Revision History and Disclaimers

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draft 1</td>
<td>July 2002</td>
<td>Internal review.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Role</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsible Engineer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Engineering Group Leader</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Engineering Manager</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

This document provides information about the Access Layer and Access Layer exposed API’s.
1. Introduction

1.1 About this document

This document is the users guide to the InfiniBand Access Layer. This document provides information about the Access Layer and Access Layer exposed API’s. This Access Layer API’s are the programming interface defined and accepted by the Linux environment and is part of the open source project.

This document describes Access Layer API’s by providing examples where appropriate. The examples provided in this document uses OS independent component library abstraction in order to be highly portable across different platforms and as well as between user mode and kernel mode. Users are encouraged to use component library functions where possible. However description of the component library API itself is outside the scope of this document also the internal design of Access Layer is out side the scope of this document.

1.2 Before you begin

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.

1.3 Acronyms and Terms

AL: Access Layer
IB: InfiniBand
IBA: InfiniBand Architecture
API: Application Programming Interface
HCA: Host Channel Adapter

1.4 What is Access Layer

Access Layer is one of the sub projects of Linux Open source InfiniBand Project. It is a software component that provides an interface to the InfiniBand fabric to multiple concurrent users. This interface exposes the capabilities of the InfiniBand Architecture and augments the verbs software transport interface with support for higher-level operations required by most users of InfiniBand.

The Access Layer is implemented in both kernel and user-level environments under Linux operating system using C as the programming language. The interfaces exposed by AL are identical in kernel mode and in user mode.

1.5 What does AL depend on

Access Layer depends on
• **Verbs Provider Driver (VPD)** of the underlying InfiniBand HCA. HCA vendor should provide this driver.

• **Component Library (CL)**—OS independent abstraction layer part of the Linux open source InfiniBand sourceforge project

• **User-Mode Verbs Provider (UVP)** library of the underlying InfiniBand HCA. HCA vendor should provide this library. This component is optional and is required only for user mode access.

### 1.6 Why you need it

Since the verbs provider driver does not allow multiple opens to its interfaces, multiple consumers may find it inconvenient to share the same interface across these users. Furthermore, a method of common access needs to be agreed upon by these consumers to share and use an HCA and its resources. The Access Layer (AL) hides the underlying IB Hardware and the verbs provider driver and the Access Layer supports multiple consumers accessing the InfiniBand fabric. It also provides a higher-level abstraction to MAD services, transparent segmentation and reassembly (RMPP), retries, timeouts, plug and play etc commonly needed by the consumers. InfiniBand protocols, middleware and management agents layered on top of the Access Layer and using its interfaces need not have to deal with the different InfiniBand HCA’s since AL provides a single entry point access to these multiple HCA’s that may be provided by different vendors.

### 1.7 Where you get it

Access Layer sources and high-level design document are available under GPL or the Intel BSD + Patent License download at the following site. [http://infiniband.sourceforge.net/](http://infiniband.sourceforge.net/)

### 1.8 Which IB HCA’s are supported

Access Layer is independent of the underlying InfiniBand HCA and can support multi vendor Infiniband HCA’s as long as the hardware driver (verbs provider drivers) of the underlying IB HCA confirms to the interface specification published in [HCA Driver DDK](http://infiniband.sourceforge.net/)

### 1.9 How to compile and load the AL

Once the sources are downloaded, follow the instructions given in the Readme file to compile and load the AL.

### 1.10 How to compile AL consumer

- **Kernel mode consumer**
  
  Include file: `<iba/ib_al.h>`
  `<iba/ib_types.h>`
  
  Compiler Option –I/$Include should point to iba directory where ib_al.h file is located.

- **User mode consumer**
Include file: <iba/ib_al.h>
<iba/ib_types.h>

Compiler Option –I/$Include should point to iba directory where ib_al.h file is located.
Libraries: -lallib -lcomplib –lpthread

1.11 Where can I get detailed information about AL API’s

The script mkdoc available in the sources will generate an HTML file called ib_al.html”. This file provides sufficient information about the API’s, data structure, return types, error codes etc.

