USERSPACE MAD ACCESS

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).

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.

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.

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");
```

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.

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 the IB_USER_MAD_ENABLE_PKEY ioctl is 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.

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.

/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
```