InfiniBand* Software Architecture
Access Layer
High Level Design

June 2002

*Other names and brands may be claimed as the property of others.
THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY
WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR
ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Intel
disclaims all liability, including liability for infringement of any proprietary rights, relating to use of information in this
specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted
herein.

This Specification as well as the software described in it is furnished under license and may only be used or copied in
accordance with the terms of the license. The information in this document is furnished for informational use only, is
subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel
Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or
any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or
transmitted in any form or by any means without the express written consent of Intel Corporation.

Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and
other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2002 Intel Corporation.
Abstract

Modern computing environments increasingly require high-bandwidth, low-latency, and 24x7 availability. This trend demands fail-safe operation, greater protection, higher isolation, deterministic behavior, and greater capacity to move data between processing nodes as well as between a processor node and I/O devices. The InfiniBand Architecture defines a point-to-point switched I/O fabric providing high-speed communication and a robust management infrastructure necessary to meet these requirements.

The InfiniBand Architecture defines a Host Channel Adapter (HCA) providing a hardware interface to the fabric. An HCA is required by the architecture to implement a verbs software transport interface. Each HCA in a system must be managed independently and can only be opened for use by one consumer at a time. Once opened, an HCA cannot be opened again until it has been closed. This architectural requirement is severely limiting in most systems, where multiple applications typically require concurrent access to the fabric. To meet these system-level requirements, a software component above the verbs interface is used to multiplex access from multiple consumers to the verbs interface of each HCA. This software component is called the InfiniBand Access Layer (AL). The definition of the InfiniBand Access Layer is outside the scope of the InfiniBand Architecture.

The document defines the InfiniBand Access Layer and describes the programming interface and high-level internal design. The Access Layer is a software component that provides an interface to the InfiniBand fabric to multiple concurrent users. The Access Layer exposes the capabilities of the InfiniBand Architecture and augments the verbs software transport interface with support for operations required by most users of InfiniBand. Users define the protocols and policies used by the Access Layer, and the Access Layer implements them under the direction of a user.

The Access Layer architecture is constructed using bottom-up, object-oriented design techniques combined with top-down functional requirements. Based on these requirements, the specific interface functions exposed by the Access Layer are created. Operations common between function interface calls are implemented using generic components. Generic components are built using standard library modules where possible. To support user-specific interface requirements, detailed modules are layered over the generic components.

The Access Layer operates in both kernel and user-level environments. I/O control codes are used to communicate between the user-level and kernel-level modules. The main features of the Access Layer are grouped into the following areas: management services, resource management, work processing, and memory management.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Introduction</td>
<td>1-1</td>
</tr>
<tr>
<td>1.1</td>
<td>Purpose and Scope</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2</td>
<td>Audience</td>
<td>1-1</td>
</tr>
<tr>
<td>1.3</td>
<td>Acronyms and Terms</td>
<td>1-1</td>
</tr>
<tr>
<td>1.4</td>
<td>References</td>
<td>1-2</td>
</tr>
<tr>
<td>1.5</td>
<td>Conventions</td>
<td>1-2</td>
</tr>
<tr>
<td>1.6</td>
<td>Before You Begin</td>
<td>1-2</td>
</tr>
<tr>
<td>2.</td>
<td>Features</td>
<td>2-1</td>
</tr>
<tr>
<td>3.</td>
<td>Design Assumptions &amp; Rules</td>
<td>3-3</td>
</tr>
<tr>
<td>4.</td>
<td>Design Overview</td>
<td>4-4</td>
</tr>
<tr>
<td>4.1</td>
<td>Verb Support</td>
<td>4-4</td>
</tr>
<tr>
<td>4.2</td>
<td>MAD Processing</td>
<td>4-5</td>
</tr>
<tr>
<td>4.3</td>
<td>Service Registration</td>
<td>4-6</td>
</tr>
<tr>
<td>4.4</td>
<td>Subnet Queries</td>
<td>4-7</td>
</tr>
<tr>
<td>4.5</td>
<td>Communication Management</td>
<td>4-8</td>
</tr>
<tr>
<td>4.6</td>
<td>Subscription Services</td>
<td>4-9</td>
</tr>
<tr>
<td>4.7</td>
<td>Multicast Services</td>
<td>4-10</td>
</tr>
<tr>
<td>4.8</td>
<td>Plug and Play</td>
<td>4-11</td>
</tr>
<tr>
<td>4.9</td>
<td>I/O Controller Services</td>
<td>4-12</td>
</tr>
<tr>
<td>4.10</td>
<td>User-Mode Support</td>
<td>4-13</td>
</tr>
<tr>
<td>5.</td>
<td>Design Details</td>
<td>5-1</td>
</tr>
<tr>
<td>5.1</td>
<td>QP and CQ Services</td>
<td>5-1</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Locking and Threading Model</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2</td>
<td>GSA/SMA</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Locking and Threading Model</td>
<td>5-3</td>
</tr>
<tr>
<td>5.3</td>
<td>MAD Special QP Alias</td>
<td>5-3</td>
</tr>
<tr>
<td>5.4</td>
<td>MAD Service</td>
<td>5-4</td>
</tr>
<tr>
<td>5.5</td>
<td>Special QP Agent</td>
<td>5-8</td>
</tr>
<tr>
<td>5.6</td>
<td>MAD QP Service</td>
<td>5-9</td>
</tr>
<tr>
<td>5.7</td>
<td>AL Manager</td>
<td>5-15</td>
</tr>
<tr>
<td>5.8</td>
<td>Generic Class Agents</td>
<td>5-17</td>
</tr>
<tr>
<td>5.9</td>
<td>Connection Agent</td>
<td>5-19</td>
</tr>
<tr>
<td>5.10</td>
<td>Service ID Resolution Agent</td>
<td>5-21</td>
</tr>
<tr>
<td>5.11</td>
<td>Service Registration Agent</td>
<td>5-23</td>
</tr>
<tr>
<td>5.12</td>
<td>Query Agent</td>
<td>5-24</td>
</tr>
<tr>
<td>5.13</td>
<td>Subscription Agent</td>
<td>5-26</td>
</tr>
<tr>
<td>5.14</td>
<td>Device Management Agent</td>
<td>5-27</td>
</tr>
<tr>
<td>5.15</td>
<td>Plug and Play Manager</td>
<td>5-29</td>
</tr>
<tr>
<td>5.16</td>
<td>User-Mode Services</td>
<td>5-34</td>
</tr>
<tr>
<td>5.16.1</td>
<td>User-Mode Access Layer Data Flow</td>
<td>5-34</td>
</tr>
<tr>
<td>5.16.2</td>
<td>User-Mode Loading and Initialization</td>
<td>5-36</td>
</tr>
</tbody>
</table>
5.16.3 User-Mode Unloading ................................................................. 5-37
5.16.4 Process Specific Cleanup ............................................................ 5-37
5.16.5 User-Mode Debugging ............................................................... 5-37
5.16.6 User-Mode Verbs Support .......................................................... 5-37
5.16.7 Notification Mechanism ............................................................. 5-38
5.16.8 Callback Processing ................................................................. 5-39

6. Initialization & Shutdown ................................................................. 6-1
7. Data Structures and APIs ............................................................... 7-3

Figures

Figure 5–1. Verbs Dispatch Support ...................................................... 4-4
Figure 5–2. MAD Processing Support .................................................... 4-5
Figure 5–3. Service Registration Support .............................................. 4-6
Figure 5–4. Subnet Queries Services ...................................................... 4-7
Figure 5–5. Communication Management ............................................ 4-8
Figure 5–6. Subscription Services ........................................................ 4-9
Figure 5–7. Multicast Services .............................................................. 4-10
Figure 5–8. Plug and Play Support ........................................................ 4-11
Figure 5–9. I/O Controller Support ....................................................... 4-12
Figure 5–10. User-Mode Support ........................................................ 4-13
Figure 6–1. QP and CQ Services - Work Request and Completion Processing .................................................. 5-1
Figure 6–2. GSA/SMA Services – Overview .......................................... 5-2
Figure 6–3. MAD Special QP Alias ....................................................... 5-3
Figure 6–4. MAD Service – Send Processing ......................................... 5-5
Figure 6–5. MAD Service – Receive Processing ...................................... 5-7
Figure 6–6. Special QP Agent – Send Processing .................................... 5-8
Figure 6–7. Special QP Agent – Receive Processing ............................... 5-9
Figure 6–8. MAD QP Service - Work Request Processing ..................... 5-10
Figure 6–9. MAD QP Service - Receive Completion Processing .......... 5-12
Figure 6–10. MAD QP Service - Send Completion Processing ............. 5-14
Figure 6–11. Access Layer Overview ................................................... 5-16
Figure 6–12. Generic Class Agent Multiplexor ...................................... 5-18
Figure 6–13. CM Agent – Connection Establishment ............................ 5-20
Figure 6–14. SIDR Agent – Service ID Resolution ............................... 5-22
Figure 6–15. Service Registration Agent .............................................. 5-23
Figure 6–16. Query Agent ................................................................. 5-25
Figure 6–17. Subscription Agent ......................................................... 5-26
Figure 6–18. Device Management Agent ............................................ 5-28
Figure 6–19. Plug and Play – Registration ............................................ 5-30
Figure 6–20. Plug and Play – Load Service ........................................... 5-31
Figure 6–21. CA PnP Service ............................................................. 5-32
Figure 6–22. IOC PnP Agent ............................................................. 5-33
Figure 6–23. User-Mode Access Layer Architecture ............................ 5-35
Figure 6–24. User-Mode Access Layer Notification Mechanism.................................5-38
1. Introduction

1.1 Purpose and Scope

This document describes the High Level Design (HLD) of the InfiniBand Access Layer. It defines the internal architecture of InfiniBand Access Layer, including inter-component dependencies, and provides sufficient design detail for an implementation based on this HLD will satisfy the product requirements.

