

Packet
The indivisible unit of IBA data transfer and routing, consisting of one or more headers, a Packet Payload, and one or two CRC's.

Packet Payload
The portion of a Packet between (not including) any Transport header(s) and the CRCs at the end of each packet. The packet payload contains up to 4096 bytes.

Packet Sequence Number
A value carried in the Base Transport Header that allows the detection and re-sending of lost packets.

Partition
A collection of Channel Adapter ports that are allowed to communicate with one another. Ports may be members of multiple partitions simultaneously. Ports in different partitions are unaware of each other's presence insofar as possible.

Partition Key Table
A table of partition keys present in each Port.

Path
The collection of links, switches, and routers a message traverses from a source Channel Adapter to a destination channel adapter. Within a subnet, a path is defined by the tuple <SLID, DLID, SL>.

Path Bits
The portion of a Local Identifier that may be changed to vary the Path through the subnet to a particular Port. If the Path Bits are zero, the Local Identifier is equal to the Base LID. The Subnet Manager through the LID Mask Control value specifies the number of Path Bits applicable to a particular port.

Path Maximum Transfer Unit
The maximum size of the Packet Payload supported along a Path from source to destination. PMTU is described in terms of the payload size, and may be 256, 512, 1024, 2048, or 4096 bytes.

Port
Location on a Channel Adapter or Switch to which a link connects. There may be multiple ports on a single Channel Adapter, each with different context information that must be maintained. Switches/switch elements contain more than one port by definition.

Protection Domain
A mechanism for associating Queue Pairs, Address Handles, Memory Windows, and Memory Registrations.

Queue Key
A construct that is used to validate a remote sender's right to access a local Receive Queue for the Unreliable Datagram and Reliable Datagram service types. If the Q_Key present in an incoming packet does not match the value stored in the receiving QP, the packet shall be dropped.

Queue Pair
Consists of a Send Work Queue and a Receive Work Queue. Send and receive queues are always created as a pair and remain that way throughout their lifetime. Its Queue Pair Number identifies the Queue Pair.

Raw Datagram
A packet that contains an IBA Local Route Header, may contain an IBA Global Route Header, but does not contain an IBA Transport header, and is not handled by IBA transport services.

Receive Queue
One of the two queues associated with a Queue Pair. The receive queue contains Work Queue Elements that describe where to place incoming data.

Reliable Connection
A Transport Service Type in which a Queue Pair is associated with only one other QP, such that messages transmitted by the send queue of one QP are reliably delivered to receive queue of the other QP. As such, each QP is said to be "connected" to the opposite QP.

Reliable Datagram
A Transport Service Type in which a Queue Pair may communicate with multiple other QPs over a Reliable Datagram Channel. A message transmitted by an RD QP's send queue will be reliably delivered to the receive queue of the QP specified in the associated Work Request. Despite the name, Reliable Datagram messages are not limited to a single packet.

Reliable Datagram Domain
An association that defines which Reliable Datagram Queue Pairs may use an End to End Context.

Remote Direct Memory
Access Method of accessing memory on a remote system without interrupting the processing of the CPU(s) on that system.

Remote Key
An opaque object, created by a verb, referring to a Memory Registration or Memory Window, used with a Virtual Address to describe authorization for the remote device to access local memory. It may also be used by the HCA hardware to identify the appropriate page tables for use in translating virtual to physical addresses.

RNR Nak
Receiver Not Ready. A response signifying that the receiver is not currently able to accept the request, but may be able to do so in the future.

Router
A device that transports packets between IBA subnets.

Send Queue
One of the two queues of a Queue Pair. The Send queue contains WQE's that describe the data to be transmitted.

Service Level
Value in the Local Route Header identifying the appropriate Virtual Lane for a packet, enabling the implementation of differentiated services. While the appropriate VL for a specific Service Level may differ over a packet's Path, the Service Level remains constant.

Signaled Completion
A modifier used for Work Requests submitted to the Send Queue specifying that a Work Completion shall be generated when the work requested completes, whether successfully or in error.

Solicited Event
A facility by which a message sender may cause an event to be generated at the recipient when the message is received.

Subnet
A set of InfiniBand Architecture Ports, and associated links, that have a common Subnet ID and are managed by a common Subnet Manager. Subnets may be connected to each other through routers.

Subnet Management Agent
An entity present in all IBA Channel Adapters and Switches that processes Subnet Management Packets from Subnet Manager(s).

Subnet Management Packet
The subclass of Management Data-grams used to manage the subnet. SMPs travel exclusively over Virtual Lane 15 and are addressed exclusively to Queue Pair Number 0.

Switch
A device that routes packets from one link to another of the same subnet, using the Destination Local Identifier field in the Local Route Header.

Target Channel Adapter
A Channel Adapter typically used to support I/O devices. TCA's are not required to support the Verbs interface.

Transport Service Type
Describes the reliability, sequencing, message size, and operation types that will be used between the communicating Channel Adapters.

Unicast
An identifier for a single port. A packet sent to a unicast address is delivered to the port identified by that address.

Unit
One or more sets of processes and/or functions attached to the fabric by one or more channel adapters.

Unreliable Connection
A Transport Service Type in which a Queue Pair is associated with only one other QP, such that messages transmitted by the send queue of one QP are, if delivered, delivered to the receive queue of the other QP. As such, each QP is said to be "connected" to the opposite QP. Messages with errors are not retried by the transport, and error handling must be provided by a higher level protocol.

Unreliable Datagram
A Transport Service Type in which a Queue Pair may transmit and receive single-packet messages to/from any other QP. Ordering and delivery are not guaranteed, and the receiver may drop delivered packets.

Un-signaled Completion
A modifier used for Work Requests submitted to the Send Queue signifying that a Work Completion is to be generated only if the requested action completes in error.

Variant CRC
A CRC covering all the fields of a packet, including those that may be changed by Switches.

Verbs
An abstract description of the functionality of a Host Channel Adapter. An operating system may expose some or all of the verb functionality through its programming interface.

Virtual Lane
A method of providing independent data streams on the same physical link.

Work Completion
The consumer-visible representation of a Completion Queue Entry. A Work Completion may be obtained when a consumer polls a Completion Queue.

Work Queue
Refers to one of Send Queue or Receive Queue.

Work Queue Element
The Host Channel Adapter's, internal representation of a Work Request. The consumer does not have direct access to Work Queue Elements.

Work Request
The means by which, a consumer requests the creation of a Work Queue Element.

1.8. Revision History

Revision	Date	Description
Revision 1	July 18, 2002	Changes to make the document vendor neutral.
Final Draft	August 16, 2002	Incorporated feedback
Revision 1.1	September 6, 2002	Adding usage guidelines for SMI and Traps, based on feedback from Mellanox
Revision 1.2	December 24, 2002	Added note on Byte Ordering Conventions
Revision 1.3	March 23rd, 2003	Added information on Callbacks, user mode support notes for passing private HCA attributes.

2. Overview

The first part of the chapter provides a brief overview of the InfiniBand architecture based systems, identifies elements of the architecture and provides a reference to available specifications to the reader. The second part details the software interface that glues the HCA with system software in an operating system environment.

2.1. InfiniBand Overview

Figure 2-1 below depicts elements of *InfiniBand Architecture* (IBA) *System Area Network* (SAN). In small configurations, a processor node can be connected to I/O nodes directly. However, in large System Area Networks many processor and I/O nodes can be interconnected over a fabric.

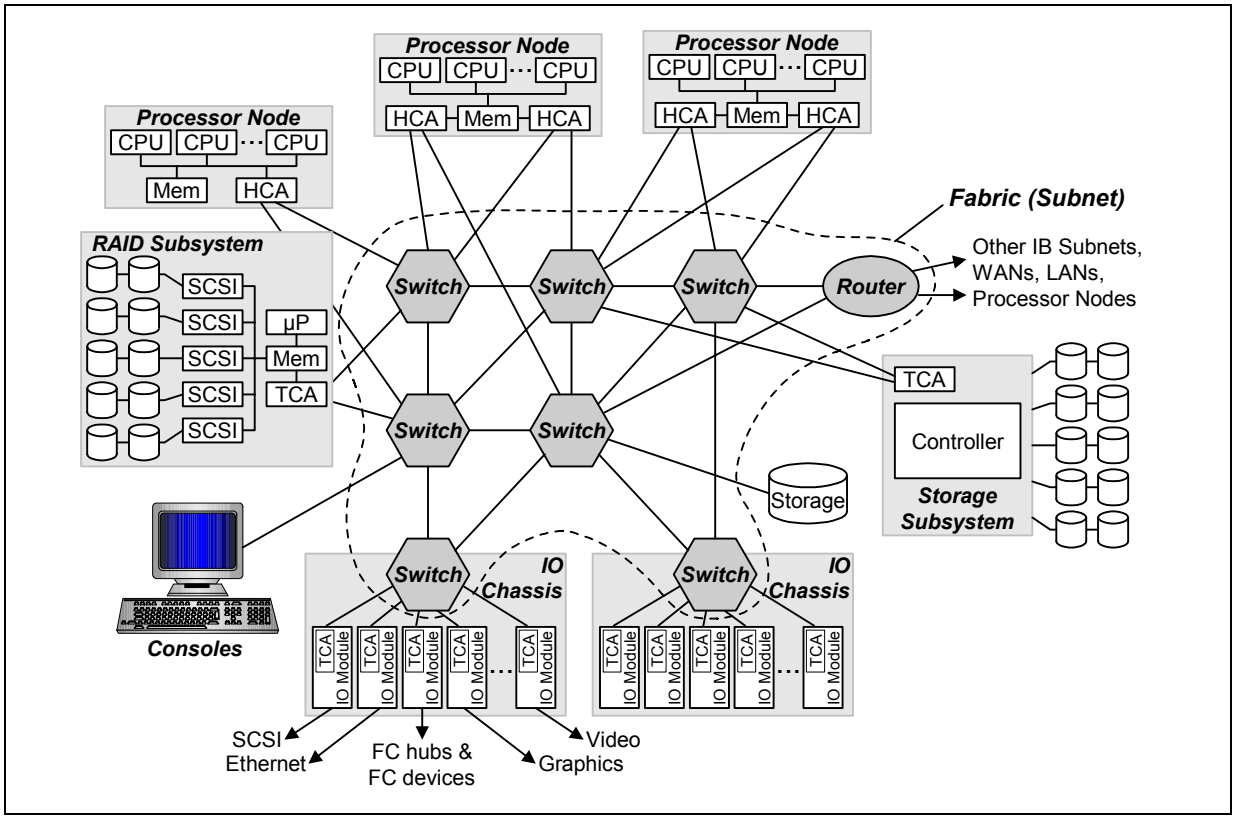

Figure 2-1 InfiniBand Architecture System Area Network

An IBA system can range from a small server with one processor and a few I/O devices to a massively parallel supercomputer installation with hundreds of processors and thousands of I/O devices. In small configurations, a processor node may be connected to I/O nodes directly. However, in large SAN's many processor and I/O nodes can be interconnected over a fabric. Furthermore, the Internet Protocol (IP) friendly nature of IBA allows bridging to an Internet, intranet, or connection to remote computer systems.

IBA defines a switched communications fabric allowing many devices to concurrently communicate with high bandwidth and low latency in a protected, remotely managed environment. An end node can communicate over multiple IBA ports and can utilize multiple paths through the IBA fabric. The potential redundancy of IBA ports and paths through the network are exploited for both fault tolerance and increased data transfer bandwidth.

The components of an IBA system described later in this section are:

- Channel Adapters
- Switches
- Routers
- Repeaters
- Links that interconnect switches, routers, repeaters and channel adapters

The basic interconnect of the InfiniBand Architecture is a link, which is a full-duplex transmission path between any two fabric elements. A link physically terminates at a port. The physical attach point to a port is either:

- A Copper Cable Connector, defined for use with a wired connection referred to as a copper cable throughout this document.
- A Fiber Optic Connector, which is defined for use with optical cables and signal converters.
- An IB Module, which is defined for use with a back-plane connector, which includes board-to-board and chip-to-chip connections.

The InfiniBand specifications define three electrical interface widths as follows:

- 1X (2 differential pair, one per direction, for a total of 4 wires)
- 4X (8 differential pair, four per direction, for a total of 16 wires)
- 12X (24 differential pair, 12 per direction, for a total of 48 wires)

2.1.1. Channel Adapters

Channel adapters are the components in processor nodes and I/O units that generate and consume packets. The IBA defines two types of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA).

3. Software Architecture Overview

Figure below shows a very high level overview of the software architecture of InfiniBand Software. This figure does not explain all the different subcomponents of the architecture. For a more detailed overview of the software architecture, please refer to the *Software Architecture Specification* (SAS) document.

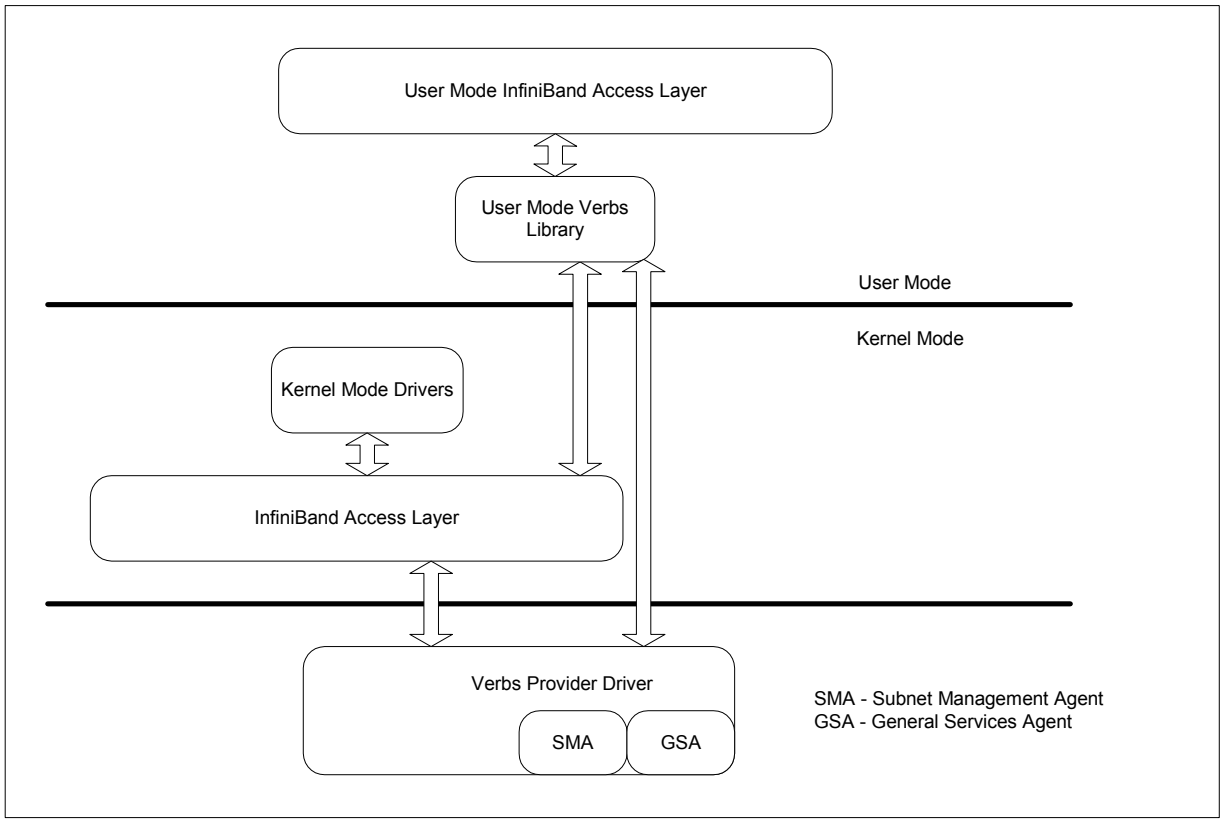

Figure 3-1 InfiniBand Software Architecture

This specification highlights the *Verbs Provider Driver* (VPD) and its sub-components. As shown above in Figure 3-1, the VPD interfaces with the *InfiniBand Access Layer* (AL). AL manages multiple adapters and different CA vendor devices, whereas VPD manages multiple instances of adapters, typically from a same manufacturer or hardware vendor. Verbs can be viewed as the software glue to the *Operating* System (OS). InfiniBand specifications also refer to a *Channel Interface* (CI), which is the combination of the VPD and the adapter hardware to provide the transport functionality.

The VPD serves the following primary functions.

- Manage multiple instances of CA's and the adapter's internal resources
- Interfaces with the OS to obtain resources necessary to function, such as device identification, and interrupt registration.
- Provides the Verbs API as exposed to the IAL.
- Provides functionality for mandatory agents such as *Subnet Management Agent* (SMA) and *General Services Agent* (GSA).

4. Verbs Provider Driver

Verbs Provider Driver is responsible for the component identification, initialization and registration with the OS to obtain kernel mode resources. Once VPD has initialized the component and ready for use, it then notifies the IB Access Layer to notify the arrival of a new HCA. Linux architecture typically uses a registration call to achieve this process. For e.g. the SCSI HBA drivers would use *scsi_register ()* call. Linux IB Access Layer exports such an interface to allow low-level drivers to register HCA devices identified in the system. The HCA would typically identify itself via the PCI configuration space. This allows the driver to identify all instances of the component via standard device discovery mechanisms. In this case the VPD would use standard *pci_register_driver ()* call to identify existing devices.

4.1. Verbs Driver Architecture

The Verbs driver manages several CA resources such as QP's, CQ's, AV's and PD's Most CA's are expected to have a large number of these resources. So it is important to efficiently utilize and manage their allocation and use in the driver.

