IBA Software Architecture
IP over IB Driver
High Level Design

Draft 2

July 2002

Revision History and Disclaimers

Rev.	Date	Notes
Draft 1	March 2002	Internal review.

Abstract

The IP over IB (IPoIB) link driver provides standardized Internet Protocol encapsulation over IBA fabrics as defined by the IETF IPoIB working group. Implemented as a standard Linux link driver, it interfaces to the lower edge of the Linux network stack through the *net_device* abstraction. The IPoIB driver gains access the HCA and subnet management services through the Abstraction Layer. All IPoIB data transfers use the unreliable datagram service. In addition to unicast transfers, IPoIB also requires the use of the IBA UD multicast service.

The primary responsibilities of the IPoIB driver are performin g address resolution to map IPv4 and IPv6 addresses to UD address vectors and the management of multicast membership. The following sections provide details on the features and goals of the design as well as the external interfaces used and exported by the IPoIB driver.

Contents

Figures

1. Design Overview

The IPoIB driver integrates into the Linux network stack as a layer-2 network driver. The network driver is responsible for constructing/deconstructing its own headers, transmitting frames, as well as receiving frames and routing them to the appropriate layer-2 network driver user. Using this model, the IPoIB driver presents a standard Linux driver interface to the bottom of the network stack and interfaces to the Channel Adapter (CA) Access Layer for IBA services.

Figure 1-1 illustrates the IPoIB driver's relationship in a complete Linux network stack.

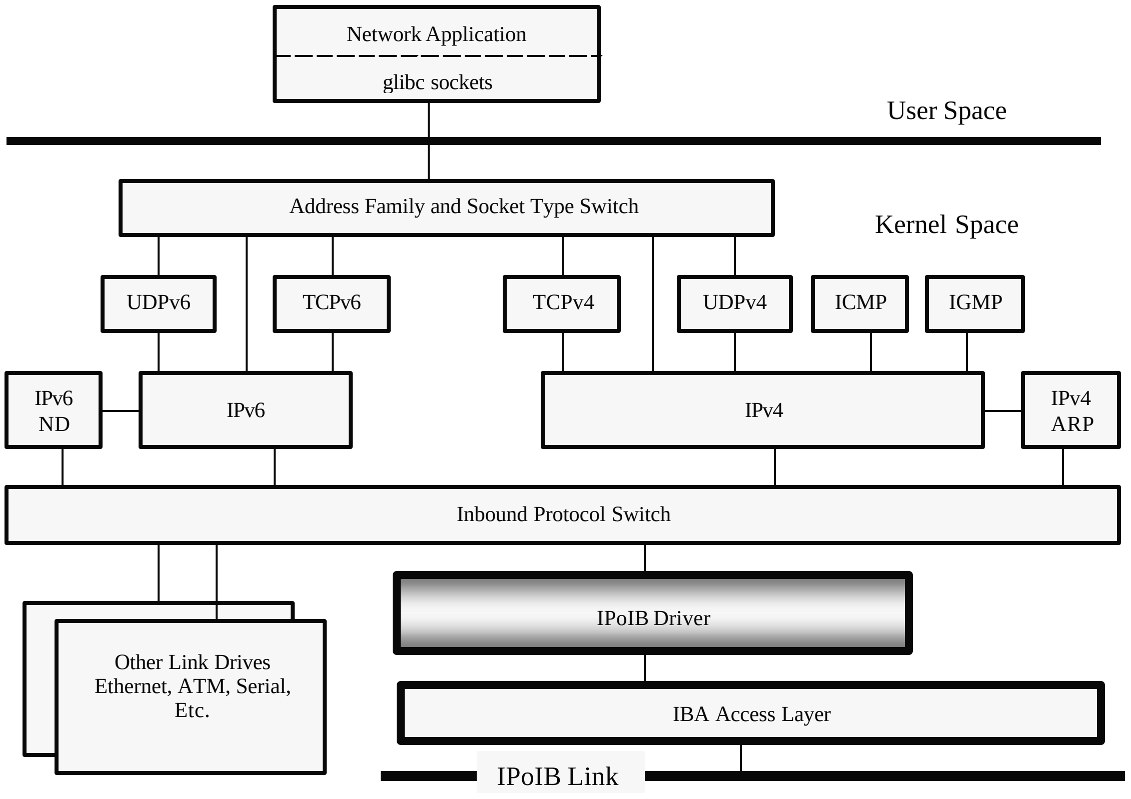


Figure 1-1. Linux Network Architecture

1.1 Data Flow

Outside of address resolution, the IPoIB driver provides a thin veneer between the IP protocol layers and the IBA Access Layer. The current version of the IPoIB draft defines a four-byte encapsulation header that contains the protocol type to identify the IPoIB user for incoming and outgoing frames. Therefore, normal data flow has very little IPoIB protocol-processing overhead.