1.2 Audience

Anyone interested in understanding the implementation of the InfiniBand Access Layer should read this document, including:

- Software developers who are integrating the separate modules into their own software projects
- Hardware developers who need an understanding of the software behavior to optimize their designs
- Evaluation engineers who are developing tests for InfiniBand-compliant devices
- Others in similar roles who need more than a basic understanding of the software

1.3 Acronyms and Terms

Access Layer (AL)

A software component that provides an interface to the InfiniBand fabric to multiple simultaneous consumers through the verbs interface of all HCAs in a system.

AL
See Access Layer.

uAL
User-mode Access Layer.

kAL
Kernel-mode Access Layer.

uDAPL
See User-Level Direct Access Transport APIs.

User-Level Direct Access Transport APIs

A set of standard, transport-independent, platform-independent, user-level Application Programming Interfaces created by the DAT Collaborative that exploit the RDMA capabilities of next-generation interconnect technologies.
1.4 References

IBA Software Architecture Verbs Provider Driver High Level Design
Intel Corporation. IBA Software Architecture Verbs Provider Driver High Level Design.

1.5 Conventions

This document uses the following typographical conventions and icons:

*Italic* is used for book titles, manual titles, URLs, and new terms.
*Bold* is used for user input (in the Installation section).
*Fixed width* is used for code definitions, data structures, function definitions, and system console output. Fixed width text is always in Courier font.

**NOTE**
Is used to alert you to an item of special interest.

**DESIGN ISSUE**
Is used to alert you to unresolved design issues that may impact the module’s design, function, or usage.

1.6 Before You Begin

Please note the following:

This document assumes that you are familiar with the *InfiniBand Architecture Specification*, which is available from the InfiniBand Trade Association at [http://www.infinibandta.org](http://www.infinibandta.org).
2. Features

The InfiniBand Access Layer provides the following required features:

Kernel and user mode access
The Access Layer consists of kernel and user mode components providing an interface for channel drivers and applications. The user mode Access Layer communicates with its kernel mode counter-part through an ioctl interface.

Meets clients needs
The Access Layer exposes all features of InfiniBand by providing a verbs level interface. In addition, the Access Layer provides support for higher-level operations required by most users of InfiniBand.

Compliant with InfiniBand Specifications
The Access Layer strives to be compliant with the InfiniBand Architecture specifications. Any known deviations are flagged.

Hot plug of fabric-attached IOCs
The Access Layer includes a PnP manager component to facilitate notification of events to clients. Clients may subscribe for notification of various events such assignment of an IOC or a port state change.

APIs allow changes with backward compatibility
The Access Layer programming interface design is intended to be generic and functionally complete to meet the needs of current and future clients without requiring changes for each HCA. However, the interface can be extended with minimal impact to existing clients should a change become necessary.

Meets Fellow Traveler requirements
It is anticipated that the requirements of Fellow Travelers will be very similar to those required by the internal clients of the Access Layer. By definition, all Fellow Travelers work together to create a design that meets the needs and is agreed to by everyone.

Meets performance requirements
The internal design of the Access Layer maximizes performance without sacrificing quality or robustness. Speed path operations are of special concern and receive the most attention in the design.

Support RedHat Linux Version 7.2
The Access Layer will be developed on the RedHat Linux Version 7.2 or later kernel. A goal for the design is the use of common kernel functions abstracted by the component library to maximize the likelihood that the code can be re-used without source code modifications on future kernel releases.

Support dynamic load, unload, and update
The Access Layer is implemented as a loadable Linux device driver. This allows the Access Layer to be easily unloaded and updated when there are no active clients.
Supports Multiple HCAs
The Access Layer does not have any internal design limitations on the number of HCAs in a system that it supports.

Support UP and MP
The Access Layer is targeted for execution on enterprise server platforms. Platforms in this class of systems range from small uniprocessor to large multi-processor systems.

Provide debug and production binaries
Checked and free versions of the Access Layer binaries are provided allowing developers to readily debug or tune the performance of client applications.

Support multiple simultaneous clients
The primary purpose of the Access Layer is to support multiple simultaneous clients.

Document external interfaces
All external interfaces exposed to clients of the Access Layer are documented in this design specification. Refer to Section 7 for a description of the Access Layer programming interface.

Meet fabric-scaling requirements
There are no built-in limitations on the size of the fabric supported by the Access Layer.

Meet common Linux requirements
The Access Layer is designed and implemented to meet the requirements of a common Linux device driver.
3. Design Assumptions & Rules

The following assumptions and rules were used in the design process of the Access Layer. An assumption is any implicit design rule or constraint that is needed but not explicitly stated as part of the design. Assumptions are listed here because they may invalidate the design if they change later.

**InfiniBand Specification**

The Access Layer was designed to provide access to a fabric based on the InfiniBand Architecture Specification Version 1.1. In addition, several specifications are currently under development by the InfiniBand Application Working Group. Some operations performed by the Access Layer are based on these preliminary AGW specifications. Changes in the InfiniBand Architecture may invalidate the Access Layer design.

**Linux OS**

Although the internal design and the use of the component library may allow the Access Layer to be ported to other operating systems, the primary target of the Access Layer implementation is for RedHat Linux Version 7.2 or later.
4. Design Overview

The Access Layer provides functionality required to access services over an InfiniBand fabric. The functionality provided by the Access Layer can be divided into the following main areas:

- Verb support
- MAD processing
- Service registration
- Subnet query support
- Communication management
- Subscription service
- Multicast support
- Plug and play support
- I/O controller services
- User-mode support

The Access Layer supports verb functionality as defined in the InfiniBand specification. It also provides value add services handling MADs. Additionally, it assists users in performing common functions by abstracting MAD functionality for the user. Abstracted MAD functions are available through the service registration, subnet query support, subscription, and communication management services. Plug and play support is handled by the Access Layer, including local and remote event reporting and I/O controller assignment. These services are explained in the following sections.

4.1 Verb Support

![Figure 4–1. Verbs Dispatch Support](image)
The Access Layer exposes all features supported by the verbs architecture. It dispatches requests to multiple channel adapters, allowing clients access to all channel adapters in the system, including those from multiple vendors. In the kernel, verb support is often limited to pass through functionality. For highest performance, pass through functionality is designed around the use of function pointer tables, and an effort has been made to have the Access Layer use the same parameters and structures as those required by the verbs provider. In such cases, the impact on performance to using the Access Layer should be minimal. User-mode support is described in more detail in section 4.10.

