Software Architecture Specification (SAS)

Revision – Draft 1
Last Print Date: 4/30/2002 - 9:03 AM

InfiniBand Linux Software
HCA Driver Development Kit

Abstract

InfiniBand™ architecture defines the interface between the OS and the HCA as the Channel Interface (CI). The specification is covered in Chapters 10 & 11 of the InfiniBand™ architecture specification Volume 1. Verb definitions in the specification are only a guiding factor for the API development effort. It provides a structure and semantics of the kinds of functions and capabilities that need to be exposed by a HCA vendor.

This specification lays out the foundation and building concepts of the HCA drivers, services it requires from the InfiniBand™ Access Layer (AL) to provide an efficient transport. There are a few concepts that are specified more clearly to address implementation issues that the architecture specification does not address. This specification will also provide different interface guidelines for vendors to develop their driver and support. It will provide all necessary information to support direct user mode access to their hardware and bypass kernel traps.

This document should not be viewed as a tutorial to InfiniBand™. The reader is expected to be aware of terms and acronyms that are InfiniBand™ related. The reader is expected to be familiar with Chapters 10 & 11 of the InfiniBand™ architecture specification, Volume 1.

Table of Contents

Table of Figures

1. Introduction

The InfiniBand™ software architecture high-level block diagram is shown below. The software architecture is specifically designed to separate services into common services generic to all HCA vendors and vendor specific functionality to provide their own value add for performance features necessary to differentiate their product. The design of the API's and the software mapping was done very carefully to not just translate the verbs definition to API's but also to provide a useful and practical solution that ensures design for scalability and robustness that will drive InfiniBand™ infrastructure and Linux to enterprise computing levels.

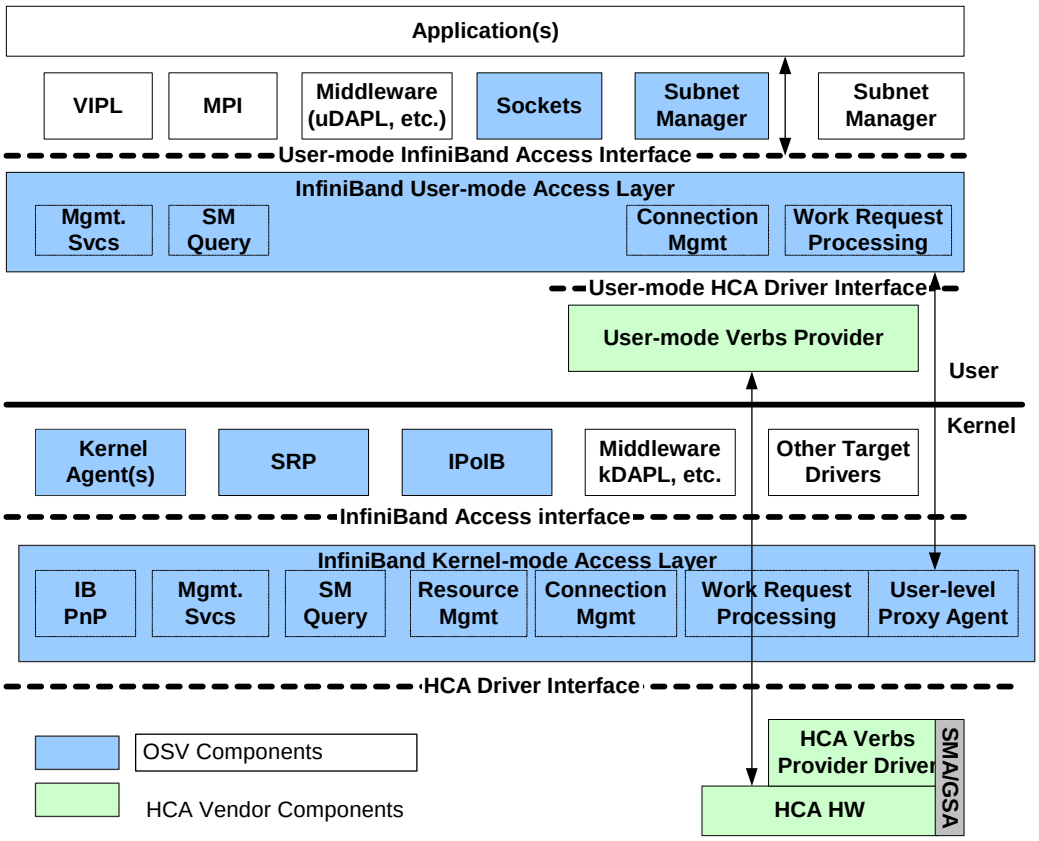

Figure 1-1 Architectural Component Block Diagram

2. Architectural Specification – HCA Specific Driver

2.1 Introduction

This chapter defines the architecture for the HCA specific, or "verbs provider", driver software. This chapter also contains detailed pictorial representations of the driver software and how it interfaces with other components. The chapter also describes how the driver software interfaces with the hardware, the operating system, and other components of the InfiniBand Linux software stack.

