Mellanox TECHNOLOGIES

RDMA Aware Networks Programming User Manual

Rev 1.2

www.mellanox.com

Mellanox Technologies
350 Oakmead Parkway
Sunnyvale, CA 94085
U.S.A.
www.mellanox.com
Tel: (408) 970-3400
Fax: (408) 970-3403

Mellanox Technologies, Ltd.
PO Box 586 Hermon Building
Yokneam 20692
Israel
Tel: +972-4-909-7200
Fax: +972-4-959-3245

RDMA Aware Networks Programming User’s Manual

Document Number: 2865

Table of Contents

Revision History

Table 1 - Revision History

Rev.	Date	Changes
Rev 1.2	Jan. 2010	Updating Programming Example Appendix A adding RDMAoE support
Rev 1.1	Oct., 2009	Integrating Low-Latency-Ethernet API, RDMA_CM, VPI and Multicast code example.
Rev 1.0	Mar. 2009	Reorganized programming example

Glossary

Access Layer	Low level operating system infrastructure (plumbing) used for accessing the interconnect fabric (VPI™, InfiniBand®, Ethernet, FCoE). It includes all basic transport services needed to support upper level network protocols, middleware, and management agents.
AH (Address Handle)	An object which describes the path to the remote side used in UD QP
CA (Channel Adapter)	A device which terminates an Infiniband link, and executes transport level functions
CI (Channel Interface)	Presentation of the channel to the Verbs Consumer as implemented through the combination of the network adapter, associated firmware, and device driver software
CM (Communication Manager)	An entity responsible to establish, maintain, and release communication for RC and UC QP service types The Service ID Resolution Protocol enables users of UD service to locate QPs supporting their desired service. There is a CM in every IB port of the end nodes.
Compare & Swap	Instructs the remote QP to read a 64-bit value, compare it with the compare data provided, and if equal, replace it with the swap data, provided in the QP.
CQ (Completion Queue)	A queue (FIFO) which contains CQEs
CQE (Completion Queue Entry)	An entry in the CQ that describe s the information about the completed WR (status size etc.)
DMA (Direct Memory Access)	Allowing Hardware to move data blocks directly to the memory, bypassing the CPU
Fetch & Add	Instructs the remote QP to read a 64-bit value and replace it with the sum of the 64-bit value and the added data value, provided in the QP.
GUID (Globally Unique IDentifier)	A 64 bit number that uniquely identifies a device or component in a subnet
GID (Global IDentifier)	A 128-bit identifier used to identify a Port on a network adapter, a port on a Router, or a Multicast Group A GID is a valid 128-bit IPv6 address (per RFC 2373) with additional properties / restrictions defined within IBA to facilitate efficient discovery, communication, and routing.
GRH (Global Routing Header)	A packet header used to deliver packets across a subnet boundary and also used to deliver Multicast messages This Packet header is based on IPv6 protocol.

Network Adapter	A hardware device that allows for communication between computers in a network. It supports verbs.
Host	A computer platform executing an Operating System which may control one or more network adapters
IB	InfiniBand
Join operation	An IB port must explicitly join a multicast group by sending a request to the SM to receive multicast packets.
lkey	A number that is received upon registration of MR is used locally by the WR to identify the memory region and its associated permissions.
LID (Local IDentifier)	A 16 bit address assigned to end nodes by the subnet manager. Each LID is unique within its subnet.
LLE (Low Latency Ethernet)	RDMA service over CEE (Converged Enhanced Ethernet) allowing IB transport over Ethernet.
NA (Network Adapter)	A device which terminates a link, and executes transport level functions.
MGID (Multicast Group ID)	IB multicast groups, identified by MGIDs, are managed by the SM. The SM associates a MLID with each MGID and explicitly programs the IB switches in the fabric to ensure that the packets are received by all the ports that joined the multicast group.
MR (Memory Region)	A set of memory buffers which have already been registered with access permissions These buffers need to be registered in order for the network adapter to make use of them. During registration an lkey and rkey are created associated with the created memory region
MTU (Maximum Transfer Unit)	The maximum size of a packet payload (not including headers) that can be sent /received from a port
MW (Memory Window)	A resource which supports incoming RDMA operations which allows low cost creation and bind when binding a MW an rkey is created
Outstanding Work Request	WR which was posted to a work queue and its completion was not polled
pkey (Partition key)	The pkey identifies a partition that the port belongs to. A pkey is roughly analogous to a VLAN ID in ethernet networking. It is used to point to an entry within the port's partition key (pkey) table. Each port is assigned at least one pkey by the subnet manager (SM).
PD (Protection Domain)	Object whose components can interact with only each other. AHs interact with QPs, and MRs interact with WQs.
QP (Queue Pair)	The pair (send queue and receive queue) of independent WQs packed together in one object for the purpose of transferring data between nodes of a network Posts are used to initiate the sending or receiving of data. There are three types of QP: UD Unreliable Datagram, Unreliable Connection, and Reliable Connection.

RC (Reliable Connection)	A QP Transport service type based on a connection oriented protocol A QP (Queue pair) is associated with another single QP. The messages are sent in a reliable way (in terms of the correctness and order if the information.)
RDMA (Remote Direct Memory Access)	Accessing memory in a remote side without involvement of the remote CPU
RDMA_CM (Remote Direct Memory Access Communication Manager)	API used to setup reliable, connected and unreliable datagram data transfers. It provides an RDMA transport neutral interface for establishing connections. The API is based on sockets, but adapted for queue pair (QP) based semantics: communication must be over a specific RDMA device, and data transfers are message based.
Requestor	The side of the connection that will initiate a data transfer (by posting a send request)
Responder	The side of the connection that will receive the data (may post a receive request)
rkey	A number that is received upon registration of MR is used to enforce permissions on incoming RDMA operations
RNR (Receiver Not Ready)	The flow in an RC QP where there is a connection between the sides but a RR is not present in the Receive side
RQ (Receive Queue)	A Work Queue which holds RRs posted by the user
RR (Receive Request)	A WR which was posted to an RQ which describes where incoming data using a send opcode is going to be written
RTR (Ready To Receive)	A QP state in which an RR can be posted and be processed
RTS (Ready To Send)	A QP state in which an SR can be posted and be processed
SA (Subnet Administrator)	The interface for querying and manipulating subnet management data
SGE (Scatter /Gather Elements)	An entry to a pointer to a full or a part of a local registered memory block. The element hold the start address of the block, size, and lkey (with its associated permissions).
S/G Array	An array of S/G elements which exists in a WR that according to the used opcode either collects data from multiple buffers and sends them as a single stream or takes a single stream and breaks it down to numerous buffers
SM (Subnet Manager)	An entity that configures and manages the subnet Discovers the network topology Assign LIDs Determines the routing schemes and sets the routing tables One master SM and possible several slaves (Standby mode) Administers switch routing tables thereby establishing paths through the fabric

SQ (Send Queue)	A Work Queue which holds SRs posted by the user
SR (Send Request)	A WR which was posted to an SQ which describes how much data is going to be transferred, its direction, and the way (the opcode will specify the transfer)
SRQ (Shared Receive Queue)	A queue which holds WQEs for incoming messages from any RC/UD QP which is associated with it More than one QPs can be associated with one SRQ.
TCA (Target Channel Adapter)	A Channel Adapter that is not required to support verbs, usually used in I/O devices
UC (Unreliable Connection)	A QP transport service type based on a connection oriented protocol, where a QP (Queue pair) is associated with another single QP. The QPs do not execute a reliable Protocol and messages can be lost.
UD (Unreliable DataGram)	A QP transport service type in which messages can be one packet length and every UD QP can send/receive messages from another UD QP in the subnet Messages can be lost and the order is not guaranteed. UD QP is the only type which supports multicast messages.
Verbs	An abstract description of the functionality of a network adapter. Using the verbs, any application can create / manage objects that are needed in order to use IB for data transfer.
VPI (Virtual Protocol Interface)	Allows the user to change the layer 2 protocol of the port.
WQ (Work Queue)	One of Send Queue or Receive Queue.
WQE (Work Queue Element)	Wookie An element in a work queue.
WR (Work Request)	A request which was posted by a user to a work queue.

1 Introduction

1.1 Scope

The VPI architecture provides a high performance, low latency and reliable means for communication among network adapters and I/O units attached to a switched, high-speed fabric. The specification defines only the line protocols for message passing, and not any APIs. It thus is amenable to improvement of reliability, scalability, speed, and efficiency by use of verbs.

The network adapter or endpoint in a VPI system uses a network adapter driver that supports verbs to connect to the access layer of the fabric.

VPI architecture permits direct user mode access to the hardware. Mellanox provides a dynamically loaded library creating access the hardware via the verbs API.

This document contains verbs and their related inputs, outputs, descriptions, and functionality as exposed through the operating system programming interface.

Note: This programming manual and its verbs are valid only for user space. See header files for the kernel space verbs.

Programming with verbs allows for customizing and optimizing the RDMA-Aware network. This customizing and optimizing should be done only by programmers with advanced knowledge and experience in the VPI systems.

1.2 Intended Audience

This document is intended for Advanced Users familiar with InfiniBand and Ethernet processes.

1.3 Online Resources

- Mellanox Technologies Web pages: http://www.mellanox.com
- Mellanox Technologies Firmware download Web page: http://www.mellanox.com/ under Firmware downloads
- Mellanox Technologies Document Distribution System (DDS): http://docs.mellanox.com under support->documentation login (requires a customer login account)

2 Overview

In order to perform RDMA operations, establishment of a connection to the remote host, as well as appropriate permissions need to be set up first. The mechanism for accomplishing this is the Queue Pair (QP). For those familiar with a standard IP stack, a QP is roughly equivalent to a socket. The QP needs to be initialized on both sides of the connection. Communication Manager (CM) can be used to exchange information about the QP prior to actual QP setup.