For outbound data, the IPoIB driver must associate the *virtual hardware address* constructed through the Address Resolution Protocol (ARP) with the appropriate Unreliable Datagram (UD) address vector handle. In addition, it must either register the `sk_buff` data buffer or copy data buffer contents to registered memory before posting to the appropriate Queue Pair (QP).

NOTE

> The decision to register `sk_buffs` or copy data to/from them will be based on performance trade-offs. Since an IPoIB frame cannot exceed the UD MTU size (2K for Profile B), it may turn out that copying may always be faster than registering/un-registering memory.

For inbound data, the IPoIB driver must construct an `sk_buff` that contains the received data and has the protocol field set appropriately. The protocol type field in the encapsulation header is used to set the protocol field. The inbound protocol switch of the Linux network stack uses the protocol field to route the `sk_buff` to the appropriate protocol layer.

Unlike outbound data, inbound data will always be copied form pre-registered memory into an allocated `sk_buff`. In addition to the possible performance considerations mentioned above, there is currently no deterministic way for the IPoIB driver to know when the `sk_buff` has been retired.

1.2 IP over IB Endpoint Identification

As discussed above, the IP over IB protocol defines a virtual hardware address to convey endpoints within the fabric. This virtual hardware address is a 20 byte value composed of the port GID + UD QPN + a reserved 8 bits for future capabilities flags. Figure 1-2 diagrams the structure of this address format.

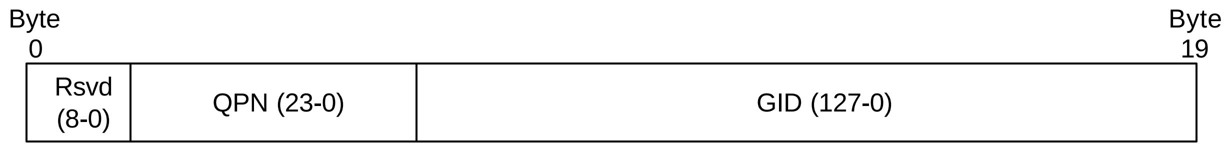


Figure 1-2. Virtual Hardware Address Format

Because an IPoIB device port (defined by the GID) may belong to more than one subnet (defined by the P_Key) or support more than one interface through a different QPN, the GID, QPN, and P_Key are all required to make an IPoIB endpoint unique. Although the P_Key is not a formal part of the IPoIB hardware address format, it is implied by the subnet associated with the network interface.

1.3 Address Resolution

Physical link addressing is unique to each type of interconnect (i.e. Ethernet, ATM, FDDI, etc.). Therefore, the IP protocol suite defines a physical address resolution process that is responsible for mapping IP addresses to physical address. For IPv4, this mapping is performed through the Address Resolution Protocol (ARP). The IPv6 protocol performs this mapping through a Neighbor Discovery (ND) protocol using ICMPv6. All of these resolution protocols require broadcast and multicast services. For IBA, this means interacting with the fabric Subnet Manager (SM) to create, join, leave, and delete multicast groups.

Due to the way IBA routes packets through the network, it is difficult for one node to tell another the complete address vector required for end-to-end communication. For this reason, the IPoIB draft defines a virtual link address detailed in Section 1.2. The sender is responsible for contacting the SM for complete addressing information to any particular end node based on the source and destination port GID. Therefore, full address resolution requires an additional step not found in other links.

The first step is the broadcast request followed by a unicast reply to exchange GID/QPN information. These steps are part of the standard ARP protocol. The next step is to contact the SM to obtain a PathRecord to the destination node. The extra step means that the ARP and ND implementation of the Linux network stack cannot be used unmodified. Although there is a relatively high overhead to this process, address caching employed by the Linux network stack and the IPoIB driver mitigates how often this process is used.

A further complication is the fixed and limited storage for a device hardware address in Linux networking structures. The standard Linux kernel defines `MAX_ADDR_LEN` to be 8 providing storage for an address up to 8 bytes long. The IPoIB virtual hardware address is 20 bytes. There are three ways of dealing with this situation:

1. Increase `MAX_ADDR_LEN` to be >= 20.
2. Map the GID+QPN virtual address to a new representation that fits within `MAX_ADDR_LEN`.
3. Change the Linux network architecture to support arbitrary length network hardware addresses.

Option 1 is the cleanest approach. This would allow the standard Linux ARP and ND modules to deal with IPoIB virtual address and produce IPoIB compliant ARP and ND frames without intervention from the IPoIB driver. The down side is that `MAX_ADDR_LEN` is a fundamental define used throughout the network stack. It will require a recompile of the Linux kernel and all associated network drivers. Third party software vendors with dependencies on any of the kernel structures using `MAX_ADDR_LEN` could not supply compatible binaries.