2.2 Design Notes

This section specifically addresses some of the design choices made in this architecture, how the API design was derived and how these decisions resulted in the final API's. This section would provide the reader specifically with deviations from the verb specification and the reason for those choices.

Table 2–1 Table of Deviations from InifiniBand Specifications

No	*Description*	*Comment*
1.	Multiple handles passed to Verb functions	Several verb specifications accept multiple handles, for e.g. create_qp take both a ca_handle_t and a pd_handle_t. Since the PD is already associated with a CA when the PD was created, the verb API's don't require both a ca_handle_t and a pd_handle_t. In all such cases just the pd_handle_t is specified in the verb API's. The verb consumer could maintain the association of the pd_handle_t to the ca_handle_t for internal tracking. This removes additional checks that would otherwise need to be performed by each verb provider functions, thereby eliminating redundant information passing and checking.
2.	Specifying a PD with all Verb API's. (create_cq and create_pd)	Verbs do not specify a PD for creating a completion queue and reliable datagram domain objects. The architecture did not specify a PD association with the CQ because the CQ could be associated with multiple QP's, where each QP could belong to multiple PD's. Operating systems could enforce this process ownership in different ways, but the PD is a better way to make that association between IB resources. If a process intends to allocate a CQ, it is expected to have a valid protection domain created before the CQ is created. If the application requires managing multiple PD's and QP's belonging to multiple PD's, it can still associate both QP's to the same CQ as long as the PD is valid for that process. The concept is extended for reliable datagram domains for consistency of all resource allocation API's.
3.	Passing handles for callback notification	Handles such as cq_handle etc are opaque objects private to each HCA specific driver. Verb requires the callback notification function to pass the handle in the notification. This handle is not of much use to the client receiving the callback. For e.g. it cannot identify its internal context associated with this resource. Moreover this would require the API consumer to take a handle, and search some data structure to identify its internal context before it can perform any useful processing. For e.g. Interrupt registration functions (say request_irq () in Linux™)

No	*Description*	*Comment*
		pass a context back provided during the registration. Most verb API's that require callbacks are modeled in the same way. For e.g. resource create calls such as create_cq etc take a context as parameter. In case of a completion callback or error notifications, the context provided during the create call is passed back to the application rather than the handle that caused the event. Applications are expected to obtain the handle from their context for further processing.
4.	SMA below the Channel Interface	Verb requires each vendor to perform all processing of Subnet Management Packets below in the Channel Interface. Some vendors provide this behavior by having processing capability in the adapter hardware. Some vendors may choose to manage this via software. The Subnet Management Agent (SMA) functionality is very generic and is not different from vendor to vendor, since the requests and responses are specified by the architecture specification. To avoid duplicating this functionality in the software by emulating QP access in software and to manage this efficiently the SMA role is divided to Vendor neutral and Vendor specific pieces. The response management is performed above in the InfiniBand Access Layer, and any related hardware management function is performed in the HCA driver software. Say setting a port state is a hardware function, which is performed by a generic ci_local_mad () function supplied by the vendor, rest of the state management is managed in a vendor neutral way. Such capability is exposed as a port capability in the port attribute structure.
5.	LID events and Port events.	There are other systems related drivers that are require to be notified by the HCA drivers on certain events, say for e.g. When the port is programmed with a new LID. These could be systems management drivers. When the SMA is managed in the Access Layer, this event can be easily passed to the consumers to notify of a change in the fabric related to local HCA's. In case where the SMA is managed in hardware, it might be required to proxy the packets even after consumption so that systems management drivers can make use of this event in a useful way. This is not required functionality by the HCA vendors, but this behavior would give a event driven method to pass those events which is a better programming model than the polling operation for driver software. A subnet manager would need to know when a port goes from down->init which could also be beneficial if the HCA vendors generate notifications for these events.
6.	Passing more than one work request for posting and completions	Post and Poll verbs accept more then one request. This model is useful for applications posting receive work request, when they usually pre-post all the receive requests. This allows the consumer to pass all once, instead of posting one at a time. For user mode functions when the HCA vendor does not provide OS bypass drivers in user-mode, these functions would be performed in the kernel via a trap to the kernel, either via a *syscall* or an *ioctl* call. This mode of passing multiple requests facilitates one trap to read and perform all the operations in a single *ioctl*, instead of making a single call to kernel for each request. Grouping allows an efficient implementation even if the vendor does not provide OS bypass mechanisms.