1.12 Who are the audience

- 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

1.13 Where can I get more information

Access Layer is a sub project under the Linux InfiniBand Open source project. Other sub projects like IPoIB, OpenSM etc are all available for download at http://infiniband.sourceforge.net/

OpenSM is an Infiniband Subnet Manager used for configuring and monitoring an IB fabric.

1.14 References

[IBA Vol1] InfiniBand Architecture Specification, Volume 1, Release 1.0.a
[IBA Vol2] InfiniBand Architecture Specification, Volume 2, Release 1.0.a

1.15 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.
2. AL API Basic

The AL API is a well-defined and very well accepted programming interface in the Linux environment. The AL API simplifies accessing the IB fabric. The AL API’s also support multiple vendors’ HCA.

The IB access layer software component exist in both user mode and in kernel mode and provides identical API interfaces for the consumers of AL. Hence this document make no distinction between a user mode consumer and a kernel mode consumer. When such difference exists, this document will explicitly state whether it is for user mode or kernel mode.

The benefit of using AL API interfaces to access IB fabric attached devices or applications are that the clients of AL need not have to change when the underlying hardware changes. AL API’s are open source and are compliant with the IB architecture specifications.

IB AL API’s expose both the verbs level interface as specified in the IB specification and higher-level operations required by most users of IB.

To know the list of features supported by the AL please refer to the IB AL High level design document available at [http://infiniband.sourceforge.net/IAL/Access/IBA_AL_HLD.pdf](http://infiniband.sourceforge.net/IAL/Access/IBA_AL_HLD.pdf)

For detailed information about the function prototypes, data structures, error codes see [http://infiniband.sourceforge.net/IAL/Access/IBAL/IBAL_mi.html](http://infiniband.sourceforge.net/IAL/Access/IBAL/IBAL_mi.html)
3. Verbs Level Interface

Access Layer API provides all the verbs level functionality as specified in the IB specification. The verbs level functionality are grouped as shown below with additional AL layer instance API’s and CA service API’s.

- AL Instance
- CA Service
- Domain Service
- Completion Queue Service
- Memory Management Service
- Address vector Service
- EE Context Service
- Queue Pair Service
- Multicast Service

3.1 AL instance

In order to talk to AL, the clients must first open the AL instance. This will be the very first call to Access Layer by the clients of AL.

AL instance API’s exposed by AL are:

- ib_open_al
- ib_close_al

Code Example

```c
#include <iba/ib_types.h>
#include <iba/ib_al.h>

ib_al_handle_t h_al;
ib_api_status_t ib_status = IB_SUCCESS;

/* Open the AL instance and get the handle to the AL */
ib_status = ib_open_al(&h_al);
if(ib_status != IB_SUCCESS)
{
    CL_PRINT(ALTS_DBG_ERROR, alts_dbg_lvl,
             ("ib_open_al failed status = %d", ib_status));
    break;
}
...

/* Close the AL instance when the program is exiting */
ib_status = ib_close_al(h_al);
```
3.2 CA services

AL provides function interfaces to retrieve the list of CA GUIDS available in the system and provides calls to access the CA and its resources on a given channel adapter.

CA Service API’s exposed by AL are:

- `ib_get_ca_guids`
- `ib_open_ca`
- `ib_query_ca`
- `ib_modify_ca`
- `ib_close_ca`

