Linux Infiniband Documentation

Release 6.8.0

The kernel development community

Jan 16, 2026

CONTENTS

1 InfiniBand Midlayer Locking 1

2 IP over InfiniBand 5

3 Intel Omni-Path (OPA) Virtual Network Interface Controller (VNIC) 9

4 Sysfs files 13

5 Tag matching logic 15

6 Userspace MAD access 17

7 Userspace verbs access 21

CHAPTER ONE

INFINIBAND MIDLAYER LOCKING

This guide is an attempt to make explicit the locking assumptions made by the InfiniBand midlayer. It describes the requirements on both low-level drivers that sit below the midlayer and upper level protocols that use the midlayer.

1.1 Sleeping and interrupt context

With the following exceptions, a low-level driver implementation of all of the methods in struct ib_device may sleep. The exceptions are any methods from the list:

- create_ah
- modify_ah
- query_ah
- destroy_ah
- post_send
- post_recv
- poll_cq
- req_notify_cq

which may not sleep and must be callable from any context.

The corresponding functions exported to upper level protocol consumers:

- rdma_create_ah
- rdma_modify_ah
- rdma_query_ah
- rdma_destroy_ah
- ib_post_send
- ib_post_recv
- ib_req_notify_cq

are therefore safe to call from any context.

In addition, the function

- ib_dispatch_event

used by low-level drivers to dispatch asynchronous events through the midlayer is also safe to call from any context.

1.1.1 Reentrancy

All of the methods in struct ib_device exported by a low-level driver must be fully reentrant. The low-level driver is required to perform all synchronization necessary to maintain consistency, even if multiple function calls using the same object are run simultaneously.

The IB midlayer does not perform any serialization of function calls.

Because low-level drivers are reentrant, upper level protocol consumers are not required to perform any serialization. However, some serialization may be required to get sensible results. For example, a consumer may safely call ib_poll_cq() on multiple CPUs simultaneously. However, the ordering of the work completion information between different calls of ib_poll_cq() is not defined.

1.1.2 Callbacks

A low-level driver must not perform a callback directly from the same callchain as an ib_device method call. For example, it is not allowed for a low-level driver to call a consumer's completion event handler directly from its post_send method. Instead, the low-level driver should defer this callback by, for example, scheduling a tasklet to perform the callback.

The low-level driver is responsible for ensuring that multiple completion event handlers for the same CQ are not called simultaneously. The driver must guarantee that only one CQ event handler for a given CQ is running at a time. In other words, the following situation is not allowed:

```
      CPU1                                    CPU2

low-level driver ->
  consumer CQ event callback:
    /* ... */
    ib_req_notify_cq(cq, ...);
                                      low-level driver ->
    /* ... */                           consumer CQ event callback:
                                          /* ... */
    return from CQ event handler
```

The context in which completion event and asynchronous event callbacks run is not defined. Depending on the low-level driver, it may be process context, softirq context, or interrupt context. Upper level protocol consumers may not sleep in a callback.

1.1.3 Hot-plug

A low-level driver announces that a device is ready for use by consumers when it calls ib_register_device(), all initialization must be complete before this call. The device must remain usable until the driver's call to ib_unregister_device() has returned.

A low-level driver must call ib_register_device() and ib_unregister_device() from process context. It must not hold any semaphores that could cause deadlock if a consumer calls back into the driver across these calls.

An upper level protocol consumer may begin using an IB device as soon as the add method of its struct ib_client is called for that device. A consumer must finish all cleanup and free all resources relating to a device before returning from the remove method.

A consumer is permitted to sleep in its add and remove methods.

CHAPTER

TWO

IP OVER INFINIBAND

The ib_ipoib driver is an implementation of the IP over InfiniBand protocol as specified by RFC 4391 and 4392, issued by the IETF ipoib working group. It is a ”native” implementation in the sense of setting the interface type to ARPHRD_INFINIBAND and the hardware address length to 20 (earlier proprietary implementations masqueraded to the kernel as ethernet interfaces).