No	*Description*	*Comment*
7.	Device Name	In Unix most device access happens via names, such as /dev/hca1 etc. Although this is traditional, such naming has always caused more trouble, especially this does not permit binding or identifying a device uniquely. Since InfiniBand™ identifies each HCA with a GUID, in the interest of portability; verbs such as *open_ca ()* take the GUID instead of a name.
8.	Asynchronous Resource Destroy	InfiniBand™ architecture specifies callbacks for both completion and error processing. Handling resource cleanup in a callback driven world must be handled properly. In order to enforce correct rules of destruction, the destroy_qp, destroy_cq API's cannot be called from the callback themselves. The access layer would maintain a different cleanup thread and call in a different thread context. This is necessary to avoid performing cleanup while the access layer could still be using the resource. Access layer API's would present an asynchronous destroy API's in order to not burden the consumers with the same restriction.

2.3 Theory of Operation – Kernel Mode Driver

This section provides details on the interfaces provided by the driver for use by the InfiniBand Access Layer and their capabilities. A later section will present a typical usage model of how the Access Layer uses the interfaces. The majority of the interfaces to the driver are derived from the Transport Verbs chapter of the InfiniBand™ architecture specification Vol1.0a. It is assumed that the driver will have verbs functions as specified in the header file ib_hca.h.

Driver initialization Interfaces

The HCA driver provides interfaces that allow the InfiniBand Access Layer open and initialize the driver, and determine the capabilities of the HCA.

Refer to the InfiniBand specification and the HAC driver API definitions for details on the specific capabilities provided by these interfaces.

In Linux, the HCA driver is a loadable kernel module that typically gets loaded during system start-up. The InfiniBand Access Driver is loaded prior to the HCA drivers. This is traditionally controlled via module dependencies and load order specified in the */etc/modules.conf* file.

If the HCA presents itself as a PCI device to the hardware, the driver registers with the PCI device driver. It provides the PCI device-id/vendor-id to the PCI driver. When the PCI driver finds an HCA that matches the PCI vendor-id/device-id, it calls the driver back providing pointers to the HCA resources. These are the typical interfaces used in Linux for PCI devices.