Once a QP is established, the verbs API can be used to perform RDMA reads, RDMA writes, and atomic operations. Serialized send/receive operations, which are similar to socket reads/writes, can be performed as well.

2.1 Available Communication Operations

2.1.1 Send/Send with Immediate

The send operation allows you to send data to a remote QP's receive queue. The receiver must have previously posted a receive buffer to receive the data. The sender does not have any control over where the data will reside in the remote host.

Optionally, an immediate 4 byte value may be transmitted with the data buffer. This immediate value is presented to the receiver as part of the receive notification, and is not contained in the data buffer.

2.1.2 Receive

This is the corresponding operation to a send operation. The receiving host is notified that a data buffer has been received, possibly with an inline immediate value. The receiving application is responsible for receive buffer maintenance and posting.

2.1.3 RDMA read

A memory region is read from the remote host. The caller specifies the remote virtual address as well as a local memory address to be copied to. Prior to performing RDMA operations, the remote host must provide appropriate permissions to access its memory. Once these permissions are set, RDMA read operations are conducted with no notification whatsoever to the remote host.

2.1.4 RDMA write/RDMA write with immediate.

Similar to RDMA read, but the data is written to the remote host. RDMA write operations are performed with no notification to the remote host. RDMA write with immediate operations, however, do notify the remote host of the immediate value.

2.1.5 Atomic Fetch and Add / Atomic Compare and Swap

These are atomic extensions to the RDMA operations.

The atomic fetch and add operation atomically increments the value at a specified virtual address by a specified amount. The value prior to being incremented is returned to the caller.

The atomic compare and swap will atomically compare the value at a specified virtual address with a specified value and if they are equal, a specified value will be stored at the address.

2.2 Transport modes

There are several different transport modes you may select from when establishing a QP. Operations available in each mode are shown below in Table 2.

Table 2 - Transport Mode capabilities

Operation	UD	UC	RC	RD
Send (with immediate)	X	X	X	X
Receive	X	X	X	X
RDMA Write (with immediate)		X	X	X
RDMA Read			X	X
Atomic: Fetch and Add/ Cmp and Swap			X	X
Max message size	MTU	2GB	2GB	2GB

2.2.1 Reliable Connection (RC)

Queue Pair is associated with only one other QP.

Messages transmitted by the send queue of one QP are reliably delivered to receive queue of the other QP.

Packets are delivered in order.

A RC connection is very similar to a TCP connection.

2.2.2 Reliable Datagram (RD)

A Queue Pair may communicate with multiple other QPs.

Messages transmitted by an RD QP send queue will be reliably delivered to the receive queue of the QP specified in the associated work request.

Despite the name, Reliable Datagram messages are not limited to a single packet. A single RD message can be up to 2GB in size.

2.2.3 Unreliable Connection (UC)

A Queue Pair is associated with only one other QP.

The connection is not reliable so packets may be lost.

Messages with errors are not retried by the transport, and error handling must be provided by a higher level protocol.

2.2.4 Unreliable Datagram (UD)

A Queue Pair may transmit and receive single-packet messages to/from any other UD QP.

Ordering and delivery are not guaranteed, and delivered packets may be dropped by the receiver. Multicast messages are supported (one to many).

A UD connection is very similar to a UDP connection.

2.3 Key Concepts

2.3.1 Send Request (SR)

An SR defines how much data will be sent, from where, how and, with RDMA, to where. struct ibv_send_wr is used to implement SRs.

2.3.2 Receive Request (RR)

An RR defines buffers where data is to be received for non-RDMA operations. If no buffers are defined and a transmitter attempts a send operation, a receive not ready (RNR) error will be sent. struct ibv_recv_wr is used to implement RRs.

2.3.3 Completion Queue

A Completion Queue is an object which contains the completed work requests which were posted to the Work Queues (WQ). Every completion says that a specific WR was completed (both successfully completed WRs and unsuccessfully completed WRs).

A Completion Queue is a mechanism to notify the application about information of ended Work Requests (status, opcode, size, source).

CQs have n Completion Queue Entries (CQE). The number of CQEs is specified when the CQ is created.

When a CQE is polled it is removed from the CQ.

CQ is a FIFO of CQEs.

CQ can service send queues, receive queues, or both.

Work queues from multiple QPs can be associated with a single CQ.

struct ibv_cq is used to implement a CQ.

2.3.4 Memory Registration

Memory Registration is a mechanism that allows an application to describe a set of virtually contiguous memory locations or a set of physically contiguous memory locations to the network adapter as a virtually contiguous buffer using Virtual Addresses.

The registration process pins the memory pages (to prevent the pages from being swapped out and to keep physical <-> virtual mapping).

During the registration, the OS checks the permissions of the registered block.

The registration process writes the virtual to physical address table to the network adapter.

When registering memory, permissions are set for the region. Permissions are local write, remote read, remote write, atomic, and bind.

Every MR has a remote and a local key (r_key, l_key). Local keys are used by the local HCA to access local memory, such as during a receive operation. Remote keys are given to the remote HCA to allow a remote process access to system memory during RDMA operations.

The same memory buffer can be registered several times (with different access permissions) and every registration results in a different set of keys.

struct ibv_mr is used to implement memory registration.

2.3.5 Memory Window

An MW allows the application to have more flexible control over remote access to its memory. Memory Windows are intended for situations where the application:

- wants to grant and revoke remote access rights to a registered Region in a dynamic fashion with less of a performance penalty than using deregistration/registration or reregistration.
- wants to grant different remote access rights to different remote agents and/or grant those rights over different ranges within a registered Region.

The operation of associating an MW with an MR is called Binding.

Different MWs can overlap the same MR (event with different access permissions).

2.3.6 Address Vector

An Address Vector is an object that describes the route from the local node to the remote node.

In every UC/RC QP there is an address vector in the QP context.

In UD QP the address vector should be defined in every post SR.

struct ibv_ah is used to implement address vectors.

2.3.7 Global Routing Header (GRH)

When global routing is used on UD QPs, there will be a GRH contained in the first 40 bytes of the receive buffer. This area is used to store global routing information, so an appropriate address vector can be generated to respond to the received packet. When using UD, the RR should always have extra 40 bytes available for this GRH.

struct ibv_grh is used to implement GRHs.

2.3.8 Protection Domain

Object whose components can interact with only each other. These components can be AH, QP, MR, and SRQ.

A protection domain is used to associate Queue Pairs with Memory Regions and Memory Windows, as a means for enabling and controlling network adapter access to Host System memory.

PDs are also used to associate Unreliable Datagram queue pairs with Address Handles, as a means of controlling access to UD destinations.

struct ibv_pd is used to implement protection domains.

2.3.9 Asynchronous Events

The network adapter may send async events to inform the SW about events that occurred in the system.

There are two types of async events:

Affiliated events: events that occurred to personal objects (CQ, QP). Those events will be sent to a specific process.

Unaffiliated events: events that occurred to global objects (network adapter, port error). Those events will be sent to all processes.

2.3.10 Scatter Gather

Data is being gathered/scattered using scatter gather elements, which include:

Address: address of the local data buffer that the data will be gathered from or scattered to.

Size: the size of the data that will be read from / written to this address.

L_key: the local key of the MR that was registered to this buffer.

struct ibv_sge implements scatter gather elements.

2.3.11 Polling

Polling the CQ for completion is getting the details about a WR (Send or Receive) that was posted.

If we have completion with bad status in a WR, the rest of the completions will be all be bad (and the Work Queue will be moved to error state).

Every WR that does not have a completion (that was polled) is still outstanding.

Only after a WR has a completion, the send / receive buffer may be used / reused / freed.

The completion status should always be checked.

When a CQE is polled it is removed from the CQ.

Polling is accomplished with the ibv_poll_cq operation.

2.4 Typical Application

This documents provides two program examples:

- The first code, RDMA_RC_example, uses the VPI verbs API, demonstrating how to perform RC: Send, Receive, RDMA Read and RDMA Write operations.
- The second code, multicast example, uses RDMA_CM verbs API, demonstrating Multicast UD.

The structure of a typical application is as follows. The functions in the programming example that implement each step are indicated in **bold.**

1. Get the device list;

   First you must retrieve the list of available IB devices on the local host. Every device in this list contains both a name and a GUID. For example the device names can be: mthca0, mlx4_1.
   Implemented in programming example by **2.5.4 resources_create.**

2. Open the requested device;

   Iterate over the device list, choose a device according to its GUID or name and open it.
   Implemented in programming example by **2.5.4 resources_create.**

3. Query the device capabilities;

   The device capabilities allow the user to understand the supported features (APM, SRQ) and capabilities of the opened device.
   Implemented in programming example by **2.5.4 resources_create.**

4. Allocate a Protection Domain to contain your resources;

   A Protection Domain (PD) allows the user to restrict which components can interact with only each other. These components can be AH, QP, MR, and SRQ.
   Implemented in programming example by **2.5.4 resources_create.**

5. Register a memory region;

   VPI only works with registered memory. Any memory buffer which is valid in the process's virtual space can be registered. During the registration process the user sets memory permissions and receives a local and remote key (lkey/rkey) which will later be used to refer to this memory buffer.
   Implemented in programming example by **2.5.4 resources_create.**

6. Create a Completion Queue (CQ);

   A CQ contains completed work requests (WR). Each WR will generate a completion queue event (CQE) that is placed on the CQ. The CQE will specify if the WR was completed successfully or not.
   Implemented in programming example by **2.5.4 resources_create.**

7. Create a Queue Pair (QP);

   Creating a QP will also create an associated send queue and receive queue.
   Implemented in programming example by **2.5.4 resources_create.**

8. Bring up a QP;

   A created QP still cannot be used until it is transitioned through several states, eventually getting to Ready To Send (RTS).
   Implemented in programming example by **2.5.6 connect_qp, 2.5.7 modify_qp_to_init, 2.5.8 post_receive, 2.5.10 modify_qp_to_rtr, and 2.5.11 modify_qp_to_rts.**

9. Post work requests and poll for completion;

   Use the created QP for communication operations.
   Implemented in programming example by **2.5.12 post_send and 2.5.13 poll_completion.**

10. Cleanup;

    Destroy objects in the reverse order you created them:

Delete QP
Delete CQ
Deregister MR
Deallocate PD
Close device
Implemented in programming example by **2.5.14 resources_destroy.**

2.5 Programming Example Synopsis – RDMA_RC_example

The following is a synopsis of the functions in the programming example, in the order that they are called.