Option 2 could be implemented without affecting the Linux kernel or compatibility with third party binary vendors. The down side is that the IPoIB driver would need to rewrite/reformat inbound and outbound ARP and IPv6 ICMP ND frames to translate from the compressed address format we advertise to the kernel and the true virtual address representation required by IPoIB.

Option 3 is listed as the architecturally “correct” thing to do. However, this is a fundamental change that would affect many lines of code. Given the level of effort, both in development and lobbying, this approach will not be perused.

Given all of the above, the IPoIB design will take option 1 and assume `MAX_ADDR_LEN` is adjusted to accommodate the 20-byte IPoIB hardware address.

1.4 IP over IB Link Boundary

Both IPv4 ARP and IPv6 Neighbor Discovery protocols depend on the ability to broadcast frames to all nodes attached to a common link. Although the IPoIB draft could have defined the IPoIB link as an IBA subnet, they chose instead to allow an IPoIB link to span multiple IBA subnets. Therefore, IPoIB must specify a mechanism to establish IPoIB link boundaries. An IPoIB link boundary is defined as all nodes that are a member of the broadcast/all-nodes multicast group for a particular IBA partition. This means that a single CA port can be home to multiple IPoIB links. From an IP perspective, each link represents an IP subnet so a single CA port can be a member of multiple IP subnets.

To support his abstraction, the IPoIB driver will create a network device instance (represented by a Linux `net_device` structure) for each IPoIB link/subnet. Each IPoIB network device will associate one UD QP to the IPoIB link. Figure 1-3 illustrates the relationships between the Linux notion of a device, IPoIB QP, CA port, and IP subnet/IPoIB link.

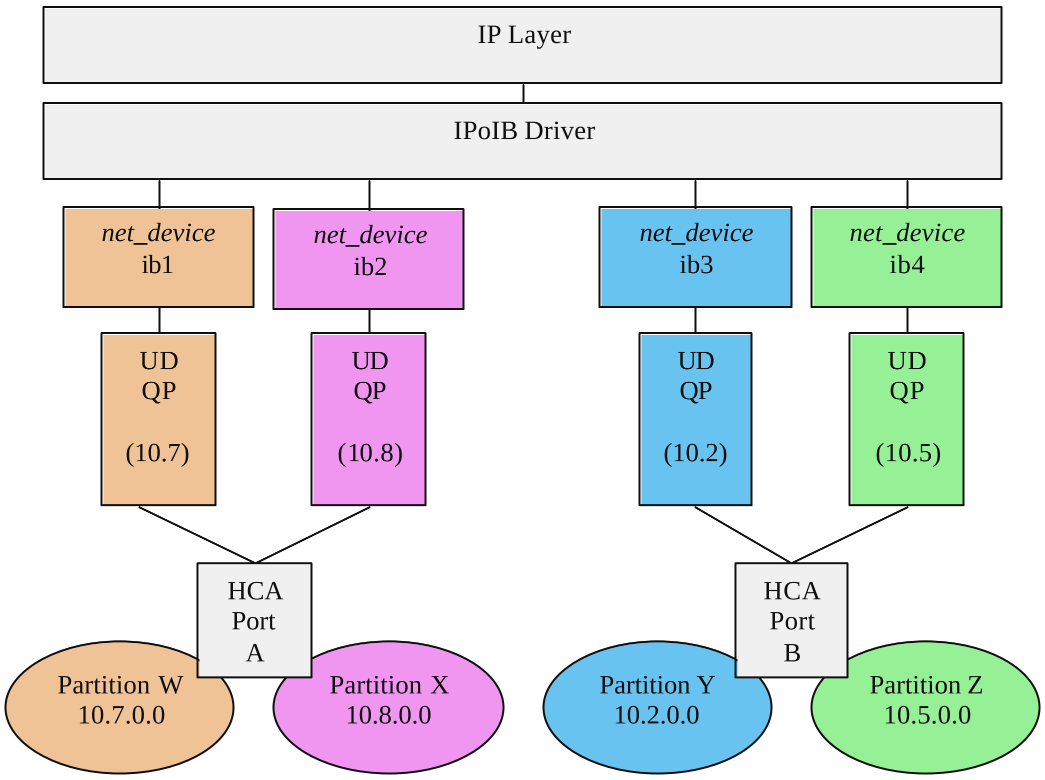


Figure 1-3. QP to IPoIB Link/Subnet Association

2. Design Details

As a component in a layered network architecture, the IPoIB driver has a service-user interface as well as a service-provider interface. The service-user interface it defined by the `net_device` and `sk_buff` structures of the Linux kernel. The service-provider for the IPoIB driver is the IBA Access Layer. The primary interface to the Access Layer is through a well-defined API that is directly callable from within the IPoIB driver. The following sections describe dataflow as it relates to these service interfaces.