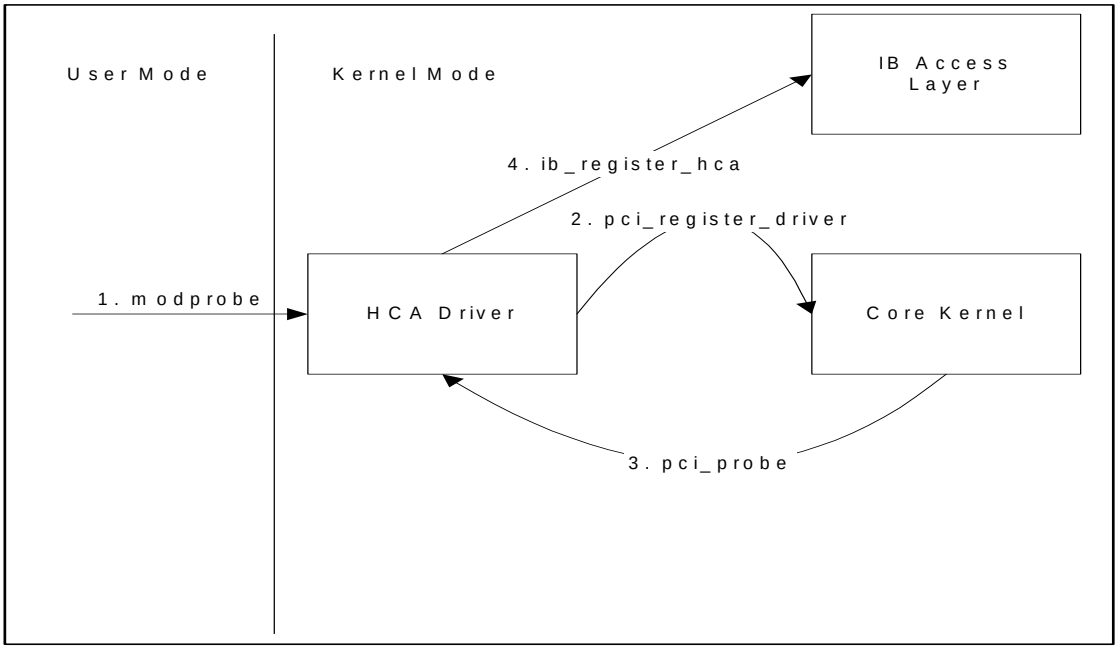

Figure 2-1 Loading the HCA driver

Driver close, unload, and cleanup

The HCA driver notifies the InfiniBand Access Layer when the driver is shutting down. This is triggered when a HCA hot-remove happens or when the HCA driver is being unloaded. The HCA driver calls the InfiniBand Access Layer to de-register the HCA leaving the system. The Access layer must remove all references to the specified HCA before the de-register API call completes.

Dynamically loaded drivers are protected from unloading by maintaining reference counts. The driver *cleanup_module ()* call must never fail. Hence it is important to maintain appropriate reference counts on each *open_hca ()* call from the InfiniBand Access Layer. A typical unload sequence is shown below.

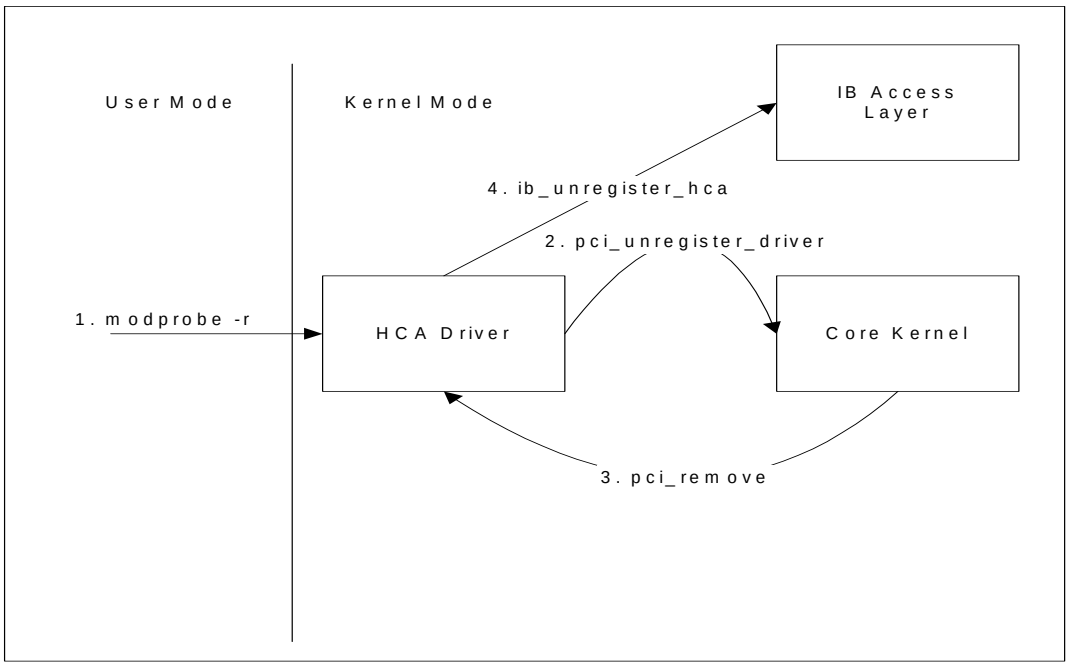

Figure 2-2 Unloading the HCA driver

Typical Driver Usage Model

This section describes a typical usage model of the driver interfaces by the InfiniBand Access Layer. It does not define every possible usage model, but outlines the way the driver interfaces could be used in a normal operation.

Initialization

The HCA driver is a loadable kernel module that typically gets loaded during system start-up. It is loaded sometime after the InfiniBand Access Layer.

If the HCA presents itself as a PCI device, the driver is instantiated by the Linux PCI driver. Upon start-up, the driver performs any required HCA initialization and sets up and initializes the driver data structures, e.g., things like the event queue, translation and protection table, etc.