2.5.1 Main

Parse command line. The user may set the TCP port, device name, and device port for the test. If set, these values will override default values in config. The last parameter is the server name. If the server name is set, this designates a server to connect to and therefore puts the program into client mode. Otherwise the program is in server mode.

Call print_config.

Call resources_init.

Call resources_create.

Call connect_qp.

If in server mode, do a call post_send with IBV_WR_SEND operation.

Call poll_completion. Note that the server side expects a completion from the SEND request and the client side expects a RECEIVE completion.

If in client mode, show the message we received via the RECEIVE operation, otherwise, if we are in server mode, load the buffer with a new message.

Sync client<->server.

At this point the server goes directly to the next sync. All RDMA operations are done strictly by the client.

***Client only ***

Call post_send with IBV_WR_RDMA_READ to perform a RDMA read of server’s buffer.

Call poll_completion.

Show server’s message.

Setup send buffer with new message.

Call post_send with IBV_WR_RDMA_WRITE to perform a RDMA write of server’s buffer.

Call poll_completion.

*** End client only operations ***

Sync client<->server.

If server mode, show buffer, proving RDMA write worked.

Call resources_destroy.

Free device name string.

Done.

2.5.2 print_config

Print out configuration information.

2.5.3 resources_init

Clears resources struct.

2.5.4 resources_create

Call sock_connect to connect a TCP socket to the peer.

Get the list of devices, locate the one we want, and open it.

Free the device list.

Get the port information.

Create a PD.

Create a CQ.

Allocate a buffer, initialize it, register it.

Create a QP.

2.5.5 sock_connect

If client, resolve DNS address of server and initiate a connection to it.

If server, listen for incoming connection on indicated port.

2.5.6 connect_qp

Call modify_qp_to_init.

Call post_receive.

Call sock_sync_data to exchange information between server and client.

Call modify_qp_to_rtr.

Call modify_qp_to_rts.

Call sock_sync_data to synchronize client<->server

2.5.7 modify_qp_to_init

Transition QP to INIT state.

2.5.8 post_receive

Prepare a scatter/gather entry for the receive buffer.

Prepare an RR.

Post the RR.

2.5.9 sock_sync_data

Using the TCP socket created with sock_connect, synchronize the given set of data between client and the server. Since this function is blocking, it is also called with dummy data to synchronize the timing of the client and server.

2.5.10 modify_qp_to_rtr

Transition QP to RTR state.

2.5.11 modify_qp_to_rts

Transition QP to RTS state.

2.5.12 post_send

Prepare a scatter/gather entry for data to be sent (or received in RDMA read case).

Create an SR. Note that IBV_SEND_SIGNALED is redundant.

If this is an RDMA operation, set the address and key.

Post the SR.

2.5.13 poll_completion

Poll CQ until an entry is found or MAX_POLL_CQ_TIMEOUT milliseconds are reached.

2.5.14 resources_destroy

Release/free/deallocate all items in resource struct.

2.6 Programming Example Synopsis – RDMA_CM API

2.6.1 Multicast – RDMA_CM code example

This code example for Multicast, uses RDMA-CM and VPI (and hence can be run both over IB and over LLE).

Notes:

1. In order to run the multicast example on either IB or LLE, no change is needed to the test's code.
2. For the IB case, a join operation is involved, yet it is performed by the rdma_cm kernel code.
3. For the LLE case, no join is required. All MGIDs are resolved into MACs at the host.
4. To inform the multicast example which port to use, you need to specify "-b <IP address>" to bind to the desired device port.

2.6.1.1 Main Flow:

1. Get command line parameters.
   - m – MC address, destination port
   - M – unmapped MC address, requires also bind address (parameter “b”)
   - s – sender flag.
   - b – bind address.
   - c – connections amount.
   - C – message count.
   - S – message size.
   - p – port space (UDP default; IPoIB)
2. Create event channel to receive asynchronous events.
3. Allocate Node and creates an identifier that is used to track communication information
4. Start the “run” main function.
5. On ending – release and free resources.

API definition files: rdma/rdma_cma.h and infiniband/verbs.h

2.6.1.2 Run

1. Get source (if provided for binding) and destination addresses – convert the input addresses to socket presentation.
2. Joining:

   A. For all connections:
   if source address is specifically provided, then bind the rdma_cm object to the corresponding network interface. (Associates a source address with an rdma_cm identifier).
   if unmapped MC address with bind address provided, check the remote address and then bind.

   B. Poll on all the connection events and wait that all rdma_cm objects joined the MC group.
3. Send & receive:

   A. If sender: send the messages to all connection nodes (function “post_sends”).

B.If receiver: poll the completion queue (function “poll_cqs”) till messages arrival.

4. On ending – release network resources (per all connections: leaves the multicast group and detaches its associated QP from the group)

The multicast example uses API man pages that can be found on Chapter 4,“RDMA_CM API,” on page 72.

3 VPI verbs API

This chapter describes the details of the VPI verbs API

3.1 Device operations

The following commands are used for general device operations, allowing the user to query information about devices that are on the system as well as opening and closing a specific device.

3.1.1 ibv_get_device_list

Template:

struct ibv_device ****ibv_get_device_list**(int *num_devices)

Input Parameters:

none

Output Parameters:

num_devices	(optional) If non-null, the number of devices returned in the array will be stored here

Return Value:

NULL terminated array of VPI devices or NULL on failure.

Description:

ibv_get_device_list returns a list of VPI devices available on the system. Each entry on the list is a pointer to a struct ibv_device.
struct ibv_device is defined as:

```
struct ibv_device
{
      struct ibv_device_ops                   ops;
      enum ibv_node_type                      node_type;
      enum ibv_transport_type                 transport_type;
      char                                    name[IBV_SYSFS_NAME_MAX];
      char                                    dev_name[IBV_SYSFS_NAME_MAX];
      char                                    dev_path[IBV_SYSFS_PATH_MAX];
      char                                    ibdev_path[IBV_SYSFS_PATH_MAX];
};
```

ops	pointers to alloc and free functions
node_type	IBV_NODE_UNKNOWN
	IBV_NODE_CA
	IBV_NODE_SWITCH
	IBV_NODE_ROUTER
	IBV_NODE_RNIC

transport_type	IBV_TRANSPORT_UNKNOWN
	IBV_TRANSPORT_IB
	IBV_TRANSPORT_IWARP
name	kernel device name eg “mthca0”
dev_name	uverbs device name eg “uverbs0”
dev_path	path to infiniband_verbs class device in sysfs
ibdev_path	path to infiniband class device in sysfs

The list of ibv_device structs shall remain valid until the list is freed. After calling ibv_get_device_list, the user should open any desired devices and promptly free the list via the **ibv_free_device_list** command.

3.1.2 ibv_free_device_list

Template:

void **ibv_free_device_list**(struct ibv_device **list)

Input Parameters:

```
list                list of devices provided from ibv_get_device_list command
```

Output Parameters:

```
none
```

Return Value:

```
none
```

Description:

ibv_free_device_list frees the list of ibv_device structs provided by **ibv_get_device_list**. Any desired devices should be opened prior to calling this command. Once the list is freed, all ibv_device structs that were on the list are invalid and can no longer be used.

3.1.3 ibv_get_device_name

Template:

const char ***ibv_get_device_name**(struct ibv_device *device)

Input Parameters:

```
device              struct ibv_device for desired device
```

Output Parameters:

```
none
```

Return Value:

```
pointer to device name char string or NULL on failure.
```

Description:

ibv_get_device_name returns a pointer to the device name contained within the ibv_device struct.

3.1.4 ibv_get_device_guid

Template:

uint64_t **ibv_get_device_guid**(struct ibv_device *device)

Input Parameters:

```
device              struct ibv_device for desired device
```

Output Parameters:

```
none
```

Return Value:

```
64 bit GUID
```

Description:

ibv_get_device_guid returns the devices 64 bit Global Unique Identifier (GUID) in network byte order.

3.1.5 ibv_open_device

Template:

struct ibv_context ***ibv_open_device**(struct ibv_device *device)

Input Parameters:

```
device              struct ibv_device for desired device
```

Output Parameters:

```
none
```

Return Value:

```
A verbs context that can be used for future operations on the device or NULL on
failure.
```

Description:

ibv_open_device provides the user with a verbs context which is the object that will be used for all other verb operations.

3.1.6 ibv_close_device

Template:

int **ibv_close_device**(struct ibv_context *context)

Input Parameters:

```
context             struct ibv_context from ibv_open_device
```

Output Parameters:

```
none
```

Return Value:

```
0 on success, -1 on failure.
```

Description:

ibv_close_device closes the verb context previously opened with ibv_open_device. This operation does not free any other objects associated with the context. To avoid memory leaks, all other objects must be independently freed prior to calling this command.