2.1 Linux Network Device Interfaces

The two primary Linux network drives interfaces are `net_device` structure and the `sk_buff` structure. The `net_device` structure defines the network interface including all supported service routines. The `sk_buff` structure defines the attributes of a link level frame.

2.1.1 net_device structure

The `net_device` structure is defined in *include/linux/netdevice.h*. The IPoIB driver is responsible for creating and initializing this structure for each device associated with a network segment under its control. The initialized structure is registered through the Linux kernel call `register_netdev().` The IPoIB driver initializes the following fields:

`name`	This fixed length array provides the humane-readable name for the device. The IPoIB driver will assign a name to each device that makes it easy for the administrator to associate the name with the physical CA and port. The naming scheme is as follows: ibX_Y_Z where X is the hex value of the CA, Y is the hex value of the port, and Z is the hex partition key value for the network segment. So the following device name is for CA 0, port 1, partition 20: ib0_1_14
`type`	This field will be set to the constant `ARPHRD_IPOIB` defined in *include/linux/if_ipoib.h*. This value (32) has been assigned to the IPoIB working group by IANA.
`flags`	This field will be initialized with the flag combination `IFF_BROADCAST|IFF_MULTICAST`
`hard_header_len`	This field will be set to the constant `IPOIB_HLEN` defined in *include/linux/if_ipoib.h*. Although the IPoIB draft only defines a four-byte encapsulation header, this value will be set to 44 to provide room for a pseudo header that contains the source and destination IPoIB virtual hardware addresses.
`mtu`	This field indicates the MTU of the underlying link. The MTU value is determined when the device joins the all-nodes multicast group. However, it cannot exceed the unreliable datagram MTU of the underlying link.

`watchdog_timeo`	This field indicates the minimum time that should pass before transmit timeout is assumed. At that time, the Linux networking infrastructure will call the routine specified in the `tx_timeout` field. This value will be set from a drive static variable that can be set when the IPoIB module is loaded. The default is the constant `DEFAULT_TX_TIMEOUT` defined in *~/ipoib.h*.
`addr_len`	This field indicates the length of the hardware address for the device. This value cannot exceed the Linux kernel value `MAX_ADDR_LEN`. The IPoIB driver will use the constant `IPOIB_ALEN` defined in *include/linux/if_ipoib.h*.
`tx_queue_len`	This field specifies the maximum number of frames that can be queued on the device's transmission queue. The IPoIB driver will set this value from a driver static variable that can be set when the IPoIB module is loaded. The default is the constant `DEFAULT_SQ_DEPTH` defined in *~/ipoib.h*.
`dev_addr`	This fixed length array holds the hardware address for the device. The maximum length is set by the Linux kernel constant `MAX_ADDR_LEN`.
`broadcast`	This fixed length array of `MAX_ADDR_LEN` holds the broadcast/all-nodes address for the device. The array will always hold the pseudo broadcast address of all ones equal to the number of bytes specified in `addr_len` and will always be mapped by the IPoIB driver to the all-nodes multicast address for this link.
`private`	This void pointer is used to reference device driver-private data. The IPoIB driver will point this to an `ipoib_context_t` structure.

The following `net_device` entries are the supported IPoIB service entry points. Unless otherwise stated, they will be initialized with the function name identical to the field pre-pended with `ipoib_`

`init`	From a network device perspective, this entry point is used to initialize the associated net_device structure and to determine if the device is available for use. On return from the initialization function, the `net_device` structure is completely initialized. This is the only field that must be initialized before calling `register_netdev()`.
`open`	This function is called when the network interface is opened (i.e. ifconfig up). On return from the open function, the network device should be completely initialized and ready for operation. It will also increment the module usage count for loadable device drivers.
`stop`	This function is called when the interface has been stopped (i.e. ifconfig down). Any system resources allocated in open should be released.
`hard_start_xmit`	This function initiates the transmission of a complete protocol frame. If the IPoIB driver has not resolved the address vector handle for the destination, it will place the frame in a holding queue until the address handle has been created.
`hard_header`	This function is responsible for constructing a device specific link header. The IPoIB driver will create a pseudo header that will be mapped to the appropriate address vector handle in `ipoib_hard_start_xmit()`.