Next, the driver registers with the InfiniBand Access Layer providing the GID of the HCA. The Access Layer provides an interface (*ib_register_ci)* to allow the driver to perform this registration. Refer to the InfiniBand Access Layer chapter for details.

The Access Layer can then open and query the driver using the GID to get the resource attributes of the HCA, such as the number of QPs, CQs, TPT, etc. The Access Layer can then also register a callback for asynchronous event notifications. This allows the Access Layer the ability to process unaffiliated events or events that are not associated with any given work request. E.g., port state events.

Queue Pair Setup in preparation for sending data

Before the InfiniBand™ Access Layer can send data across the fabric; it must allocate protection domains, set up queue pairs and completion queues, and establish connections with remote nodes.

A protection domain must be allocated using the *ci_allocate_pd ()* interface so that it can be associated with the queue pair when it is created. The Access Layer or upper layers of software

can allocate a separate protection domain for each of its clients or client processes to provide memory protection between clients and/or processes. If the connection that the access layer is intending to create is a reliable datagram, the access layer must allocate a reliable datagram protection domain.

After obtaining a PD, the Access Layer creates a completion queue to be associated with the work queue pair. This is done using the *ci_create_cq ()* Interface. Next the Access Layer allocates the queue pair using the *ci_create_qp ()* passing the CQ, PD, and other desired attributes of the QP. The connection manager component of the Access Layer can then use the Modify QP interface to modify the state of the queue pair during the connection establishment process.

Memory Registration in preparation for sending data

After the queue pairs have been allocated and set up for communication, the Access Layer must register all memory buffers that contain the user data buffers to/from which data will be transferred. To accomplish this, the Access Layer uses the memory registration interfaces of the driver.

The Register Memory Region interface is used to register virtual addresses and the Register Physical Region is used to register physical memory regions. The Access Layer can use the Register Shared Memory Region to register memory that is shared between protection domains. It can use the memory window routines to allocate a memory window and then later bind the memory window to a memory region using a work request, but no mater which registration mechanism is used; all memory buffers that are described in work requests must be registered.

Posting Work Requests

Once the Access Layer has allocated and initialized queue pairs and completion queues and registered the memory buffers associated with a data transfer, the Access Layer uses the Post Send and Post Receive interfaces to post work requests.

The diagram below illustrates the typical flow of posting a send or receive work request.

1. Channel Driver posts Work Request to Access Layer.
2. Access Layer Posts the Work Request to the appropriate driver.
3. HCA driver builds WQE, posts it to the H/W queue, and rings the doorbell.
4. HCA Driver returns to the Access Layer.
5. Access Layer returns to the Channel Driver.
6. HCA transfers data to/from the data buffer asynchronously while processing the WQE.