Code Example

```c
intn_t guid_count;
ib_net64_t *p_ca_guid_array;

/* Get the Local CA Guid count */
ib_status = ib_get_ca_guids(ph_al, NULL, &guid_count);
if(ib_status != IB_INSUFFICIENT_MEMORY)
{
    CL_PRINT( ALTS_DBG_ERROR, alts_dbg_lvl,
            ("ib_get_ca_guids failed status = %d\n", (uint32_t)ib_status) );
    break;
}

/* allocate memory for ca_guid_array, guid_count tells the total ca guids available */
p_ca_guid_array = (ib_net64_t*)cl_malloc(guid_count * sizeof(ib_net64_t));
ib_status = ib_get_ca_guids(ph_al, p_ca_guid_array, &guid_count)
if(ib_status != IB_SUCCESS)
{
    CL_PRINT( ALTS_DBG_ERROR, alts_dbg_lvl,
            ("ib_get_ca_guids failed with status = %d\n", ib_status) );
    break;
}
```

3.3 Domain Services

AL provides function interfaces to create and destroy protection domain and reliable datagram domains.

Domain Service API’s exposed by AL are:

- `ib_alloc_pd`
- `ib_dealloc_pd`
- `ib_alloc_rdd`
- `ib_dealloc_rdd`

Code Example
ib_status = ib_alloc_pd(h_ca, IB_PDT_NORMAL, &my_context, &h_pd);
if(ib_status != IB_SUCCESS)
{
    CL_PRINT( ALTS_DBG_ERROR, alts_dbg_lvl, 
    ("ib_alloc_pd failed status = %d", ib_status) );
    break;
}
...
/* alts_pd_destroy_cb() function will be called when the resource is destroyed */
ib_status = ib_dealloc_pd(h_pd, alts_pd_destroyCb);
...

void
alts_pd_destroy_cb(
    void *context
)
{
    CL_ENTER( ALTS_DBG_VERBOSE, alts_dbg_lvl, ("alts_pd_destroy_cb") );
    /* in destroy pd callback handler */
    ...
    CL_EXIT( ALTS_DBG_VERBOSE, alts_dbg_lvl );
    return;
}

3.4 Completion Queue Services

AL provides function interfaces to manage completion queues and to process completions. When a signaled completion occurs on the completion queue, the client will be notified either through callbacks or signaling mechanisms.

Note: A valid PD handle needs to be passed as the input argument for creating the Completion Queue

Completion Queue API’s exposed by AL are:

- ib_create_cq
- ib_query_cq
- ib_modify_cq
- ib_destroy_cq
- ib_poll_cq
- ib_rearm_cq

Code Example

```c
uint32_t cq_size;
ib_status = ib_alloc_pd(h_ca, IB_PDT_NORMAL, &pd_context, &h_pd);
...
cq_create.size = CQ_SIZE;
cq_create.pfn_comp_cb = alts_cq_comp_cb; //Setting a callback function
...```
ib_status = ib_create_cq(h_pd,&cq_create, NULL,alts_cq_err_cb, &h_cq);

ib_status = ib_query_cq(h_cq,&cq_size); //to get the cq size

... 

cq_size += 50; //modify the cq size
ib_status = ib_modify_cq(h_cq,&cq_size);
...

ib_status = ib_destroy_cq(h_cq, alts_cq_destroy_cb);
...

void alts_cq_destroy_cb(
   void *context
)
{
   CL_ENTER( ALTS_DBG_VERBOSE, alts_dbg_lvl, 
      ("alts_cq_destroy_cb"));

   /* in destroy cq handler */
...

   CL_EXIT( ALTS_DBG_VERBOSE, alts_dbg_lvl);
   return;
}

3.5 Memory Management Services

This set of API’s provides the client the ability to register and deregister the memory with the channel adapter. Also provides the API’s to register a memory region that is shared across multiple processes.

Memory management API’s exposed by AL are:

- ib_reg_mem
- ib_reg_phys
- ib_reg_shared
- ib_reg_shmid
- ib_rereg_mem
- ib_rereg_phys
- ib_dereg_mr
- ib_create_mw
- ib_query_mw
- ib_bind_mw
- ib_destroy_mw
ib_status = ib_alloc_pd(h_ca, IB_PDT_NORMAL, &pd_context, &h_pd); //passing null context

/*
 * Allocate the virtual memory which needs to be registered
 */
mask = MEM_ALLIGN - 1;
ptr = cl_malloc(MEM_SIZE + MEM_ALLIGN - 1);

ASSERT(ptr);
ptr_align = ptr;

if(((unsigned)ptr & mask) != 0){
    ptr_align = (char *)((unsigned)ptr+mask) & ~mask;
}
virt_mem.vaddr = ptr_align;
virt_mem.length = MEM_SIZE;
virt_mem.access_ctrl = (IB_AC_LOCAL_WRITE | IB_AC_MW_BIND);
/*
 * Register the memory region
 */
ib_status = ib_reg_mem(h_pd, &virt_mem, &lkey, &rkey, &h_mr);
if(ib_status != IB_SUCCESS){
    CL_PRINT(ALTS_DBG_ERROR, alts_dbg_lvl, "ib_reg_mem failed status = %d", ib_status);
    break;
}
...
ib_status = ib_dereg_mr(h_mr);

### 3.6 Address Vector Services

This set of API’s provides Address vector services to clients of the Access Layer.

Address vector service API’s exposed by AL are:

- `ib_create_av`
- `ib_query_av`
- `ib_modify_av`
- `ib_destroy_av`

---

**Code Example**