`rebuild_header`	This function is used to rebuild an existing link header within a `sk_buff` structure.
`tx_timeout`	This function will be called when a transmit fails to complete within the time specified in `watchdog_timeo`.
`get_status`	This function returns status information in a `net_device_status` structure. The IPoIB driver maintains an instance of the structure in the `ipoib_context_t` structure associated with each device.
`set_config`	This interface method is for link drivers that use I/O and DMA resources. This is not applicable for the IPoIB driver so this field will be set to `NULL`.
`do_ioctl`	This function provides support for device-specific ioctl commands. An example would be mapping an IPoIB ARP pseudo hardware address into a complete address vector.
`set_multicast_list`	This function is called when the multicast list for the device changes and when the flags associated with address filtering are changed. The IPoIB driver uses this as the signaling mechanism to join or leave an IPoIB multicast group.
`set_mac_address`	This function provides a method for changing the MAC address of a network device. Since this is not applicable to the IPoIB driver, this entry will be set to `NULL`.
`change_mtu`	This function provides a method for changing the MTU of a network device. Since this is not applicable to the IPoIB driver, this entry will be set to `NULL`.
`header_cache`	This function supports link header caching by initializing a `hh_cache` structure entry from the results of an ARP reply.
`header_cache_update`	This function provides a method for updating an `hh_cache` structure entry when addressing information changes.
`hard_header_parse`	This function copies the source hardware address from a `sk_buff`.

2.1.2 Inbound Data Flow

The IPoIB driver is notified of inbound network frames through a completion queue callback. Both send and receive callback funnel through a single completion queue serviced by `ipoib_interrupt()`. This routine determines if it is a send or receive event. For receive events, `ipoib_rx()` is called.

2.1.2.1 ipoib_rx ()

This function is responsible for creating an `sk_buff` containing the received data. (The reader is referred to section 2.4 for a discussion on buffer management.) As part of creating an `sk_buff`, this routine must set the following fields:

`dev`	Set to the `net_device` for this frame.
`protocol`	Set to the "ethertype" for this frame.

The device is derived from the `ipoib_context_t` pointer that is provided as Completion Queue (CQ) callback context. The ethertype is derived from IPoIB encapsulation header.

When the `sk_buff` has been completely processed by the IPoIB driver, it is posted to the Linux kernel networking infrastructure by calling Linux kernel function `netif_rx()`. At this point, the IPoIB driver is no longer responsible for the `sk_buff` and its contents.

2.1.3 Outbound Data Flow

When the Linux network stack has selected the appropriate link driver to send a frame out on, it calls several driver functions to prepare and send it. It calls the `ipoib_hard_header()` entry point to allow the driver to create a link header and then `ipoib_hard_start_xmit()` to send what should be a well formed frame ready for transmission. When the frame has been sent, the `ipoib_interrupt()` callback will be called and the send buffer placed back in the send buffer queue. (The reader is referred to section 2.4 for a complete discussion on buffer management.)

2.1.3.1 ipoib_hard_header()

This function allows the driver to append a link specific header to the frame. Although the draft IPoIB protocol only defines the encapsulation header as a four byte protocol type, some reference to the destination address needs to be associated with the frame at this time because source/destination addressing information is not presented at the `ipoib_hard_start_xmit()` interface. For this reason, the IPoIB driver will create a pseudo header containing the source and destination address passed to the function.

2.1.3.2 ipoib_hard_start_xmit()

This function is called to place a complete frame on the link. The IPoIB driver will de-reference the pseudo header it build in `ipoib_hard_header()` to extract the destination address token. Using the address token, the appropriate address vector handle is located. It is possible that that all the needed information to create an address vector handle is not currently available. For example, a PathRecord lookup may be in progress. When the handle is not available, the `sk_buff` is queued to a pending transmit queue associated with the address token. When enough information to create a handle becomes available, all `sk_buff` structures waiting on this information will be de-queued and transmitted at that time by calling `ipoib_send()`. Refer to Section 2.1.4 for a complete discussion on the shadow address resolution process. Refer to Section 2.2.2 for a complete discussion on `ipoib_send()` send completion processing.

2.1.4 Shadow ARP and Neighbor Discovery

Due to the way IBA frames are addressed, it is not practical to have complete address vector information returned in ARP and Neighbor Discovery responses. Instead, these responses return the target GID and QPN. The GID is used in a subsequent PathRecord lookup to return all the information to form an address vector and create an address vector handle needed to transmit frames over a UD QP. To deal with this extra PathRecord transaction, the IPoIB driver provides a “shadow” ARP/ND facility.

The shadow facility creates a list of `ipoib_addr_token_t` structures linked to the `ipoib_context_t` structure associated with appropriate `net_device` structure. New entries are added whenever the IPoIB driver needs to track hardware address properties of an address not currently in the list. These properties include the IPoIB hardware address, components of the address vector, and when address re-mapping is required, the address token. Although an entry will be created any time address information needs to be retrieved or stored, inbound ARP replies and Neighbor Advertisements will generally trigger the process.