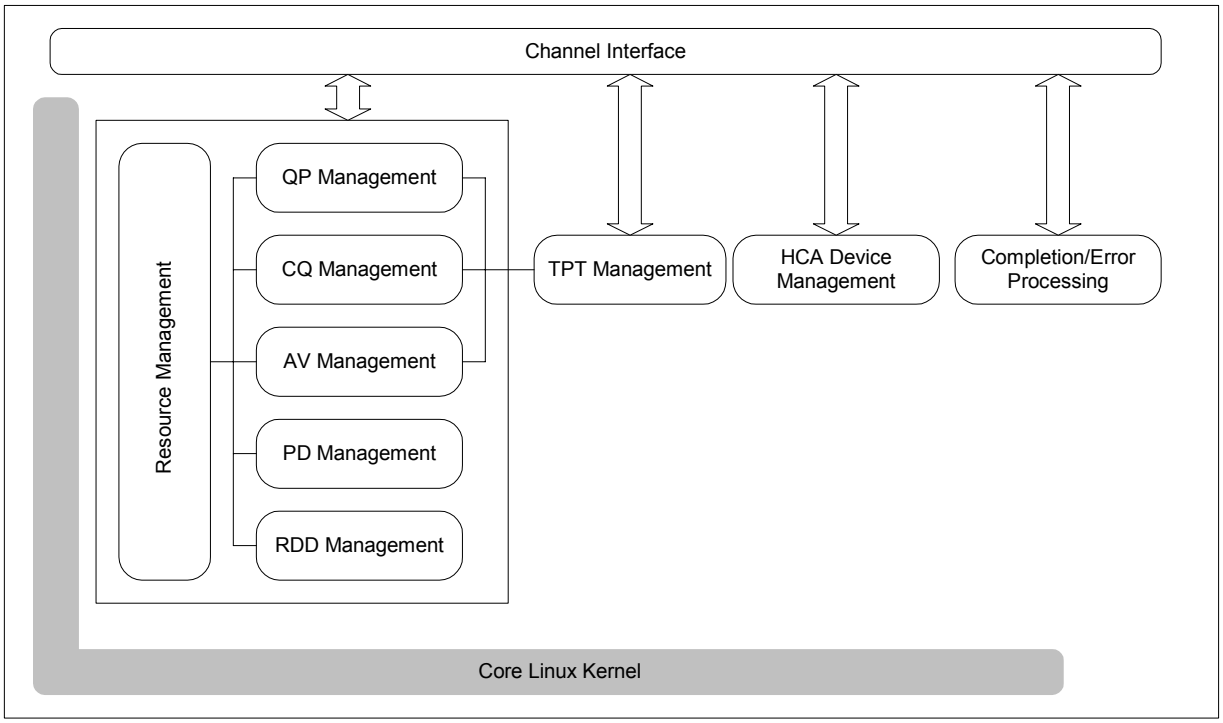

Figure 4-1 Verbs Driver Architecture

The figure above provides a brief view of what functions exist in the drivers. The salient functional pieces are described below.

- Resource Management – Manages allocation of unallocated resources. A generic scheme is suggested later in this guide, which allows managing multiple types of resources.
- Memory management – InfiniBand architecture requires true virtual to physical mapping without involving the host CPU. Memory management schemes available in the HCA make such translations possible.
- HCA device Management – This refers to managing multiple host adapters available in a system and the management of the adapters themselves.

- Completion and error processing – This refers to the interrupt handling and event queue management. Processing completion events, identifying the right context associated with the resource necessary to perform the notification.
- Special QP management – Verbs also has to support certain mandatory agents necessary to support the QP0 and QP1 QPs as described in the InfiniBand architecture specifications.

4.2. Verb Groups

Verbs define an abstract definition of functionality provided to a host by a CI. Chapter 11 of the InfiniBand architecture specifications defines certain verbs as mandatory, and some features are optionally supported by the verbs. Figure 4-2 depicts the different verb groups, and their inter-dependencies to some extent. In general the architecture defines the following verb groups.

- Transport resource management – Covers the following
  - HCA access functions
  - Protection domain management
  - Reliable Datagram domain management
  - Address handle management for Unreliable Datagram (UD) QPs
  - Special QP access verbs
  - General QP, End-to-End context management
  - Completion queue management.
  - Memory management services
- Multicast services for UD QP's
- Work Request processing
- Event notification and handling.

The subnet management agent in the Verbs Driver must be compliant with the specification in Chapter 14 of Volume 1 of the InfiniBand Architecture specifications in terms of wire level formats for requests and responses.

NOTE

Deviations from InfiniBand Specification

In Figure 4-2 below the Completion Queue operations require a valid protection domain. Although the InfiniBand Specifications do not specify this dependency due to the fact that many QP's in different protection domains can still have the same Completion Queue. The association of a CQ to a protection domain gives it the added implementation ease for user mode Completion Queues. Completion Queues are required to have doorbells to enable the CQ for event notifications. For user mode processes, such access must be facilitated from the process address space. Protection domains provide the process level protection required to expose direct user mode IO capabilities of InfiniBand.

4.3. Byte-Ordering Conventions

All multi-byte quantifies defined by the InfiniBand™ specifications that are exchanged between nodes are retained in network byte order to avoid frequent conversions. An example would be QP number, P_KEY, Q_KEY, LID, GUID's, PSN values etc. Such quantities are reflected in the API as ib_net16_t, ib_net32_t or ib_net64_t defined in the ib_types.h header file.

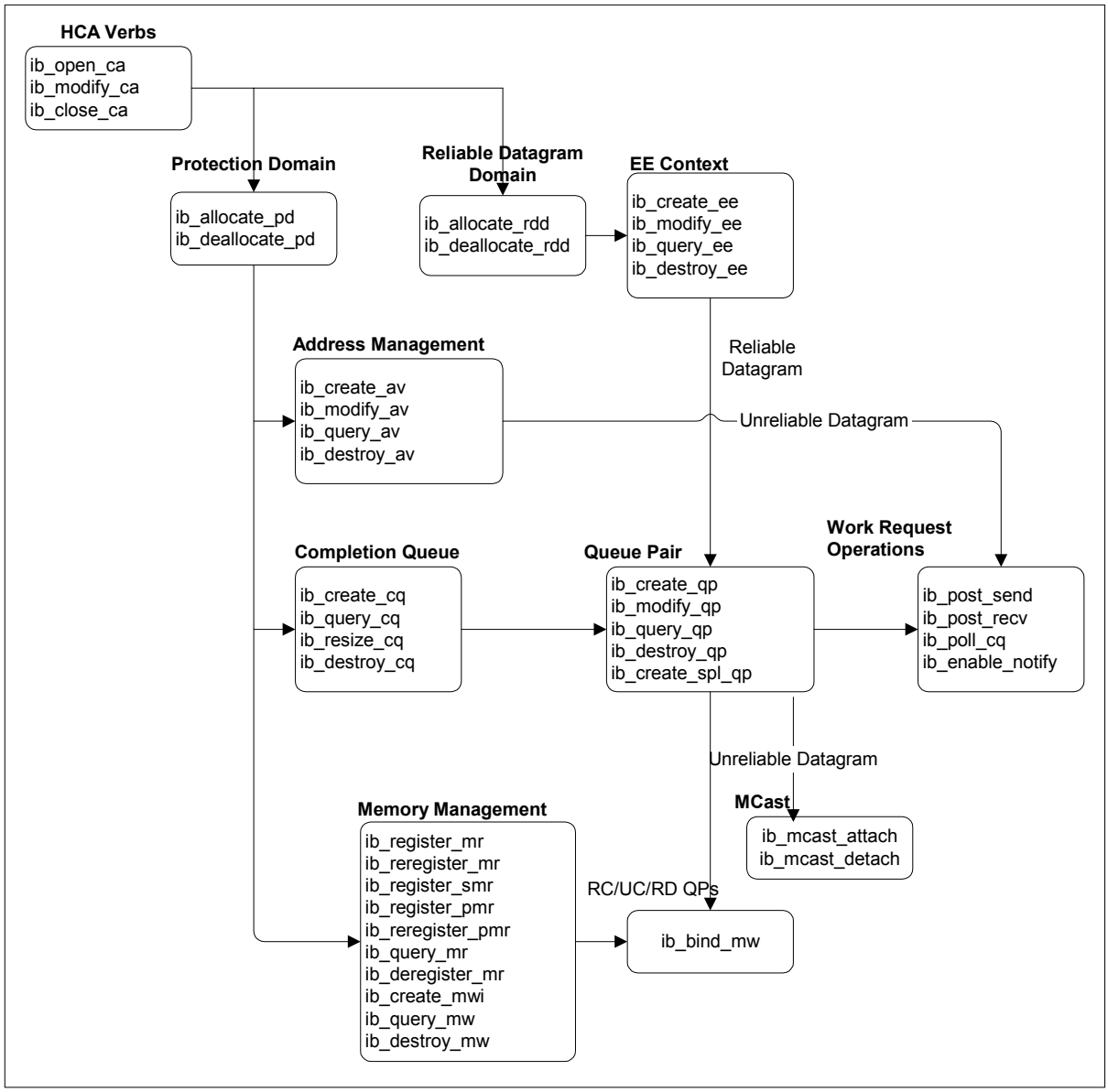

Figure 4-2 Verb groups and relationships

4.4. Verb Classes

The following table lists the different verbs, mandatory or optional. In addition the table also lists if some of the verbs can be implemented in user mode for speed path operations. This table mirrors Table 78 of in the Volume 1.0.a of the specifications. For the verb classes identified by privileged, the operation is performed in the kernel mode driver via either *syscall ()* or *ioctl ()* interfaces. For user-level access, it implies the user mode process directly communicates with hardware with the assistance of the user mode verb plug-in library.

Table 4-1 Table of Verb Classes and Supported Features

Verb	*Mandatory/Optional*	*Privileged or User Level*
Open HCA	Mandatory	Privileged
Close HCA	Mandatory	Privileged
Query HCA	Mandatory	Privileged
Modify HCA	Mandatory	Privileged
Allocate PD	Mandatory	Privileged
Deallocate PD	Mandatory	Privileged
Allocate RDD	Optional RD Service	Privileged
Deallocate RDD	Optional RD Service	Privileged
Create AV	Mandatory	Privileged
Query AV	Mandatory	Privileged
Modify AV	Mandatory	Privileged
Destroy AV	Mandatory	Privileged
Create QP	Mandatory	Privileged
Modify QP	Mandatory	Privileged
Query QP	Mandatory	Privileged
Destroy QP	Mandatory	Privileged
Get Special QP	Mandatory	Privileged
Create CQ	Mandatory	Privileged
Query CQ	Mandatory	Privileged
Resize CQ	Mandatory	Privileged
Destroy CQ	Mandatory	Privileged
Create EE	Optional RD Service	Privileged
Modify EE	Optional RD Service	Privileged
Query EE	Optional RD Service	Privileged
Destroy EE	Optional RD Service	Privileged
Register Memory Region	Mandatory	Privileged
Register Physical Memory Region	Mandatory	Privileged
Query Memory Region	Mandatory	Privileged
Deregister Memory Region	Mandatory	Privileged
Reregister Memory Region	Mandatory	Privileged
Reregister Physical Memory	Mandatory	Privileged
Register Shared Memory Region	Mandatory	Privileged
Allocate Memory Window	Mandatory	Privileged
Query Memory Window	Mandatory	Privileged
Bind Memory Window	Mandatory	User Level
Deallocate Memory Window	Mandatory	Privileged
Attach QP to Multicast Group	Optional UD Multicast Service	Privileged
Detach QP From	Optional UD Multicast	Privileged

Verb	*Mandatory/Optional*	*Privileged or User Level*
Multicast Group	Service	
Post Send	Mandatory	User Level
Post Recv	Mandatory	User Level
Poll CQ	Mandatory	User Level
Request Completion Notification	Mandatory	User Level
Local Mad Operation	Not defined in InfiniBand Specifications.	Kernel Mode (Optional)

NOTE

Deviation from the InfiniBand Specifications

The following changes are added to the Linux Verbs Specification for ease of implementation and to provide more flexibility to the API's.

1. Set async error handler, completion handler calls were removed as separate set functions. This was necessary since the *ci_open_ca ()* itself took the callbacks to be registered. This avoids confusing cases, when the event is ready for dispatch to an earlier registered callback function, when the
2. InfiniBand Specifications define the SMA and GSA to be part of the Channel Interface (CI). The issue is that both the QP's are required above the verbs also for Subnet Manager and Communication Manager functionalities among many. Some hardware vendors may provide this as a real QP interface, so that some processor agent in the HCA does the processing of MAD's. Not all HCA's may have this facility, moreover the SMA functionality is well defined by the InfiniBand Specifications, hence each vendor do not need to re-invent the wheel. In order to not complicate the implementation for most HCA vendors, who would otherwise emulate these functionalities, the Linux Implementation provides this for MAD's directed at the local HCA.

   Note: This interface is not visible above the Access Layer. Access Layer provides an interface similar to QP for the aliased QP's. The work requests submitted to the aliased QP's exposed by the InfiniBand Access Layer would automatically use the *ci_local_mad ()* API's.

   Vendors supporting the special QP with native QP semantics do not need to provide the implementation for *ci_local_mad ()*. The CA attribute Structure also exposes this capability via a Structure member. Please refer to the API documentation for more details.
3. InfiniBand Specifications does not specify an association between the PD and the Completion Queue. This association is necessary to restrict doorbell space access from user mode to processes that only own the Completion Queue.
4. Verbs in general specify many handles to be passed, some of which are redundant. Since in most cases where both a CA handle and PD handle is required, just a PD handle would be sufficient, since the PD was allocated on a CA. The vendor could track the CA handle within a PD handle. Hence in almost all API's the number of handles to the Verb API's is far reduced. Please refer to the `ib_ci.h` for relevant information on specific API's.

5. Driver Initialization

This section covers the basic initialization steps that an driver developer would need to perform before the component can be used. We will cover all steps necessary from identification, to setup, before the Verbs driver registers with the Access Layer via *ib_register_ca ().*

5.1. Identifying Host Channel Adapter

Each vendor would have a unique vendor and device id advertised in the PCI configuration space. Most new architectures including 3GIO use the same method to enumerate devices attached to the system. Each vendor HCA would have these values obtained from the PCI SIG. Table 5-1 below shows some sample values for your reference

Table 5-1 Sample Device ID Table

Device Type	*Vendor ID*	*Device ID*	*PCI BaseClass*	*PCI SubClass*
Intel® GEN1 HCA	0x8086	0x3101	0x2	0x80

HCA driver would use existing OS enumeration schemes to identify the HCA using the device id fields as shown above. Linux PCI driver registration functions allow a single registration to specify more than one device id combinations. Refer to *pci_register_driver ()* in the Linux kernel sources. The steps are listed below

1. Register with PCI driver subsystem
2. PCI driver calls the probe function for each available device on the PCI bus. Linux PCI driver calls the *pci_probe* function and provides a pointer to the *pci_dev* structure.
3. HCA driver determines *Register Base Address* necessary to be able to control and configure the hardware. For e.g. a Linux kernel mode driver, we would use the *pci_resource_start (pci_dev, 0)*, and *pci_resource_len (pci_dev, 0)* to obtain the start physical address and length of the register space.
4. HCA driver then uses these values to map and obtain a kernel virtual address for this physical address for use in the kernel mode driver.
5. HCA driver now performs various initializations on the component to make it ready for use.
6. Registers with the IB Access Layer indicating that this HCA is ready for use via the *ib_register_ca ()*. As part of the registration, it also sends the HCA EUI64 identifier to identify the HCA being added to the system.

The exact order of the component on discovery is vendor dependent. The sequences illustrated here should be treated as reference only.

Figure 5-1 and Figure 5-2 illustrate the example load and unload sequences for a typical HCA driver.

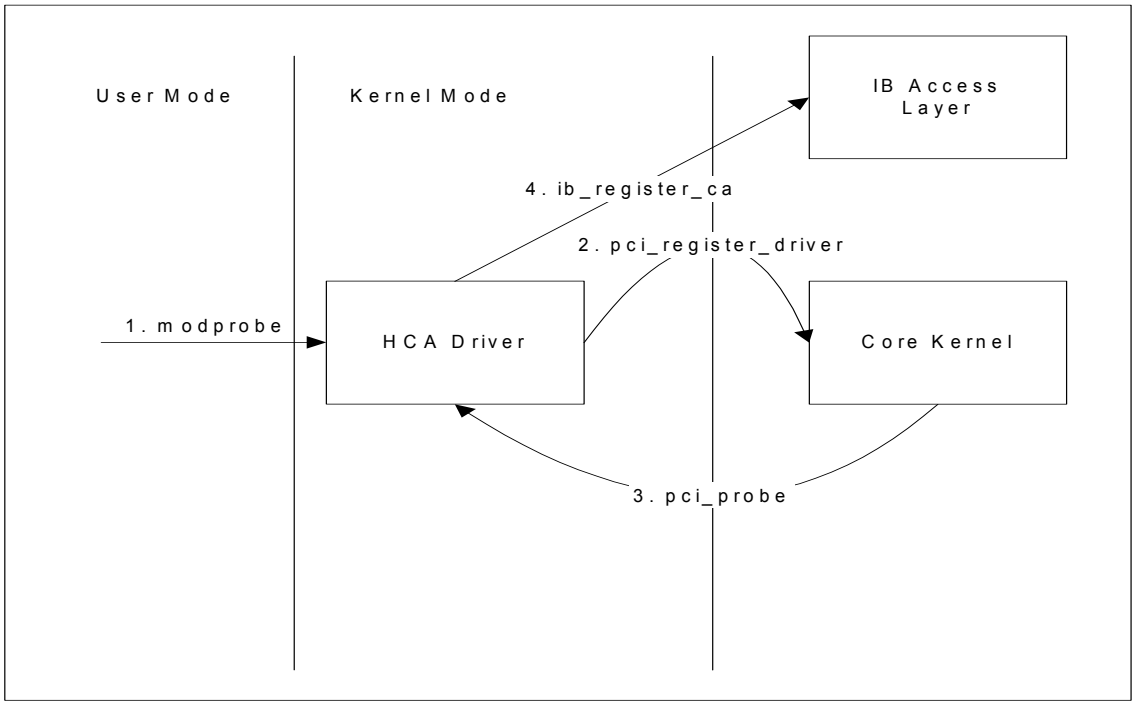

Figure 5-1 Driver Load sequence

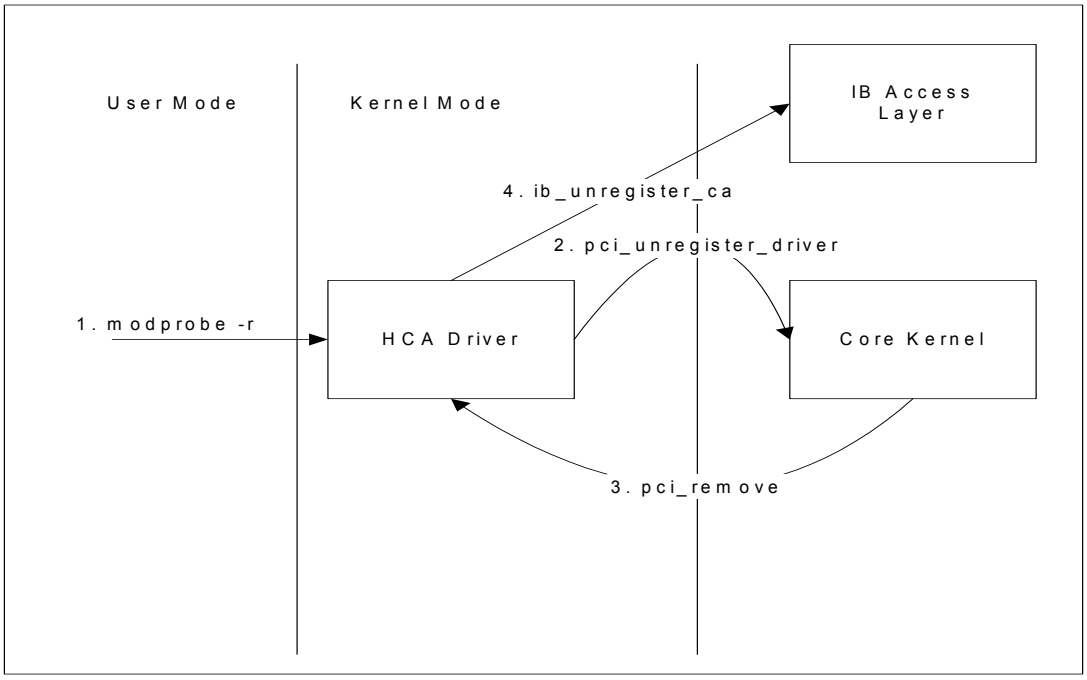

Figure 5-2 Driver Unload Sequence

5.2. Initializing the Host Channel Adapter

Most HCA's would provide several advanced configuration options to suite any OS environment and any specific application requirement, for example configuring the page size support for doorbell space could be different for different operating environments. Linux* x86 requires a 4K stride. Itanium® Architecture has a configurable page size. Current RedHat distributions are shipped with 16K page sizes. The following sections identify some the general configuration sequence.