4.2 MAD Processing

![Diagram of MAD Processing Support](image)

The Access Layer provides extensive support for MAD processing. The Access Layer will segment large send requests into appropriately sized MADs, reassemble segmented MADs upon receiving multiple packets, and perform transaction management. Based on user-specified settings, MAD Services will retry send operations until a response is received. Any queue pair may be configured to provide MAD support, and all Access Layer owned QP’s 0 and 1 will automatically be configured for MAD support. Thus, MAD Services are the interface to both the GSA and SMA.
4.3 Service Registration

Along with providing MAD support, the Access Layer provides services that abstract common MAD uses. The Access Layer supports the ability for local clients to advertise services to other clients throughout the fabric. It uses standard service registration mechanisms to register services with Subnet Administration. The Service Registration Agent relieves clients from having to use MADs directly. The Service Registration Agent handles sending and receiving MADs, locating Subnet Administration, and retrying registration requests.
4.4 Subnet Queries

The Query Agent issues queries to Subnet Administration at the request of a user. The Query Agent relieves clients from having to use MADs directly. The Query Agent handles sending and receiving MADs, locating Subnet Administration, and retrying queries. The Query Agent supports a small number of commonly executed queries. Typical queries supported by the Query Agent include obtaining path records and service information of remote clients.
4.5 Communication Management

The CM Agent assists in connecting queue pairs and end-to-end contexts, and performing service ID resolution. The CM Agent handles connection establishment between remote queue pairs by executing the standard communication management protocol. The CM Agent is responsible for transitioning the state of the queue pairs, and placing queue pairs and end-to-end contexts into and out of a time wait state. In addition to connection establishment, the CM Agent supports datagram communication by providing service ID registration and resolution mechanisms.
4.6 Subscription Services

The Subscription Agent permits a user to register for notification of events that occur on the fabric. Unlike plug and play events, subscription is typically used for notification of remote events. Such events are not directly visible by the local Access Layer and must be provided by a remote Class Manager, such as the subnet or configuration manager.
4.7 Multicast Services

The Multicast Service permits a user to attach or detach an unreliable datagram queue pair a multicast group. The Multicast Service registers the queue pair with Subnet Administration. Once attached, the queue pair receives datagrams addressed to the multicast group.

Figure 4–7. Multicast Services
4.8 Plug and Play

PnP Services report local and fabric events to registered clients. Plug and play events include the addition and removal of a channel adapter, port up and down events, port LID and PKEY configuration changes, and I/O controller assignments. The Load Agent works with the PnP Services to load kernel-mode clients when pre-defined events occur. In order to load clients in response to plug and play events, the Access Layer uses a device configuration file to associate clients with plug and play events.
4.9 I/O Controller Services

The Access Layer supports I/O controllers for both host and target systems. As a target system, it provides device management capabilities that may be used to register an I/O controller. Registered I/O controllers are reported to the configuration manager, which assigns the controller to particular host systems. As a host system, the Access Layer allows clients to register for notification when an I/O controller is assigned to a local port. When an I/O controller is assigned, the Access Layer automatically retrieves information about the services offered by the I/O controller and the I/O unit on which it resides. This information is reported to registered clients.

Figure 4–9. I/O Controller Support
4.10 User-Mode Support

The Access Layer provides both kernel-mode and user-mode support. It is designed to simplify the implementation of the user-level verbs provider library, while providing high-performance access for user-mode applications to access the InfiniBand fabric. The user-mode component performs IOCTL handling to communicate with kernel-mode components, and it interacts with the user-mode verbs provider library.
5. Design Details

5.1 QP and CQ Services

A QP Service is responsible for posting work requests to a hardware queue pair. All data sent or received on a QP Service is opaque to the Access Layer. A CQ Service is responsible for managing work completions on an associated QP Service. Each QP Service must be bound to one or two CQ Services, with users able to direct send and receive completions to different CQ Services.

The QP Service is the basic queue pair abstraction upon which other more complex queue pair services are constructed. Likewise, the CQ Service is the basic completion queue abstraction upon which other more complex completion queue abstractions are constructed. The operation of the QP and CQ Services are highlighted below.

![Diagram](image)

**Figure 5–1. QP and CQ Services - Work Request and Completion Processing**

1. A user of the QP Service posts a send or receive work request to the QP service. The QP Service validates the request.
2. The QP Service formats a work request to the verbs interface and posts it to the underlying QP.
3. The user issues a request to the Access Layer to poll the CQ Service associated with the given QP Service. The user may poll on a periodic interval or may be responding to some notification, such as a completion event. The poll operation operates synchronously.
4. The CQ service retrieves all completed work requests from the underlying completion queue. Completed work requests are returned to the user.
5.1.1 Locking and Threading Model

The QP Service provides serialization posting work requests. However, the implementation of the QP Service shall not serialize the posting of work requests with their completion.

The CQ Service provides serialization removing completed work requests. However, the implementation of the CQ Service shall not serialize completing work requests with their posting.

5.2 GSA/SMA

The Access Layer provides GSA and SMA services on a per port basis. When enabled, each port on all channel adapters owned by the Access Layer will have an instance of a GSA and SMA. Clients are given access to the GSA and SMA via MAD Special QP Alias and MAD Services. The following diagram provides an overview of the GSA and SMA services. Additional details for each identified component are described in later sections.

As shown by the diagram, clients access the GSA and SMA by creating a MAD Special QP Alias. This QP service acts as an alias to QP 0 or QP 1. For each type of MAD that a client wishes to send or receive, they create a MAD Service. Each MAD Service handles the processing of MADs for a given management class and version. For additional details, see sections 5.3 and 5.4.
5.2.1 Locking and Threading Model

The GSA and SMA services provide serialization to QP 1 and QP 0, respectively. Since the GSA and SMA services need to support multiple clients, all locking must be done within the GSA and SMA services themselves. Where possible, locks are held only long enough to protect access to the underlying hardware queue pairs.

The GSA and SMA services support receiving work requests from multiple clients and therefore must be multi-thread aware. Additionally, completion processing in the GSA and SMA will assume multi-threaded operation. Clients of the GSA and SMA must provide re-entrant code for completion processing. The GSA and SMA services will execute in the thread context in which they are invoked. It is expected that completion processing will be executed within a tasklet thread.

5.3 MAD Special QP Alias

MAD Special QP Alias provides an interface to QP 0 or QP 1. Together with MAD Services, the MAD Special QP Alias permits multiple clients to send management datagrams out of QP 0 and QP 1. A single client owns each MAD Special QP Alias, but all MAD Special QP Aliases are tied to either the GSA or SMA at creation. Unlike other QP Services, clients may only post sends to a MAD Special QP Alias, and the data sent must be in the format of a management datagram. In this case, the data is not transparent to the Access Layer.

Before a MAD Special QP Alias can be used to transfer data, it must be bound to one or more MAD Services. Each MAD Service handles MADs for a specified class and version and given methods. A MAD Service belongs to a single MAD Special QP Alias, but a MAD Special QP Alias can reference multiple MAD Services. MAD Services perform segmentation and reassembly and transaction processing.

The operation of sending a MAD over a MAD Special QP Alias is defined below.

![Diagram of MAD Special QP Alias](image-url)
1. A user posts the MAD to a MAD Service associated with the MAD Special QP Alias.
2. The MAD Service retrieves a work permit from its associated MAD Special QP Alias. If it cannot obtain a work permit, the send operation is rejected.
3. Upon completion of sending a MAD, the MAD service returns the work permit back to the MAD Special QP Alias.
4. The user is notified of the MAD completion asynchronously via a callback.

5.4 MAD Service

A MAD Service is responsible for handling sent and received management datagrams. MAD Services are responsible for transaction management and segmentation and reassembly of large MADs. When sending a MAD for which a response is expected, a MAD Service will retry the send operation until a response is received, the send operation times out waiting for the response, or the user cancels the transaction. If a sent MAD is larger than 256 bytes, a MAD Service will automatically segment the MAD into multiple packets, until all packets have been sent.

Although users may not post buffers to a MAD Service to receive incoming data, MAD Services are still responsible for reassembling segmented MADs. MAD Services register with MAD Dispatch Services in order to receive incoming data. When a MAD Dispatch Service receives data, it routes the received data to the correct MAD Service for further processing.

The interaction of a MAD Service with the user and the MAD Dispatch Services are shown in the following figures. For simplification, the internal operation of the MAD Dispatch Service is shown in section 5.6 as part of the MAD QP Service.
1. The user invokes a MAD Service to send a MAD. The MAD Service verifies that it knows how to handle the MAD and the type of operation being requested for it.

2. The MAD Service retrieves a work permit from its associated MAD Special QP Alias. If a work permit cannot be obtained, the operation is rejected. If the MAD requires a response, it is given to the Transaction Agent component of the MAD Service. The Transaction Agent is responsible for sending the MAD and issuing retries until a response has been received. When retrying send operations, the Transaction Agent ensures that the send is outstanding only once to the underlying queue pair.

3. The Transaction Agent hands the MAD to the Segmentation Agent. The Segmentation Agent is responsible for dividing the MAD into separate packets and issuing sends for each packet.

4. To track the process of the segmented MAD, the Segmentation Agent requests structures from the global SAR Resource Manager. These structures are used to track the state of the segmentation and are returned to the Segmentation Agent synchronously. If a segmentation buffer cannot be obtained, the operation is rejected.
5. The Segmentation Agent posts a send operation to a MAD Dispatch Service. MAD Dispatch Services are responsible for handling send operations from multiple MAD Services.