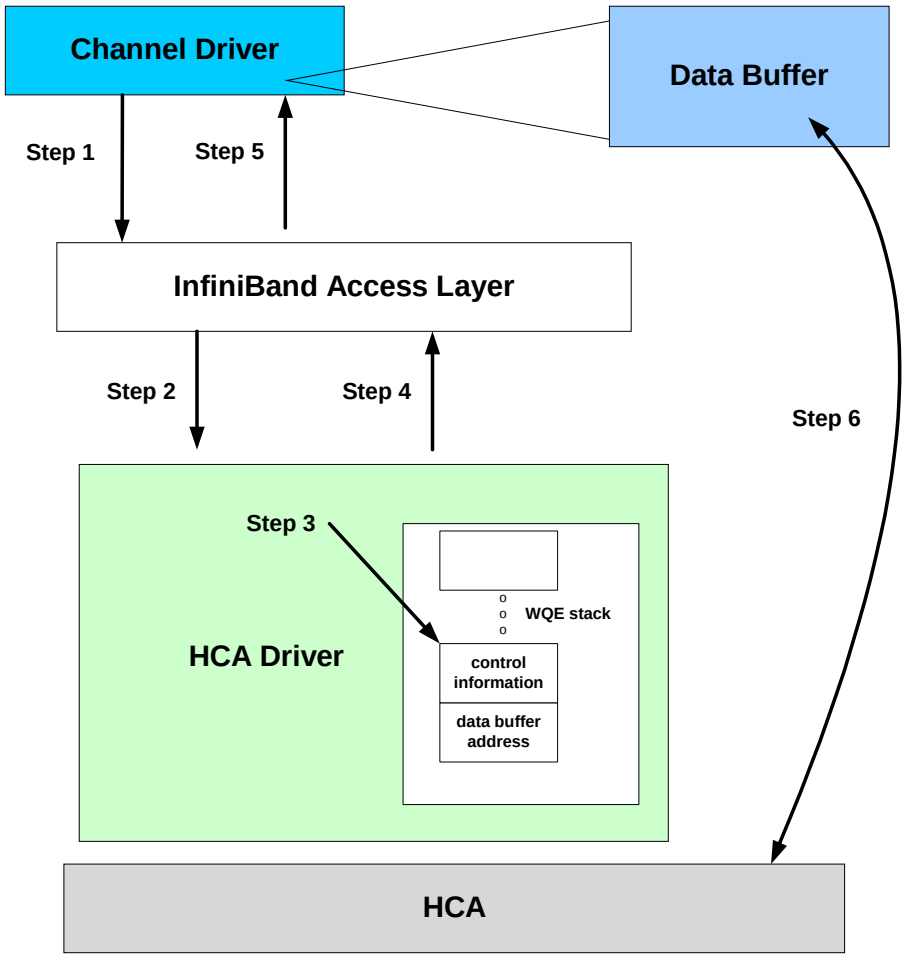

Figure 2-3 Posting Work Requests

Processing Completions

Once work requests have been posted to the Send or Receive queues and the H/W has completed processing the work requests, the Access Layer can process the associated completions.

The diagram below shows the typical flow of the completion processing.

1. HCA writes a completion queue entry to CQ entry queue managed by the HCA driver.
2. HCA may write an event queue entry to notify the HCA driver about a completion queue event.
3. HCA hardware delivers interrupt that the OS will eventually deliver to the HCA specific driver for processing.
4. Interrupt processing routine retrieves the event queue and processes the event generated by HCA in order.
5. HCA driver delivers the CQ notification to the Access Layer for further processing.
6. Access Layer performs a *ci_poll_cq ()* operation to retrieve the contents of the Completion queue entry.
7. HCA driver now copies the CQ entry to application buffer translating data in a generic format as specified by in the API specification.