2.1 Partitions and P_Keys

When the IPoIB driver is loaded, it creates one interface for each port using the P_Key at index 0. To create an interface with a different P_Key, write the desired P_Key into the main interface's /sys/class/net/<intf name>/create_child file. For example:

```
echo 0x8001 > /sys/class/net/ib0/create_child
```

This will create an interface named ib0.8001 with P_Key 0x8001. To remove a subinterface, use the ”delete_child” file:

```
echo 0x8001 > /sys/class/net/ib0/delete_child
```

The P_Key for any interface is given by the ”pkey” file, and the main interface for a subinterface is in ”parent.”

Child interface create/delete can also be done using IPoIB's rtnl_link_ops, where children created using either way behave the same.

2.2 Datagram vs Connected modes

The IPoIB driver supports two modes of operation: datagram and connected. The mode is set and read through an interface's /sys/class/net/<intf name>/mode file.

In datagram mode, the IB UD (Unreliable Datagram) transport is used and so the interface MTU has is equal to the IB L2 MTU minus the IPoIB encapsulation header (4 bytes). For example, in a typical IB fabric with a 2K MTU, the IPoIB MTU will be 2048 - 4 = 2044 bytes.

In connected mode, the IB RC (Reliable Connected) transport is used. Connected mode takes advantage of the connected nature of the IB transport and allows an

MTU up to the maximal IP packet size of 64K, which reduces the number of IP packets needed for handling large UDP datagrams, TCP segments, etc and increases the performance for large messages.

In connected mode, the interface's UD QP is still used for multicast and communication with peers that don't support connected mode. In this case, RX emulation of ICMP PMTU packets is used to cause the networking stack to use the smaller UD MTU for these neighbours.

2.3 Stateless offloads

If the IB HW supports IPoIB stateless offloads, IPoIB advertises TCP/IP checksum and/or Large Send (LSO) offloading capability to the network stack.

Large Receive (LRO) offloading is also implemented and may be turned on/off using ethtool calls. Currently LRO is supported only for checksum offload capable devices.

Stateless offloads are supported only in datagram mode.

2.4 Interrupt moderation

If the underlying IB device supports CQ event moderation, one can use ethtool to set interrupt mitigation parameters and thus reduce the overhead incurred by handling interrupts. The main code path of IPoIB doesn't use events for TX completion signaling so only RX moderation is supported.

2.5 Debugging Information

By compiling the IPoIB driver with CONFIG_INFINIBAND_IPOIB_DEBUG set to 'y', tracing messages are compiled into the driver. They are turned on by setting the module parameters debug_level and mcast_debug_level to 1. These parameters can be controlled at runtime through files in /sys/module/ib_ipoib/.

CONFIG_INFINIBAND_IPOIB_DEBUG also enables files in the debugfs virtual filesystem. By mounting this filesystem, for example with:

```
mount -t debugfs none /sys/kernel/debug
```

it is possible to get statistics about multicast groups from the files /sys/kernel/debug/ipoib/ib0_mcg and so on.

The performance impact of this option is negligible, so it is safe to enable this option with debug_level set to 0 for normal operation.

CONFIG_INFINIBAND_IPOIB_DEBUG_DATA enables even more debug output in the data path when data_debug_level is set to 1. However, even with the output disabled, enabling this configuration option will affect performance, because it adds tests to the fast path.

2.6 References

Transmission of IP over InfiniBand (IPoIB) (RFC 4391)
http://ietf.org/rfc/rfc4391.txt

IP over InfiniBand (IPoIB) Architecture (RFC 4392)
http://ietf.org/rfc/rfc4392.txt

IP over InfiniBand: Connected Mode (RFC 4755)
http://ietf.org/rfc/rfc4755.txt

CHAPTER

THREE

INTEL OMNI-PATH (OPA) VIRTUAL NETWORK INTERFACE CONTROLLER (VNIC)

Intel Omni-Path (OPA) Virtual Network Interface Controller (VNIC) feature supports Ethernet functionality over Omni-Path fabric by encapsulating the Ethernet packets between HFI nodes.