3.2 Verb context operations

The following commands are used once a device has been opened. These commands allow you to get more specific information about a device or one of its ports, create completion queues (CQ), completion channels (CC), and protection domains (PD) which can be used for further operations.

3.2.1 ibv_query_device

Template:

int **ibv_query_device**(struct ibv_context *context, struct ibv_device_attr *device_attr)

Input Parameters:

```
context              struct ibv_context from ibv_open_device
```

Output Parameters:

```
device_attr          struct ibv_device_attr containing device attributes
```

Return Value:

```
0 on success, -1 on failure.
```

Description:

ibv_query_device retrieves the various attributes associated with a device. The user should allocate a struct ibv_device_attr, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this struct.
struct ibv_device_attr is defined on the following page.

```
struct ibv_device_attr
{
      char                                             fw_ver[64];
      uint64_t                                         node_guid;
      uint64_t                                         sys_image_guid;
      uint64_t                                         max_mr_size;
      uint64_t                                         page_size_cap;
      uint32_t                                         vendor_id;
      uint32_t                                         vendor_part_id;
      uint32_t                                         hw_ver;
      int                                              max_qp;
      int                                              max_qp_wr;
      int                                              device_cap_flags;
      int                                              max_sge;
      int                                              max_sge_rd;
      int                                              max_cq;
      int                                              max_cqe;
      int                                              max_mr;
      int                                              max_pd;
      int                                              max_qp_rd_atom;
      int                                              max_ee_rd_atom;
      int                                              max_res_rd_atom;
```

```
        int                             max_qp_init_rd_atom;
        int                             max_ee_init_rd_atom;
        enum ibv_atomic_cap             atomic_cap;
        int                             max_ee;
        int                             max_rdd;
        int                             max_mw;
        int                             max_raw_ipv6_qp;
        int                             max_raw_ethy_qp;
        int                             max_mcast_grp;
        int                             max_mcast_qp_attach;
        int                             max_total_mcast_qp_attach;
        int                             max_ah;
        int                             max_fmr;
        int                             max_map_per_fmr;
        int                             max_srq;
        int                             max_srq_wr;
        int                             max_srq_sge;
        uint16_t                        max_pkeys;
        uint8_t                         local_ca_ack_delay;
        uint8_t                         phys_port_cnt;
}
```

fw_ver	Firmware version
node_guid	Node global unique identifier (GUID)
sys_image_guid	System image GUID
max_mr_size	Largest contiguous block that can be registered
page_size_cap	Supported page sizes
vendor_id	Vendor ID, per IEEE
vendor_part_id	Vendor supplied part ID
hw_ver	Hardware version
max_qp	Maximum number of Queue Pairs (QP)
max_qp_wr	Maximum outstanding work requests (WR) on any queue
device_cap_flags	IBV_DEVICE_RESIZE_MAX_WR IBV_DEVICE_BAD_PKEY_CNTR IBV_DEVICE_BAD_QKEY_CNTR IBV_DEVICE_RAW_MULTI IBV_DEVICE_AUTO_PATH_MIG IBV_DEVICE_CHANGE_PHY_PORT IBV_DEVICE_UD_AV_PORT_ENFORCE IBV_DEVICE_CURR_QP_STATE_MOD IBV_DEVICE_SHUTDOWN_PORT IBV_DEVICE_INIT_TYPE IBV_DEVICE_PORT_ACTIVE_EVENT IBV_DEVICE_SYS_IMAGE_GUID IBV_DEVICE_RC_RNR_NAK_GEN IBV_DEVICE_SRQ_RESIZE IBV_DEVICE_N_NOTIFY_CQ IBV_DEVICE_XRC
max_sge	Maximum scatter/gather entries (SGE) per WR for non-RD QPs
max_sge_rd	Maximum SGEs per WR for RD QPs
max_cq	Maximum supported completion queues (CQ)

max_cqe	Maximum completion queue entries (CQE) per CQ
max_mr	Maximum supported memory regions (MR)
max_pd	Maximum supported protection domains (PD)
max_qp_rd_atom	Maximum outstanding RDMA read and atomic operations per QP
max_ee_rd_atom	Maximum outstanding RDMA read and atomic operations per End to End (EE) context (RD connections)
max_res_rd_atom	Maximum resources used for incoming RDMA read and atomic operations
max_qp_init_rd_atom	Maximium RDMA read and atomic operations that may be initiated per QP
max_ee_init_atom	Maximum RDMA read and atomic operations that may be initiated per EE
atomic_cap	IBV_ATOMIC_NONE - no atomic guarantees
	IBV_ATOMIC_HCA - atomic guarantees within this device
	IBV_ATOMIC_GLOB - global atomic guarantees
max_ee	Maximum supported EE contexts
max_rdd	Maximum supported RD domains
max_mw	Maximum supported memory windows (MW)
max_raw_ipv6_qp	Maximum supported raw IPv6 datagram QPs
max_raw_ethy_qp	Maximum supported ethertype datyagram QPs
max_mcast_grp	Maximum supported multicast groups
max_mcast_qp_attach	Maximum QPs per multicast group that can be attached
max_total_mcast_qp_attach	
	Maximum total QPs that can be attached to multicast groups
max_ah	Maximum supported address handles (AH)
max_fmr	Maximum supported fast memory regions (FMR)
max_map_per_fmr	Maximum number of remaps per FMR before an unmap operation is required
max_srq	Maximum supported shared receive queues (SRCQ)
max_srq_wr	Maximum work requests (WR) per SRQ
max_srq_sge	Maximum SGEs per SRQ
max_pkeys	Maximum number of partitions
local_ca_ack_delay	Local CA ack delay
phys_port_cnt	Number of physical ports

3.2.2 ibv_query_port

Template:

int **ibv_query_port**(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr *port_attr)

Input Parameters:

`context`	`struct ibv_context from` **`ibv_open_device`**
`port_num`	`physical port number (1 is first port)`

Output Parameters:

`port_attr`	`struct ibv_port_attr containing port attributes`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_query_port retrieves the various attributes associated with a port. The user should allocate a struct ibv_port_attr, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this struct.
struct ibv_port_attr is defined as follows:

```
struct ibv_port_attr
{
       enum ibv_port_state                        state;
       enum ibv_mtu                               max_mtu;
       enum ibv_mtu                               active_mtu;
       int                                        gid_tbl_len;
       uint32_t                                   port_cap_flags;
       uint32_t                                   max_msg_sz;
       uint32_t                                   bad_pkey_cntr;
       uint32_t                                   qkey_viol_cntr;
       uint16_t                                   pkey_tbl_len;
       uint16_t                                   lid;
       uint16_t                                   sm_lid;
       uint8_t                                    lmc;
       uint8_t                                    max_vl_num;
       uint8_t                                    sm_sl;
       uint8_t                                    subnet_timeout;
       uint8_t                                    init_type_reply;
       uint8_t                                    active_width;
       uint8_t                                    active_speed;
       uint8_t                                    phys_state;
};
```

`state`	`IBV_PORT_NOP`
	`IBV_PORT_DOWN`
	`IBV_PORT_INIT`
	`IBV_PORT_ARMED`
	`IBV_PORT_ACTIVE`
	`IBV_PORT_ACTIVE_DEFER`

max_mtu	Maximum Transmission Unit (MTU) supported by port. Can be: IBV_MTU_256 IBV_MTU_512 IBV_MTU_1024 IBV_MTU_2048 IBV_MTU_4096
active_mtu	Actual MTU in use
gid_tbl_len	Length of source global ID (GID) table
port_cap_flags	N/A
max_msg_sz	Maximum message size
bad_pkey_cntr	Bad P_Key counter
qkey_viol_cntr	Q_Key violation counter
pkey_tbl_len	Length of partition table
lid	First local identifier (LID) assigned to this port
sm_lid	LID of subnet manager (SM)
lmc	LID Mask control (used when multiple LIDs are assigned to port)
max_vl_num	Maximum virtual lanes (VL)
sm_sl	SM service level (SL)
subnet_timeout	Subnet propagation delay
init_type_reply	Type of initialization performed by SM
active_width	Currently active link width
active_speed	Currently active link speed
phys_state	Physical port state

3.2.3 ibv_query_gid

Template:

int **ibv_query_gid**(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid)

Input Parameters:

```
context              struct ibv_context from ibv_open_device
port_num             physical port number (1 is first port)
index                which entry in the GID table to return (0 is first)
```

Output Parameters:

```
gid                  union ibv_gid containing gid information
```

Return Value:

```
0 on success, -1 on failure.
```

Description:

ibv_query_gid retrieves an entry in the port's global identifier (GID) table. Each port is assigned at least one GID by the subnet manager (SM). The GID is a valid IPv6 address composed of the globally unique identifier (GUID) and a prefix assigned by the SM.

The user should allocate a union ibv_gid, pass it to the command, and it will be filled in upon successful return. The user is responsible to free this union.
union ibv_gid is defined as follows:

```
union ibv_gid
{
       uint8_t                                raw[16];
       struct
       {
           uint64_t                           subnet_prefix;
           uint64_t                           interface_id;
       } global;
};
```

3.2.4 ibv_query_pkey

Template:

int **ibv_query_pkey**(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey)

Input Parameters:

`context`	`struct ibv_context from` **`ibv_open_device`**
`port_num`	`physical port number (1 is first port)`
`index`	`which entry in the pkey table to return (0 is first)`

Output Parameters:

`pkey`	`desired pkey`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_query_pkey retrieves an entry in the port's partition key (pkey) table. Each port is assigned at least one pkey by the subnet manager (SM). The pkey identifies a partition that the port belongs to. A pkey is roughly analguous to a VLAN ID in ethernet networking.