8. HCA driver returns back to Access Layer.

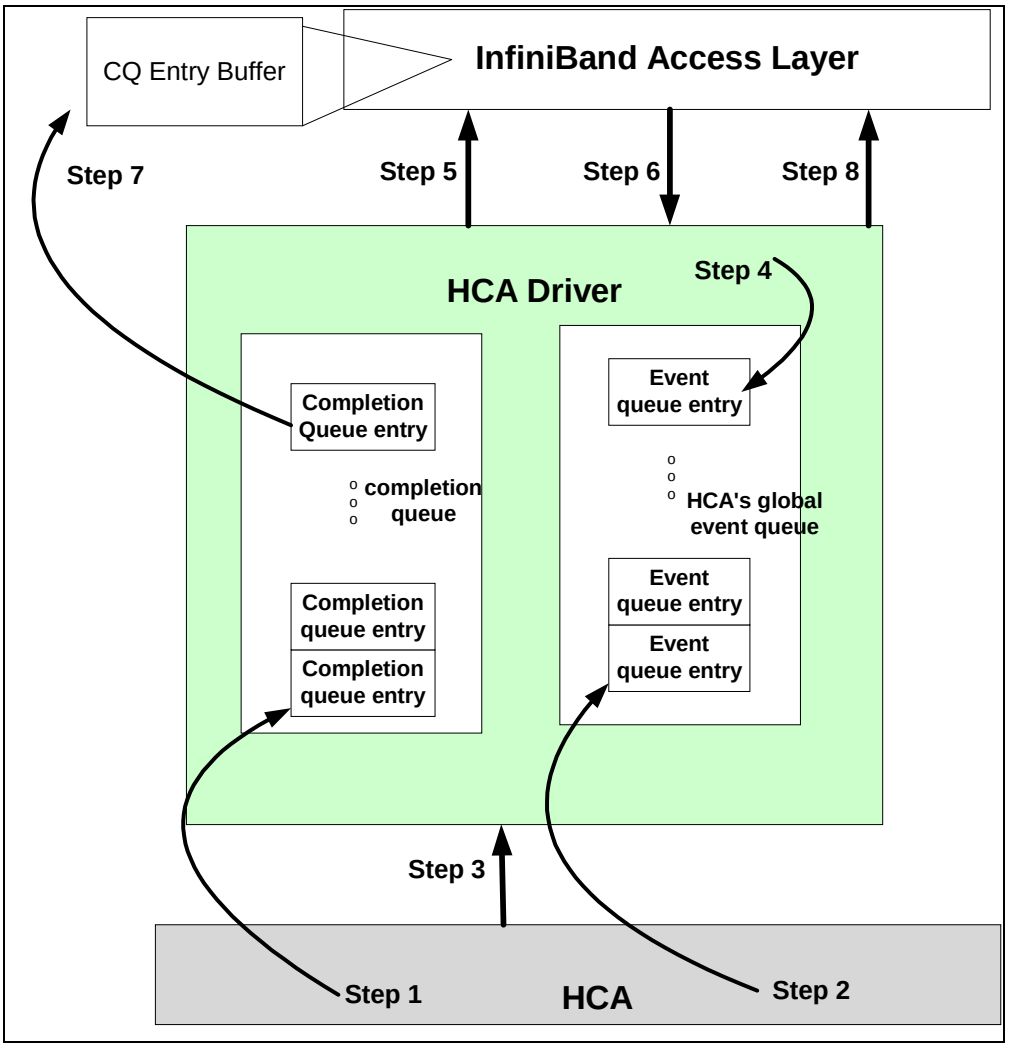

Figure 2-4 Completion Processing

Memory Deregistration

Once the work requests have completed, the Access Layer can use the Deregister memory interfaces to deregister the memory buffers. This frees up the TPT entries for use in subsequent memory registration operations. It is not uncommon for memory buffers to be registered and deregistered for each I/O operation, but it is possible for the Access Layer or upper layers of software to reuse the same I/O buffers and thus only need to register the memory once. This can lead to increased performance in some applications.

Destroying Queue Pairs and Cleaning up after Disconnections

Typically when a connection with a remote node is terminated, the Access Layer will release the resources associated with that connection for use in subsequent connections. The Access Layer uses the Destroy Queue Pair interface to free the queue pair for subsequent use. It uses the Destroy Completion Queue interface to release the CQ resources. It uses Deallocate Protection Domain and Deallocate Reliable Datagram Domain to free the PD resources that were associated with the QP.

2.4 User Mode Architecture

InfiniBand™ architecture permits direct user mode access to the hardware. The user mode architecture is designed to meet the following high level needs with a goal of providing robust solutions for InfiniBand™ software and hardware products.

- The HCA vendor would be required to provide a dynamically loadable library to aid direct user mode access to the hardware. If the vendor is not able to provide such a library, then the access layer would still permit access to the hardware for user mode applications via kernel support to the speed path data transfer operations (DTO). Direct user mode access bypassing kernel interface will be better performing than requiring kernel support to perform the same.
    - User-mode support is provided by a central dispatcher function in user and kernel mode via the InfiniBand™ access layer. HCA support library in user mode is not expected to communicate directly with its driver module. This facilitates the kernel mode Access Layer as the only agent authenticating resource creations and validation of handles passed from user mode to kernel mode.
- Provide ability to vendor specific processing for user mode support. For e.g. some vendors may allocate buffers for Completion Queue in user mode and program hardware appropriately, and some may choose to do the buffer allocation in kernel mode driver and map those to user mode after creation. In order to aid this behavior the interface functions will provide a pre processing trigger and a post-processing trigger for each verb support in user mode.