3.1 Architecture

The patterns of exchanges of Omni-Path encapsulated Ethernet packets involves one or more virtual Ethernet switches overlaid on the Omni-Path fabric topology. A subset of HFI nodes on the Omni-Path fabric are permitted to exchange encapsulated Ethernet packets across a particular virtual Ethernet switch. The virtual Ethernet switches are logical abstractions achieved by configuring the HFI nodes on the fabric for header generation and processing. In the simplest configuration all HFI nodes across the fabric exchange encapsulated Ethernet packets over a single virtual Ethernet switch. A virtual Ethernet switch, is effectively an independent Ethernet network. The configuration is performed by an Ethernet Manager (EM) which is part of the trusted Fabric Manager (FM) application. HFI nodes can have multiple VNICs each connected to a different virtual Ethernet switch. The below diagram presents a case of two virtual Ethernet switches with two HFI nodes:

```
                             +-------------------+
                             |      Subnet/      |
                             |     Ethernet      |
                             |      Manager      |
                             +-------------------+
                                /          /
                              /           /
                            /            /
                          /             /
+-----------------------------+  +------------------------------+
|  Virtual Ethernet Switch    |  |  Virtual Ethernet Switch     |
|  +---------+    +---------+ |  | +---------+    +---------+   |
|  | VPORT   |    |  VPORT  | |  | |  VPORT  |    |  VPORT  |   |
+--+---------+----+---------+-+  +-+---------+----+---------+---+
         |                 \        /                 |
         |                   \    /                   |
         |                     \/                     |
         |                    /  \                    |
         |                  /      \                  |
     +-----------+------------+  +-----------+------------+
     |   VNIC    |    VNIC    |  |    VNIC   |    VNIC    |
     +-----------+------------+  +-----------+------------+
     |          HFI           |  |          HFI           |
     +------------------------+  +------------------------+
```

The Omni-Path encapsulated Ethernet packet format is as described below.

Bits	Field
Quad Word 0:	
0-19	SLID (lower 20 bits)
20-30	Length (in Quad Words)
31	BECN bit
32-51	DLID (lower 20 bits)
52-56	SC (Service Class)
57-59	RC (Routing Control)
60	FECN bit
61-62	L2 (=10, 16B format)
63	LT (=1, Link Transfer Head Flit)
Quad Word 1:	
0-7	L4 type (=0x78 ETHERNET)
8-11	SLID[23:20]
12-15	DLID[23:20]
16-31	PKEY
32-47	Entropy
48-63	Reserved
Quad Word 2:	
0-15	Reserved
16-31	L4 header
32-63	Ethernet Packet
Quad Words 3 to N-1:	
0-63	Ethernet packet (pad extended)
Quad Word N (last):	
0-23	Ethernet packet (pad extended)
24-55	ICRC
56-61	Tail
62-63	LT (=01, Link Transfer Tail Flit)

Ethernet packet is padded on the transmit side to ensure that the VNIC OPA packet is quad word aligned. The 'Tail' field contains the number of bytes padded. On the receive side the 'Tail' field is read and the padding is removed (along with ICRC, Tail and OPA header) before passing packet up the network stack.

The L4 header field contains the virtual Ethernet switch id the VNIC port belongs to. On the receive side, this field is used to de-multiplex the received VNIC packets to different VNIC ports.

3.2 Driver Design

Intel OPA VNIC software design is presented in the below diagram. OPA VNIC functionality has a HW dependent component and a HW independent component.

The support has been added for IB device to allocate and free the RDMA netdev devices. The RDMA netdev supports interfacing with the network stack thus creating standard network interfaces. OPA_VNIC is an RDMA netdev device type.

The HW dependent VNIC functionality is part of the HFI1 driver. It implements the verbs to allocate and free the OPA_VNIC RDMA netdev. It involves HW resource allocation/management for VNIC functionality. It interfaces with the network stack and implements the required net_device_ops functions. It expects Omni-Path encapsulated Ethernet packets in the transmit path and provides HW access to them. It strips the Omni-Path header from the received packets before passing them up the network stack. It also implements the RDMA netdev control operations.