6. When the send operation completes, the MAD Dispatch Service notifies the Segmentation Agent of the completion. If packets remain to be sent, the Segmentation Agent posts additional send operations, repeating steps 6 and 7 until all packets have been transferred.

7. Once all packets have been sent, the Segmentation Agent notifies the Transaction Agent that all data has been sent. If a response is expected, the Transaction Agent queues the operation until a response is received. The Transaction Agent is responsible for retrying the operation.

8. Once the send operation is complete and a response to the MAD has been received, if one is expected, the Transaction Agent returns the work permit to the MAD Special QP Alias.

9. Upon completion of the send, the MAD Service invokes the user’s MAD send completion callback to notify them of the completed request.

10. Finally, the Segmentation Agent returns the Segmentation Tracking structure to the global SAR Resource Manager.

MAD Services handle received MADs in a manner similar to handling sends. The MAD Dispatch Service matches a received MAD with a MAD Service, based on the MADs class, version, and method. The received MAD is given to the MAD Service for additional processing, such as reassembly. The following figure provides an overview of the MAD Service handling a received MAD.
1. The MAD Service registers with the MAD Dispatch Service to receive incoming MADs. The registration process allows the MAD Dispatch Service to match unsolicited MADs with a given MAD Service capable of handling that MAD. Only one MAD Service may be registered to receive unsolicited MADs for a given management class, method, and version.

2. After receiving a MAD, the MAD Dispatch Service routes the received MAD to the proper MAD Service. The Reassembly Agent receives the MAD and checks to see if this is part of a multiple packet MAD.

3. If the received MAD was segmented into multiple packets, the Reassembly Agent synchronously requests a Reassembly Tracking structure from the global SAR Resource Manager. It uses the Reassembly Tracking structure to maintain the state of the reassembly.

4. Once the received MAD has been completely reassembled, it is given to the Transaction Agent. If the MAD was in response to a MAD sent by this MAD Service, the Transaction Agent matches the response with the sent MAD.

Figure 5–5. MAD Service – Receive Processing
5. The MAD Service invokes the user’s callback to notify them of the received MAD. If the MAD was in response to a sent MAD, the sent MAD will also complete at this time.

6. The Reassembly Agent returns any Reassembly Tracking structures to the global SAR Resource Manager.

7. The Reassembly Agent returns all received MAD buffers to the MAD Dispatch Service.

5.5 Special QP Agent

Special QP Agent is a generic name given to the GSA and SMA. Most MAD Services will reside over one of these two agents. The GSA and SMA are similar in that both behave as MAD Dispatch Services. Both send and receive MADs on behalf of multiple clients. From the viewpoint of the Special QP Agents, a client is a MAD Service. The following two diagrams show the behavior of the Special QP Agents sending and receiving MADs.

![Diagram](image)

**Figure 5–6. Special QP Agent – Send Processing**

1. A newly created MAD Service registers with the MAD Dispatch Service. This causes the MAD Dispatch Service to assign a 32-bit client ID to the MAD Service.

2. The MAD Service posts a send request to the Special QP Agent. Since the Special QP Agent handles multiple MAD services from separate clients, the To Send Queue provides synchronization between the multiple clients. The To Send Queue queues send operations until the underlying hardware queue pair can accept additional requests.

3. When the underlying queue pair, owned by the MAD Dispatch Service, can accept additional work requests, the To Send Queue posts queued send MADs to the MAD Dispatch Service.
4. The queue pair owned by the MAD Dispatch Service is bound to a MAD CQ Service. When a completion occurs on a MAD CQ Service, it uses data referenced by the work request identifier to invoke post-processing on the related MAD Dispatch Service.

5. The MAD Dispatch Service routes the completion to the correct MAD Service.

The next figure shows the high-level overview of processing a receive completion.

---

**Figure 5–7. Special QP Agent – Receive Processing**

1. A newly created MAD Service registers with the MAD Dispatch Service for unsolicited MADs.

2. The queue pair owned by the MAD Dispatch Service is bound to a MAD CQ Service. When a completion occurs on a MAD CQ Service, it uses data referenced by the work request identifier to invoke post-processing on the related MAD Dispatch Service.

3. Since the received MAD may be used in post-processing, the MAD Dispatch Service tries to obtain a buffer from a MAD Buffer Manager. If a buffer is available, the MAD Dispatch Service posts the buffer on the receive queue of the underlying queue pair. This helps to ensure that additional traffic can be received.

4. The MAD Dispatch Service routes the received MAD to the correct MAD Service.

Users should refer to section 5.6 for details on the operation of the MAD Dispatch Service and its relationship to MAD Services.

### 5.6 MAD QP Service

The MAD QP service assists in the sending and receiving of management datagrams. Conceptually, it can be viewed as a cross between normal QP Services and MAD Special QP Alias. A MAD QP Service
provides all of the MAD support afforded to a MAD Special QP Alias, but is not restricted to running over QP 0 or 1. Each MAD QP Service has its own underlying hardware queue pair. MAD QP Services may be used for redirection purposes.

The following diagram shows the operation of posting send and receive work requests to a MAD QP Service. In this example, the MAD QP service owns the underlying queue pair. The send and receive operations are numbered separately.

Figure 5–8. MAD QP Service - Work Request Processing

R1. A MAD QP Service is automatically created with a corresponding MAD Dispatch Service. Whereas a MAD Special QP Alias uses the MAD Dispatch Service of the GSA or SMA, MAD QP Services provide their own dispatching agent. As new MAD Services are created and
associated with a MAD QP Service, they register for unsolicited MADs with the MAD Dispatch Service.

R2. The user posts a receive work request to the MAD QP service. The MAD QP service assigns a work permit to the request. If no work permits are available, the request is rejected. The work permit is used to match a request with a completion and assists with any post-processing.

R3. The user’s work request identifier is saved in the work permit and the request is handed to the underlying QP Service.

S1. Similar to the receive path, newly created MAD Services register with the MAD Dispatch Service. The MAD Dispatch Service assigns a 32-bit client identifier to the MAD Service, which it uses to track send and response completions.

S2. The user posts a send work request to the MAD Service.

S3. The MAD Service obtains a work permit for the request. If no work permits are available, the request is rejected. The work permit is used to match a request with a completion and assists with any post-processing.

S4. The MAD Service posts a send request to the MAD Dispatch Service. Since the MAD Dispatch Service handles requests from multiple MAD services, it tracks the request based on the previously assigned client identifier.

S5. A work request to send the MAD is posted to the underlying queue pair.

As shown in the diagram, the MAD Dispatch Service is comprised of: a QP Service, Send Client Manager, Unsolicited Dispatcher, and a Completion Agent.

The Send Client Manager assigns each MAD Service a 32-bit client identifier. This identifier is automatically inserted into MADs as the upper 32-bits of the transaction ID. Affected MADs are those sent for which a response is expected. This permits the MAD Dispatch Service to route received responses to the correct MAD Service.

The Unsolicited Dispatcher is responsible for distributing unsolicited received MADs to the correct MAD service. Unsolicited MADs are those MADs without the response bit set in the MAD header. The Unsolicited Dispatcher routes the MAD to a MAD Service based on the MAD’s class, version, and method. The look-up mechanism employed by the Unsolicited Dispatcher is not defined in the HLD, but it is anticipated that only a small number of MAD services that handle unsolicited MADs will be associated with a MAD Dispatcher.

The Completion Agent component of the MAD Dispatcher is specific to the owner of the MAD Dispatcher. For example, the GSA and SMA will have separate implementations of the Completion Agent than that required by the MAD QP Service.

The operation of completing a work request is shown the next figures.
Figure 5–9. MAD QP Service - Receive Completion Processing

1. The Completion Agent polls the MAD CQ Service for completions. The MAD CQ Service removes all completed work requests from the underlying completion queue. The MADs are given to the Completion Agent for post-processing. The Completion Agent is part of the MAD Dispatch Service.
2. The Completion Agent directs the received MAD to either the Send Client Manager or Unsolicited Dispatcher.
   a. MADs that have the response bit set are given to the Send Client Manager. The Send Client Manager uses the transaction identifier to direct the received MAD to the correct client.
   b. MADs that do not have the response bit set are given to the Unsolicited Dispatcher. The Unsolicited Dispatcher tries to match a registered MAD Service with the received MAD. If a service is not found, the MAD is marked as such and no further processing occurs on the MAD.

3. The MAD is routed to the correct MAD Service
   a. The Send Client Manager routes the MAD directly using the upper 32-bits of the transaction identifier.
   b. The Unsolicited Dispatcher routes the MAD to a MAD Service that handles unsolicited MADs for the given class, version, and method.