```c
ib_av_attr_t alts_av_attr; //Address Vector attribute structure
alts_av_attr.port_num = local_port_num; //Port number through which the packet has to leave
alts_av_attr.dlid = destldid;
alts_av_attr.sl = 0;
alts_av_attr.static_rate = 0;
alts_av_attr.path_bits = 0;
... ib_status = ib_alloc_pd(h_ca, IB_PDT_NORMAL, (void *)&pd_context, &h_pd);
...```
ib_status = ib_create_av(h_pd, &alts_av_attr, &h_av);
...

ib_status = ib_destroy_av(h_av);
...

3.7 EE Context Services

This set of API’s provides the clients the ability to create and manage end-to-end contexts.

EE Context service API’s exposed by AL are:

- ib_create_eec
- ib_query_eec
- ib_modify_eec
- ib_destroy_eec

Code Example

ib_status = ib_alloc_rdd(h_ca, &h_rdd);
...
ib_status = ib_create_eec(h_rdd,
    NULL, /* Context */
    alts_eec_err_cb, /* Error call back function */
    &alts_eec_attr,
    &h_eec);
...

3.8 Queue Pair Services

This set of API’s provides the clients the ability to create, manage and post work requests.

Queue Pair service API’s exposed by AL are:

- ib_create_qp
- ib_query_qp
- ib_modify_qp
- ib_destroy_qp
- ib_post_send
- ib_post_recv

Code Example

/*
 * Set Queue pair attributes and call ib_create_qp to create a queue pair. This created queue pair
 * will be in reset state when created.
 */
ib_cq_create_t cq_create_attr;
...
qp_create_attr.h_rdd = NULL; //Since this is not RD
3.9 Multicast Services

This set of API’s provides the clients the ability to join and leave multicast groups.

Multicast service API’s exposed by AL are:

- ib_join_mcast
- ib_leave_mcast

Code Example

```c
ib_mcast_req_t mcast_req;

...}
mcast_req.p_mcast_gid   = create_broadcast_mgid( pkey, &mgid );
mcast_req.create        = FALSE;
mcast_req.mcast_context = context;
mcast_req.pfn_mcast_cb  = ipoib_broadcast_join_callback;
mcast_req.timeout_ms    = mcast_timeout;
mcast_req.retry_cnt     = mcast_retry_cnt;

status = ib_join_mcast( context->h_qp, &mcast_req, context->h_bcast );
if ( status == IB_SUCCESS )
{
    ...
}
```
4. Higher level Operation Interfaces

Higher-level operation interfaces provide value added functionality commonly needed by most clients. The type of functionality includes some of the following:

- Mad Processing
- Service Registration
- Subnet Query Support
- Communication Management
- Subscription Service
- Plug and Play support
- I/O controller service

4.1 MAD Services

MAD service API’s exposed by the AL provides support for MAD processing. All the segmentation and reassembly and the transaction management are handled inside the Access Layer. Mad services will also retries of send operations based on the clients initial settings. Any UD queue pair may be configured to provide MAD support. Access Layer owned QP0 and QPI are automatically configured for MAD support and are available as the aliased queue pairs. These aliased queue pair access through the MAD services are also the interfaces to the GSA and SMA.