The OPA VNIC module implements the HW independent VNIC functionality. It consists of two parts. The VNIC Ethernet Management Agent (VEMA) registers itself with IB core as an IB client and interfaces with the IB MAD stack. It exchanges the management information with the Ethernet Manager (EM) and the VNIC netdev. The VNIC netdev part allocates and frees the OPA_VNIC RDMA netdev devices. It overrides the net_device_ops functions set by HW dependent VNIC driver where required to accommodate any control operation. It also handles the encapsulation of Ethernet packets with an Omni-Path header in the transmit path. For each VNIC interface, the information required for encapsulation is configured by the EM via VEMA MAD interface. It also passes any control information to the HW dependent driver by invoking the RDMA netdev control operations:

```
+-------------------+ +----------------------+
|                   | |       Linux          |
|     IB MAD        | |      Network         |
|                   | |       Stack          |
+-------------------+ +----------------------+
         |               |          |
         |               |          |
+----------------------------+      |
|                            |      |
|      OPA VNIC Module       |      |
|  (OPA VNIC RDMA Netdev     |      |
|     & EMA functions)       |      |
|                            |      |
+----------------------------+      |
            |                       |
            |                       |
   +------------------+             |
   |     IB core      |             |
   +------------------+             |
            |                       |
            |                       |
+--------------------------------------------+
|                                            |
|      HFI1 Driver with VNIC support         |
|                                            |
+--------------------------------------------+
```

CHAPTER FOUR

SYSFS FILES

The sysfs interface has moved to Documentation/ABI/stable/sysfs-class-infiniband.

CHAPTER

FIVE

TAG MATCHING LOGIC

The MPI standard defines a set of rules, known as tag-matching, for matching source send operations to destination receives. The following parameters must match the following source and destination parameters:

- Communicator
- User tag - wild card may be specified by the receiver
- Source rank – wild car may be specified by the receiver
- Destination rank – wild

The ordering rules require that when more than one pair of send and receive message envelopes may match, the pair that includes the earliest posted-send and the earliest posted-receive is the pair that must be used to satisfy the matching operation. However, this doesn’t imply that tags are consumed in the order they are created, e.g., a later generated tag may be consumed, if earlier tags can’t be used to satisfy the matching rules.

When a message is sent from the sender to the receiver, the communication library may attempt to process the operation either after or before the corresponding matching receive is posted. If a matching receive is posted, this is an expected message, otherwise it is called an unexpected message. Implementations frequently use different matching schemes for these two different matching instances.

To keep MPI library memory footprint down, MPI implementations typically use two different protocols for this purpose:

1. The Eager protocol- the complete message is sent when the send is processed by the sender. A completion send is received in the send_cq notifying that the buffer can be reused.

2. The Rendezvous Protocol - the sender sends the tag-matching header, and perhaps a portion of data when first notifying the receiver. When the corresponding buffer is posted, the responder will use the information from the header to initiate an RDMA READ operation directly to the matching buffer. A fin message needs to be received in order for the buffer to be reused.

5.1 Tag matching implementation

There are two types of matching objects used, the posted receive list and the unexpected message list. The application posts receive buffers through calls to the MPI receive routines in the posted receive list and posts send messages using the MPI send routines. The head of the posted receive list may be maintained by the hardware, with the software expected to shadow this list.

When send is initiated and arrives at the receive side, if there is no pre-posted receive for this arriving message, it is passed to the software and placed in the unexpected message list. Otherwise the match is processed, including rendezvous processing, if appropriate, delivering the data to the specified receive buffer. This allows overlapping receive-side MPI tag matching with computation.