4. Post-processing, such as reassembly, may require delaying the return of the received MAD buffer to the user. However, to ensure that additional MADs may still be received, the work permit associated with the received MAD are returned to the RQ Service of the MAD QP Service. The user will be notified that a buffer was received through the poll operation in step one, which indicates that additional receive buffers may be posted to the MAD QP Service.

5. Once all post-processing has completed on the received MAD, the MAD Service hands the completed MAD to the user through an asynchronous callback.
1. The Completion Agent polls the MAD CQ Service for completions. The MAD CQ Service removes all completed work requests from the underlying completion queue. The MADs are given to the Completion Agent for post-processing. The Completion Agent is part of the MAD Dispatch Service.
2. The Completion Agent directs the completed MAD send operations to the Send Client Manager. The Send Client Manager uses the transaction identifier to direct the completed MAD to the correct client.

3. The Send Client Manager routes the MAD to the correct MAD Service using the upper 32-bits of the transaction identifier.

4. Post-processing, such as segmentation, may require delaying the return of the sent MAD buffer to the user. In addition, to ensure that forward progress can be made on the send operation, the work permit associated with the sent MAD is held by the MAD Service until the send operation has completed. Once the send operation has completed successfully, timed out, or has been canceled, the associated work permit is returned to the SQ Service of the MAD QP Service. Until the work permit is returned to the MAD QP Service, further send operations are restricted.

5. Once all post-processing has completed on the sent MAD, the MAD Service hands the completed MAD to the user through an asynchronous callback.

5.7 AL Manager

The AL Manager is a unique, global component of the Access Layer. The AL Manager is at the top of the ownership hierarchy, with all other services under the AL Manager either directly or indirectly. The AL Manager initializes the Access Layer for use by clients and cleans up system resources when shutting down. It is responsible for routing requests to the appropriate services.

The AL Manager provides PnP Services, an Asynchronous Processing Agent, a MAD Buffer Agent, and SAR Resource Manager. It also manages Global Dispatchers, which route requests to the correct Port Agents.
The PnP Services are responsible for reporting local plug and play events to registered clients. Users register with the PnP Services for notification of locally occurring events, or the assignment of I/O controllers to a local port. Once registered, a client will receive notification from the PnP Services that a given event has occurred. Users may register for the addition or removal of a channel adapter, a change in the state of a port, or a change in a port’s configuration.

The Asynchronous Processing Agent is an active component with its own system thread. It is used to change execution from tasklet priority to system thread priority. It also provides serialization of work, such as invoking user callbacks, to ensure that callbacks occur to a client in the proper order. The Asynchronous Processing Agent uses a single thread to guarantee serialization and minimize system resource requirements.

The MAD Buffer Agent maintains a global pool of data buffers that are used internally when sending MADs. Several services by the Access Layer require the ability to send MADs in response to a client request. These services use the MAD Buffer Agent to obtain a MAD and return the MAD to the MAD Buffer Agent once the request is complete. It is expected that this will reduce system resource usage and minimize the amount of memory that must be registered with the underlying channel adapters.
buffers provided by the MAD Buffer Agent are usable on any channel adapter in the system. It is expected the MAD Buffer Agent will be able to service all internal Access Layer services in a timely fashion. If an internal service requires more buffers than the global MAD Buffer Agent can provide, the service can be modified to use a private MAD Buffer Agent in a future release of the software.

The SAR Resource Manager maintains a global pool of structures used to track segmentation and reassembly of MADs. All MAD Services obtain SAR structures from the global SAR Resource Manager. A global SAR Resource Manager helps to reduce system resource requirements. It is anticipated that most MADs that are sent or received will not require SAR, so will not use the SAR Resource Manager.

For every channel adapter in the system, the AL Manager creates a CA Manager. Each CA Manager maintains a list of function pointers in a Verbs Dispatch Table. A CA Manager dispatches functions through the Access Layer to its corresponding verbs provider driver. In addition to maintaining the Verbs Dispatch Table, a CA Manager provides device management support. The CA DM Agent allows clients to register I/O controllers with a given channel adapter. The I/O controllers are reported to a device manager, which exposes them to the fabric. This allows a channel adapter to appear as an I/O unit.

Although not part of the AL Manager, there are a number of Port Agents created by sub-components of the AL Manager. Port Agents permit the Access Layer to be enabled and disabled on a per port basis. The Port Agents supported by the Access Layer include: CM, Service Registration, Subscription, Query, and IOC Port Agents. Although the specific operation of each Port Agent is unique, they share some common functionality. A Port Agent is bound to a single instance of the GSA, sends and receives MADs, and can be considered a value-add service provided by the Access Layer. Port Agents are typically used to send and receive MADs on a specific port.

In order to direct user requests to an appropriate Port Agent, the AL Manager contains a set of Global Dispatchers. Not shown in the preceding figure, there exists a Global Dispatcher for each mentioned Port Agent. Global Dispatchers are responsible for directing requests to the correct Port Agent and maintaining any global information required by all Port Agents. As an example, the CM Global Dispatcher would maintain a list of active connections and listen requests. Calls to form a connection would be routed by the CM Global Dispatcher to a specific CM Port Agent. The CM Port Agent would be responsible for sending the MADs necessary to establish the connection.

### 5.8 Generic Class Agents

With respect to this HLD, a Class Agent is a generic term applied to an Access Layer service residing over the GSA. The Access Layer provides several types of Class Agents, all of which follow a common structure. Class Agents include the CM Port Agent, SIDR Port Agent, Service Registration Port Agent, Query Port Agent, Subscription Port Agent, DM Port Agent, and IOC Port Agent. Access Layer Class Agents perform MAD processing in response to a client’s request. The following diagram illustrates the operation of a Class Agent. Items shown in gray are dependent on the specific implementation of the Class Agent. Two of the gray components are shown outside of the Generic Class Agent Multiplexor but must exist for proper operation.
1. The user makes a request of the Generic Class Agent. Typically requests require obtaining a MAD to send to a remote entity. The specific format of the MAD is dependent on the type of Class Agent being used. For example, a user may invoke the CM Agent with a connection request. The CM Agent will obtain a MAD in order to send a connection request message to the remote endpoint.
2. The MAD Request Queue requests a MAD from the global MAD Buffer Manager. If a MAD cannot be obtained, the user request is failed due to a lack of resources. If a MAD is available, it is removed from the buffer pool and formatted based on the user’s request.

3. The MAD is submitted to a To Send Queue. The To Send Queue ensures that requests from multiple users who may not be aware of each other do not overflow the underlying MAD Special QP Alias.

4. When the MAD Special QP Alias can handle additional work requests - i.e. it has Work Permits available - the MAD is submitted to the MAD Service for processing. For details on the operation of the MAD Service, see section 5.4.

5. The MAD Service posts the send request to the GSA.

6. The GSA returns the sent MAD to the MAD Service. It also routes any responses that were received as a result of the sent MAD to the MAD Service. Typically, the sent MAD is part of a transaction request and a response is expected. In such cases, the MAD Service does not complete the sent MAD until the response has been received.

7. Once all MADs have been sent and received for this transaction, the MAD Service hands the MADs to a Generic Completion Processing Agent.

8. The Generic Completion Processing Agent restarts operation on the To Send Queue since additional Work Permits are now available on the MAD Special QP Alias.

9. The Generic Completion Processing Agent passes the MADs to an implementation specific Completion Processing Agent who interprets the MAD and forms a response to the user.

10. The user is notified of the result of their request. If possible, the data received in the actual MAD is given to the user. This avoids a data copy from the received MAD to an internal buffer. The MAD buffer is owned by the Access Layer on return from the user’s asynchronous callback, so must be copied by the user if the data is desired beyond the duration of the callback.

11. The MAD buffer used send the request is returned to the global MAD Buffer Manager.

5.9 Connection Agent

The Connection Agent is responsible for establishing connections between queue pairs. Along with the SIDR Agent, it is part of the general communication manager. The Connection Agent handles sending and receiving MADs that are part of the connection protocol, including path migration. The high-level operation of a user requesting and receiving a reply to a connection request is shown next.
S1. A user issues a connection request. The connection request is given to the Global CM Agent, which uses the CM Port Dispatcher to route the request to the Connection Agent on the correct CM Port Agent.

S2. The CM Port Dispatcher uses the connection path record information to route the request to the Connection Agent component of the proper CM Port Agent.

S3. The Connection Agent synchronously obtains a structure needed to track the state of the connection once it is established.

S4. The Connection Agent synchronously obtains a structure needed to track the state of the connection request while the request is in progress.

S5. The Connection Agent transitions the local QP Service to the correct state before sending the connection request information.
S6. A request is made to the global MAD Buffer Manager for a MAD. The MAD will be used to send the connection request data.