As with the standard ARP cache, to prevent shadow entries from becoming stale they must be purged/refreshed from time to time. This could be done through a timer facility or by having the IPoIB driver look for outbound ARP requests and Neighbor Solicitations. Current thinking is to use the more deterministic outbound frame integration method. It is a little more straightforward to implement and will insure coherence with the Linux neighbor cache.

2.1.4.1 Shadow ARP State Machine

In all cases, an IPoIB hardware address is required to create an `ipoib_addr_token_t` entry. When the entry is created, it is marked with a PATHRECORD_PENDING state and a PathRecord lookup request is given to the Access Layer using the GID component of the IPoIB address. When the result of the PathRecord is successfully returned, all PathRecord information is stored with the entry, an address vector handle is created through the Access Layer, and the state changed to RESOLVED. If either the PathRecord lookup or address handle creation was not successful, the entry will be deleted and the error logged.

Frames that could not be sent because the state of the address token was not RESOVED are queued to the entry. When the entry transitions from PATHRECORD_PENDING to RESOLVED, all queued frames will be sent to the fabric at that time. In the event of a PathRecord lookup failure, all pending entries will be freed.

2.2 HCA Access Layer Interfaces

The IPoIB driver will interface to the HCA using the Access Layer APIs and service routines. This will insulate the driver from HCA vendor specifics.

2.2.1 Inbound Data Flow

The arrival of inbound data is signaled through the completion queue callback `ipoib_interrupt()`. This routine will interrogate the work completion elements to determine if send or receive events have completed.

For receive events, the GRH header is stripped, and `ipoib_rx()` function is called to construct an sk_buff and send the received data up through the Linux network stack. Refer to Section 2.1.2.1for a discussion on the `ipoib_rx()` function.

2.2.2 Outbound Data Flow

Outbound data is handled by the `ipoib_send()` function. The routine will determine the fastest method for transferring the data. The available methods are mapping the supplied buffer to the HCA or copying into a pre-allocated and registered buffer. At initialization, a pool of transmit buffers will be pre-allocated and registered for this purpose.

2.2.2.1 Send Completions

When the frame has been sent, the `ipoib_interrupt()` callback will be called. Depending on whether the original buffer of the sk_buff was mapped or if it was copied into a pre-allocated buffer, the send buffer is either placed back in the send buffer pool or unregistered from the HCA. (The reader is referred to section 2.4 for a discussion on buffer management.)

2.3 Locking and Threading Model

The IPoIB driver does not explicitly employ threading in the design. The transmit path is always run under the context of the Linux network stack and cannot block. The receive path is run under the context of the Access Layer callback for completion queue events. Finally, PathRecord lookups are run under the context of the Access Layer callback for completed SM events.

It is assumed that all thread contexts running in the IPoIB driver cannot be blocked. For this reason, non-preemptive `cl_spinlock_t` locks protect all linked lists. Due to the number of lists involved and to provide scaling in a multiprocessor environment, the IPoIB driver employs reference counting in conjunction with fine-grained locking. Any list item with a non-zero reference count cannot be deleted. Inversely, any list item that transitions to a reference count of zero will be removed from its associated list and deleted.

Due to the non-preemptive nature of spin locks, they will only be taken long enough to adjust reference counts and/or to add and remove list items.

2.4 Buffer Strategy

The Linux network buffer management strategy transfers ownership of the `sk_buff` with the buffer. It is not retained by the entity that initially allocated it. This approach means that the IPoIB driver cannot recycle permanently registered receive `sk_buff` buffers. Instead, it must choose between allocating, registering, posting, and un-registering an `sk_buff` or using permanently registered memory and copying received data into an allocated `sk_buff`. The approach taken will depend on the registration overhead vs. the overhead of a memory copy.

2.5 Driver Initialization

The IPoIB driver is implemented as a loadable kernel module. It can be loaded through the PnP Manager facility of the Access Layer or explicitly by an administrator using the insmod(8) command. Runtime configuration can be done through the common configuration file method or by command line arguments to insmod(8) using the *symbol=value* method. Command line arguments will always take precedence over configuration file entries.

On module entry, the driver will perform the following steps:

1. Open an interface to the Access Layer by calling `ib_open_al()`.
2. Request local CA notifications by calling `ib_reg_notify()` specifying both CA and port events. All events will be process by a common callback routine.

At this point, the remaining initialization will be driven by CA and port callback events.

2.6 Network Interface Initialization

The IPoIB driver creates network interfaces dynamically as it becomes aware of IBA partitions that have been configured as IPoIB subnets. This process is event driven by Access Layer callbacks registered during driver initialization.