When a receive-message is posted, the communication library will first check the software unexpected message list for a matching receive. If a match is found, data is delivered to the user buffer, using a software controlled protocol. The UCX implementation uses either an eager or rendezvous protocol, depending on data size. If no match is found, the entire pre-posted receive list is maintained by the hardware, and there is space to add one more pre-posted receive to this list, this receive is passed to the hardware. Software is expected to shadow this list, to help with processing MPI cancel operations. In addition, because hardware and software are not expected to be tightly synchronized with respect to the tag-matching operation, this shadow list is used to detect the case that a pre-posted receive is passed to the hardware, as the matching unexpected message is being passed from the hardware to the software.

CHAPTER

SIX

USERSPACE MAD ACCESS

6.1 Device files

Each port of each InfiniBand device has a ”umad” device and an ”issm” device attached. For example, a two-port HCA will have two umad devices and two issm devices, while a switch will have one device of each type (for switch port 0).

6.2 Creating MAD agents

A MAD agent can be created by filling in a struct ib_user_mad_reg_req and then calling the IB_USER_MAD_REGISTER_AGENT ioctl on a file descriptor for the appropriate device file. If the registration request succeeds, a 32-bit id will be returned in the structure. For example:

```
struct ib_user_mad_reg_req req = { /* ... */ };
ret = ioctl(fd, IB_USER_MAD_REGISTER_AGENT, (char *) &req);
if (!ret)
        my_agent = req.id;
else
        perror("agent register");
```

Agents can be unregistered with the IB_USER_MAD_UNREGISTER_AGENT ioctl. Also, all agents registered through a file descriptor will be unregistered when the descriptor is closed.

2014

a new registration ioctl is now provided which allows additional fields to be provided during registration. Users of this registration call are implicitly setting the use of pkey_index (see below).

6.3 Receiving MADs

MADs are received using read(). The receive side now supports RMPP. The buffer passed to read() must be at least one struct ib_user_mad + 256 bytes. For example:

If the buffer passed is not large enough to hold the received MAD (RMPP), the errno is set to ENOSPC and the length of the buffer needed is set in mad.length.

Example for normal MAD (non RMPP) reads:

```
struct ib_user_mad *mad;
mad = malloc(sizeof *mad + 256);
ret = read(fd, mad, sizeof *mad + 256);
if (ret != sizeof mad + 256) {
        perror("read");
        free(mad);
}
```

Example for RMPP reads:

```
struct ib_user_mad *mad;
mad = malloc(sizeof *mad + 256);
ret = read(fd, mad, sizeof *mad + 256);
if (ret == -ENOSPC)) {
        length = mad.length;
        free(mad);
        mad = malloc(sizeof *mad + length);
        ret = read(fd, mad, sizeof *mad + length);
}
if (ret < 0) {
        perror("read");
        free(mad);
}
```

In addition to the actual MAD contents, the other struct ib_user_mad fields will be filled in with information on the received MAD. For example, the remote LID will be in mad.lid.

If a send times out, a receive will be generated with mad.status set to ETIMEDOUT. Otherwise when a MAD has been successfully received, mad.status will be 0.

poll()/select() may be used to wait until a MAD can be read.

6.4 Sending MADs

MADs are sent using write(). The agent ID for sending should be filled into the id field of the MAD, the destination LID should be filled into the lid field, and so on. The send side does support RMPP so arbitrary length MAD can be sent. For example:

```
struct ib_user_mad *mad;

mad = malloc(sizeof *mad + mad_length);
```

```
/* fill in mad->data */

mad->hdr.id  = my_agent;        /* req.id from agent registration */
mad->hdr.lid = my_dest;         /* in network byte order... */
/* etc. */

ret = write(fd, &mad, sizeof *mad + mad_length);
if (ret != sizeof *mad + mad_length)
        perror("write");
```

6.5 Transaction IDs

Users of the umad devices can use the lower 32 bits of the transaction ID field (that is, the least significant half of the field in network byte order) in MADs being sent to match request/response pairs. The upper 32 bits are reserved for use by the kernel and will be overwritten before a MAD is sent.

6.6 P_Key Index Handling

The old ib_umad interface did not allow setting the P_Key index for MADs that are sent and did not provide a way for obtaining the P_Key index of received MADs. A new layout for struct ib_user_mad_hdr with a pkey_index member has been defined; however, to preserve binary compatibility with older applications, this new layout will not be used unless one of IB_USER_MAD_ENABLE_PKEY or IB_USER_MAD_REGISTER_AGENT2 ioctl's are called before a file descriptor is used for anything else.