S7. The Connection Agent posts the MAD to the Generic Class Agent Multiplexor. The Generic Class Agent Multiplexor posts the MAD to the GSA.

R1. The Generic Class Agent Multiplexor receives a response to the connection request and returns it to the Connection Agent.

R2. The Connection Agent searches through its list of active and recently active connections to ensure that the connection information given by the remote side is not part of a stale connection. This is done to prevent a queue pair from being re-used too quickly, which can result in the queue pair improperly receiving stale data.

R3. The Connection Agent transitions the state of the QP Service to indicate that it is now ready to send and receive data.

R4. The Connection Agent queues a request to callback the user with the Asynchronous Processing Agent. The Asynchronous Processing Agent serializes connection establishment callbacks to the user to simplify the user’s connection handling. This avoids issues where two callbacks are invoked in response to received MADs, but the callbacks reach the user in the opposite order than when the MADs were received.

R5. The Asynchronous Processing Agent invokes the user’s callback to notify them of the progress of the connection request.

5.10 Service ID Resolution Agent

The SIDR Agent is responsible for service ID registration and resolution. Along with the CM Agent, it is part of the general communication manager. The SIDR Agent handles sending and receiving MADs that are part of the service ID resolution protocol. The high-level operation of a user requesting and receiving a reply to a SIDR request is shown next.
S1. A user issues a SIDR request. The SIDR request is given to the Global CM Agent, which uses the CM Port Dispatcher to route the request to the correct SIDR Agent.

S2. The CM Port Dispatcher uses the path record information to route the request to the SIDR Agent component of the proper CM Port Agent.

S3. If the SIDR request is registering a service with the local CM Agent, the SIDR Agent synchronously obtains a structure needed to register the service.

S4. If the SIDR request is a request for service ID resolution with a remote CM, the SIDR Agent synchronously obtains a structure needed to track the state of the SIDR request while the request is in progress.
S5. A request is made to the global MAD Buffer Manager for a MAD. The MAD will be used to send the SIDR request data.

S6. The SIDR Agent posts the MAD to the Generic Class Agent Multiplexor. The Generic Class Agent Multiplexor posts the MAD to the GSA.

R1. The Generic Class Agent Multiplexor receives a response to the SIDR request and returns it to the SIDR Agent.

R2. The SIDR Agent invokes the user’s callback to notify them of result of the SIDR request.

### 5.11 Service Registration Agent

The Service Registration Agent allows a user to advertise a service to the subnet. Services are advertised on a per port basis by registering their availability with the subnet administrator. The high-level operation of a user registering a service is shown below.

![Figure 5–15. Service Registration Agent](image)

1. A user issues a request to register a service with Subnet Administration.
2. The Global Service Registration Agent routes the request to the appropriate Service Registration Port Agent. A service is registered with Subnet Administration for a specified port.

3. The Service Registration Port Agent provides a Registration Manager as part of the Generic Class Agent Multiplexor. The Registration Manager is equivalent to the MAD Request Queue and Completion Processing Agent components shown with the Generic Class Agent Multiplexor. (See section 5.8.) The Registration Manager sends a MAD to Subnet Administration to register the service on the local port.

4. The response from Subnet Administration is returned through the Generic Class Agent Multiplexor to the Registration Manager.

5. The result of the registration request is returned to the user.

5.12 Query Agent

The Query Agent provides a means to obtain commonly used information from the subnet administrator and relieves clients from the burden of having to issue MADs directly. The Query Agent handles sending and receiving MADs, locating the subnet administrator, and retrying queries. The high-level operation of the Query Agent is shown below.
1. A user issues a Subnet Administration query.

2. The Global Query Agent routes the request to the appropriate Query Port Agent. Since a different Subnet Manager may configure each port, users specify from which port a query should be issued.

3. The Query Port Agent provides a Query Manager as part of the Generic Class Agent Multiplexor. The Query Manager is equivalent to the MAD Request Queue and Completion Processing Agent components shown with the Generic Class Agent Multiplexor. (See section 5.8.) The Query Manager sends a MAD to Subnet Administration to query for the requested data.

4. The response from Subnet Administration is returned through the Generic Class Agent Multiplexor to the Query Manager.

5. The result of the query is returned to the user.
5.13 Subscription Agent

The Subscription Agent allows users to register for notification of events that occur on the fabric. When the Class Manager that monitors the specified event delivers a report, the Subscription Agent notifies all registered users. The high-level operation of the Subscription Agent is shown below.

1. The user issues a request to subscribe for notification of an event reported to a class manager.
2. The Global Subscription Agent routes the request to the appropriate Subscription Port Agent. Since a different Subnet Manager may configure each port, users specify from which port a subscription should be issued.
3. The Subscription Service checks with the Report Service to see if the local node has already registered with Subnet Administration for the specified event. If the local node has already registered for the event, the subscription request is added to the list of currently subscribed clients.
maintained for that event and the operation completes. Otherwise a subscription request is made to the class manager.

4. The Subscription Service issues a query to the Query Manager in order to locate the class manager that monitors for the specified event.

5. The Query Manager returns path information to the Subscription Service.

6. Using the returned path information, the Subscription Service issues a subscription request to the remote class manager through the Generic Class Agent Multiplexor. The Subscription Service is equivalent to the MAD Request Queue and part of the Completion Processing Agent components shown with the Generic Class Agent Multiplexor. (See section 5.8.)

7. The response from the class manager is returned through the Generic Class Agent Multiplexor to the Subscription Service.

8. The Report Service is updated with the result of the subscription request. The Report Service creates an entry for the event and adds the user as a notification recipient for the event.

9. The result of the subscription is returned to the user.

The following steps describe the action taken by the subscription service upon receiving a report for the occurrence of an event.

T1. The Generic Class Agent Multiplexor hands the report MAD to the Report Service. Along with the Subscription Service, the Report Service comprises the Completion Processing Agent component shown with the Generic Class Agent Multiplexor. (See section 5.8.)

T2. The Report Service locates the event being reported and queues a request for work to the Asynchronous Processing Agent. This allows the notification callbacks to the clients to be serialized and allows execution at a lowered priority level.

T3. The Asynchronous Processing Agent calls back into the Report Service. The Report Service checks if any users still want notification of the reported event. Note that the Asynchronous Processing Agent calls the Report Service rather than the user directly in order to protect against race conditions unsubscribing for an event when it is being reported.

T4. The Report Service invokes a callback to each user registered for notification of the event.

5.14 Device Management Agent

The DM Agent is responsible for reporting I/O controller information in response to queries from a class manager, such as the configuration manager (CFM). The DM Agent allows a user to view the local node as an I/O unit. Users create I/O controllers, assign services to them, and then register the controllers with the local DM Agent.
C1. A user issues a request to the Access Layer to create an I/O controller. I/O controllers are assigned to a specific channel adapter in the system, and are available on all ports of that channel adapter.

C2. The Global CA Dispatcher determines which channel adapter the I/O controller is being created on and hands the request to the proper CA DM Agent. The CA DM Agent creates a new
controller object, which is returned to the user. On creation, the I/O controller does not provide any services.

A1. The user adds one or more service entries to the created I/O controller. Each service entry is associated with a single I/O controller.

E1. The user enables the I/O controller on the selected channel adapter.

E2. The CA DM Agent signals that all ports on the channel adapter should now report I/O controller information when requested by a remote management agent. If this is the first controller enabled on the channel adapter, this results in all DM Port Agents enabling the port for device management support.

Q1. When a query for I/O controller profile information is received, the Generic Class Agent Multiplexor gives it to the DM Responder. The DM Responder is equivalent to the Completion Processing Agent component shown with the Generic Class Agent Multiplexor. (See section 5.8.)

Q2. The DM Responder forms a list of all I/O controllers available on the local channel adapter. It reports only those controllers that have been registered with the CA DM Agent.

Q3. The DM Responder responds to the query with a list of all active I/O controllers.

### 5.15 Plug and Play Manager

Users register with the PnP Manager to receive notification of local channel adapter and port events and the assignment of I/O controllers. The PnP Manager works in conjunction with a Load Agent to automatically load clients whenever a desired event occurs. A high level view of registering with the PnP Manager is shown below.
Figure 5–19. Plug and Play – Registration

1. The user registers for PnP events with a PnP Management Service. The IOC PnP Service processes registrations for I/O controllers, whereas the CA PnP Service processes registrations for local channel adapter and port events. A Registration Service inserts a new registration request into a Registration List and checks to see if the current system information fulfills the registration request. For example, if the user is registering for port up events, the Registration Service determines if any ports on the local channel adapters are active.