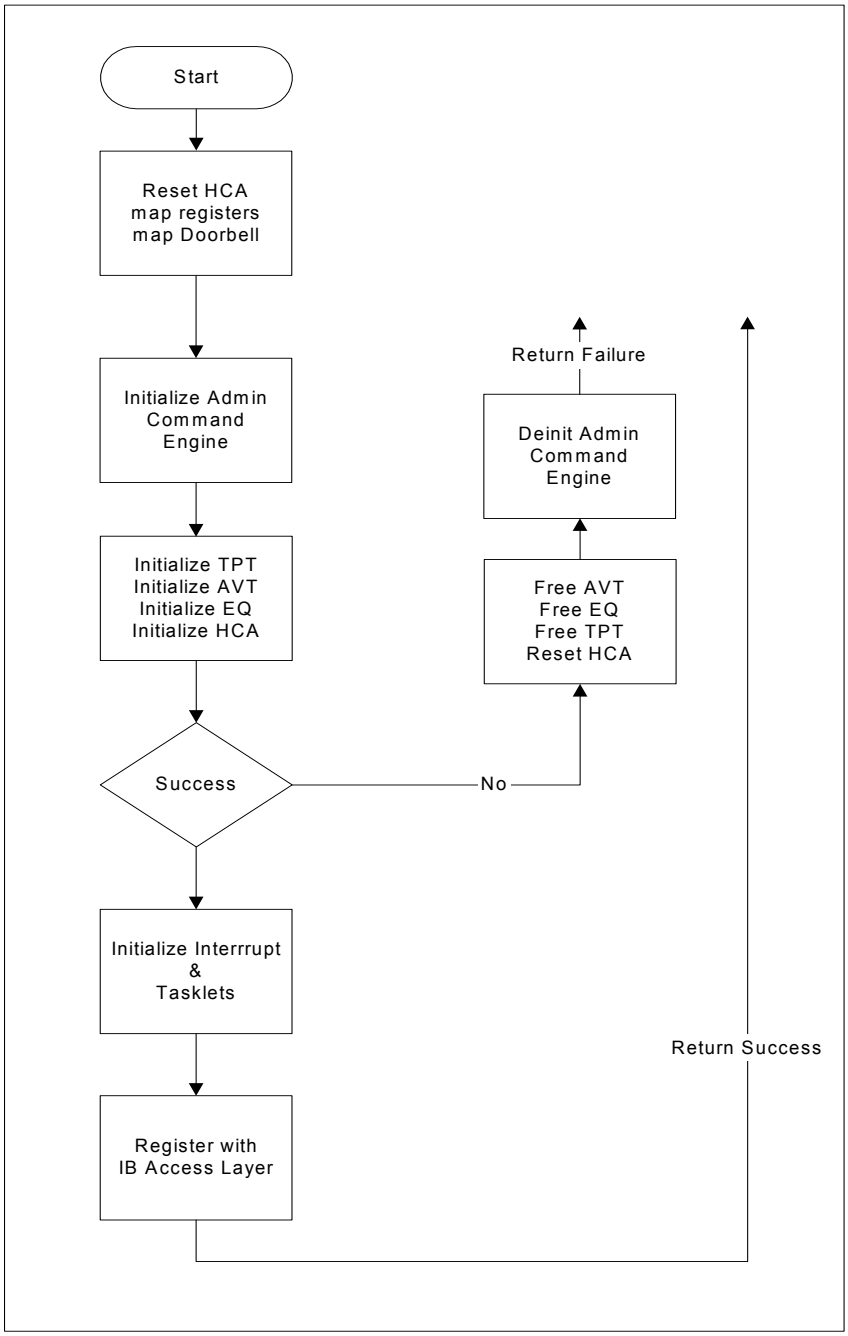

Figure 5-3 Initializing the HCA

Figure 5-3 above shows a typical initialization sequence for initializing the HCA in a Linux driver environment.

HCA vendors may abstract different initialization steps and provide a simple programmatic interface to initialize the component. Once HCA is identified and resources mapped for driver use, the first step that the driver would perform is to issue a reset to initialize the component. The sequence of steps is listed below assuming such abstractions are available.

1. Issue Reset to hardware
2. Initialize and map device specific space for driver use.
3. Initialize Context structures required to track this instance of the HCA
4. Initialize system resources in system memory such as
    a. Memory Management interfaces
    b. Address Vector Table
    c. Event Queue
    d. General resource allocation strategy for QPs and CQs
5. Perform the Initialize HCA administrative command.
6. Retrieve EUI64 identifiers for the adapter and all ports supported.
7. Register with OS for interrupt processing
8. Inform the Access Layer about new HCA with the verb interface structure via the *ib_register_ca()*.

NOTE

For exact details on retrieving EUI64 identifiers refer to the vendors Programmers reference manual.

Some typical parameters required to initialize the HCA are listed below. Each vendor may require different setting, these are provided purely as an example.

Table 5-2 Initialize HCA Parameters

Parameter	*Comments*
Doorbell Area Stride (PAGE_SIZE value)	Linux IA-32=4K Linux IA-64 = 16K
Maximum QP's	Vendor Configurable
Maximum CQ's	Vendor Configurable

5.2.1. Memory Management

HCA implementations must support a mechanism to translate virtual addresses to physical addresses. This is necessary so that the HCA can perform DMA operations without involving host CPU. This facility is referred to as the Virtual To Physical Translation Table (VPTT) in this document.

VPTT is used for both internal resources and to register memory regions with a HCA. HCA's may choose to create the Work Queue Element (WQE) rings and the Completion Queue Entry (CQE) rings in host memory. HCA's would be required to read or write from these memory locations, which may be formed by completely disjoint physical pages. Hence VPTT may be used for managing internal resources and for registering client specific memory regions for performing data transfer operations to or from these memory locations.

Modern operating systems have a limited set of physically contiguous memory for some special purpose drivers that cannot handle non-contiguous memory regions. In embedded operating systems most memory allocation calls may naturally be physically contiguous. In such cases, certain HCA vendors may provide a physical mode operation, where there is no virtual to physical translation, and hence would improve system performance in such special operating environments.

The allocation and maintenance of VPTT in HCA drivers is further explained in Memory Management sections, later in this document.

5.3. Address Vector Initialization

Address Vectors (AV) is used to provide remote node address information for Unreliable datagram (UD) transport service type. AV's can exist in system memory, or device memory located locally in the HCA.

An HCA vendor may choose to support AVT's in many different forms.

- AVT's in device memory – This requires that each address vector creation be managed in conjunction with the device. Privileged mode driver manages operations such as create, modify and destroy address vectors.
- AVT in system memory managed by kernel mode driver – Some HCA vendors may choose to provide all AVT in system memory, yet controlled by a kernel mode agent. This may be required since AVT has protection domain parameters, which must be validated, that only a kernel mode trusted agent could perform.
- AVT in user space – Some HCA vendors may choose to provide AVT in user mode local to the process address space. In such implementations the entry for protection domain cannot be validated, since a trusted mode agent did not provide the information. A vendor specific library in user mode may require working in conjunction with its kernel mode driver to exchange critical information necessary to achieve this transparently.
- A combination of the above schemes.

Depending on where the AVT's are located the HCA driver may need to allocate kernel memory and initialize them prior to the HCA being advertised to the Access Layer.

Some important considerations that a HCA vendor should consider for exposing address vectors in user mode are listed below.

5.3.1. Important User Space AVT Considerations

When AVT is provided in user address space, the vendor would typically require implementing some mechanism to do the following for compliance.

1. Pass an L_KEY and virtual address of memory registered in process address space. This is required so that the HCA can DMA the AVT entry directly from user address space.
2. The protection domain of the registered memory region that holds the L_KEY must match the protection domain of the QP for which this AVT is being used.
3. The port number of the AVT must match the port number for the QP for which this AVT is being used.
4. The protection domain entry in AVT must be ignored, since only a kernel mode agent can perform protection domain checks.

5.4. Event Queue Initialization

HCA's utilize event queues to indicate attention condition that requires more processing from the driver. Events indicate one of the following conditions. Events do not correspond to interrupt generation. Some HCA may be capable of generating events, but the interrupt generation may be shut off unless requested by the kernel mode HCA driver. Since HCA's may have practically several thousand QP's and CQ's, if the HCA generates events for notifying conditions, that may impede system performance severely.

The event queues must be sized large enough to hold all the possible event generation sources. If the HCA cannot generate an event, it's considered a fatal error, and HCA is expected to immediately shut

down all its services. The event generation scenarios are vendor specific; some sample events are shown below. Please consult the vendor programmer's guide for the HCA you are using to get more specific information.

- Receive and Send Queue Completion Events
- QP or EE state change events
- Receive and Send Queue errors
- Completion queue error events
- Port State change and violation trap events

All the above-mentioned event sources could share a single event queue for a HCA. Some vendors may choose to provide multiple event queues, but co-coordinating them for correct event delivery may be a problem. If the interrupt source is still the same, there is little chance to process events in parallel. It's the completion queue processing that needs to be done in parallel. Since each process would own its own set of completion queues, this comes natural with the right usage model. A driver trying to improve this on its own may hurt OS performance and scheduling methods.

5.4.1. Sizing the Event Queue

Sizing the event queue is one of the most important aspects of the initialization sequence performed by the HCA driver.

The following table provides an example of the number of events each source can generate. In the following example we assume 256K QPs, 256K CQs and 2 ports.

Table 5-3 Event Generation Sources

Resource	*Events Generated*	*HCA Maximum*	*Total Possible EQE*
Queue Pairs	4 events • RTR-RTS/RTS-SQD • Path Migrated event • QP Catastrophic event • Connection established event	256K QP's	256K * 3 = 768K Events
Completion Queues	2 events • Completion notification • CQ Error Notification	256K CQ's	256K * 2 = 512K Events
Port Events	2 events • Port State Change • P_KEY Violation • Q_KEY Violation • Buffer Overrun • Link Integrity	2	2 * 5 = 10 Events

The maximum number of events that a configuration can generate depends on the number of resources that could be allocated and used. The number of possible events can be calculated as follows

MAX_EVENTS = MAX_QP * 4 + MAX_CQ * 2 + MAX_PORTS * 5

5.5. Setting up Multicast

Multicast support is essential for supporting IP over IB. Due to design dependencies for name resolution; the same support is indirectly essential to support Sockets Direct Protocol (SDP). If the HCA vendor supports this capability they must set the appropriate capability bits in the attribute structure to indicate that it can handle multicast operations. If the vendor supports this capability, then it also exports the following information.

- Number of multicast groups supported.
- Number of QP's per multicast group
- Verbs support to add or remove a QP from multicast group.

NOTE

Multicast LIDs can be overloaded. This means that a single multicast LID can refer to more than one multicast-gid. The implementation to support this is vendor specific. InfiniBand architecture does not specify how this capability is handled or made known to the configuration manager. The best choice would be to have a 1-1 mapping, and the overloading can be avoided totally for some efficient HW implementation choices.

NOTE

The vendor implementations are expected to be ready to handle any verb calls before the registration returns. Some implementations of the Access Layer may require notifying upper level protocol drivers about a new HCA arrival, and they may start making verb calls to allocate resources. One such example would be an SMA, who may start allocating the special QP for use on notification immediately.

6. Transport Resource Management

The following sections show some implementation options for managing the large number of resources presented by the HCA in an optimal fashion. The resource management algorithms are only suggested guidelines; an implementer can choose a different scheme as it fits the operating environment.

Resources presented by an HCA can be categorized in 2 different classifications.

- Resources that need to be identified very quickly, since they are required in speed path operations. QP's and CQ's fall in this category since they are required to identify the handles or contexts that needs to be passed up to the consumer to indicate a completion event.
- Resources that need to be just tracked and free list maintained. PD's, RDD's, AV's are such resources. Speed path operations don't need lookup based on resource numbers for these resources.

Resource management for each requirement can be generic, and the particular implementation only abstracts the management of the resource, the resource manager for that class maintains identification or fingerprinting resource numbers.

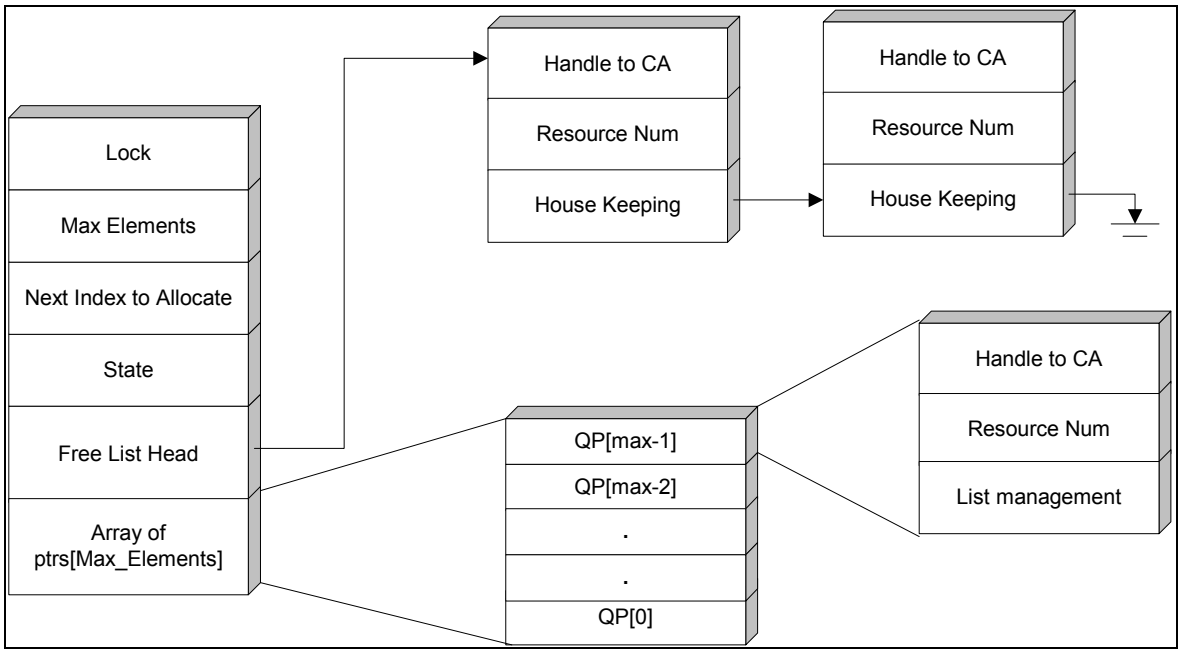

Figure 6-1 Resource Management Structures

Resource such as QP's and CQ's which need to access context values for callback purposes would require a 2 step process for allocation. First the resource is obtained from a free list, once allocated and handed to the client; they occupy slots in the pointer array until deallocated. For purpose of this discussion we will assume that the HCA provides the resource number that needs attention. For example, the CQ number is reported when a completion queue event is being reported to driver software. HCA driver uses this number to directly reference the resource that needs to be attended. This mechanism allows immediate lookup since these operations are speed path sensitive. The size of the array should be sized to hold the maximum number of resources expected. Alternately one could choose a hash bucket to perform another level of lookup. The size of the array is a very small percentage of memory required to accommodate such large resources by the HCA driver. So the array size should be cause for concern.

Certain other resources such as Protection domains, Address Vectors, Reliable Datagram domains do not need access in the speed path operations. Hence the array holding pointers to resources is not a

required. A simple allocation and de-allocation flow is provided below in Figure 6-2. The general idea is that the driver provides a simple scheme that is applicable for all resource types and extended based on special needs as shown above.