When the notification indicates a CA has been added, a new `ipoib_ca_t` structure is allocated and added to the list of existing CAs. Likewise, `ipoib_port_t` structures are allocated and added to the list of existing ports associated with a CA when a port-up notification is received.

When the driver receives a P_Key change notification and determines that a P_Key has been added, it will attempt to initialize a network interface by performing the following steps:

1. Allocate a `ipoib_context_t` structure to be associated with the interface
2. Create a completion queue for sends and receives by calling `ib_create_cq()`.
3. Create a QP for this potentialinterface by calling `ib_create_qp()` and associate it with the completion queue for the port.
4. Initialize the QP for Unreliable Datagram service by calling `ib_init_dgrm_svc()`.
5. Allocate memory the `net_device` structure
6. Allocate and register send and receive buffers.
7. Attempt to join the broadcast group by calling `ib_join_mcast()`. Perform the following steps on a successful callback:
    1. Register the `net_device` structure with the Linux network infrastructure by calling `register_netdev().`
    2. Post receive buffers.

At this point, an IPoIB network interface is available for configuration by Linux network initialization scripts or by the system administrator. The relationship of the IPoIB driver data structures is documented in Section 3.1.

NOTE

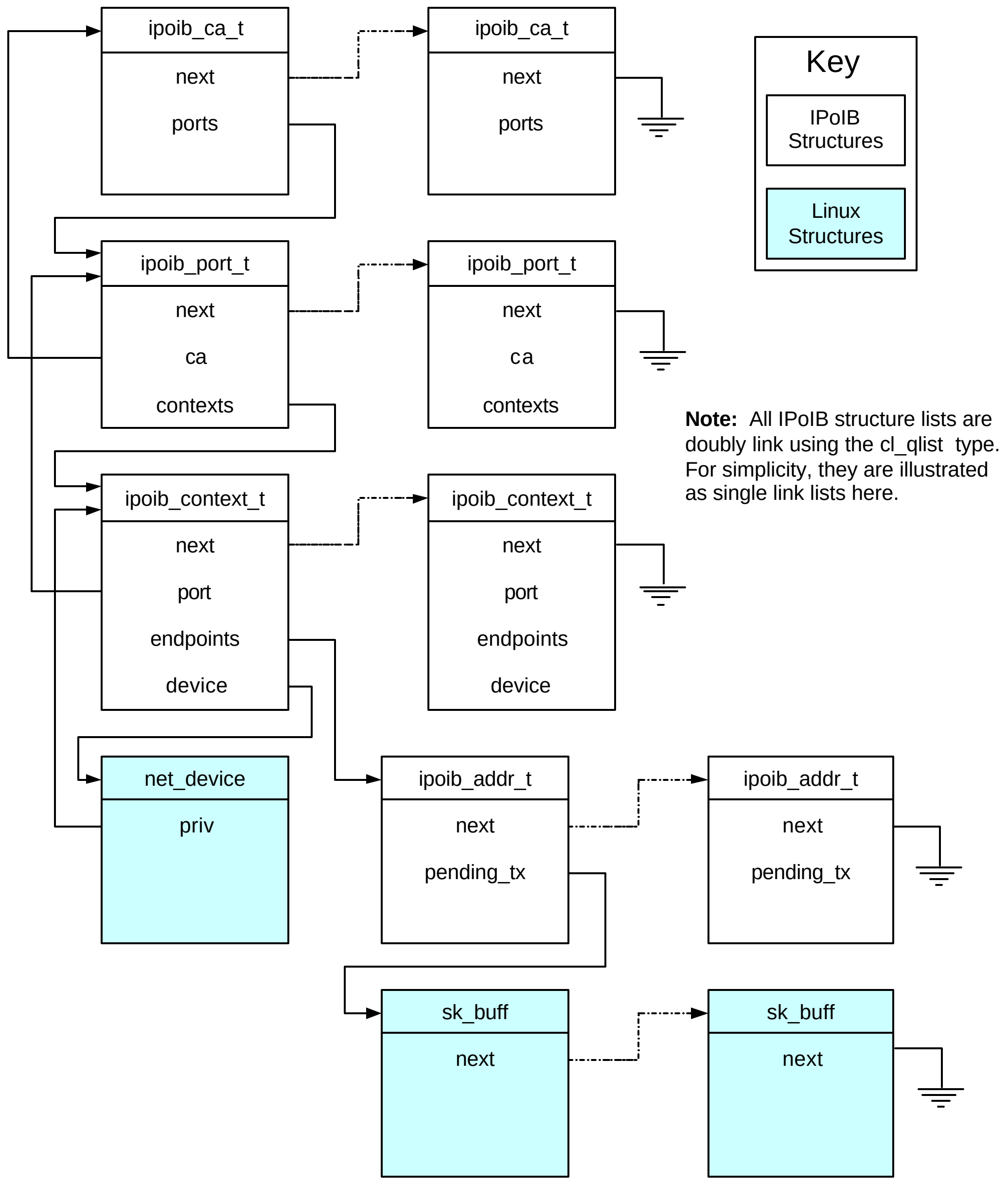
The method for discovering IPoIB links described above assumes that link broadcast groups exist at the time a port is assigned an IPoIB link related P_Key. It also assumes that joining an IBA multicast group is a separate operation from creating one. As of this writing, there are changes being considered by the IBA that would make joining a non-existent multicast group an implicit group create. This would render the dynamic link discovery method outlined above inoperable. Therefore, the implementation will also contain conditional code that will compare newly assigned P_Keys to a list of configured keys to determine if a multicast join operation should be attempted.