The user passes in a pointer to a uint16 that will be filled in with the requested pkey. The user is responsible to free this uint16.

3.2.5 ibv_alloc_pd

Template:

struct ibv_pd ***ibv_alloc_pd**(struct ibv_context *context)

Input Parameters:

`context`	`struct ibv_context from` **`ibv_open_device`**

Output Parameters:

`none`

Return Value:

`pointer to created protection domain or NULL on failure.`

Description:

ibv_alloc_pd creates a protection domain (PD). PDs limit which memory regions can be accessed by which queue pairs (QP) or completion queues (CQ), providing a degree of protection from unauthorized access. The user must create at least one PD to use VPI verbs.

3.2.6 ibv_dealloc_pd

Template:

int **ibv_dealloc_pd**(struct ibv_pd *pd)

Input Parameters:

pd	struct ibv_pd from **ibv_alloc_pd**

Output Parameters:

none

Return Value:

0 on success, errno on failure.

Description:

ibv_dealloc_pd frees a protection domain (PD). This command will fail if any other objects are currently associated with the indicated PD.

3.2.7 ibv_create_cq

Template:

struct ibv_cq ***ibv_create_cq**(struct ibv_context *context, int cqe, void *cq_context, struct ibv_comp_channel *channel, int comp_vector)

Input Parameters:

`context`	`struct ibv_context from` **`ibv_open_device`**
`cqe`	`Minimum number of entries CQ will support`
`cq_context`	`(Optional) User defined value returned with completion events`
`channel`	`(Optional) Completion channel`
`comp_vector`	`(Optional) Completion vector`

Output Parameters:

`none`

Return Value:

`pointer to created CQ or NULL on failure.`

Description:

ibv_create_cq creates a completion queue (CQ). A completion queue holds completion queue events (CQE). Each Queue Pair (QP) has an associated send and receive CQ. A shared receive queue (SRQ) also has an associated CQ. A single CQ can be shared for sending and receiving as well as be shared across multiple QPs.

The parameter cqe defines the minimum size of the queue. The actual size of the queue may be larger than the specified value.

The parameter cq_context is a user defined value. If specified during CQ creation, this value will be returned as a paramater in **ibv_get_cq_event** when using a completion channel (CC).

The parameter channel is used to specify a CC. A CQ is merely a queue that does not have a built in notification mechanism. When using a polling paradigm for CQ processing, a CC is uneccessary. The user simply polls the CQ at regular intervals. If, however, you wish to use a pend paradigm, a CC is required. The CC is the mechanism that allows the user to be notified that a new CQE is on the CQ.

The parameter comp_vector is used to specify the completion vector used to signal completion events. It must be >=0 and < context->num_comp_vectors.

3.2.8 ibv_resize_cq

Template:

int **ibv_resize_cq**(struct ibv_cq *cq, int cqe)

Input Parameters:

`cq`	`CQ to resize`
`cqe`	`Minimum number of entries CQ will support`

Output Parameters:

`none`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_resize_cq resizes a completion queue (CQ).

The parameter cqe must be at least the number of outstanding entries on the queue. The actual size of the queue may be larger than the specified value.

3.2.9 ibv_destroy_cq

Template:

int **ibv_destroy_cq**(struct ibv_cq *cq)

Input Parameters:

```
cq                  CQ to destroy
```

Output Parameters:

```
none
```

Return Value:

```
0 on success, errno on failure.
```

Description:

ibv_destroy_cq frees a completion queue (CQ). This command will fail if there are any queue pairs (QP) that still have the specified CQ associated with them.

3.2.10 ibv_create_comp_channel

Template:

struct ibv_comp_channel ***ibv_create_comp_channel**(struct ibv_context *context)

Input Parameters:

`context` struct ibv_context from **ibv_open_device**

Output Parameters:

`none`

Return Value:

`pointer to created CC or NULL on failure.`

Description:

ibv_create_comp_channel creates a completion channel. A completion channel is a mechanism for the user to receive notifications when new completion queue event (CQE) has been placed on a completion queue (CQ).

3.2.11 ibv_destroy_comp_channel

Template:

int **ibv_destroy_comp_channel**(struct ibv_comp_channel *channel)

Input Parameters:

`channel`	`struct ibv_comp_channel from` **`ibv_create_comp_channel`**

Output Parameters:

`none`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_destroy_comp_channel frees a completion channel. This command will fail if there are any completion queues (CQ) still associated with this completion channel.

3.3 Protection domain operations

Once you have established a protection domain (PD), you may create objects within that domain. This section describes operations available on a PD. These include registering memory regions (MR), creating queue pairs (QP) and address handles (AH).

3.3.1 ibv_reg_mr

Template:

struct ibv_mr ***ibv_reg_mr**(struct ibv_pd *pd, void *addr, size_t length, enum ibv_access_flags access)

Input Parameters:

pd	protection domain, struct ibv_pd from **ibv_alloc_pd**
addr	memory base address
length	length of memroy region in bytes
access	access flags

Output Parameters:

none

Return Value:

pointer to created memory region (MR) or NULL on failure.

Description:

ibv_reg_mr registers a memory region (MR), associates it with a protection domain (PD), and assigns it local and remote keys (lkey, rkey). All VPI commands that use memory require the memory to be registered via this command. The same physical memory may be mapped to different MRs allowing different permissions or PDs to be assigned to the same memory, depending on user requirements.

Access flags may be:

```
IBV_ACCESS_LOCAL_WRITE        Allow local host write access
IBV_ACCESS_REMOTE_WRITE       Allow remote hosts write access
IBV_ACCESS_REMOTE_READ        Allow remote hosts read access
IBV_ACCESS_REMOTE_ATOMIC      Allow remote hosts atomic access
IBV_ACCESS_MW_BIND            Allow memory windows on this MR
```

Local read access is implied and automatic.

Any VPI operation that violates the access permissions of the given memory operation will fail. Note that the queue pair (QP) attributes must also have the correct permissions or the operation will fail.

If IBV_ACCESS_REMOTE_WRITE or IBV_ACCESS_REMOTE_ATOMIC is set, then IBV_ACCESS_LOCAL_WRITE must be set as well.

struct ibv_mr is defined as follows:

```
struct ibv_mr
{
	struct ibv_context		*context;
	struct ibv_pd			*pd;
	void				*addr;
	size_t				length;
	uint32_t			handle;
	uint32_t			lkey;
	uint32_t			rkey;
};
```

3.3.2 ibv_dereg_mr

Template:

int **ibv_dereg_mr**(struct ibv_mr *mr)

Input Parameters:

`mr` `struct ibv_mr from` **`ibv_reg_mr`**

Output Parameters:

`none`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_dereg_mr frees a memory region (MR). The operation will fail if any memory windows (MW) are still bound to the MR.

3.3.3 ibv_create_qp

Template:

struct ibv_qp ***ibv_create_qp**(struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)

Input Parameters:

pd	struct ibv_pd from ibv_alloc_pd
qp_init_attr	initial attributes of queue pair

Output Parameters:

qp_init_attr	actual values are filled in

Return Value:

pointer to created queue pair (QP) or NULL on failure.

Description:

ibv_create_qp creates a QP. When a QP is created, it is put into the RESET state.
struct qp_init_attr is defined as follows:

```
struct ibv_qp_init_attr
{
      void                                        *qp_context;
      struct ibv_cq                               *send_cq;
      struct ibv_cq                               *recv_cq;
      struct ibv_srq                              *srq;
      struct ibv_qp_cap                           cap;
      enum ibv_qp_type                            qp_type;
      int                                         sq_sig_all;
      struct ibv_xrc_domain                       *xrc_domain;
};
```

qp_context	(optional) user defined value associated with QP.
send_cq	send CQ. This must be created by the user prior to calling ibv_create_qp.
recv_cq	receive CQ. This must be created by the user prior to calling ibv_create_qp. It may be the same as send_cq.
srq	(optional) shared receive queue. Only used for SRQ QP’s.
cap	defined below.
qp_type	must be one of the following: IBV_QPT_RC IBV_QPT_UC IBV_QPT_UD IBV_QPT_XRC
sq_sig_all	If this value is set to 1, all work requests (WR) will generate completion queue events (CQE). If this value is set to 0, only WRs that are flagged will generate CQE’s (see **ibv_post_send**).
xrc_domain	(Optional) Only used for XRC operations.

struct ibv_qp_cap is defined as follows:

```
struct ibv_qp_cap
{
      uint32_t                              max_send_wr;
      uint32_t                              max_recv_wr;
      uint32_t                              max_send_sge;
      uint32_t                              max_recv_sge;
      uint32_t                              max_inline_data;
};
```

max_send_wr	Maximum number of outstanding send requests in the send queue.
max_recv_wr	Maximum number of outstanding receive requests (buffers) in the receive queue.
max_send_sge	Maximum number of scatter/gather elements (SGE) in a WR on the send queue.
max_recv_sge	Maximum number of SGEs in a WR on the receive queue.
max_inline_data	Maximum size in bytes of immediate data on the send queue.

3.3.4 ibv_destroy_qp

Template:

int **ibv_destroy_qp**(struct ibv_qp *qp)

Input Parameters:

```
qp                  struct ibv_qp from ibv_create_qp
```

Output Parameters:

```
none
```

Return Value:

```
0 on success, errno on failure.
```

Description:

ibv_destroy_qp frees a queue pair (QP).

3.3.5 ibv_create_ah

Template:

struct ibv_ah ***ibv_create_ah**(struct ibv_pd *pd, struct ibv_ah_attr *attr)

Input Parameters:

pd	struct ibv_pd from ibv_alloc_pd
attr	attributes of address

Output Parameters:

none

Return Value:

pointer to created address handle (AH) or NULL on failure.