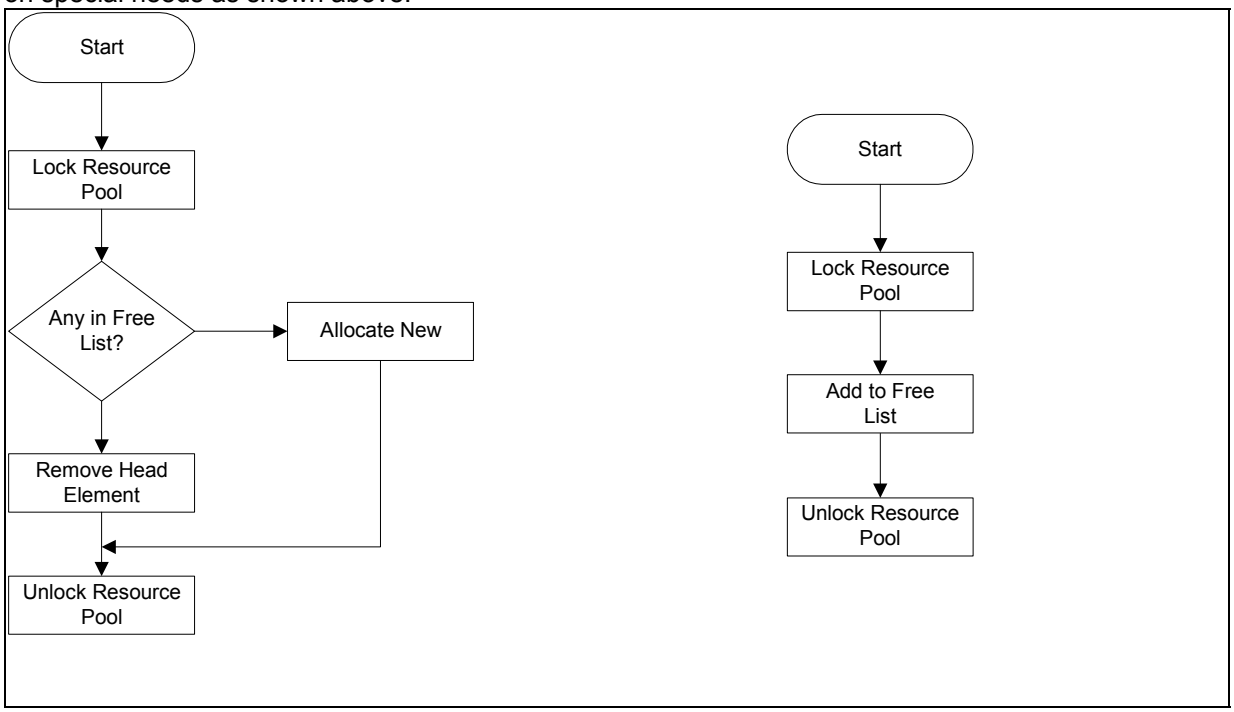

Figure 6-2 Generic Resource Management

6.1. Accessing the HCA

A verb consumer must acquire a handle to the adapter before acquiring any transport resource such as QP's and CQ's. The following sections explain the HCA access verbs.

6.1.1. OpenHCA

OpenHCA verb returns an opaque handle that identifies the HCA instance being open. The InfiniBand Access Layer is the only layer that makes this call. Typically the input parameter would be the HCA's EUI64 identifier that identifies the local HCA. Vendor specific HCA driver allocates a kernel data-structure to identify this open instance. InfiniBand specifications require that this verb only be called once. Driver software must track this so deny further attempts to open this HCA instance.

Figure 6-3 shows some of the fields that the CA handle returned to the verbs consumer holds. The important fields are

- A pointer to the HCA attributes.
- Housekeeping flags can hold certain state information about the CA handle itself.
- Lock member provides serialization access to this structure.
- InUse member keeps track of active callbacks in progress.

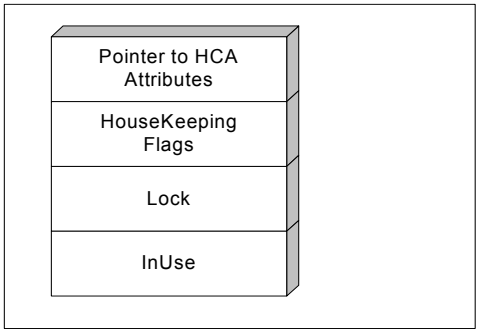

Figure 6-3 HCA Handle

Since the API's are very asynchronous in nature, it is important that the drivers keep track of structure reference counts. This is necessary so that when a active callback is in progress, the verbs driver would not succeed a *ci_close_ca ()*. The *InUse* member is incremented each time a completion or error callback is invoked to the access layer. The *ci_close_ca ()* must block until pending callbacks in progress are complete. For this reason, the consumer must never call the *ci_close_ca ()* from interrupt context.

6.1.2. QueryHCA

QueryHCA verb allows a consumer to inquire properties of the HCA. HCA driver software must maintain limits on resources, and return capability appropriately to the consumer. This verb reports the following data.

- Identifying data about the HCA, which is HCA's EUI64 identifier, IEEE vendor ID, and the device ID.
- Maximum allowed limits on certain hardware resources such as QP's, CQ's, PD's, RDD's and Multicast Group related limits.
- Number of port supported
- Optional features supported
    - Violation counter support, support for raw QP's, Reliable Datagram, Alternate path migration,
    - Ability to change primary port during a SQD->RTS state change.
- Per port data such as
    - Port's EUI64 identifier, Subnet Manager Info, Port States, violation counters etc.

An example of the memory layout of HCA attributes for a 2 port HCA is shown in Figure 6-4 below.

The important note is that the port attributes are located in an array form, so port0 attributes is followed immediately by port1 and so on.

NOTE

When the data is returned to user mode or to a different address space than the kernel, the function that performs the copy must also fix the pointers relevant to the user mode address after performing the copy to user mode buffer. The Access Layer will perform these when the proxy performs the copy to user mode. The buffer received by any vendor driver will receive only have kernel mode addresses. The function *ib_copy_ca_attr ()* can assist in this regard.

When drivers indent to pass some private attributes, such as any special capabilities based on driver loaded to the user mode plug-in, they can keep the extra data at the end of the hca attributes. Setting a larger value in the attributes->size member of the ca attributes structure, AL and the User mode Proxy copy the entire size to user mode.

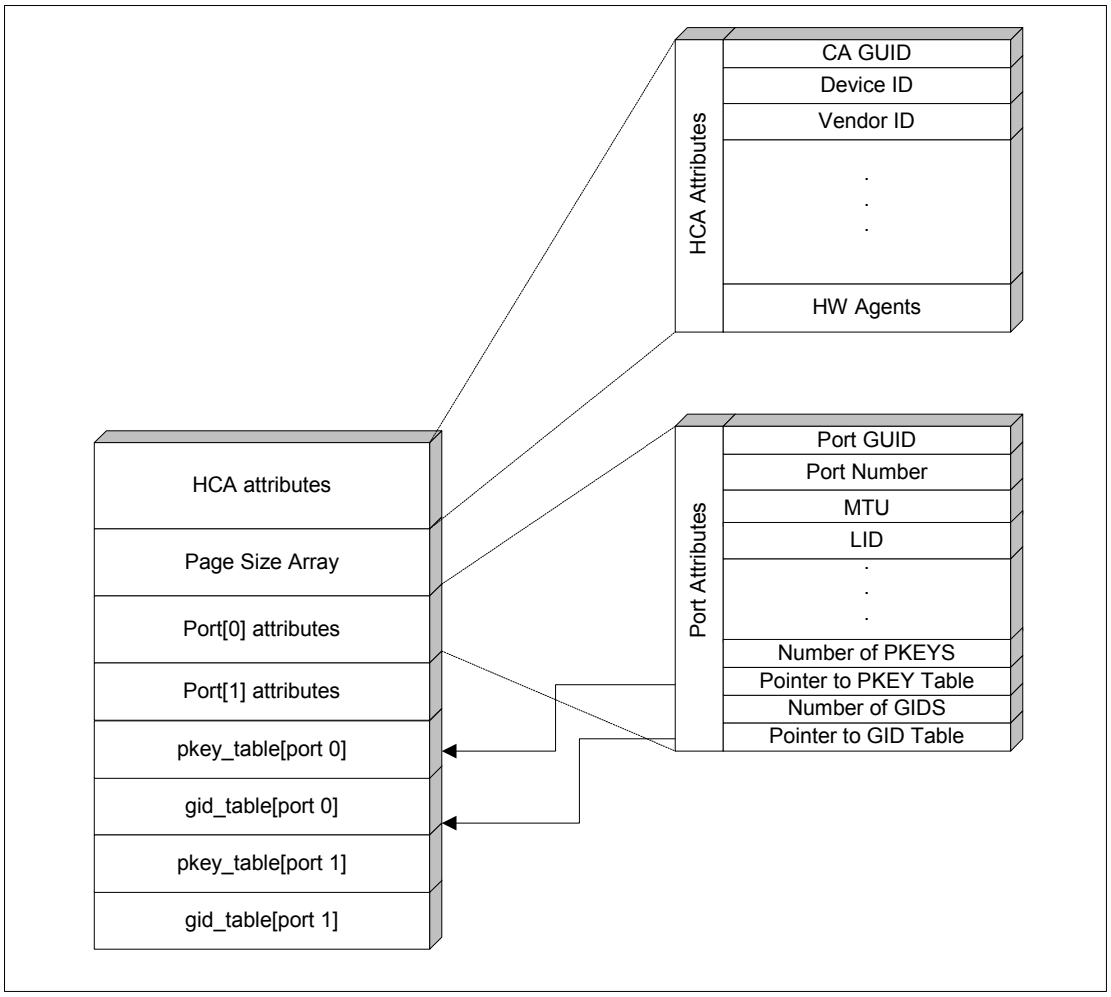

Figure 6-4 Sample HCA Attribute Layout

The total size required for reporting HCA attributes can be computed as below.

```
total_ca_attr_size = sizeof (ib_ca_attr_t) + num_page_supported *
                  sizeof (uint32_t) + num_ports * sizeof (port_attr_t);

for each port
{
      total_ca_attr_size += num_pkeys * sizeof (uint16_t);
      total_ca_attr_size += num_gids * sizeof (ib_gid_t);
}
```

The access layer has some functions that can be used to duplicate the ca attribute structure. They are; *ib_dup_ca_attr, ib_free_ca_attr* and *ib_copy_ca_attr.* Please refer to the documentation on these API's for more information.

6.1.3. ModifyHCA

Modify HCA allows certain attributes to be set by the HCA Driver. This verb is also used to set certain capabilities that can be returned upon query by a subnet manager (SM). The attributes that can be modified by this verb are listed below.

Table 6-1 Modify HCA Attributes

Attribute	*Comment*
IB_CA_MOD_IS_CM_SUPPORTED	Indicates if a connection manager is available on this port
IB_CA_MOD_IS_SNMP_SUPPORTED	Indicates that SNMP agent is present on this port
IB_CA_MOD_IS_DEV_MGMT_SUPPORTED	Indicates that SNMP device management agent is present on this port.
IB_CA_MOD_IS_VEND_SUPPORTED	Indicates that Vendor Management Support is present on this port
IB_CA_MOD_IS_SM	Indicates if Subnet Manager is locally running on this port.
IB_CA_MOD_IS_SM_DISABLED	Indicates if an out of band mechanism turned of SM support on the port. See compliance C16-49.
IB_CA_MOD_QKEY_CTR	Support for initializing the Q_KEY violation counter. This counter is optional, and support must be indicated by setting capability in ca_attr_t.
IS_CA_MOD_PKEY_CTR	Support for initializing the P_KEY violation counter. This counter is optional, and support must be indicated by setting capability in ca_attr_t.

This API operates only on a port. Although the Verb defines that one must be passed for each port, this is cumbersome to manage. Hence the Linux implementation only takes a port level attribute. The verb will co-operate with the internal SMA that supplies data so that the new values are indicated when the port attributes are queried the next time via a SMP, either locally via the *ci_local_mad ()* interface, or externally requested via a SMP request if the SMA is resident on the HCA.

The Verb provider does not need to perform any authentication checks in order to perform these changes. It is expected that the IB Access Layer must have done any necessary authentication prior to invoking the vendor specific interface call.

The verb specification only specifies IsSM, IsSNMP, IsDevMgmt and IsVendor along with the P_KEY and Q_KEY violation counters as required parameters for the *ci_modify_qp* () call. The other values are left to implementation-defined mechanisms. In this implementation the API provides the same mechanism for all capability settings, which are advertised in PortInfo capabilities.

NOTE

- The authentication mechanism is not determined at this time. In a later version of the software, the Access Layer may require a special user account to exist and check if the admin account user is performing these changes, otherwise it must return E_PERM as error.
- Port numbering starts at 1. Port number is not an index.

6.1.4. CloseHCA

This verb indicates that the InfiniBand Access Layer is not going to need any more services from the HCA driver. When making this call it is expected that the caller has destroyed all resources created on this HCA. The HCA driver does very minimal tracking on behalf of its clients, and hence it's the responsibility of the Access Layer to ensure proper cleanup.

CloseHCA also needs to ensure that there are no pending callbacks pending when the destroy call is complete. There are two possible solutions. The verb is considered synchronous like the *free_irq ()* call in the kernel hence the CloseHCA verb cannot be called from interrupt context. The CloseHCA verb would need to spin until the count drops to zero, and any future callbacks must be prevented.

This verb blocks until any pending callback from the HCA driver returns, then sets a flag to indicate that the handle is destroyed before returning back to the consumer.

In Figure 6-5, the "Block Thread" could be a tight spin loop if the call is made from non-blocking sections of the driver, such as timer callbacks etc. It is advised that this call be allowed only when in passive mode, and that the thread can be suspended until this condition can be reached.

Before returning to the client, the HCA driver frees up any resource allocated to track the open instance.

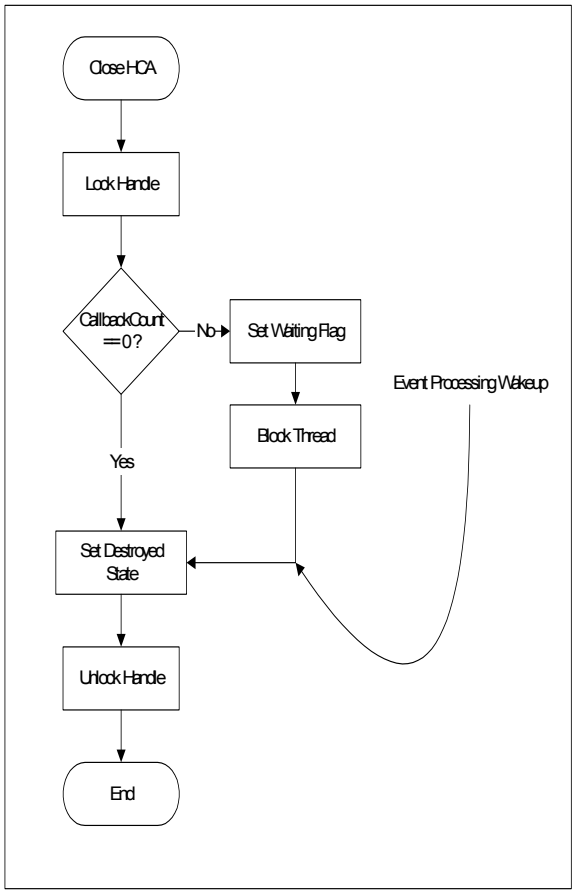

Figure 6-5 CloseHCA

6.2. Protection Domain

Protection domain is used as a form of access security for memory regions, address vectors and queue pairs as defined by the InfiniBand Architecture. The Architecture requires a protection domain before a QP is allocated. Most vendors who expect to support user mode IO, will have a form of notification handshake from user mode. In most HCA's this is a device memory address mapped to the process address space. We call this address the "doorbell" in this document.

In order to protect the notifications from errant processes ringing doorbells, it is required that the doorbell be page aligned and also be located in the process address space for user mode access. In order to make this process level association for Completion queues, the Linux Implementation requires a protection domain for allocating a Completion Queue, although the architecture does not require one.

Some vendors may have the protection domain object as a plain integer for verification purposes to perform access checks, and a separate memory space for user mode access. In such cases the protection domain handle must internally allocate one such address from device memory space if the protection domain is intended for use in user mode.

NOTE

The doorbell stride must be equal to the PAGE_SIZE of the kernel. For IA32 Linux, this value is 4K, and 16K for RedHat® based Itanium® distributions.

6.2.1. Protection Domain Allocation

Protection domain is required to create any resource on a HCA. The relationship is identified in Figure 4-2. The only resource that does not require a protection domain according to InfiniBand Specifications is the Completion Queue resource. For reasons mentioned above, in order to protect Completion queues in user mode, the association to protection domain for a CQ is extended to provide the same purpose of access check and security.

Protection domain resources can be maintained using a generic pool of free resources, just maintaining the next available resource as shown in Figure 6-2. The scheme of using an array to track resources by number is not required to maintain free protection domain resources.

NOTE

Some HCA vendors may choose to provide a special device memory address, which can operate on any CQ or QP. Which for purpose of this document is called the Privileged Notification Address (PNA). This device memory can be used for all kernel mode verb support, since there is no special protection required. The vendor could still use a different protection domain when asked for, but re-use the same PNA for kernel mode applications since they are all in the same address space.

Figure 6-6 shows the fields tracked as part of a PD handle. The protection domain number identifies the resource number in use. The reference count field is used whenever a resource being created belongs to this protection domain. The resources are:

- Completion Queue
- Queue Pairs and EE's
- Create Address Vector
- Register Memory operations

The reference count gets incremented with each resource association with a protection domain, and decremented during each resource being disassociated. The List management fields are used when the PD is being freed.