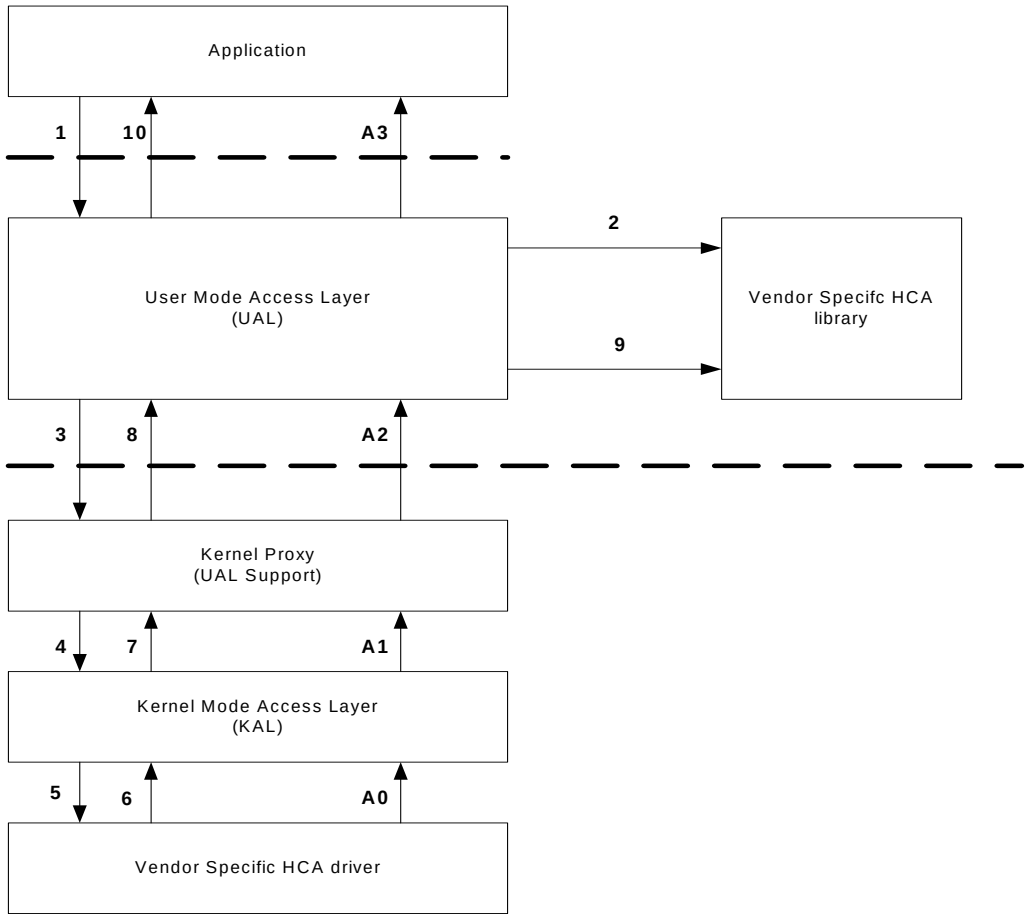

Figure 2-5 User Mode Support Flow

The different steps in interactions are listed here for creating a completion queue. Some vendors would prefer to allocate memory for user mode framework in the context of the user address space. Some vendors due to certain hardware restrictions might create them in kernel mode and perform any necessary memory mappings of a kernel mode buffer. Using user address for allocation and maintenance is the preferred approach since the operating system may have a restricted number of linear kernel address space limitations. Moreover the traditional process accounting for the kernel allocated memory for user mode process is not tracked in per process cost structure in most Unix operating systems.

Allocating CQ memory in user mode

1. Application calls UAL to allocate a CQ
2. UAL call the user mode support library if one is available. This calls is called *pre-create* which is performed before the actual creation of the CQ. The vendor is expected to perform any setup required for OS bypass in this step. It allocates a buffer so that it can pass and receive information from its kernel mode driver. For e.g. the Vendor library may choose to now allocate memory in user mode, and pass the buffer addresses to its kernel mode HCA driver.
3. UAL passes this driver private data to the kernel proxy component in the kernel. The kernel proxy support component then copies the private driver specific data to a kernel mode buffer before calling the KAL for the resource creation.
4. Kernel proxy calls the kernel mode AL for the CQ creation
5. KAL now calls the kernel mode HCA driver for the CQ to be created passing a pointer to the kernel mode buffer that contains the data, which was passed, by the user mode HCA specific library.
6. Kernel mode HCA driver now uses the buffers passed from user-mode. Using OS specified functions to pin the buffer down the CQ memory, perform the real creation of the CQ and returns back to KAL.
7. KAL performs resource-tracking operations and returns a handle back to the kernel proxy component.
8. User kernel proxy now performs some additional tracking so that it can facilitate asynchronous notifications in user mode for this CQ and returns back to UAL
9. UAL prepares and calls the user mode HCA specific library for *post-create* calls.
10. UAL now prepares the ib_cq_handle and returns back to user mode application that made the call.

In cases where the driver allocates the buffers, the user mode component may perform the memory mapping in the *post-create* calls. The advantages of the UAL facilitating the kernel dialogue between the user mode HCA library and the kernel mode HCA driver are.

- Single point of cleanup trigger when user program exits or performs an abnormal exit.
- Fewer drivers interfaces necessary and simplifies the infrastructure and client tracking from the HCA drivers.
- All callbacks are facilitated from the UAL and the kernel proxy. Any necessary threads in order to perform the notification is not required to be maintained by each vendor simplifying the code that would otherwise be duplicated in each vendor library.