2. The PnP Services report the current state of the system. For example, if an I/O controller exists that meets the PnP registration request the IOC PnP Service reports it to the user asynchronously. Likewise, the CA PnP Service reports the current state of all channel adapters and ports that match the registration request. The PnP Services submit work requests to the Asynchronous Processing Agent to report current system information.

3. The Asynchronous Processing Agent calls the PnP Service back. The Asynchronous Processing Agent invokes the PnP Service rather than the user directly in order to provide synchronization between a user registering and deregistering for an event.

4. The PnP Service invokes a user callback with the current system information.
A Load Service is used to automatically load drivers when a specified event occurs. The Load Service uses a Device Config File to register unloaded clients for events. When it detects that such an event has occurred, it responds by loading the associated driver. For example, a client may use the Device Config File to automatically load when a port goes active or when a specific I/O controller is assigned to the local node.

**Figure 5–20. Plug and Play – Load Service**

1. The Device Config File Parser reads the Device Config File. The Device Config File contains a list of modules and events that will cause each to load. The Device Config File can logically be viewed as calls to register for PnP events by the unloaded modules.

2. The Device Config File Parser interprets the data in the Device Config File and hands PnP registration requests to the File Registration Agent.

3. The File Registration Agent uses the standard PnP Manager registration interface to register for notification of all events specified in the Device Config File.
4. When a specified event occurs, the PnP Manager notifies the File Registration Agent. The File Registration Agent locates the module associated with the event.

5. The File Registration Agent issues a request to the Module Loader to load the specified driver.

6. The Module Loader uses the operating specific mechanisms to load the user.

After a user has been loaded, it can register for local and I/O controller events. By returning the current system information during the registration process, the PnP Service can assist a user in determining which event resulted in their being loaded. The following diagram illustrates the operation of the CA PnP Service.

![Diagram of CA PnP Service]

**Figure 5–21. CA PnP Service**

1. A local event is delivered to the CA Report Service.

2. The CA Report Service schedules work with the Asynchronous Processing Agent in order to report the events with registered users. This serializes event notification to the users.

3. The Asynchronous Processing Agent calls the CA Report Service back. The CA Report Service searches the Registration List for any users registered for the event at the head of the CA Event Queue.

4. If a matching registration request is found, the CA Report Service notifies the user of the event.

The following diagram highlights the operation of reporting a newly assigned I/O controller to a registered user.
1. An I/O controller is assigned to a port on a local channel adapter. Notification of the assignment is delivered to the IOC Port Agent.

2. The IOC Port Agent issues queries to obtain path information to the I/O unit containing the controller.

3. The Query Service returns the requested path information to the user IOC Port Agent.
4. The IOC Service Agent uses the path information to query the device manager on the I/O unit for the service entries associated with the assigned I/O controller.

5. The device manager on the I/O unit returns the service entries.

6. The IOC Port Agent notifies the IOC Record Manager of the new I/O controller.

7. The IOC Record Manager updates the IOC Record List with the new I/O controller information. It notifies the IOC Report Service that a new I/O controller is now available.

8. The IOC Report Service schedules work with the Asynchronous Processing Agent in order to report the assigned I/O controller with registered users. This serializes event notification to the users.

9. The Asynchronous Processing Agent calls the IOC Report Service back. The IOC Report Service searches the Registration List for any users registered for the reported I/O controller.

10. If a matching registration request is found, the IOC Report Service notifies the user of the assignment.

5.16 User-Mode Services

User-mode applications such as Subnet Management, uDAPL, etc., interact with the InfiniBand user-mode Access Layer (uAL) to make access the InfiniBand fabric. uAL is a dynamically loadable, shared library that provides a set of services to the applications using the user-mode InfiniBand Access Layer Interface. uAL is automatically loaded when an application linked with uAL library is loaded.

uAL extends the Access Layer kernel-mode support and to user-mode applications as appropriate. Broadly, these include the support for the verbs interface, connection management, management services, work request processing, and Plug and Play.

uAL interacts with the kernel-mode Access Layer through a User-level Proxy Agent in the kernel. uAL provides a means to plug-in a vendor-supplied User-mode Verbs Provider Library (UVP), however, uAL does not require a vendor library. If an HCA vendor has provided a UVP library, uAL makes use of this library to facilitate verbs related calls including speed-path operations such as posting and completing work requests.

For all the non-verbs related functionality, uAL interacts with InfiniBand kernel-mode Access Layer (kAL) through the User-level Proxy Agent. The Proxy Agent is the main entry point for ioctls originating from uAL. The Proxy Agent forwards ioctls to specific sub-components within kAL. Depending on the specific ioctl command, the Proxy Agent may directly call the kernel-mode Access Layer API to satisfy the request or an internal interface between kAL and the Proxy Agent. For example, non-verbs calls from the user process will result in the Proxy Agent calling the kernel-mode Access Layer API to complete the request whereas all verbs calls result in the Proxy Agent calling a kAL provided internal interface to allow additional private arguments to be passed from the vendor-supplied UVP to the VPD.

5.16.1 User-Mode Access Layer Data Flow

The following diagram describes the overall architecture of the user-mode support for Access Layer. For purposes of illustration, the User-level Proxy Agent is shown logically separate from kAL, however it is actually implemented as an integral part of kAL. The diagram below is representative of how the user-mode support for verbs is provided.
The uAL data flow is illustrated with the following example:

1. An application calls a verbs-related API in the Access Layer.
2. uAL calls the vendor-supplied UVP library through a pre-ioctl call provided by the vendor. The UVP can return a buffer to uAL, which will be passed to the kernel VPD.
3. uAL issues an ioctl to forward the request to the kernel proxy.
4. The Kernel Proxy calls kAL. The buffer from the UVP is given to the kAL verbs call.
5. kAL calls the VPD, passing a reference to the buffer from the UVP.

6. The VPD performs the requested operation and places any output data into the buffer.

7. kAL returns to Kernel Proxy with the output buffer information.

8. The Kernel Proxy returns the result through the ioctl to uAL.

9. uAL calls UVP library through a post-ioct1 call provided by the vendor. The buffer from step 2 is returned to the UVP.

10. uAL returns the result to application.

uAL provides callback threads to propagate asynchronous events from kAL. These threads invoke application callback functions. Although the illustration shows one thread, it should be construed as representative. The implementation of uAL may utilize different threads based on the type of event. More information on the uAL notification method can be found in a later section. The asynchronous event data flow is illustrated as follows:

A0. The VPD generates an asynchronous event callback to kAL.

A1. kAL generates an asynchronous event callback to the User-level Proxy Agent.

A2. The User-level Proxy Agent signals the asynchronous notification uAL.

A3. uAL generates an asynchronous event callback to the application.

5.16.2 User-Mode Loading and Initialization

When the application opens a channel adapter, uAL loads the appropriate vendor HCA library (UVP), which is also a dynamically loadable, shared library. The name of the vendor HCA library is obtained through an ioctl to the User-level Proxy Agent in the kernel. The name is available to the proxy agent when the VPD registers the HCA interface with kAL. The loading of UVP is transparent to the application. The UVP library remains loaded as long as the application has resources bound to the respective HCA. If the vendor library name string is null or there is no vendor library installed in the system, uAL will issue ioctl calls to the User-level Proxy Agent to support all calls.

uAL interfaces with the User-level Proxy Agent in the kernel. It uses the Linux character-mode device special file /dev/iba. The User-level Proxy Agent is part of the loadable kernel-mode Access Layer (kAL) device driver. When kAL is loaded, /dev/iba is created to provide user-mode support.

When uAL is loaded in user-mode, it opens the device /dev/iba. For those kernel-mode services needed for user-mode support and which are not InfiniBand-centric, the device /dev/Syshelper is opened. The SystemHelper is a separate loadable kernel module with built-in support for services useful for user-mode support. Examples of System Helper services are:

- OS Wait Objects
- Support for mapping of shared buffers between kernel and user-mode
- Other notification mechanisms useful for user-mode support

The User-level Proxy Agent in kernel-mode and the system helper interface module in the kernel are built using a common device framework in kernel mode. The device framework is expected to provide abstraction of functions used in a Linux device driver implementation. These functions include:

- A method to register a component-specific open and close function.
- A method to register a component-specific ioctl handler. The common device framework is expected to provide function to copy an input ioctl buffer from user-mode to kernel-mode, and vice versa.
• A method to register a component-specific memory map function (mmap() interface).

When a user-mode process opens /dev/iba, the device open performs the initialization required to support user-mode access on a per-process basis. Subsequently, ioctls from the user-mode process will be passed with the process-specific context as well as the parameters of the ioctl to the User-level Proxy Agent. The Proxy Agent then keeps track of per-process resources created with the help of the process-specific context. During initialization, uAL also creates the necessary threads and wait objects needed to process callbacks from kernel mode.