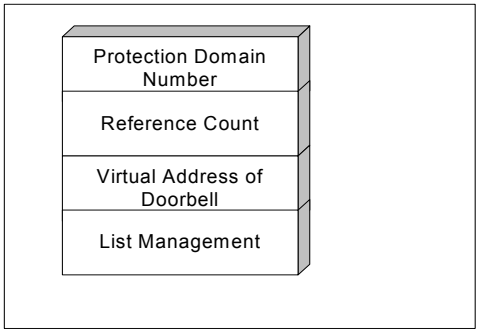

Figure 6-6 PD Handle

6.2.2. Preparing Doorbells for User Mode Access

User mode processes require a mechanism to directly notify hardware for fast IO purposes. Doorbells are required for the following purposes

- Posting Work Request to Send Queue.
- Posting Work Request to the Receive Queue.
- Enable notifications on the Completion Queue to request event notification when a new entry is added to the CQ.

If a HCA vendor supports such a facility, then the vendor is required to support a library in user mode to support vendor specific functionality to facilitate user mode access.

All resource allocations are performed via the kernel proxy agent that is a component of the Access Layer that facilitates user mode support. In order to use this mechanism the steps required by a HCA vendor are:

1. Vendor plugin *uvp_pre_allocate_pd ()* provides the vendor a chance to prepare a buffer to receive information from its own kernel mode library. The vendor library would setup a buffer to receive the physical address of the doorbell, which required to be mapped to process user address space.
2. The user mode support now performs the ioctl call, and passes an appropriate sized buffer requested by the vendor library to its kernel mode agent. The presence of the buffer is an indication that this resource is being allocated for user mode. The vendor kernel mode library now copies relevant information that would be passed back to the user mode library.
3. Vendor plugin *uvp_post_allocate_pd ()* is called after a successful call to allocate a protection domain. The *umv_buf_t* holds the data that was sent from the verbs kernel mode driver. In this case we assume that the physical address of the doorbell was passed. Now the vendor library performs an *mmap ()* call to obtain a user virtual address for a kernel virtual address.

NOTE

The association of a protection domain with a doorbell is only an example illustration. Every vendor may have their own unique way to allocate these objects. The library calls in user mode provide additional flexibility for each vendor to invent their own schemes to support these options.

6.3. Reliable Datagram Domain

Most HCA vendors do not manage Reliable Datagram Domain (RDD) as a hardware resource. This is because a RDD object is created and co-managed potentially between multiple processes, so that they can reuse the same End-to-End Context resource between processes. This could be just a 32bit number that identifies a RDD used to create an EE for use in RD style connections. HCA driver writer simply needs to maintain what RDD's numbers are allocated and manage the resources. The scheme depicted in Figure 6-2 can be used to maintain RDD resources.

Reliable datagram domain handle and related information is shown on the right in Figure 6-7. When creating an EE or a RD QP, this reference count is incremented. When an attempt is made to destroy the RDD handle, if the reference count is not zero, then the HCA driver must indicate that the resource is in use.

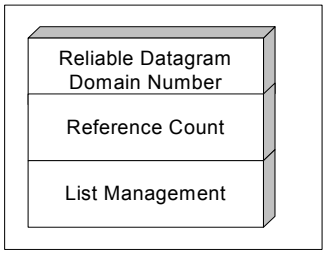

Figure 6-7 RDD Handle

6.4. Address Vector Management

Address Vectors provide local and remote address information for all Unreliable datagram (UD) PostSend calls. Some HCA vendors may provide facility to use create address vectors in user mode. As discussed before in section 5.3, there are several methods. In this example we will assume 2 cases, which should cover most common cases.

6.4.1. Address Handles Allocated in Kernel mode

In this example, we assume that the address vectors are allocated in kernel mode. During HCA initialization kernel mode driver has done the following.

1. Allocate virtually contiguous memory required to hold a pre-defined MAX_AV number of address vectors.
2. Register the memory required with the HCA. This is required so that the HCA could compute where to DMA the address vector from memory.
3. Indicate the start of the virtual address, the L_KEY of the registered region, and the number of address vectors available.

For the purpose of this discussion we will assume that the HCA requires the index to the address vector, in order to determine where in physical memory the AV entry is present. Since the HCA is aware of the base for the AVT, it can compute what the virtual address is, and hence the physical address of the AVT entry.

6.4.1.1. AV Handle for Kernel Mode Address Vectors

AV Handle tracks the address vector handles created and managed by the HCA driver software. The AV resource manager allocates these structures on demand. Since the AVT table is a virtually contiguous table, HCA driver can identify the system memory associated with this index. When address vectors are released, the handles are recycled to the free pool for future allocation. Typical information stored as part of the address vector is shown on the right in Figure 6-8.

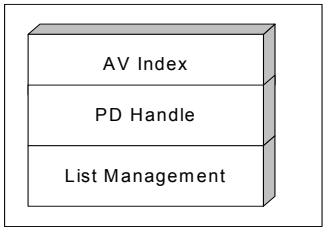

Figure 6-8 Address Vector Handle

6.4.1.2. User Mode Access for Kernel Mode Address Vectors

In the above example the index of the AVT is the only necessary info that the HCA requires for accessing an AVT. So the sequences of steps are:

1. Setup some user mode buffers to receive the index of the AVT during the *uvp_pre_create_av ()*
2. Proxy agent calls the kernel mode driver and passes a buffer to pass this information back to user mode library.
3. User mode plugin *uvp_post_create_av ()* is called with the AVT. The umv_buf_t passed now contains the index of the AVT that the user mode library would use to allocate and store in the user mode buffer for future use.

In future *ci_post_send ()* calls to the send queue of an UD QP, the user mode post routine would now have enough knowledge about how to construct the AV information necessary for this specific HCA.

6.4.2. Managing AV Entries in user mode

Some HCA vendors may choose to keep the AV for user mode applications entirely in user mode. A possible implementation and method to handle user mode access is outlined in this section.

HCA vendors are required to perform some extra checks in order to be compliant with the InfiniBand Specifications.

1. AV entry memory area must be pinned, and an L_KEY produced for the memory region holding the AV entry.
2. Protection Domain checks: - Each AV entry has a protection domain entry. Since the address vectors are hosted in user mode, the value of the PD cannot be trusted. InfiniBand Specifications require that the PD of the AV entry and the PD of the QP for which it's being used must be the same. The Vendor would require ignoring the PD in the AV entry record, but instead use the PD of the memory region used to register this user mode address vector.
3. Port number checks: - This is similar to the PD check. The AVT has a port number field, which must be validated. In the case the AV entry is allocated in kernel mode, the kernel mode driver would have checked and returned an immediate error. In the case the AV entry is hosted in user address space, the HCA would require to perform run time checks for validity of the QP and the AV entry port number to match, or return an appropriate error as required by the Verbs specification.

6.4.2.1. User Mode access for Address Vectors in User Address Space

There is much more the user mode library is expected to do to support address vectors completely in user address space. The primary factors that govern that are.

1. Memory registration cannot be performed for each and every address vector being allocated. This would negate any performance gain of allocating them in user mode, since a kernel transition is not avoided.
2. User mode Address Vector creation would now need to track the L_KEY of the region and the address in combination to indicate an address vector. Since in the case as shown in Section 6.4.1, the user mode could just trust the index returned from kernel mode driver.
3. Manage a pool of address vectors for each HCA, Port number and PD tuple. When the poll is empty, then a new buffer needs to be registered and tracked for cleanup operations.

The interaction between the HCA library in user mode and the driver in kernel mode is depicted below.

1. During the *uvp_pre_create_av ()* the user mode acquires a large buffer and passes the address via the umv_buf_t to its kernel mode driver.
2. Kernel mode driver, registers the buffer, and passes the L_KEY to the use mode HCA library via the umv_buf_t.
3. User mode library now initializes its poll of AV entries and returns one handle back to user mode application to satisfy the request.
4. On future calls to *uvp_pre_create_av ()*, if a buffer is available from the pre-registered pool, then the call would allocate a handle and return IB_VERBS_PROCESSING_DONE to indicate that a kernel mode call is not necessary any more. The memory pinned for this purpose must be cleaned when the original address vector is being destroyed as part of final cleanup.

The user mode library will also short circuit the user mode plugin *uvp_pre_destroy_av ()*, *uvp_pre_query_av ()* and *uvp_pre_modify_av ()* are also required to use the special return code IB_VERBS_PROCESSING_DONE, in order to avoid making the call via the proxy kernel agent.

6.5. Managing Queue Pairs

Queue pairs are managed just like any other resource management as described in Figure 6-2. Since QP's resource access would be required during callback or event generation time, its important to be able to perform a quick lookup based on resource numbers. The assumption here is that the HCA would provide the HCA driver the resource number to identify the context in kernel mode for event propogation. If the HCA has any special requirements for managing special QP's then the driver must

ensure that those are reserved before registering with the Access Layer. The HCA driver must also have no assumptions on the sequence, or ordering of allocation to rely on the allocation pattern of the Access Layer.

For e.g. if an HCA requires that the real QP0 and QP1 be reserved for port 0, then these resources must be reserved and never be available when a another *ci_create_qp ()* calls is made on behalf of a client.

6.5.1. Creating Queue Pairs

Create Queue Pair involves allocating a QP resource, registering the creation with the HCA using an appropriate administrative command, setting initial states and QP query attributes before returning the newly created QP handle. The type of QP being created is typically communicated to HCA via the administrative command to create a queue pair. An example list of parameters is shown below as reference in Table 6-2.

Table 6-2 WQE Creation Parameters

Feature	Comment
WQE Depth	Maximum Number of WQE's.
Scatter Gather Lists	Number of SGL's per WQE.
WQE Ring Start Address	Must be registered with the HCA.
Completion Queue	CQ number for the Send & Receive Queue
QP Signaling Type	Signaled always, or Signaled via each WQE
Protection Domain	May additionally indicate doorbell pages valid for this QP.
WQE Page Size	Page size for the WQE memory being registered.
Type of QP	Transport type for this QP (RC, UC, RD, UD, Raw, Special QPX)

The algorithm for creating a QP is shown in Figure 6-9. The primary tasks in allocating a QP for use would involve the following.

- Allocate a free QP
- Compute the size of the WQE's for Send and Receive Queues and allocate memory for WQE. The number of SGL determines the WQE sizes, and the numbers of WQE's determine the space to hold the WQE's.
- Allocate VPTT to present the address space as a virtually contiguous space to the HCA driver.
- Define the parameters and perform the administrative command for HCA to inform the HCA to initialize any necessary context for this QP.

NOTE

The scenarios explained below and the parameters are just example parameters. Each vendor requirement may include additional requirements.

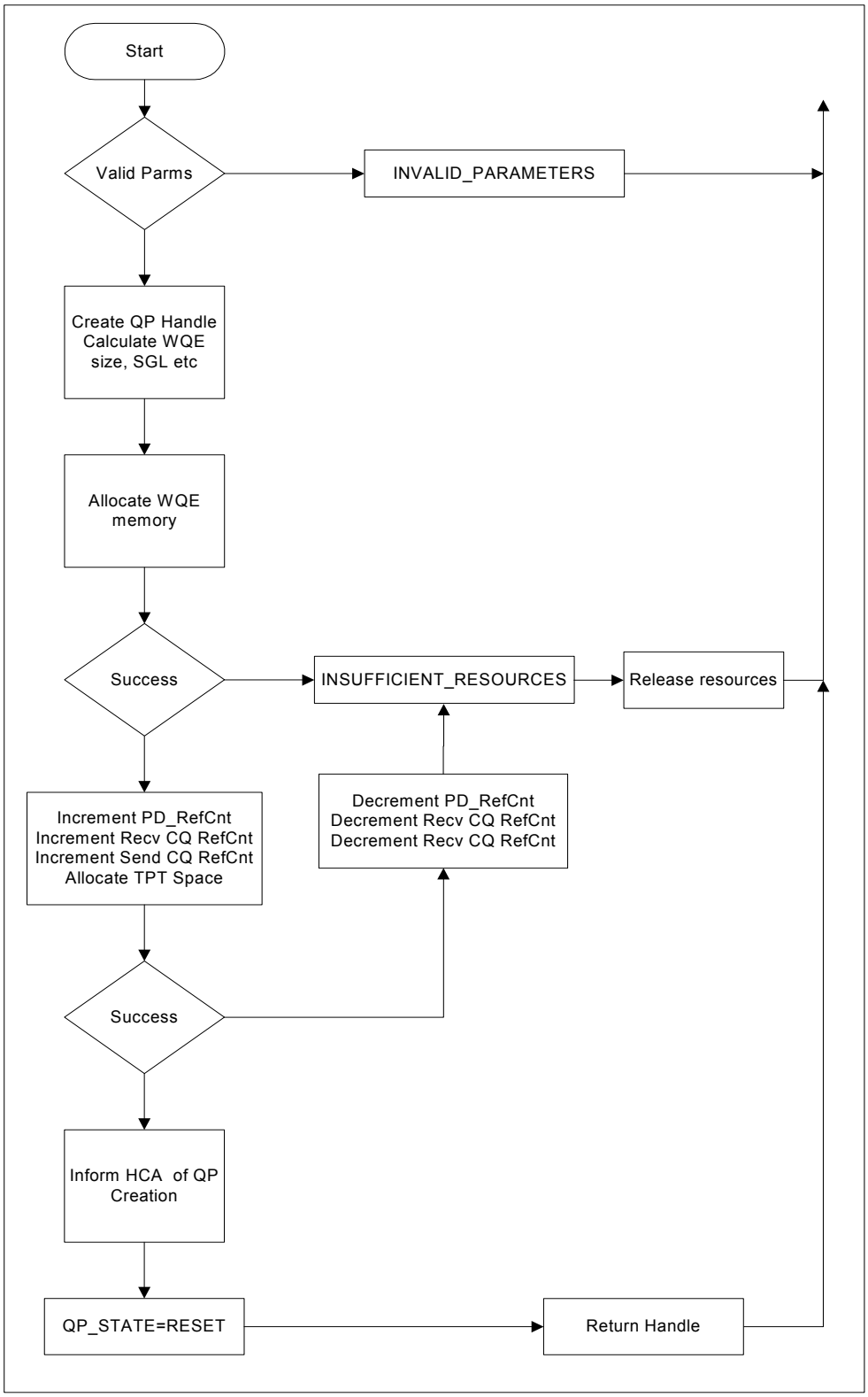

Figure 6-9 Creating a QP or EE

6.5.1.1. Queue Pair Handle

The Queue Pair handle is used to keep track of all details of the queue pair while in use. Some of the fields that are required are shown in Figure 6-10.

- Handle to open instance to track the HCA on which the resource was created.
- Handle to protection domain to which this QP belongs.
- Send and Receive Queue information hold the following information of the specific queues
  - Pointer to CQ controlled per WQ data-structure
  - Index where the next WQE to post
  - Number of WQE's posted
  - Ring Size for this Queue
  - Pointer to WQE ring base
- Generic QP specific Information
  - QP Number
  - L_KEY of registered region for WQE space.
  - QP Type
  - Number of SGL's etc.

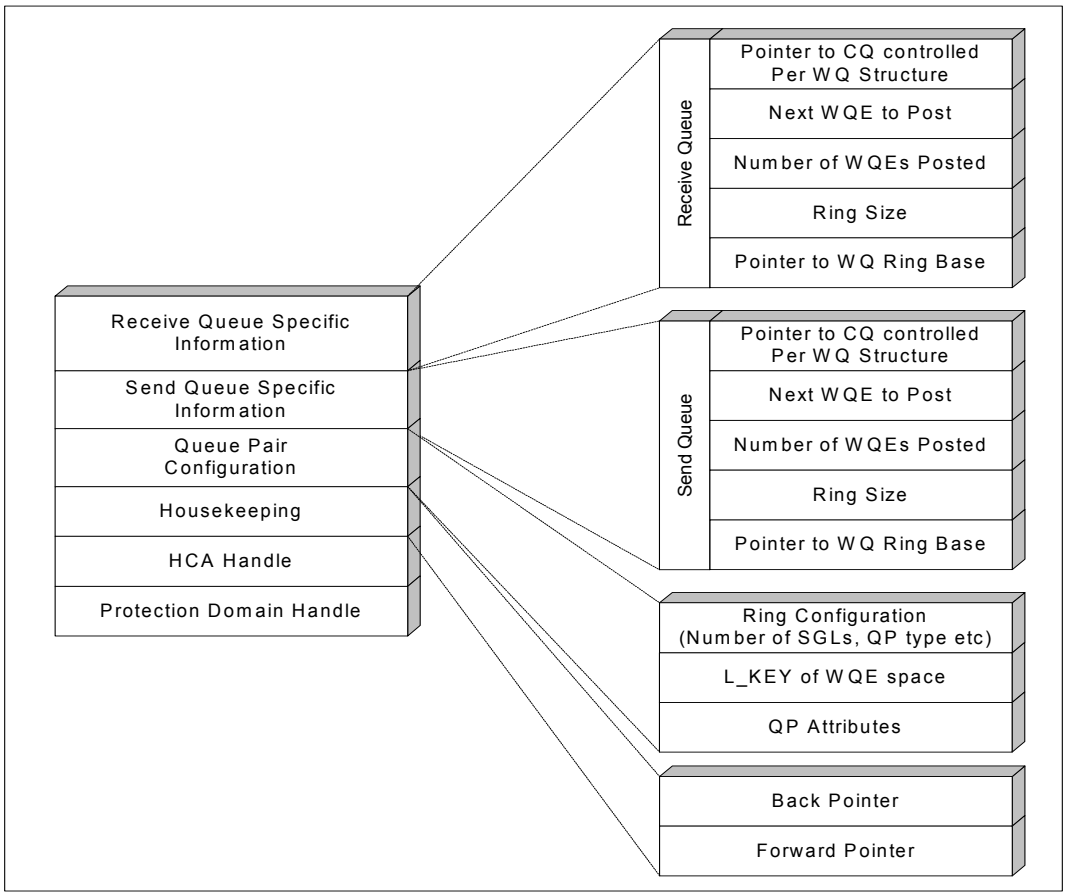

Figure 6-10 QP Handle Data-Structure

6.5.2. Modifying QP Attributes

A verb consumer uses this interface to modify QP states, and also to change some operating parameters for the QP depending on if that change is supported. Figure 6-11 below shows the permitted QP states and how ModifyQP verb can transition states. HCA driver may also keep track of

the current state information in the QP handle. This is handy to satisfy the QueryQP verb, in addition certain compliance statements in Work Request Processing require that the verbs return immediate error if posts to the send side is attempted when the QP is not in RTS state.