3. Data Structures and APIs

All internal data structures are defined in ~/ipoib.h. All public data structures are defined in <linux/if_ipoib.h>.

3.1 Structure Relationships

The following diagram provides a schematic of the structures use in the IPoIB driver.



3.2 APIs

The IPoIB driver provides a user mode and kernel mode interface for returning PathRecord information associated with an IP address. This interface is necessary to implement the Sockets Direct Protocol (SDP) and Direct Access Provider Library (DAPL) as well as provide administrative diagnostic information.

3.3 struct ipoib_path_info

Both kernel and user-mode APIs use the `ipoib_path_info` structure. This structure and `SIOCGIPOIBPATH` are defined in <linux/if_ipoib.h>. On input, the `sin_family` and associated `addr` fields for the target of the query must be initialized. All PathRecord information comes directly from the associated MAD and is in network order along. IP addresses are specified in network order while the `sin_family` is specified in host order.

```
#define SIOCGIPOIBPATH  (SIOCDEVPRIVATE + 0)
struct ipoib_path_info {
      sa_family_t       sin_family;       /* address family */
      union {
            uint8_t     v4_addr[4];       /* ipv4 destination address */
            uint8_t     v6_addr[16];      /* ipv6 destination address */
      } addr;
      uint8_t           dest_gid[16];     /* destination port GID */
      uint8_t           source_gid[16];   /* source port GID */
      uint16_t          dest_lid;         /* destination LID */
      uint16_t          source_lid;       /* source LID */
      uint32_t          flow_lable;       /* flow label */
      uint16_t          pkey;             /* partition key */
      uint16_t          service_level;    /* service level */
      uint16_t          mtu;              /* path MTU selector*/
      uint16_t          static_rate;      /* static rate selector */
      uint16_t          packet_lifetime;  /* packet lifetime */
      uint8_t           hop_limit;        /* hop limit */
      uint8_t           tclass;           /* traffic class */
};
```

3.3.1 Kernel-Mode API

The IPoIB driver will export the following function call:

```
int
ipoib_get_path( IN OUT struct ipoib_path_info *req,
                IN     void *context,
                IN     void (*callback)(IN int status,
                                        IN void *context,
                                        IN struct ipoib_path_info*) );
```

Where 0 is returned if the request was satisfied and -EAGAIN is returned if information is not currently available. If -EAGAIN is returned and the callback argument is non-null, the callback will be invoked when the outcome of the request is known. The status argument of the callback will be 0 on success and

-ENXIO if a node with the specified address could not be found. The context value of the callback will be the same as passed to the ipoib_get_path() call. When successfully completed, the `ipoib_path_info` structure will be updated with valid information.

3.3.2 User-Mode API

The IPoIB driver will implement the following ioctl:

```
int
ioctl( sock, SIOCGIPOIBPATH, struct ipoib_path_info *req );
```

This is a blocking call until the outcome is known. A return value of 0 indicates the path information was resolved. ENXIO is returned if a node with the specified address could not be found. If the socket is marked non-blocking and the information is not known at the time of the call, EAGAIN will be returned.

4. Installing, Configuring, and Uninstalling

4.1 Installing

The IPoIB driver is installed as a standard Linux loadable module. It is can be loaded into the kernel explicitly using the insmod(8) command or automatically us ing the PnP facility of the Access Layer.

4.2 Configuring

The following configuration variables can be specified on the insmod(8) command line or in the Access Layer PnP configuration file:

sq_depth	Send queue depth
rq_depth	Receive queue depth
mcast_timeout	Multicast join request timeout in milliseconds
mcast_retry_cnt	Multicast join request retries
timeout	Transmit timeout in jiffies
debug	Set debug message level

4.3 Uninstalling

The IPoIB driver is uninstalled as a standard Linux loadable module using the rmmod(8) command. All IPoIB network devices will be brought down and all resources will be released.