In September 2008, the IB_USER_MAD_ABI_VERSION will be incremented to 6, the new layout of struct ib_user_mad_hdr will be used by default, and the IB_USER_MAD_ENABLE_PKEY ioctl will be removed.

6.7 Setting IsSM Capability Bit

To set the IsSM capability bit for a port, simply open the corresponding issm device file. If the IsSM bit is already set, then the open call will block until the bit is cleared (or return immediately with errno set to EAGAIN if the O_NONBLOCK flag is passed to open()). The IsSM bit will be cleared when the issm file is closed. No read, write or other operations can be performed on the issm file.

6.8 /dev files

To create the appropriate character device files automatically with udev, a rule like:

```
KERNEL=="umad*", NAME="infiniband/%k"
KERNEL=="issm*", NAME="infiniband/%k"
```

can be used. This will create device nodes named:

```
/dev/infiniband/umad0
/dev/infiniband/issm0
```

for the first port, and so on. The InfiniBand device and port associated with these devices can be determined from the files:

```
/sys/class/infiniband_mad/umad0/ibdev
/sys/class/infiniband_mad/umad0/port
```

and:

```
/sys/class/infiniband_mad/issm0/ibdev
/sys/class/infiniband_mad/issm0/port
```

CHAPTER

SEVEN

USERSPACE VERBS ACCESS

The ib_uverbs module, built by enabling CONFIG_INFINIBAND_USER_VERBS, enables direct userspace access to IB hardware via ”verbs,” as described in chapter 11 of the InfiniBand Architecture Specification.

To use the verbs, the libibverbs library, available from https://github.com/linux-rdma/rdma-core, is required. libibverbs contains a device-independent API for using the ib_uverbs interface. libibverbs also requires appropriate device-dependent kernel and userspace driver for your InfiniBand hardware. For example, to use a Mellanox HCA, you will need the ib_mthca kernel module and the libmthca userspace driver be installed.

7.1 User-kernel communication

Userspace communicates with the kernel for slow path, resource management operations via the /dev/infiniband/uverbsN character devices. Fast path operations are typically performed by writing directly to hardware registers mmap()ed into userspace, with no system call or context switch into the kernel.

Commands are sent to the kernel via write()s on these device files. The ABI is defined in drivers/infiniband/include/ib_user_verbs.h. The structs for commands that require a response from the kernel contain a 64-bit field used to pass a pointer to an output buffer. Status is returned to userspace as the return value of the write() system call.

7.2 Resource management

Since creation and destruction of all IB resources is done by commands passed through a file descriptor, the kernel can keep track of which resources are attached to a given userspace context. The ib_uverbs module maintains idr tables that are used to translate between kernel pointers and opaque userspace handles, so that kernel pointers are never exposed to userspace and userspace cannot trick the kernel into following a bogus pointer.

This also allows the kernel to clean up when a process exits and prevent one process from touching another process's resources.

7.3 Memory pinning

Direct userspace I/O requires that memory regions that are potential I/O targets be kept resident at the same physical address. The ib_uverbs module manages pinning and unpinning memory regions via get_user_pages() and put_page() calls. It also accounts for the amount of memory pinned in the process's pinned_vm, and checks that unprivileged processes do not exceed their RLIMIT_MEMLOCK limit.

Pages that are pinned multiple times are counted each time they are pinned, so the value of pinned_vm may be an overestimate of the number of pages pinned by a process.

7.4 /dev files

To create the appropriate character device files automatically with udev, a rule like:

```
KERNEL=="uverbs*", NAME="infiniband/%k"
```

can be used. This will create device nodes named:

```
/dev/infiniband/uverbs0
```

and so on. Since the InfiniBand userspace verbs should be safe for use by non-privileged processes, it may be useful to add an appropriate MODE or GROUP to the udev rule.