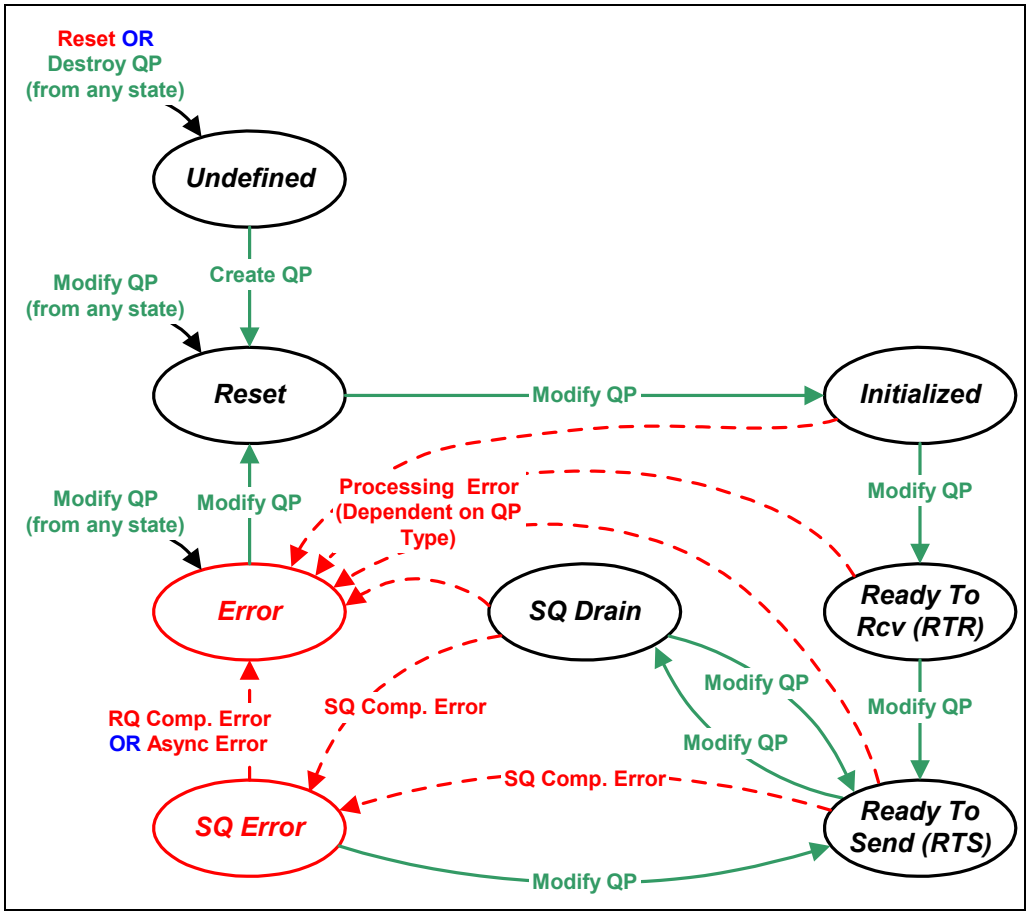

Figure 6-11 QP States

InfiniBand architecture allows several parameters to be modified during each these QP state transitions. Some need to be supported by all HCA's and some are optional and supported only if the CI permits those operations. The options are listed in Table 79 in Chapter 11 of the InfiniBand Specifications Volume 1. Please refer to the vendor specific HCA programmer's reference on what values are applicable to a specific HCA. HCA drivers are expected to export such optional capability via the CA attributes structure, so that the Access Layer can have enough knowledge about a particular capability of an HCA.

The verb provider must keep track for QP's in connected state, a reset followed by the next RTR transition must not happen before the TIME WAIT state has expired.

Each QP state transition is unique and discussion is very vendor specific. Such discussion is beyond the scope of this document.

NOTE

Deviation from the InfiniBand Specifications

The access layer provides facilities that destroy resource hierarchy, so that the client does not need to perform and track resources. The specification recommends that the communication manager manages

the TIME WAIT state for a QP and not release it until the time has expired. This behavior implies the following.

- CQ's cannot be destroyed until this QP is destroyed
- PD's cannot be destroyed until the CQ's and QP's are destroyed.
- AV's cannot be destroyed until the PD's are destroyed.
- *ib_close_ca ()* cannot be called until all the above resources are destroyed.

This would also mean that strictly the application couldn't quit until these conditions are satisfied. If the application has several QP's this time to wait will be significant and noticeable. On the other hand the HCA driver without any complexity can very easily maintain this state.

Functionally this does satisfy the sprit of the requirement, which is that the same QP must not be used for another connection before the TIME WAIT period has expired.

 **NOTE**

When processing PostSend () and PostRecv () verbs, the HCA driver is required to return invalid state as an immediate error if the QP is not in the appropriate state. For performance reasons, the HCA driver maintains the state information in the QP data-structures, as each transition is successful. This eliminates expensive query operations to HCA hardware to query current QP states in speed path operations. This state caching is also critical to user mode verb implementations.

6.5.3. User Mode Interactions when creating a QP

When creating a QP for user mode, the user mode vendor specific library may be required to perform special functions in order to create the Work Queue element memory required by the HCA in user address space. This is necessary to support direct user mode communication with the HCA for speed path IO operations. Some typical sequences are shown below.

1. When the vendor plugin *uvp_pre_create_qp ()* the vendor user mode library allocates memory required for holding the WQE rings. The virtual address, size information is passed to the kernel mode driver via the umv_buf_t structure.
2. Kernel mode driver perform validity of the user mode memory passed, and then pins the pages using available OS interfaces. Please consult the raw IO patches for kiovec interfaces to perform this on Linux.
3. The kernel mode driver also registers this memory, so that the L_KEY can be used to inform the HCA about where the WQE rings are located in system memory. This internal memory is registered only for local access and could use the PD for the QP itself as a protection domain for such registrations.
4. Kernel mode driver also needs to indicate the HCA about valid doorbell segment for this QP in user mode. This is required so that if a doorbell is received from a doorbell space that is not authorized, the HCA will not malfunction, but rather ignore the doorbell ring and not process the WQE's erroneously.

From now on the user mode does not need to make kernel transitions for submitting work requests. The vendor specific code could format data as required by the HCA, and notify the HCA via the doorbell.

6.5.4. Destroying a QP

A client calls the DestroyQP verb in order to release QP resources back to the verb provider driver. Typically the destroy operation would involve informing the HCA via the DestroyQP administrative command, which performs DestroyQP operation. Once the QP is destroyed, HCA driver must guarantee that the QP is now returned to reset state, and that any pending completions on behalf of the QP is flushed with error.

The destroy QP verb is a synchronous call, and it will block the caller until all callbacks in progress complete.

NOTE

Verb provider must guarantee the following. This is particularly important if the HCA driver is keeping data-structures and context on behalf of the QP in the Completion Queue tracking structures. Vendors may need this capability to provide additional functionality, or to supplement hardware behavior. As an example consider the following scenario.

1. Verb consumer has pending completion queue entries that still remain and are not retrieved via the *ci_poll_cq ()*.
2. Consumer calls the *ci_destroy_qp ()*.
3. There are no active callbacks in progress, so the consumer has destroyed QP context and successfully returns from the verb.
4. Verb consumer now calls *ci_poll_cq ()*. If this call requires context related to the QP, then such data must still be available to complete the call successfully. The CQ entries being retrieved cannot fail because the QP was destroyed.

The same requirement exists for performing *ci_modify_qp ()* that resets the QP state. This requires that the WQE's will be start from the beginning of the WQE ring and should ensure that the verb must be still able to retrieve earlier completions that existed in the CQ before the QP RESET was performed, as part of the *ci_poll_cq ()* interface.

6.5.5. Important Notes to HCA Driver Writers

This section documents some important issues to keep in mind when writing HCA drivers, especially related to maintaining QP related information.

1. A verb provider must track for QP overflow information. If the consumer submits more requests without processing the corresponding completion records could cause a queue full condition to occur.
2. The Completion Queue processing often requires some tracking to happen on the Queue Pair handle. This is required to keep track of completions. It is important that the driver writer not have cross locks between the *ci_poll_cq ()* and the *ci_post_send ()* routines. This may affect the performance since the post and poll routines would be single threaded.
3. When the QP is being destroyed or a QP is being reset, information necessary to track completions must not be lost. An example would be, some HCA implementations might require tracking additional context as part of the QP handle in order to process completions per WQE. Say to report the Work Request ID (WRID) in the WR. Destroying a QP, or performing a RESET operation on the QP must not affect or alter the ability to poll existing completions in the CQ.
4. When a QP is being destroyed, or transitioned to RESET state should not affect the client ability to Poll the CQ, or affect reporting completions on other QP's that may be bound to the same CQ.
5. When a QP is transitioned from RESET to Init State, the WQE processing always starts from the first WQE. In cases where the QP is either being DRAINED, or moved from SQErr to RTS state, the WQE's are processed from where it stopped last
6. If the QP was part of a multicast group, then remove this QP from all groups this QP is a member.
7. Ensure that no callbacks will be generated once the *ci_destroy_qp ()* call has successfully completed.
8. Properly manage reference counts on the CQ and PD on behalf of this QP. If not properly managed, could lead to resources that can never be destroyed.

9. Preserve any context on behalf of the QP that may be required to process any pending completion queue entries.
10. Deallocate memory, and free VPTT resources used to register memory region to hold the WQE rings.
11. Ensure that any connected QP or EE has expired the TIME WAIT state before returning the QP back to free pool as use to another consumer.
12. If *ci_destroy_qp ()* has completed successfully, and the same QP was re-allocated via a *ci_create_qp ()*, then any old pending events in the event queue must not be delivered to the new owner of the QP. Vendors may choose to track this condition and not return the QP to the free list, until it is certain that no new events are pending that needs to be reported.

6.6. Managing End-To-End Contexts

Some vendors may implement End-To-End contexts as a special attribute to queue pairs. Hence the allocation of a local EEC may be nothing more than allocating a QP from the QP resource pool. The only difference is the during the creation time, the other QP attributes such as WQE base, PD etc do not need to be specified in the administrative command. Hence the QP_HANDLE and the EE_HANDLE can effectively be the same in the example driver under discussion. The algorithm as shown in Figure 6-9 must be used for EE allocation.

The local EE state management is also identical to the QP state management as specified in the IB Spec. Figure 6-11 illustrates the different states and the actions permitted on the local EEC in each state.

6.7. Special Queue Pair Management

The management of special QP is very vendor specific; hence we will only list some general concepts, roles and responsibilities. For more specific information, please refer the programmer's reference manual for that specific HCA under consideration.

What is special about these QP's are that they are multiplexed among different consumers. Hence it creates special issues, and cannot be treated as normal QP's.

6.7.1. SMI QP

SMI QP is also called the QP0 in the architecture specifications. QP0 is the management QP and is the only QP required first to bring the InfiniBand nodes configured by a subnet manager. Most HCA's may require that the real QP0 on the HCA be used only as the special QP0. This is required since the QP number is transmitted in the LRH when packets are sent out. It is possible to use a different QP as a QP0 for port 0, but real QP0 cannot be used for any other purpose.

If the HCA supports an SMA on board, the vendor must expose that capability via setting the `hw_agents` in the `ca_attr_t` structure when registering with the access Layer. This means that when a SM needs to configure the local HCA using direct route SMPs, they would still be posted like normal requests.

This is a special case, since for configuring the local HCA; the HOP_COUNT in the direct route SMPs must be 0. The InfiniBand specifications require that no direct routed SMPs with hop count 0, is sent on the wire. The expectation is that the on board processor would intercept before sending them on the wire.

If the HCA does not posses such capability, then the SMA in the access layer controls these functions. Since the SMA functionality is spec defined, this module is generic to any HCA. For local operations

with HOP_COUNT set as 0 for direct routed SMPs the access layer would use the *ci_local_mad ()* interface to perform configure operations.

6.7.2. GSI QP

GSI QP is called as the QP1 in the architecture specifications. QP1 is the general services QP that can be used for practically any general purpose from connection establishment to running configuration services. Similar to the QP0 interface, the real QP1 from the HCA may be required to be used nothing other than the QP1 functionality. Some implementations may choose to reserve these only for that dedicated function to avoid the QP being used as an application QP.

Unlike the QP0 interface, there are no direct route issues with the QP1. Hence agents in the same node would still get packets looped back due to the Self Address Packet (SAP) requirement, which states that packets are looped back if the source and destination address is the same.

The only gotcha is that a performance manager, which is a vendor specific functionality and hence cannot be resident above the Access Layer. The special QP transport service in Access Layer will verify if the service is for a performance agent, and if the HCA does not support `hw_agents`, then the *ci_local_mad ()* is used to retrieve the information from the local HCA.

Since there is just one physical QP1 for each port, the access layer provides other mechanisms to abstract and provide a QP like service.

6.7.3. Event Generation

All HCA vendors are expected to generate asynchronous events as required by the InfiniBand specifications for notifying software layers about this exception condition. However there are some possible choices to permit flexibility for HCA vendors.

If the HCA supports ability to generate events when a port is available (i.e. port is in ACTIVE or ACTIVE_DEFER states), then the HCA must generate an event to indicate that the port is available.

NOTE

The IB Access Layer depends on this functionality for a lot of the plug and play aspects of the driver. If this notification is not supported, certain drivers may not be loaded or unloaded as expected. It is strongly suggested that the HCA vendor support this feature. Even though the InfiniBand specifications suggest this as a HCA capability advertised by the HCA driver, this would be a required attribute for the software to work correctly.

6.7.4. Trap Generation

Certain HCA's may not support on board processing ability for processing of SMP or MAD messages. Such implementations would use the asynchronous event method. The event structures defined in `ib_types.h` can be used to indicate such conditions.

Table 6-3 Extended Asynchronous Events

Feature	Comment
IB_AE_PKEY_TRAP	Local HCA generating P_KEY violation Trap
IB_AE_QKEY_TRAP	Local HCA generating Q_KEY violation Trap
IB_AE_BUF_OVERUN	Excessive buffer overrun threshold reached
IB_AE_LINK_INTEGRITY	Local link Integrity threshold reached
IB_AE_MKEY_TRAP	Bad M_KEY access attempted

6.8. Completion Queue Management

Each Send and Receive Queue must be associated with a CQ. The CQ is the only means for a verb consumer to obtain completion information from the associated queues. The association between the QP and the CQ remains until the QP is destroyed.

CQ Handle is used to track CQ specific information. CQ's have the following properties or restrictions on their usage.

Some of the parameters that decide the size of the Completion Queue which are vendor specific: -

- Size of each completion queue entry
- CQ entries are written to by HCA and read from verb interfaces to poll completion queue entries.
- Total number of WQE's in send and receive queue of a QP being associated with this CQ
- Number of QP's associated with the CQ.

Vendor may have additional restrictions on memory start address alignment restrictions, which the vendor specific code in kernel and user mode would manage co-operatively.

A sample of elements that would be typically tracked by an HCA vendor is shown below in Figure 6-12. Each vendor requirement is unique, please consult the programmers reference manual for the component that you are using to evaluate additional requirements.

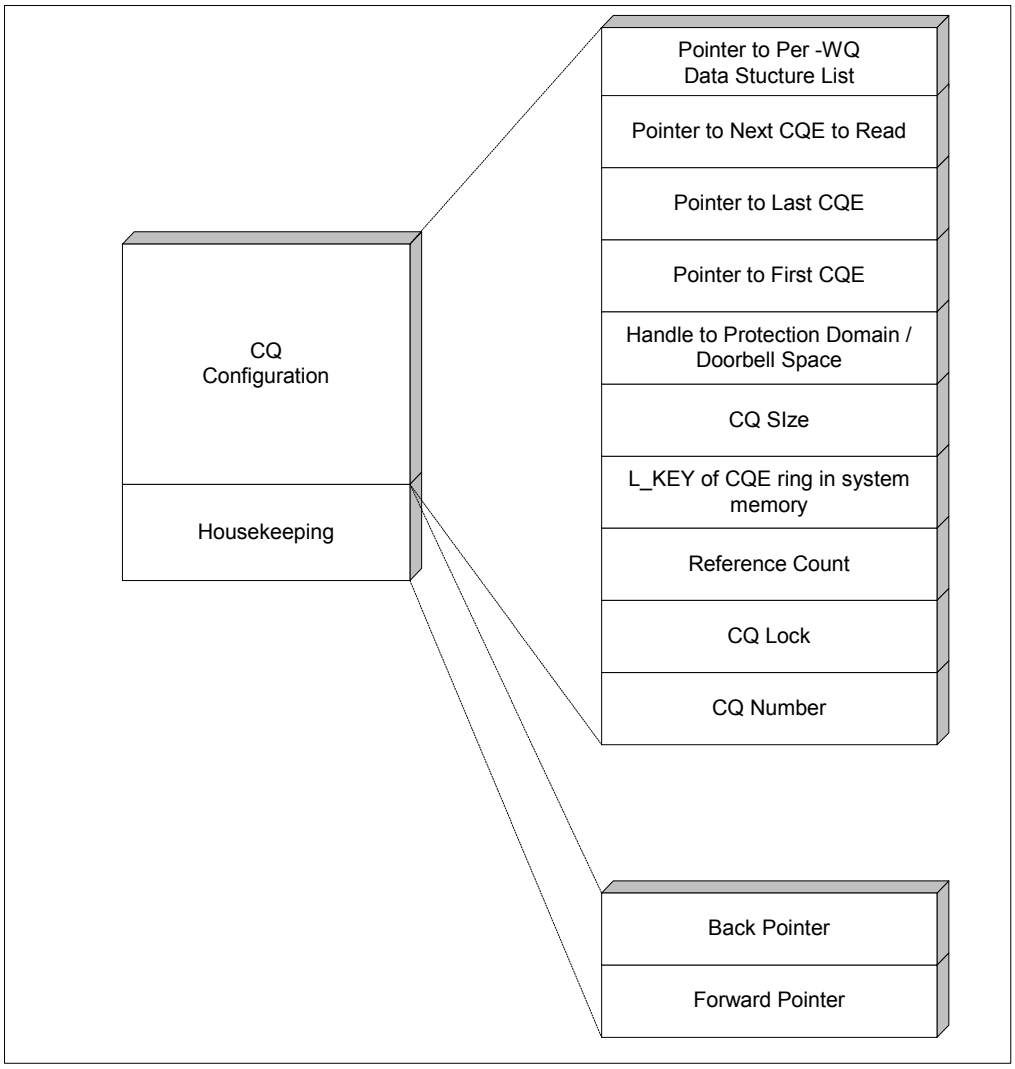

Figure 6-12 CQ Handle

6.8.1. Managing CQ for User Mode Access

Completion queues are a critical part to enable direct IO from user mode. In order to facilitate this the basic requirements are: -

- Completion Queue Entry ring must be located in user mode, so that when notifications are done, the entries can be pulled directly from user mode.
- Doorbell method available for the CQ from the process address space using which the doorbell can be enabled to generate events to indicate future completion notifications.