Description:

ibv_create_ah creates an AH. An AH contains all of the necessary data to reach a remote destination. In connected transport modes (RC, UC) the AH is associated with a queue pair (QP). In the datagram transport modes (RD, UD), the AH is associated with a work request (WR) or work completion (WC).
struct ibv_ah_attr is defined as follows:

```
struct ibv_ah_attr
{
      struct ibv_global_route                grh;
      uint16_t                               dlid;
      uint8_t                                sl;
      uint8_t                                src_path_bits;
      uint8_t                                static_rate;
      uint8_t                                is_global;
      uint8_t                                port_num;
};
```

grh	defined below
dlid	destination lid
sl	service level
src_path_bits	source path bits
static_rate	static rate
is_global	this is a global address, use grh.
port_num	physical port number to use to reach this destination

struct ibv_global_route is defined as follows:

```
struct ibv_global_route
{
      union ibv_gid                          dgid;
      uint32_t                               flow_label;
      uint8_t                                sgid_index;
```

```
    uint8_t                                 hop_limit;
    uint8_t                                 traffic_class;
};
```

dgid	destination GID (see **ibv_query_gid** for definition)
flow_label	flow label
sgid_index	index of source GID (see **ibv_query_gid)**
hop_limit	hop limit
traffic_class	traffic class

3.3.6 ibv_destroy_ah

Template:

int **ibv_destroy_ah**(struct ibv_ah *ah)

Input Parameters:

ah	struct ibv_ah from **ibv_create_ah**

Output Parameters:

none

Return Value:

0 on success, errno on failure.

Description:

ibv_destroy_ah frees an address handle (AH).

3.4 Queue pair bringup (ibv_modify_qp)

Queue pairs (QP) must be transitioned through an incremental sequence of states prior to being able to be used for communication.

QP States:

RESET	Newly created, queues empty.
INIT	Basic information set. Ready for posting to receive queue.
RTR	Ready to Receive. Remote address info set for connected QPs, QP may now receive packets.
RTS	Ready to Send. Timeout and retry parameters set, QP may now send packets.

These transitions are accomplished through the use of the **ibv_modify_qp** command.

Template:

int **ibv_modify_qp**(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask)

Input Parameters:

qp	struct ibv_qp from **ibv_create_qp**
attr	QP attributes
attr_mask	bit mask that defines which attributes within attr have been set for this call

Output Parameters:

none

Return Value:

0 on success, errno on failure.

Description:

ibv_modify_qp transitions a QP from one state to another. Its name is a bit of a misnomer, since you can not use this command to modify qp attributes at will. There is a very strict set of attributes that may be modified during each transition, and transitions must occur in the proper order. The following subsections describe each transition in more detail.

struct ibv_qp_attr is defined as follows:

```
struct ibv_qp_attr
{
	enum ibv_qp_state		qp_state;
	enum ibv_qp_state		cur_qp_state;
	enum ibv_mtu			path_mtu;
	enum ibv_mig_state		path_mig_state;
	uint32_t			qkey;
	uint32_t			rq_psn;
	uint32_t			sq_psn;
```

```
        uint32_t                                dest_qp_num;
        int                                     qp_access_flags;
        struct ibv_qp_cap                       cap;
        struct ibv_ah_attr                      ah_attr;
        struct ibv_ah_attr                      alt_ah_attr;
        uint16_t                                pkey_index;
        uint16_t                                alt_pkey_index;
        uint8_t                                 en_sqd_async_notify;
        uint8_t                                 sq_draining;
        uint8_t                                 max_rd_atomic;
        uint8_t                                 max_dest_rd_atomic;
        uint8_t                                 min_rnr_timer;
        uint8_t                                 port_num;
        uint8_t                                 timeout;
        uint8_t                                 retry_cnt;
        uint8_t                                 rnr_retry;
        uint8_t                                 alt_port_num;
        uint8_t                                 alt_timeout;
};
```

The following values select one of the above attributes and should be OR'd into the attr_mask field:

```
        IBV_QP_STATE
        IBV_QP_CUR_STATE
        IBV_QP_EN_SQD_ASYNC_NOTIFY
        IBV_QP_ACCESS_FLAGS
        IBV_QP_PKEY_INDEX
        IBV_QP_PORT
        IBV_QP_QKEY
        IBV_QP_AV
        IBV_QP_PATH_MTU
        IBV_QP_TIMEOUT
        IBV_QP_RETRY_CNT
        IBV_QP_RNR_RETRY
        IBV_QP_RQ_PSN
        IBV_QP_MAX_QP_RD_ATOMIC
        IBV_QP_ALT_PATH
        IBV_QP_MIN_RNR_TIMER
        IBV_QP_SQ_PSN
        IBV_QP_MAX_DEST_RD_ATOMIC
        IBV_QP_PATH_MIG_STATE
        IBV_QP_CAP
        IBV_QP_DEST_QPN
```

3.4.1 RESET to INIT

When a queue pair (QP) is newly created, it is in the RESET state. The first state transition that needs to happen is to bring the QP in the INIT state.

Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE           IBV_QPS_INIT
pkey_index / IBV_QP_PKEY_INDEX    pkey index, normally 0
port_num / IBV_QP_PORT            physical port number (1...n)
qp_access_flags /
    IBV_QP_ACCESS_FLAGS           access flags (see ibv_reg_mr)

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY                qkey (see ibv_post_send)
```

Optional Attributes:

```
none
```

Effect of transition:

Once the QP is transitioned into the INIT state, the user may begin to post receive buffers to the receive queue via the **ibv_post_recv** command. At least one receive buffer must be posted before the QP can be transitioned to the RTR state.

3.4.2 INIT to RTR

Once a queue pair (QP) has receive buffers posted to it, it is now possible to transition the QP into the ready to receive (RTR) state.

Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE          IBV_QPS_RTR
path_mtu /  IBV_QP_PATH_MTU      IB_MTU_256
                                 IB_MTU_512 (recommended value)
                                 IB_MTU_1024
                                 IB_MTU_2048
                                 IB_MTU_4096

*** Connected QPs only ***
ah_attr / IBV_QP_AV              an address handle (AH) needs to be created and
                                 filled in as appropriate. Minimally,
                                 ah_attr.dlid needs to be filled in.
dest_qp_num / IBV_QP_DEST_QPN    QP number of remote QP.
rq_psn / IBV_QP_RQ_PSN           starting receive packet sequence number (should
                                 match remote QP's sq_psn)
max_dest_rd_atomic /
    IBV_MAX_DEST_RD_ATOMIC       maximum number of resources for incoming RDMA
                                 requests
min_rnr_timer /
    IBV_QP_MIN_RNR_TIMER         minimum RNR NAK timer (recommended value: 12)
```

Optional Attributes:

```
*** All QPs ***
qp_access_flags /
    IBV_QP_ACCESS_FLAGS          access flags (see ibv_reg_mr)
pkey_index / IBV_QP_PKEY_INDEX   pkey index, normally 0

*** Connected QPs only ***
alt_ah_attr / IBV_QP_ALT_PATH    AH with alternate path info filled in

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY               qkey (see ibv_post_send)
```

Effect of transition:

Once the QP is transitioned into the RTR state, the QP begins receive processing.

3.4.3 RTR to RTS

Once a queue pair (QP) has reached ready to receive (RTR) state, it may then be transitioned to the ready to send (RTS) state.

Required Attributes:

```
*** All QPs ***
qp_state / IBV_QP_STATE          IBV_QPS_RTS

*** Connected QPs only ***
timeout / IBV_QP_TIMEOUT         local ack timeout (recommended value: 14)
retry_cnt / IBV_QP_RETRY_CNT     retry count (recommended value: 7)
rnr_retry / IBV_QP_RNR_RETRY     RNR retry count (recommended value: 7)
sq_psn / IBV_SQ_PSN              send queue starting packet sequence number
                                 (should match remote QP's rq_psn)
max_rd_atomic
    / IBV_QP_MAX_QP_RD_ATOMIC    number of outstanding RDMA reads and atomic
                                 operations allowed.
```

Optional Attributes:

```
*** All QPs ***
qp_access_flags /
    IBV_QP_ACCESS_FLAGS          access flags (see ibv_reg_mr)

*** Connected QPs only ***
alt_ah_attr / IBV_QP_ALT_PATH    AH with alternate path info filled in
min_rnr_timer /
    IBV_QP_MIN_RNR_TIMER         minimum RNR NAK timer

*** Unconnected QPs only ***
qkey / IBV_QP_QKEY               qkey (see ibv_post_send)
```

Effect of transition:

Once the QP is transitioned into the RTS state, the QP begins send processing and is fully operational. The user may now post send requests with the **ibv_post_send** command.

3.5 Active queue pair operations

Once a queue pair is completely operational, you may query it, be notified of events, and conduct send and receive operations on it. This section describes the operations available to perform these actions.

3.5.1 ibv_query_qp

Template:

int **ibv_query_qp**(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask, struct ibv_qp_init_attr *init_attr)

Input Parameters:

qp	struct ibv_qp from **ibv_create_qp**
attr_mask	bitmask of items to query (see **ibv_modify_qp**)

Output Parameters:

attr	struct ibv_qp_attr to be filled in with requested attributes
init_attr	struct ibv_qp_init_attr to be filled in with initial attributes

Return Value:

0 on success, errno on failure.

Description:

ibv_query_qp retrieves the various attributes of a queue pair (QP) as previously set through **ibv_create_qp** and **ibv_modify_qp**.