MAD service API’s exposed by AL are:

- `ib_init_dgrm_svc`
- `ib_reg_mad_svc`
- `ib_send_mad`
- `ib_cancel_mad`
- `ib_coalesce_mad`

Code Example

```c
ib_status = ib_open_al(&h_al);

ib_status = ib_get_ca_guids(h_al, ca_guid_array, &guid_count);

ib_status = ib_open_ca(h_al, ca_guid_array[0], //Default open the first CA
alts_ca_err_cb, //CA error Callback
&ca_context, //ca_context
&h_ca);
```

/*
* USE IB_PDT_ALIAS type for IB_QPT_QP0_ALIAS and IB_QPT_QP1_ALIAS QPs. For any other queuepair
* type user IB_PDT_NORMAL
*/

ib_status = ib_alloc_pd(h_ca, IB_PDT_NORMAL, NULL, &h_pd);
...

qp_create_attr.qp_type = IB_QPT_MAD; //Creating MAD Service QP type
qp_create_attr.h_rdd = NULL;
qp_create_attr.sq_depth = SQ_DEPTH;
qp_create_attr.rq_depth = RQ_DEPTH;
qp_create_attr.sq_sge = 0x01;
qp_create_attr.rq_sge = 0x01;
qp_create_attr.sq_signal = TRUE;

/* h_sq_cq and h_rq_cq is set to NULL if qp_type = IB_QPT_MAD, IB_QPT_QP0_ALIAS, IB_QPT_QP1_ALIAS */
qp_create_attr.h_sq_cq = h_cq;
qp_create_attr.h_rq_cq = h_cq;

ib_status = ib_create_qp(
    h_pd,
    &qp_create_attr,
    &qp_context,
    qp_err_cb, //QP error Callback function
    0, //Qkey is set to Zero. Let AL manage my Qkey
    &h_qp);

/*
* This call binds the queue pair to a given port and transitions its
* state to ready to send and receive data. */

ib_status = ib_init_dgrm_svc(h_qp, dgrm_info);
...

mad_svc.mad_svc_context = &my_svc_context;
mad_svc.pfn_mad_send_cb = mad_send_cb; //Send callback function
mad_svc.pfn_recv_cb = mad_recv_cb; //Recv callback function
mad_svc.mgmt_class = IB_MCLASS_MY_CLASS;
mad_svc.mgmt_version = MY_CLASS_BASE_VERSION;
mad_svc.header_size = sizeof(my_class_hdr_t);
mad_svc.p_method_array = NULL; //Not supporting unsolicited

/*This routine registers a queue pair as using a particular management
class */

ib_status = ib_reg_mad_svc(h_qp, &mad_svc, &h_mad_svc);
...

virt_mem.vaddr = ptr_align;
virt_mem.length = MEM_SIZE;
virt_mem.access_ctrl = (IB_AC_LOCAL_WRITE | IB_AC_MW_BIND);
/*
* Register the memory region
*/

ib_status = ib_reg_mem(h_pd, &virt_mem, &lkey, &rkey, &h_mr);
...

ib_status = ib_post_recv(h_qp, p_r_wr, &p_failure_wr);
IBA Software Architecture
Access Layer
Users Guide

... ib_status = ib_send_mad(h_mad_svc, p_mad_send_wr, NULL, &h_mad_send);
...

4.2 Service Registration Services

Service Registration service API’s exposed by the AL provide the ability for local clients to advertise services to other clients throughout the fabric. It uses standard service registration mechanisms to register services with the Subnet Administrator. 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.