The sequence of operations that a verb provider library in user mode would perform in conjunction with the kernel mode driver is listed below: -

1. When the vendor plugin call *uvp_pre_create_cq ()* is called, the library performs memory allocation to satisfy the HCA specific requirements, and passes the buffer and size to its kernel mode driver via the umv_buf_t structure.
2. The proxy agent in kernel passes this data to the kernel mode driver. The kernel mode HCA driver will pass the data back to the kernel mode Verbs Driver.
3. HCA driver now is required to perform any specific setup to create this completion queue. As part of this setup, the driver will internally register the memory, so that the HCA can DMA completion records directly to user mode.
4. Notifications are directed to the proxy, which facilitates arranging notifications to the client informing about an interrupt condition.
5. Now the *ci_poll_cq ()* needs to only reformat data as expected by the consumer. As part of the poll, the vendor specific code may need to perform additional data collection from user mode to extract any extended data that needs to be returned to the consumer, such as the context.

6.8.2. Resize Completion Queue

As described in Section 6.8, the size of the completion queue depends on the number of Queues and the depth for which the Queues are configured. A client may choose a default size for the CQ at time of its creation. When additional queues need to be associated with this CQ, the consumer must determine if the current size of the CQ is sufficient. If the current size is smaller than what is required, then this verb is called to resize the CQ. The resize operation can also be used to shrink the size of the CQ. The verb implementation could choose if it should really shrink or keep the current size, it's a policy left to the implementer of the verb.

HCA's are expected to support the resize CQ without any loss of completion records. Typical interactions would be

1. Create a new CQ area with new requested size
2. Driver copies the old data remaining in the old CQ to the new CQ
3. Destroy the old CQ

If the resize operations fails, for e.g. existing items are more than the new size of the CQ being created, then the operation must fail, and the old CQ is required to be left intact.

For user mode support, the interactions are similar to what is explained in Section 6.8.1. Except that the data-copy may be performed in user mode, once the kernel mode driver has performed the physical switch of the CQ.

6.8.3. Destroy Completion Queue

Completion queues cannot be destroyed if they have any queues still bound to the CQ. If the CQ's reference count is not zero, then a busy status is returned to consumer to properly cleanup references before destroying the CQ. When the reference count is zero, this would mean any QP related context managed as part of CQ is moved to the pending lists. All such context structures can now be removed and memory reclaimed.

In order to ensure that any pending events be delivered before the CQ is recycled and reallocated to another application, HCA driver must ensure that stale events are not delivered to the new owner of the CQ. Mechanism to ensure this is vendor specific.

6.9. Multicast

Multicast operations require verbs to add and remove QP's from multicast group. The capability for multicast support must be advertised via the ca attributes structure to indicate to the Access Layer if such services are permitted on this HCA.

The exact management of these operations is vendor specific, and hence not discussed in any detail here in this document.

7. Memory Management

InfiniBand Architecture provides sophisticated high performance operations like remote DMA and direct user mode IO. In order to achieve these goals of performance, robustness and simplicity, the architecture defines appropriate memory management mechanisms.

Memory management provides a mechanism to allow the consumer to describe a set of virtually contiguous memory locations or a set of physically contiguous memory locations to the HCA. The HCA uses these descriptions to perform DMA operations to and from host memory without intervening the host CPU. For the purpose of this discussion we will assume that the HCA has two different memory registration resources.

- Region Entry – which holds the parameters, required describing the registration, such as virtual address, length, access rights, the R_KEY and L_KEY values.
- Translation Entries – Which represent the different physical pages that describe the translation.

For simplicity we will assume that the two regions can be completely different and managed as separate resources.

7.1.1. Memory Management Verbs

Memory Registration is a process that describes virtual memory, or a set of physical pages to the HCA. Memory registration verbs produce 2 distinct keys as described by the InfiniBand Architecture.

L_KEY	A 32bit opaque quantity that specifies local access rights for a memory region
R_KEY	A 32bit opaque quantity that specifies remote access rights for a memory region

The HCA uses these keys in combination with the virtual address to determine the exact physical pages that needs to be considered in a data transfer operation.

7.1.1.1. Register Memory Region

Register memory region verb prepares a virtually addressed memory region for use by the HCA. The primary input parameters are

- HCA handle
- Protection domain
- Virtual address of memory region being registered
- Length of bytes being registered
- Access Control rights for local and remote access

On successful completion this verb returns the following to the verb consumer.

- Memory region handle
- L_KEY
- R_KEY (Optional)

Some of the data the needs to be tracked as part of memory handles is listed below

- PD Handle – Required to keep reference counts for Protection Domain objects
- Virtual Address & Length – Represent the original virtual address that was provided by the verb consumer
- Region entry – Points the region entry block that represents this registration
- TE Ref counts – This is used when performing shared memory register that shares the same translation indices.
- List of locked pages – This member represents the pages represented by this translation.

The steps involved to perform a memory registration are

- Validate parameters if local & remote access rights provided are consistent
- Increment reference counts to the protection domain being provided
- Obtain and populate VPTT resources necessary to register this memory region.

An example of the flow of events is shown below in Figure 7-1. In the chart, for physical mode registrations, the verb is required to generate a pseudo virtual address. It is important to note that the applications cannot use this as a valid address in the process address space. The generated virtual address is only for the purpose of the HCA to identify the translation entries so that it can locate the physical pages where the IO must be performed.

7.1.1.2. Register Physical Memory Region

Register Physical Memory region, is logically similar to Register Memory Region with the following differences.

- No need to pin memory down, since it is expected that memory is already pinned.
- Consumer passes a ***list of memory pages***, ***page size*** for the buffers, the ***offset*** in the first page for the region being registered, and the ***length*** of the region being registered.

The process of registering memory is same as presented in Figure 7-1, except that the memory it is not required to pin pages down. The consumer passes a requested virtual address, if this address is not acceptable to the HCA driver; it is permitted to pass a new virtual address back to the consumer. The major differences between the physical memory registration and register virtual memory is shown as a "gray" box in Figure 7-1.

7.1.1.3. Re-Register Memory Region

Re-Register memory region is logically equivalent to a memory de-registration followed by a registration. HCA drivers may choose to optimize re-use of resources to the extent possible. For example, if the number of pages required is less than or equal to the previous VPTT resources held by the previous registration, then the VPTT resources can be reused in this registration.

HCA driver must de-reference the count on the old PD if its different from the new PD for which the memory region is being registered.

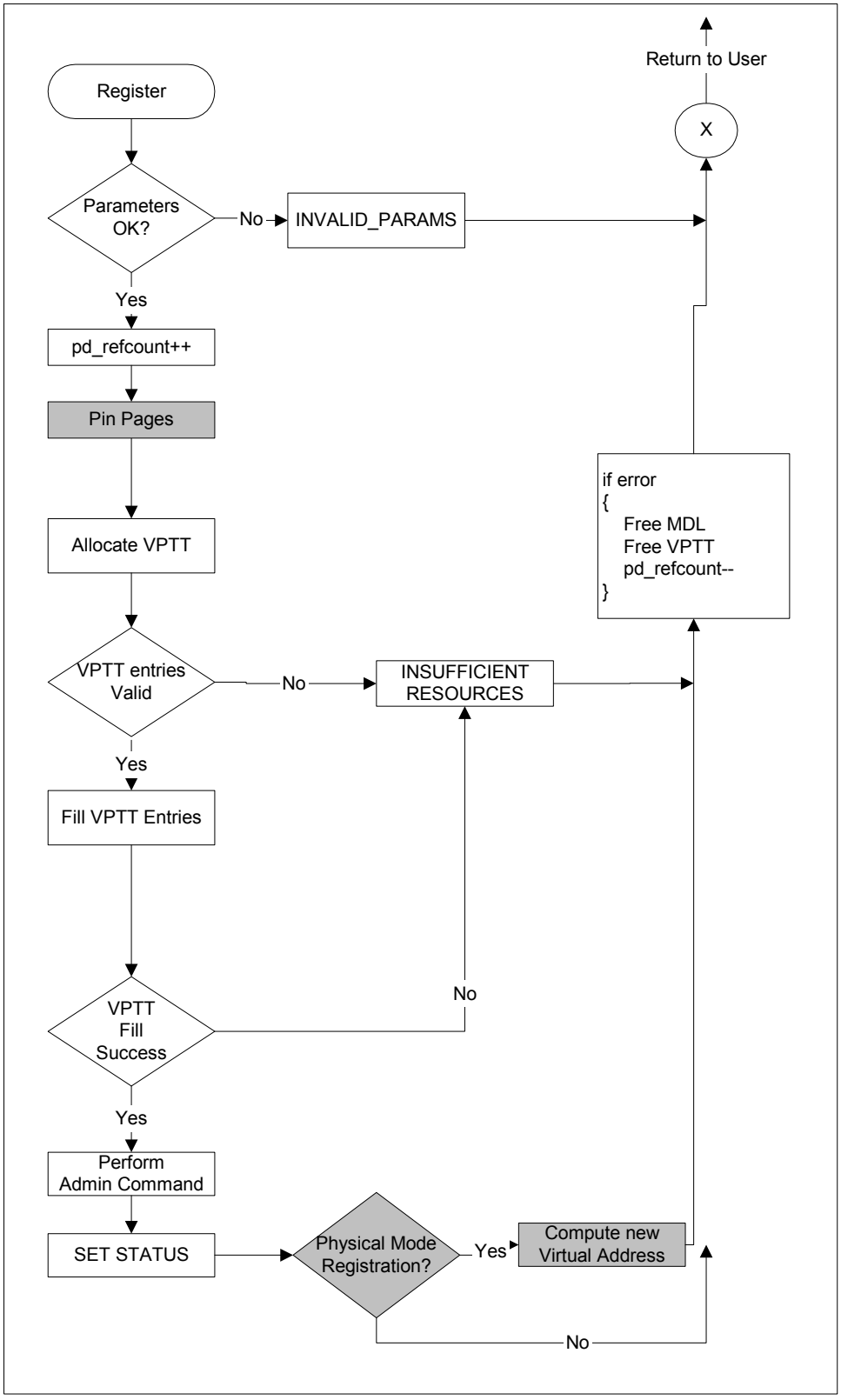

Figure 7-1 Register Memory Region

NOTE

InfiniBand Specifications requires that the memory handle produced by either register virtual or register physical memory can use the same handle for a re-register memory region, or register shared memory region call subsequently. Hence to appropriately un-pin pages it is required that the HCA driver performs some tracking on how the handle was constructed, so the de-registration process can deal with the correct treatment of these handles and perform the necessary unpinning of user mode buffers submitted during the register call.

7.1.1.4. Query Memory Region

Query memory region verb returns the properties of the registered memory region. The attributes that this verb returns are listed below.

- Memory protection bounds, obtained from the region handle maintained by the driver.
- PD handle obtained from the region handle maintained by the driver
- L_KEY and R_KEY of the registered region.

The methods to obtain these could be simply stored in the handle themselves, or vendor could have a unique way to retrieve them.

7.1.1.5. Register Shared Memory Region

Register shared memory verb provides an existing region handle, and requests another registration be performed. The protection domain and the access rights can be different from the reference handle provided as input. Depending on the implementation of VPTT by a specific vendor it may or may not allow sharing the registration resources. Since VPTT areas could be quite large depending on the size of the region being registered, any sharing to take advantage would be a good utilization of resources.

The vendor implementation must also guarantee that the VPTT resources will maintain proper reference counts so that if one region is being de-registered, the VPTT resources are not freed. This would cause severe malfunction and cause system memory corruption.

In the example flow for the shared registration process, we assume that the VPTT resources can be shared.

In the figure below, `master_mr` refers to the original memory region. The `slave_mr` refer to the newly registered memory region that is sharing the translation resources with the master memory region.

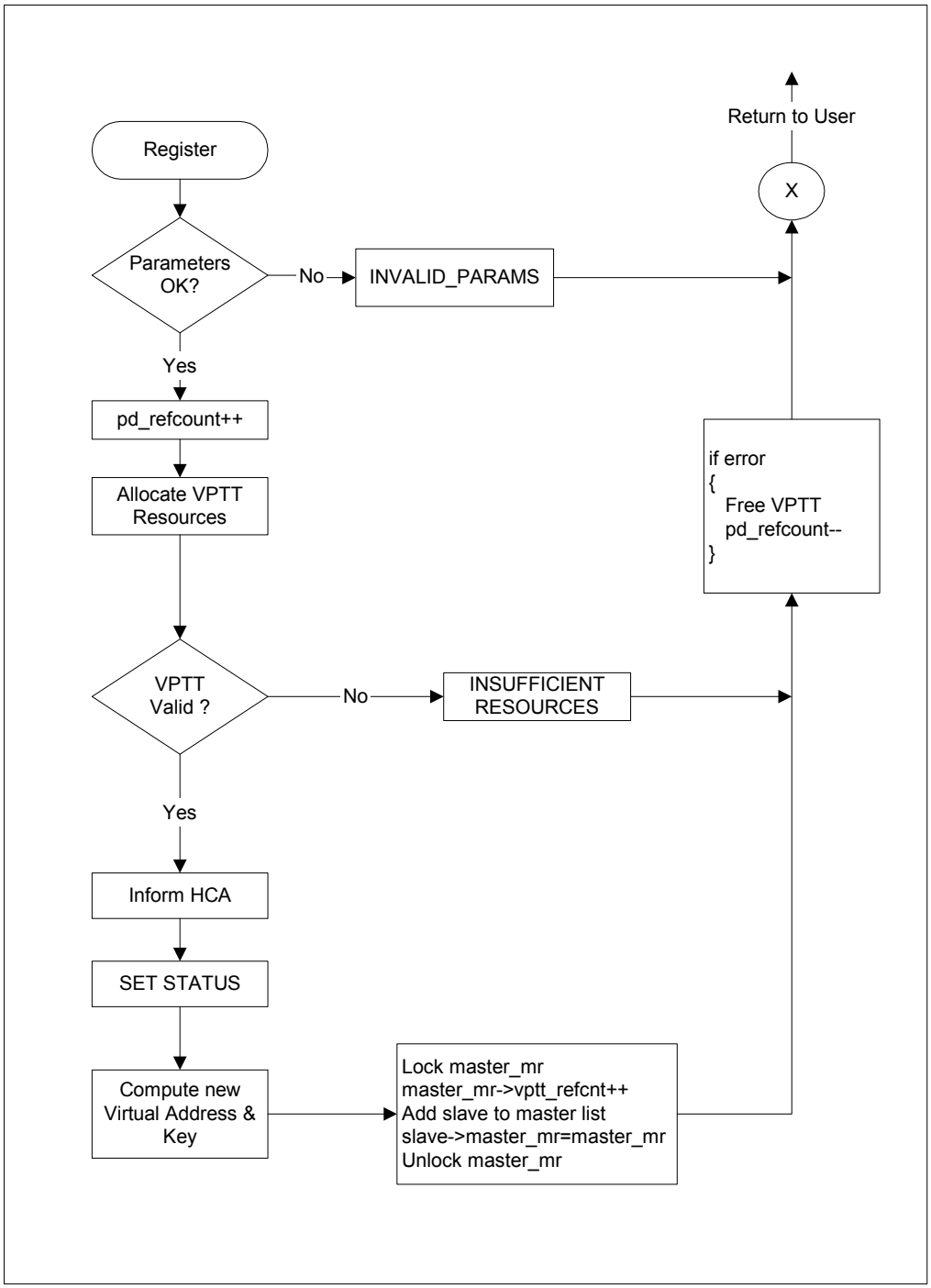

Figure 7-2 Register Shared Region

7.1.1.6. Destroy Memory Region

When a consumer has no more need for a registered memory region, this verb can be used to release the translation resources back to the Channel Interface. Some of the primary responsibilities for this verb are listed below:

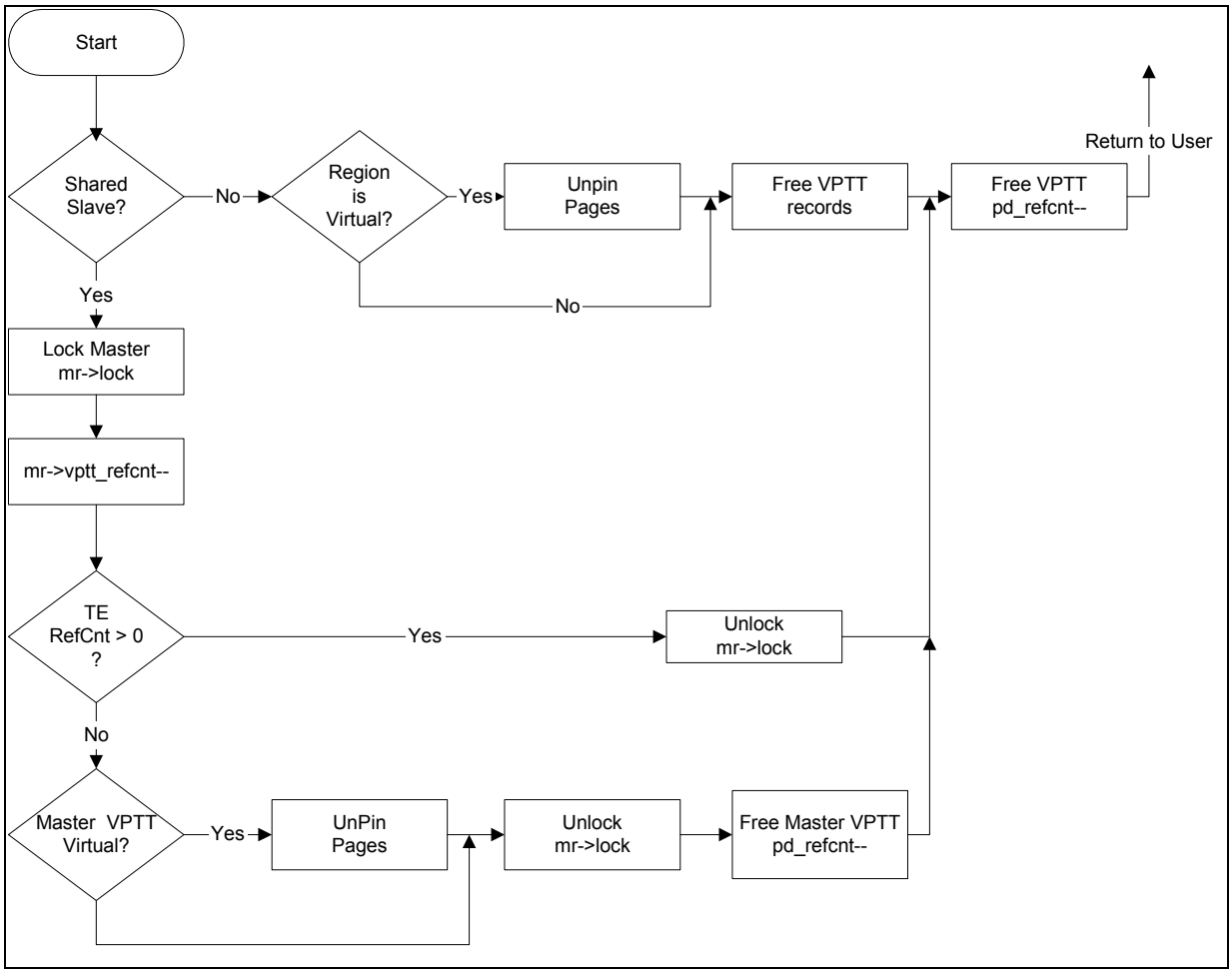

Figure 7-3 Destroy Memory Region

In the above example flow, the key points to note are.