The user should allocate a struct ibv_qp_attr and a struct ibv_qp_init_attr and pass them to the command. These structs will be filled in upon successful return. The user is responsible to free these structs.
struct ibv_qp_init_attr is described in **ibv_create_qp** and struct ibv_qp_attr is described in **ibv_modify_qp**.

3.5.2 ibv_post_recv

Template:

int **ibv_post_recv**(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)

Input Parameters:

qp	struct ibv_qp from **ibv_create_qp**
wr	first work request (WR) containing receive buffers

Output Parameters:

bad_wr	pointer to first rejected WR

Return Value:

0 on success, errno on failure.

Description:

ibv_post_recv posts a linked list of WRs to a queue pair's (QP) receive queue. At least one receive buffer must be posted to the receive queue to transition the QP to RTR. Receive buffers are consumed as the remote peer executes send or send with immediate operations. Receive buffers are **NOT** used for RDMA operations. Processing of the WR list is stopped on the first error and a pointer to the offending WR is returned in bad_wr.

struct ibv_recv_wr is defined as follows:

```
struct ibv_recv_wr
{
	uint64_t				wr_id;
	struct ibv_recv_wr			*next;
	struct ibv_sge				*sg_list;
	int					num_sge;
};
```

wr_id	user assigned work request ID
next	pointer to next WR, NULL if last one.
sg_list	scatter array for this WR
num_sge	number of entries in sg_list

struct ibv_sge is defined as follows:

```
struct ibv_sge
{
	uint64_t				addr;
	uint32_t				length;
	uint32_t				lkey;
};
```

addr	address of buffer
length	length of buffer

lkey	local key (lkey) of buffer from **ibv_reg_mr**

3.5.3 ibv_post_send

Template:

int **ibv_post_send**(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)

Input Parameters:

qp	struct ibv_qp from **ibv_create_qp**
wr	first work request (WR)

Output Parameters:

bad_wr	pointer to first rejected WR

Return Value:

0 on success, errno on failure.

Description:

ibv_post_send posts a linked list of WRs to a queue pair's (QP) send queue. This operation is used to initiate all communication, including RDMA operations. Processing of the WR list is stopped on the first error and a pointer to the offending WR is returned in bad_wr.

The user should not alter or destroy AHs associated with WRs until the request has been fully executed and a work completion entry (WCE) has been retrieved from the corresponding completion queue (CQ) to avoid unexpected behaviour.

The buffers used by a WR can only be safely reused after the WR has been fully executed and a WCE has been retrieved from the corresponding CQ. However, if the IBV_SEND_INLINE flag was set, the buffer can be reused immediately after the call returns.

struct ibv_send_wr is defined as follows:

```
struct ibv_send_wr
{
       uint64_t                                  wr_id;
       struct ibv_send_wr                        *next;
       struct ibv_sge                            *sg_list;
       int                                       num_sge;
       enum ibv_wr_opcode                        opcode;
       enum ibv_send_flags                       send_flags;
       uint32_t                                  imm_data;/* network byte order */
       union
       {
           struct
           {
               uint64_t                          remote_addr;
               uint32_t                          rkey;
           } rdma;
           struct
           {
               uint64_t                          remote_addr;
               uint64_t                          compare_add;
               uint64_t                          swap;
```

```
                uint32_t                        rkey;
            } atomic;
            struct
            {
                struct ibv_ah                   *ah;
                uint32_t                        remote_qpn;
                uint32_t                        remote_qkey;
            } ud;
        } wr;
        uint32_t                                xrc_remote_srq_num;
};
```

wr_id	user assigned work request ID
next	pointer to next WR, NULL if last one.
sg_list	scatter/gather array for this WR
num_sge	number of entries in sg_list
opcode	IBV_WR_RDMA_WRITE IBV_WR_RDMA_WRITE_WITH_IMM IBV_WR_SEND IBV_WR_SEND_WITH_IMM IBV_WR_RDMA_READ IBV_WR_ATOMIC_CMP_AND_SWP IBV_WR_ATOMIC_FETCH_AND_ADD
send_flags	(optional) see below
imm_data	immediate data to send in network byte order
remote_addr	remote virtual address for RDMA/atomic operations
rkey	remote key (from **ibv_reg_mr** on remote) for RDMA/atomic operations
compare_add	compare value for compare and swap operation
swap	swap value
ah	address handle (AH) for datagram operations
remote_qpn	remote QP number for datagram operations
remote_qkey	Qkey for datagram operations
xrc_remote_srq_num	shared receive queue (SRQ) number for the destination extended reliable connection (XRC). Only used for XRC operations.

send flags:

IBV_SEND_FENCE	set fence indicator
IBV_SEND_SIGNALED	send completion event for this WR. Only used for QPs that had the sq_sig_all set to 0
IBV_SEND_SEND_SOLICITED	set solicited event indicator
IBV_SEND_INLINE	send data in sge_list as inline data.

struct ibv_sge is defined in **ibv_post_recv**.

3.5.4 ibv_req_notify_cq

Template:

int **ibv_req_notify_cq**(struct ibv_cq *cq, int solicited_only)

Input Parameters:

`cq`	`struct ibv_cq from` **`ibv_create_cq`**
`solicited_only`	`only notify if WR is flagged as solicited`

Output Parameters:

`none`

Return Value:

`0 on success, errno on failure.`

Description:

ibv_req_notify_cq arms the notification mechanism for the indicated completion queue (CQ). When a completion queue entry (CQE) is placed on the CQ, a completion event will be sent to the completion channel (CC) associated with the CQ. If the solicited_only flag is set, then only CQE's for WR's that had the solicited flag set will trigger the notification.

The user should use the **ibv_get_cq_event** operation to receive the notification.

The notification mechanism will only be armed for one notification. Once a notification is sent, the mechanism must be re-armed with a new call to **ibv_req_notify_cq**.

3.5.5 ibv_get_cq_event

Template:

int **ibv_get_cq_event**(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context)

Input Parameters:

`channel`	`struct ibv_comp_channel from` **`ibv_create_comp_channel`**

Output Parameters:

`cq`	`pointer to completion queue (CQ) associated with event`
`cq_context`	`user supplied context set in` **`ibv_create_cq`**

Return Value:

`0 on success, errno on failure.`

Description:

ibv_get_cq_event waits for a notification to be sent on the indicated completion channel (CC). Note that this is a blocking operation. The user should allocate pointers to a struct ibv_cq and a void to be passed into the function. They will be filled in with the appropriate values upon return. It is the user's responsibility to free these pointers.

Each notification sent MUST be acknowledged with the **ibv_ack_cq_events** operation. Since the **ibv_destroy_cq** operation waits for all events to be acknowledged, it will hang if any events are not properly acknowledged.

Once a notification for a completion queue (CQ) is sent on a CC, that CQ is now "disarmed" and will not send any more notifications to the CC until it is rearmed again with a new call to the **ibv_req_notify_cq** operation.

This operation only informs the user that a CQ has completion queue entries (CQE) to be processed, it does not actually process the CQEs. The user should use the **ibv_poll_cq** operation to process the CQEs.

3.5.6 ibv_ack_cq_events

Template:

void **ibv_ack_cq_events**(struct ibv_cq *cq, unsigned int nevents)

Input Parameters:

`cq`	`struct ibv_cq from ibv_create_cq`
`nevents`	`number of events to acknowledge (1...n)`

Output Parameters:

`none`

Return Value:

`none`

Description:

ibv_ack_cq_events acknowledges events received from **ibv_get_cq_event**. Although each notification received from **ibv_get_cq_event** counts as only one event, the user may acknowledge multiple events through a single call to **ibv_ack_cq_events**. The number of events to acknowledge is passed in nevents and should be at least 1. Since this operation takes a mutex, it is somewhat expensive and acknowledging multiple events in one call may provide better performance.

See **ibv_get_cq_event** for additional details.

3.5.7 ibv_poll_cq

Template:

int **ibv_poll_cq**(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)

Input Parameters:

cq	struct ibv_cq from ibv_create_cq
num_entries	maximum number of completion queue entries (CQE) to return

Output Parameters:

wc	CQE array

Return Value:

number of CQEs in array wc or -1 on error

Description:

ibv_poll_cq retrieves CQEs from a completion queue (CQ). The user should allocate an array of struct ibv_wc and pass it to the call in wc. The number of entries available in wc should be passed in num_entries. It is the user's responsibility to free this memory.

The number of CQEs actually retrieved is given as the return value.