Service Registration service API’s exposed by AL are:

- ib_reg_svc
- ib_dreg_svc

Code Example

... /*Registers a service with SA */

ib_status = ib_open_al(&h_al);
...
reg_svc_req.svc_rec.svc_lease = LEASE_PERIOD;
reg_svc_req.svc_rec.svc_flags = SVC_FLAG;
reg_svc_req.svc_rec.svc_name = SVC_NAME;
reg_svc_req.svc_rec.svc_id = SVC_ID;
reg_svc_req.svc_rec.port_guid = p_port_attr->port_guid;
reg_svc_req.timeout_ms = 100;
reg_svc_req.retry_cnt = IB_FLAGS_SYNC;
reg_svc_req.svc_context = &my_svc_context;
reg_svc_req.pfn_reg_svc_cb = reg_svc_cb;

ib_status = ib_reg_svc(h_al, &reg_svc_req, &h_reg_svc);
...

4.3 Subnet Query Services

Subnet query services API’s exposed by the AL provides the ability for clients to query the Subnet Administrator. This service supports a small number of commonly used queries like getting path records, port records, Node records, service record by Service IDs and service record by Service Names.

Subnet query service API’s exposed by AL is:

- ib_query
4.4 Communication Management Services

Communication management service API’s exposed by the AL provide the ability for clients to establish connection between remote queue pairs by executing the standard communication management protocol.

Connection Management service API’s exposed by AL is:

- `ib_cm_listen`
- `ib_cm_cancel`
- `ib_cm_req`
- `ib_cm_rep`
- `ib_cm_rtu`
- `ib_cm_rej`
- `ib_cm_mra`
- `ib_cm_lap`
- `ib_cm_apr`
- `ib_cm_dreq`
- `ib_cm_drep`
- `ib_force_apm`

4.5 Service Id Resolution Services

Service Id resolution service API’s exposed by the AL provide the ability for clients to register and perform SIDR.

Connection Management service API’s exposed by AL is:

- `ib_reg_sidr`
- `ib_dreg_sidr`
- `ib_sidr_req`
- `ib_sidr_rep`

/* TBD */
ib_cm_listen{

ib_status = ib_open_al(&h_al);
...
4.6 Subscription Services

Subscription service API’s exposed by the AL provide the ability for clients 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.

Subscription service API’s exposed by AL are:

- `ib_subscribe`
- `ib_unsubscribe`

Code Example

```c
... 
ib_status = ib_open_al(&h_al);
... 
ib_status = ib_subscribe(h_al, &sub_req, &h_sub);
... 
```

4.7 Plug and Play Services

Plug and play service API’s exposed by the AL provide the ability for clients to register for local plug and play events.

Plug and play service API’s exposed by AL are:

- `ib_reg_pnp`
- `ib_dreg_pnp`

Code Example

```c
... 
ib_status = ib_open_al(&h_al);
... 

pnp_req.pnp_context = &pnp_context;
pnp_req.pfn_pnp_cb = ipoib_al_callback; //Call back function
pnp_req.pnp_class = IB_PNP_CA; //CA CLASS

ib_status = ib_reg_pnp(h_al, &pnp_req, &h_pnp);
... 
```

4.8 I/O Controller services

I/O controller service API’s exposed by the AL provide the ability for the target systems to export IOC’s to the fabric.

I/O controller services API’s exposed by AL are:

- `ib_create_ioc`
- `ib_destroy_ioc`
• ib_reg_ioc
• ib_add_svc_entry
• ib_remove_svc_entry
• ib_reject_ioc

Code Example

... 
/* TBD */ 
ib_create_ioc(
5. Data Structures and APIs

The script mkdoc available in the sources will generate an HTML file called ib_al.html”. This file provides sufficient information about the API’s, data structures, return types, error codes etc. mkdoc uses robodoc to generate HTML files. Sources and robodoc are downloaded from http://infiniband.sourceforge.net/