- Maintaining a reference count on the shared VPTT resources
- If the registered memory region has no more shared registrations, then the memory region is unpinned.
- The current RE is always freed back to free pool.
- The master is not freed until there are no more regions sharing the VPTT record space.

Please keep in mind the above are only sample choices, each vendor situation is different. Please refer to the programmer's reference manual of the HCA for more information.

7.2. Memory Windows

InfiniBand architecture defines memory windows to provide a more efficient way to grant access rights to remote end nodes in a more dynamic fashion than using proper memory registration. Memory Windows allow the verbs consumer in the following situations.

- Need to grant and revoke access rights to a remote end node.
- Need for different access rights to different remote agents, for different regions of memory previously registered by the client.

In order to utilize memory windows, a verb consumer must register a region of memory, and then open different regions for access using memory windows. A typical sequence would be as follows.

1. Client registers a memory region.
2. Allocates a memory window
3. Posts a Memory Window Bind.
4. Posts a Send to exchange R_KEY information with remote agent.
5. After indication that the intended remote operation completed, the verb consumer unbinds the window by posting with a length of zero.

After unbind the verb consumer can utilize this window entry to point to a different memory region and perform similar use of bind and unbind operations. Memory Window bind operation is useful for the following reasons.

- Very light weight compared to normal memory registration that requires pinning down memory.
- Operations can be performed from user mode without taking a kernel transition compared to memory register operations that require kernel transitions to enforce validation and performing the operation of locking down user memory pages.
- Protects the R_KEY from being utilized for stale requests. Each time a new bind is issued the R_KEY must be different from the old one. This ensures that stale requests will not succeed, since the R_KEY will be different between each bind operation.

 **NOTE**

HCA drivers must ensure that the PD reference counts are maintained properly when utilizing memory window verbs. User mode code must maintain its own copy of PD reference counts other than what is maintained in kernel, since the post window bind operations

8. Work Request Processing

Work Request processing encompasses the operation of submitting work requests to the HCA for processing. HCA driver converts the Work Request to internal WQE formats, which is vendor specific. HCA driver converts the Work Request, which is a generic work description to an internal format specific to the HCA.

- Send WQE – To allow Send Command.
- RDMA Read – For posting RDMA Read commands to the Send Queue
- RDMA Write – For posting RDMA Write commands to the Send Queue
- Atomic – Posting Compare Exchange and Fetch add IB commands on RC/RD QP's Send Queue
- Memory Window Bind – Posting a memory window bind command to the Send Queue of a RC/RD queue pair.
- Receive – This is a generic WQE for posting Receive commands to the Receive Queue of any type of QP.

Once the work is formatted in a WQE the user vendor specific code informs the HCA to indicate that work is available via writing to the doorbell space. This operation may be permitted from user mode, if the doorbells are mapped to user mode.

If the verb has knowledge on available WQE's, then it can format more than one request and inform the HCA that there are n# of WQE's available. In order to permit this facility, the Linux implementation of the *ci_post_send ()* verb accepts extra parameters. Please consult the file *ib_ci.h* for more information.

- h_qp refers to the Handle to the QP for which the work request is submitted.
- A Pointer to head of the list of work requests.
- A pointer to pointer of failed work requests, which could not be submitted.

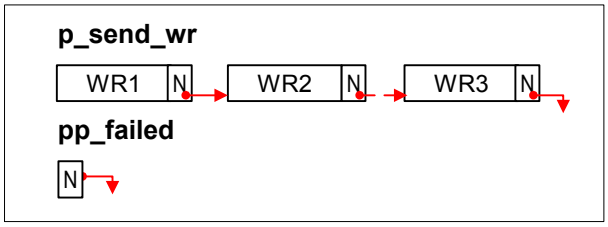

Figure 8-1 Initial Post Parameters

The list must be a null terminated list. If any of the requests was not submitted, then the driver will terminate the list with only successful requests submitted. The failed list will point to the first request that failed the post operation.

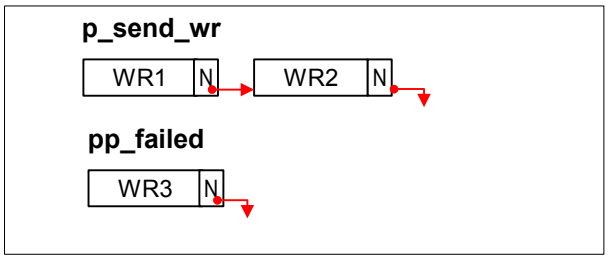

Figure 8-2 Post Parameters on Completion

In the above example, two work requests were posted and the hence the list was terminated at WR2. pp_failed parameter is set to the first WR that was not successfully posted.

8.1.1. Posting Work Requests

Posting Work Requests to the QP involves the following steps.

1. Verify if current QP state is appropriate for the post operation. For e.g. if QP is not in Init, RTR or Reset states when a WR is posted to the Send Queue.
2. Verify any other parameter checks that require returning an immediate error on post operations.
3. Ensure that there is a WQE available for posting this work request.
   - HCA driver must ensure that the WQE's won't be overwritten. A client can expect to keep posting, and expect to receive IB_INSUFFICIENT_RESOURCES as error value. This error must not change the state of the QP, or other work requests already posted.
   - It is also important that the design does not require locking between the poll and post operations, since these are speed path critical for performance and the driver should attempt to **not** single thread both the activities.
4. Prepare the WQE based on the operation type and the QP type as required by the HCA. The vendor specific code may require to do additional checks, or housekeeping say to retrieve the Work Request ID (WRID) etc in a vendor specific way.
5. Indicate to the HCA that work is now available via writing to the doorbell space.

NOTE

HCA drivers storing the WRID for later retrieval should keep fast access methods to that there is no expensive search when the *ci_poll_cq ()* call is made. Optimizing for performance is critical in this area.

8.2. Completion Processing

Completion queue processing encompasses operations permitted on a CQ. This would include polling a CQ for completion entries and enabling a CQ for notification when the next CQE is added to this CQ.

8.2.1. Polling Completion Queue

PollCQ retrieves work completion from the specified CQ. Work Completion indicates that a WR submitted to a QP associated with this CQ has completed. The CQE contains important information necessary to the consumer to identify the work request submitted, the status of the operation etc. The important fields reported by a CQE are: -

- 64bit WRID submitted with the WR when posted to the QP
- Type of the operation that was completed.
- Length of data sent/received
- Source QP/EE for datagram QPs.
- Path Attributes necessary to communicate with the remote end for UD QP's.
- Free count indicating the number of WQE's freed by this completion queue entry for RD QP's. This is required since the completions may be reported out of order by the HCA. This information can be used in conjunction with the Post WR verbs to ensure that WQE rings do not overflow.

The poll verb in the Linux implementation accepts two lists.

- One to provide a list of free WC's to the poll_cq verb.
- A second one to provide a list of successfully polled CQE's.

The list of CQE's successfully retrieved is a null terminated list. Hence the chain that was submitted to the *ci_poll_cq ()* verb will break the list and give it back to the consumer with a list that is free, and a list that has completion processing done. The parameters when the call to *ci_poll_cq ()* is made are shown below in Figure 8-3.

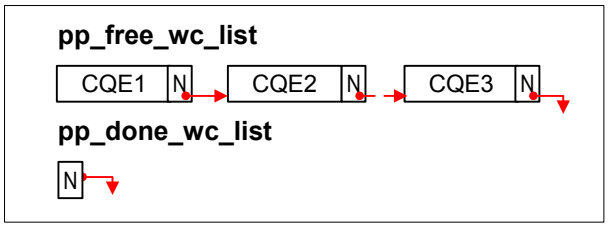

Figure 8-3 Initial PollCQ Parameters

In this example we assume that two entries are retrieved via the poll call. The state of the parameters is shown below in Figure 8-4.

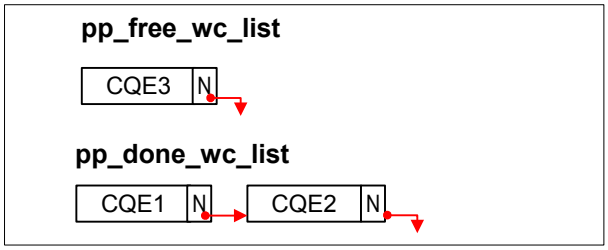

Figure 8-4 PollCQ Parameters on Completion

The list interface was provided for possible optimizations possible with hardware interactions. Most receive side processing will have many receive descriptors and would avoid the user making repeated calls. A second benefit is that in order to prevent CQ overflow, the HCA driver may require notifying hardware on the number of completions retrieved. This facility essentially reduces the number of times the HCA needs to be notified, and thereby reducing the number of device IO transactions on the bus.

8.2.2. Requesting for Completion Notification

Requesting completion notification requests the verbs provider to call its completion handler when the next completion entry of the specified type gets added to the specified CQ. The notification must not be generated for entries already existing in the CQ. InfiniBand Specifications specify two types of notification requests.

- Notify only on the next solicited completion event only
- Notify on the next solicited or unsolicited completion.

HCA hardware will maintain a CQ state to manage this notification request. The driver software and HCA hardware compliment each other in managing these notifications. This operation can be invoked from the user mode. Most HCA vendors may provide a doorbell space to inform the HCA about the type of notification request being made.

8.2.3. Important CQ related Notes to Driver Writer

HCA hardware and software must work together to achieve the required behavior. Lists of potential issues are listed below: -

1. HCA driver and hardware must co-manage and ensure that CQ overflow is prevented. CQ overflow is not recoverable and the first QP that caused this error would enter error state. The expectation is that existing entries are available to consumers without loss of CQ entries.
2. A QP may be reset, or destroyed. Such actions must not intervene with the ability to report completions on behalf of the CQ. Vendors using alternate store to track work request ID's must take special care that old completions not yet retrieved will still be reported correctly with the exact WRID's submitted when the WR was posted.

3. Driver writers must strive to have no locks between the post and poll side of the verbs. This is purely a performance concern that may otherwise compromise work request processing by serializing the completion processing and posting of work requests.
4. When CQ's are destroyed, HCA driver must ensure that any undelivered events to the CQ owner will not be delivered to a new CQ owner.
5. When QP's are destroyed, the HCA driver must ensure that the QP's are not made available until the time wait period has expired.
6. When a CQ is being notified to report an event for new entry added to the CQ, the driver software and hardware would require to ensure that a notification for an existing entry in the CQ will not cause an event to be generated.
7. Race free notification request is critical, i.e when the client issues a notification request, the HCA may be in process of generating the CQ entry. Due to hardware latencies or Chipset behaviors the entry may not have yet made its way to the system memory. HCA driver and the hardware must have some mechanism to guarantee that events or CQ entries will not be lost in flight.

8.3. Avoiding Race between Polling and CQ Arming

InfiniBand specifications require the verb consumer to poll all entries to completion. There is always a race between the polling and rearm action. This is due to the fact that the hardware keeps placing entries in the CQ, and the poll is an asynchronous operation. In addition the InfiniBand specifications require not generating interrupts for entries already existing in the CQ. In order to avoid this race and ensure that either the completions are pulled out during a poll operation, or be guaranteed of an interrupt generation, the following sequence of arming and check is recommended.

```
cq_notify_callback()
{
     boolean_t rearm_needed = TRUE;
     ib_wc_t   *p_empty, *p_filled = NULL;

recheck:

     p_empty = get_free_elements ();
     poll_status = ib_poll_cq( h_cq, &p_empty, &p_filled )

     if ( ( rearm_needed == FALSE ) && ( filled == NULL ) )
     {
         return;
     }
     /* Put free elements back */
     if ( p_empty )
        return_to_free ( p_empty );
     /* Got new entries, process them */
     process_elements( p_filled );
     ib_rearm_cq ( h_cq );
     rearm_needed = FALSE;
     goto recheck;
}
```

Figure 8-5 Poll CQ / Rearm Algorithm

9. Interrupt and Event Processing

Interrupt processing is the primary notification from the hardware that a serviceable condition has occurred. Most of what happens here is completely vendor specific, hence we will provide some general information that might be applicable to most HCA vendors.

InfiniBand HCA's have numerous resources, such as QP's, CQ's with several thousands of them in certain configurations. Each resource can generate an event, which is an interrupt condition. Interrupt generation and servicing the interrupt at this rate can cause severe performance degradation. The event queues provide a nice alternate to traditional interrupt scheme for these resources.

Interrupts are mostly one shot in the HCA. When a first entry is added to the EQ and if the EQ were requested to generate an interrupt, then the HCA would generate a physical interrupt. Unlike traditional IO devices, the HCA's do not stop functioning once the interrupt is delivered. It still keeps functioning and uses the event queue for notifying future completions and error conditions.

Interrupt routines typically schedule a tasklet in Linux for deferred processing. In the deferred processing mode, each event queue entry is processed, and the appropriate owner is notified of the event. The tasklet typically processes all events submitted before requesting for an interrupt generation the next time. Functionally they work similar to how the CQ's function, i.e. CQ entries are generated and processed by client until there are no more completions. Then the client requests the HCA to inform when a new CQ entry is added to the CQ.

NOTE

The entire InfiniBand stack assumes that no callbacks are generated from interrupt context. The purpose is that the stack uses the light weight spin-lock method, and does not stop interrupts from happening. The lock ordering is also very important, spinlock_bh () followed by spin_lock_irq () works fine. If the stack used spinlock_irq () followed by spin_lock_bh (), this does not guarantee that interrupts are disabled, hence may end up with critical sections not locked.

Verbs Provider driver is the only driver that would require spin_lock_irq () to protect its resources, Hence all Verbs provider drivers must issue all completion and event callbacks from a tasklet context for the lock ordering to work correctly.

10. User Mode Support via Plugin

Throughout the entire document, we have shown some recommended procedures to handle native user mode support. These are guidelines, and individual vendors may choose to have other alternative approaches, within the framework provided by the Access Layer. Please refer to the header file ib_uvp.h to understand the different plugin API's and how to work with AL to provide native user mode support as a HCA vendor.

The following table highlights how to handle different error cases and how success or failure is determined.

Table 10-1 Error Case Handling with user mode Plug-in

No.	*Pre-ioctl*	*Kernel Ioctl*	*Post-ioctl*	*Comments*
1.	SUCCESS	SUCCESS	SUCCESS	Return successful return and pass handle back to user mode consumer.
2.	SUCCESS	SUCCESS	FAILURE	If any resource was allocated by the ioctl, such as say create_qp (), then the QP is released. Failure is returned to the consumer.
3.	SUCCESS	FAILURE	**FAILURE**	If the kernel ioctl was a failure, the post routine must return failure. The access layer will assert if the plugin returned SUCCESS.
4.	FAILURE	NOT DONE	NOT DONE	If the pre-ioctl call fails, then the ioctl operation is itself not performed.

The user mode plugin must rely on the status in the UMV_BUFFER that is kept for communication between the user mode and kernel mode components. If the UMV_BUFFER is carrying a **bad** status, it is expected that the kernel mode verb also returned bad status, and fails the resource creation call.

The post-ioctl plugin call is invoked always **if** the ioctl to kernel is performed. This is designed so that the user mode plugin gets a chance to cleanup any resource allocated due to an invocation of the pre-ioctl plugin.

When the plugin returns failure to a post-ioctl plugin, if the effect of the ioctl is a resource creation, then the cleanup will be performed, before returning bad status to the verb consumer. For calls such as modify operations, the post-plugin is mostly a informational notification. Hence no ioctl will be performed. For e.g. if the call was a modify QP to RTS, this cannot be reversed back to old state. On such API's the access layer code may perform asserts to ensure that correct status is returned by the verb plugin's.