5.16.3 User-Mode Unloading

When a user-mode process exits gracefully, it unloads the uAL library. The unload routine in uAL will make a close_device call to the /dev/iba device perform any cleanup necessary (e.g., destroying callback threads, wait objects, etc.).

The close_device call to /dev/iba performs the necessary cleanup and resource de-allocation in kAL on behalf of the calling process. The closing of the /dev/iba device could be as a result of the user-mode process exiting normally or abnormally. Resources are de-allocated regardless of how the user-mode application exits.

5.16.4 Process Specific Cleanup

User-mode applications can exit gracefully cleaning up all the resources allocated or can exit abnormally (upon a Ctrl-C or other error). When a process exits abnormally, the proxy is responsible for cleaning up all allocated InfiniBand resources. Since many of the resources allocated are in the kernel on behalf of the user process, per-process resource tracking is needed in the kernel for handling the cleanup of an abnormally terminated process. The User-level Proxy Agent in kAL keeps track of per-process resources.

When a process exits, the kernel mode device (/dev/iba) will receive a close_device call with a process-specific context. From this call, the User-level Proxy Agent in kAL can determine which process is exiting and scan the resource tracking pool for any resources remaining to be deallocated for that process. If any kernel resources for the process remain, the Proxy Agent will perform the appropriate cleanup to free the resources.

5.16.5 User-Mode Debugging

uAL will be designed to support application debugging using GDB. Since the uAL library is a shared library, debugging of uAL will not be possible\(^1\). If an application developer needs to debug uAL, the library should be statically linked with the application.

5.16.6 User-Mode Verbs Support

uAL supports a vendor-supplied plug-in library for user-mode verbs. uAL defines the interface that a vendor library must support to fit into the uAL architecture. Through this interface, uAL calls the UVP twice for each verbs related call. The first call occurs before the kernel ioctl to allow UVP to do any pre-ioctl processing (e.g., allocating QP memory). The second call occurs after the kernel ioctl completes to allow post-ioctl processing (e.g., mapping kernel memory to user space). The model does not require the vendor to provide a pre/post ioctl interface for each verbs call. uAL will only invoke the pre/post calls that are provided by vendor; all other calls are passed directly to the kernel through the ioctl interface.

\(^1\) A static library can be debugged through GDB.
5.16.7 Notification Mechanism

A user-mode application using the InfiniBand Access Layer can potentially receive different types of asynchronous events and callbacks from the User-level Proxy Agent. Together, the User-level Proxy Agent and uAL deliver all the callbacks from the kernel-mode drive to the user-mode application. Since a direct callback from kernel mode to user mode is not possible, the following mechanism is used to convey an asynchronous event from kernel-mode to user-mode.

A thread in user-mode waits on an event object that is signaled from the User-level Proxy Agent in the kernel. Upon an asynchronous event occurring in the kernel, the User-level Proxy Agent queues the event and signals the appropriate wait object. The user-mode thread wakes up and retrieves further information about the event via an ioctl to the proxy agent. The diagram below illustrates this approach.

![Diagram](image)

**Figure 5–24. User-Mode Access Layer Notification Mechanism**

1. A user mode thread created by uAL waits on a process specific wait object. This thread handles the callback once the User-level Proxy Agent notifies it.
2. kAL makes a callback to the User-level Proxy Agent in the kernel. Since both these components are in the kernel, this is a direct function call.

3. The User-level Proxy Agent queues the callback context passed by kAL to the appropriate context queue. There is a separate queue for each thread in uAL waiting to handle callbacks. For example, there will be a thread in uAL and a corresponding queue in the proxy agent for completion callbacks, and a similar pair for error callbacks. Once the context is queued, the proxy agent signals the wait object. When the wait object is signaled, the user thread waiting in step 1 wakes up.

4. uAL performs an ioctl to the User-level Proxy Agent to get the contexts from the corresponding context queue. Multiple callback contexts can be de-queued and copied to uAL through this ioctl.

5. From the context provided by proxy agent, uAL locates the application context and callback function and performs a callback to the application. Since the application and uAL are both in user-mode, this is a direct function call.

5.16.8 Callback Processing

The AL API provides several callback types. uAL must handle these callbacks from the kernel on behalf of the application. Based on the type and priority of the callback, uAL groups callbacks into three different categories. Each callback group has a dedicated wait object and user-mode thread for receiving notifications related to the group. In addition, there is a common thread pool allocated for all callback groups to process the callback. When the notification thread is woken up, it queues the callback-processing request to the thread pool, and then goes back waiting for the notification. One of the available threads from the thread pool will get the context related to the callback and make the appropriate application callback. Once the callback thread is done, it returns to the free state, ready to process other callbacks.

The callback groups associated with uAL are given below. Refer to the AL API for a detailed list of callbacks.

5.16.8.1 Connection Manager Callbacks

All Connection Manager related callbacks (e.g., listen callback, request callback, SIDR callbacks, etc.) are grouped together in this type. As mentioned earlier, a dedicated wait object and user mode thread (process specific) are used to handle these callbacks. Once notified of a callback, the notification thread uses the available resources in the thread pool to process the callback.

5.16.8.2 Completion Callbacks

This group handles all work completion callbacks. The behavior of the callback processing is similar to Connection Manager callbacks. Completion callbacks are given high priority when queued to the thread pool. This means when the thread pool starts processing a callback request, it handles the completion callback request before any Connection Manager callbacks or Generic callbacks.

5.16.8.3 Generic Callbacks

This involves all the other generic callbacks such as error callbacks, destroy callbacks, device callbacks, etc. The behavior of the callback processing is similar to the other two groups.
6. Initialization & Shutdown

Once loaded, the Access Layer loads internal modules in the order shown below.

1. The AL Manager is created and initialized. The AL Manager is global throughout the Access Layer and is responsible for the order in which the components are initialized.

2. The AL Manager creates the Asynchronous Processing Manager. The Asynchronous Processing Manager is used by several components and is not directly dependent on other components operating.

3. The Local PnP Agent is created. Most other services are dependent on receiving notification of Local PnP Events. Internal agents and services register with the Local PnP Agent for notification of the additional and removal of channel adapters. The Local PnP Agent notifies its registered clients in the order in which they registered.

4. The Global CA Manager is created. It allocates and initializes the global MAD Buffer Manager. The Global CA Manager registers for notification of channel adapter changes with the Local PnP Agent.
   a. When a new channel adapter is added to the system, the Global CA Manager creates a CA Manager for the newly added channel adapter. The new CA Manager initializes the CA function table. The MAD Buffer Manager then registers the global pool of MAD buffers with the new channel adapter. If the buffers cannot be registered, the new channel adapter is rejected.

5. The AL Manager creates a global SMA Manager. This registers with the Local PnP Agent for notification of channel adapter changes with the Local PnP Agent.
   a. When a new channel adapter is added to the system, the global SMA Managers creates one Port SMA for each port on the channel adapter. Each allocates their respective queue pairs on the new channel adapter and initializes it for use.
   b. When a Port SMA changes the configuration of a port, it notifies the Local PnP Agent of the event. The Local PnP Agent is then responsible for notifying all registers clients of the event.

6. The AL Manager creates a global GSA Manager. The manager registers with the Local PnP Agent for notification of port up and down events.
   a. When a port goes active for the first time, the global GSA Manager creates a Port GSA. The Port GSA allocates the queue pair on the active port. Subsequent port up and down events result in the global GSA Manager simply notifying the affected Port GSA of the event.

7. The AL Manager creates the following agents in the order shown:
   - Query Agent
   - Subscription Agent
   - IOC PnP Agent
   - Service Registration Agent
   - CM Agent
Each agent is global and part of the AL Manager. All agents register for port events. When a port goes active for the first time, each global agent creates a port agent. Each port agent registers with the proper Port GSA in order to send and receive MADs on the specified port. Subsequent port up and down events result in the global managers notifying the affected port agents.

8. The AL Manager creates and initializes the User-Mode Proxy agent.

9. Finally, the AL Manager creates the Load Agent. The Load Agent is initialized last to ensure that all other services have been notified of local plug and play events before the Load Agent attempts to load modules in response to system events. On creation, the Load Agent only registers for notification of channel adapter changes. As new adapters are added, it registers for port changes. This guarantees that it does not load modules for port events before underlying services can be properly initialized on newly active ports.

The shutdown process is the reverse of the initialization process. Internal components remove themselves when receiving notification of the removal of channel adapters.
7. **Data Structures and APIs**

To view the data structures and APIs associated with this component, click on [InfiniBand Access Layer](#).