CQs must be polled regularly to prevent an overrun. In the event of an overrun, the CQ will be shut down and an async event IBV_EVENT_CQ_ERR will be sent.

struct ibv_wc is defined as follows:

```
struct ibv_wc
{
	uint64_t					wr_id;
	enum ibv_wc_status				status;
	enum ibv_wc_opcode				opcode;
	uint32_t					vendor_err;
	uint32_t					byte_len;
	uint32_t					imm_data;/* network byte order */
	uint32_t					qp_num;
	uint32_t					src_qp;
	enum ibv_wc_flags				wc_flags;
	uint16_t					pkey_index;
	uint16_t					slid;
	uint8_t						sl;
	uint8_t						dlid_path_bits;
};
```

wr_id	user specified work request id as given in **ibv_post_send** or **ibv_post_recv**
status	IBV_WC_SUCCESS IBV_WC_LOC_LEN_ERR IBV_WC_LOC_QP_OP_ERR IBV_WC_LOC_EEC_OP_ERR IBV_WC_LOC_PROT_ERR IBV_WC_WR_FLUSH_ERR IBV_WC_MW_BIND_ERR IBV_WC_BAD_RESP_ERR IBV_WC_LOC_ACCESS_ERR IBV_WC_REM_INV_REQ_ERR IBV_WC_REM_ACCESS_ERR IBV_WC_REM_OP_ERR IBV_WC_RETRY_EXC_ERR IBV_WC_RNR_RETRY_EXC_ERR IBV_WC_LOC_RDD_VIOL_ERR IBV_WC_REM_INV_RD_REQ_ERR IBV_WC_REM_ABORT_ERR IBV_WC_INV_EECN_ERR IBV_WC_INV_EEC_STATE_ERR IBV_WC_FATAL_ERR IBV_WC_RESP_TIMEOUT_ERR IBV_WC_GENERAL_ERR
opcode	see **ibv_post_send**
vendor_err	vender specific error
byte_len	number of bytes transferred
imm_data	immediate data
qp_num	local queue pair (QP) number
src_qp	remote QP number
wc_flags	see below
pkey_index	index of pkey (valid only for GSI QPs)
slid	source local identifier (LID)
sl	service level (SL)
dlid_path_bits	destination LID path bits

flags:

IBV_WC_GRH	global route header (GRH) is present in UD packet
IBV_WC_WITH_IMM	immediate data value is valid

3.5.8 ibv_init_ah_from_wc

Template:

int **ibv_init_ah_from_wc**(struct ibv_context *context, uint8_t port_num, struct ibv_wc *wc, struct ibv_grh *grh, struct ibv_ah_attr *ah_attr)

Input Parameters:

context	struct ibv_context from **ibv_open_device.** This should be the device the completion queue entry (CQE) was received on.
port_num	physical port number (1..n) that CQE was received on
wc	received CQE from **ibv_poll_cq**
grh	global route header (GRH) from packet (see description)

Output Parameters:

ah_attr	address handle (AH) attributes

Return Value:

0 on success or -1 on error

Description:

ibv_init_ah_from_wc initializes an AH with the necessary attributes to generate a response to a received datagram. The user should allocate a struct ibv_ah_attr and pass this in. If appropriate, the GRH from the received packet should be passed in as well. On UD connections the first 40 bytes of the received packet may contain a GRH. Whether or not this header is present is indicated by the IBV_WC_GRH flag of the CQE. If the GRH is not present on a packet on a UD connection, the first 40 bytes of a packet are undefined.

On return ah_attr will be filled in. ah_attr may then be used in the **ibv_create_ah** function. The user is responsible for freeing ah_attr.

Alternatively, **ibv_create_ah_from_wc** may be used instead of this operation.

3.5.9 ibv_create_ah_from_wc

Template:

struct ibv_ah ***ibv_create_ah_from_wc**(struct ibv_pd *pd, struct ibv_wc *wc, struct ibv_grh *grh, uint8_t port_num)

Input Parameters:

`pd`	`protection domain (PD) from` **`ibv_alloc_pd`**
`wc`	`completion queue entry (CQE) from` **`ibv_poll_cq`**
`grh`	`global route header (GRH) from packet`
`port_num`	`physical port number (1..n) that CQE was received on`

Output Parameters:

`none`

Return Value:

`created address handle (AH) on success or -1 on error`

Description:

ibv_create_ah_from_wc combines the operations **ibv_init_ah_from_wc** and **ibv_create_ah**. See the description of those operations for details.

4 RDMA_CM API

See Section Appendix B. for the selected API used in the multicast example.

4.1 Event Channel Operations

4.1.1 rdma_create_event_channel

Template:

struct rdma_event_channel * rdma_create_event_channel (void)

Input Parameters:

`void` no arguments

Output Parameters:

`none`

Description:

Opens a channel used to report communication events. Asynchronous events are reported to users through event channels.

Notes:

Event channels are used to direct all events on an rdma_cm_id. For many clients, a single event channel may be sufficient, however, when managing a large number of connections or cm_id's, users may find it useful to direct events for different cm_id's to different channels for processing.

All created event channels must be destroyed by calling rdma_destroy_event_channel. Users should call rdma_get_cm_event to retrieve events on an event channel.

Each event channel is mapped to a file descriptor. The associated file descriptor can be used and manipulated like any other fd to change its behavior. Users may make the fd non-blocking, poll or select the fd, etc.

See Also:

rdma_cm, rdma_get_cm_event, rdma_destroy_event_channel

4.1.2 rdma_destroy_event_channel

Template:

void **rdma_destroy_event_channel** (struct rdma_event_channel *channel)

Input Parameters:

```
channel                The communication channel to destroy.
```

Output Parameters:

```
none
```

Description:

Close an event communication channel. Release all resources associated with an event channel and closes the associated file descriptor.

Notes:

All rdma_cm_id's associated with the event channel must be destroyed, and all returned events must be acked before calling this function.

See Also:

rdma_create_event_channel, rdma_get_cm_event, rdma_ack_cm_event

4.2 Connection Manager (CM) ID Operations

4.2.1 rdma_create_id

Template:

int **rdma_create_id** (struct rdma_event_channel *channel, struct rdma_cm_id **id, void *context, enum rdma_port_space ps)

Input Parameters:

channel	The communication channel that events associated with the allocated rdma_cm_id will be reported on.
id	A reference where the allocated communication identifier will be returned.
context	User specified context associated with the rdma_cm_id.
ps	RDMA port space.

Output Parameters:

none

Description:

Creates an identifier that is used to track communication information.

Notes:

Rdma_cm_id's are conceptually equivalent to a socket for RDMA communication. The difference is that RDMA communication requires explicitly binding to a specified RDMA device before communication can occur, and most operations are asynchronous in nature. Communication events on an rdma_cm_id are reported through the associated event channel. Users must release the rdma_cm_id by calling rdma_destroy_id.

PORT SPACES	Details of the services provided by the different port spaces are outlined below.
RDMA_PS_TCP	Provides reliable, connection-oriented QP communication. Unlike TCP, the RDMA port space provides message, not stream, based communication.
RDMA_PS_UDP	Provides unreliable, connectionless QP communication. Supports both datagram and multicast communication.

See Also:

rdma_cm, rdma_create_event_channel, rdma_destroy_id, rdma_get_devices, rdma_bind_addr, rdma_resolve_addr, rdma_connect, rdma_listen, rdma_set_option

4.2.2 rdma_destroy_id

Template:

int **rdma_destroy_id** (struct rdma_cm_id *id)

Input Parameters:

`id`	`The communication identifier to destroy.`

Output Parameters:

`none`

Description:

Destroys the specified rdma_cm_id and cancels any outstanding asynchronous operation.

Notes:

Users must free any associated QP with the rdma_cm_id before calling this routine and ack an related events.

See Also:

rdma_create_id, rdma_destroy_qp, rdma_ack_cm_event

4.2.3 rdma_resolve_addr

Template:

int **rdma_resolve_addr** (struct rdma_cm_id *id, struct sockaddr *src_addr, struct sockaddr *dst_addr, int timeout_ms)

Input Parameters:

id	RDMA identifier.
src_addr	Source address information.This parameter may be NULL.
dst_addr	Destination address information.
timeout_ms	Time to wait for resolution to complete.

Output Parameters:

none

Description:

Resolve destination and optional source addresses from IP addresses to an RDMA address. If successful, the specified rdma_cm_id will be bound to a local device.

Notes:

This call is used to map a given destination IP address to a usable RDMA address. The IP to RDMA address mapping is done using the local routing tables, or via ARP. If a source address is given, the rdma_cm_id is bound to that address, the same as if rdma_bind_addr were called. If no source address is given, and the rdma_cm_id has not yet been bound to a device, then the rdma_cm_id will be bound to a source address based on the local routing tables. After this call, the rdma_cm_id will be bound to an RDMA device. This call is typically made from the active side of a connection before calling rdma_resolve_route and rdma_connect.

InfiniBand Specific

This call maps the destination and, if given, source IP addresses to GIDs. In order to perform the mapping, IPoIB must be running on both the local and remote nodes.

See Also:

rdma_create_id, rdma_resolve_route, rdma_connect, rdma_cre-ate_qp, rdma_get_cm_event, rdma_bind_addr, rdma_get_src_port, rdma_get_dst_port, rdma_get_local_addr, rdma_get_peer_addr

4.2.4 rdma_bind_addr

Template:

int **rdma_bind_addr** (struct rdma_cm_id *id, struct sockaddr *addr)

Input Parameters:

`id`	`RDMA identifier.`
`addr`	`Local  address information. Wildcard values are permitted.`

Output Parameters:

`none`

Description:

Associates a source address with an rdma_cm_id. The address may be wildcarded. If binding to a specific local address, the rdma_cm_id will also be bound to a local RDMA device.

Notes:

Typically, this routine is called before calling rdma_listen to bind to a specific port number, but it may also be called on the active side of a connection before calling rdma_resolve_addr to bind to a specific address. If used to bind to port 0, the rdma_cm will select an available port, which can be retrieved with rdma_get_src_port.

See Also:

rdma_create_id, rdma_listen, rdma_resolve_addr, rdma_create_qp, rdma_get_local_addr, rdma_get_src_port

4.2.5 rdma_join_multicast

Template:

int **rdma_join_multicast** (struct rdma_cm_id *id, struct sockaddr *addr, void *context)

Input Parameters:

id	Communication identifier associated with the request.
addr	Multicast address identifying the group to join.
context	User-defined context associated with the join request.

Output Parameters:

none

Description:

Joins a multicast group and attaches an associated QP to the group.

Notes:

Before joining a multicast group, the rdma_cm_id must be bound to an RDMA device by calling rdma_bind_addr or rdma_resolve_addr. Use of rdma_resolve_addr requires the local routing tables to resolve the multicast address to an RDMA device, unless a specific source address is provided. The user must call rdma_leave_multicast to leave the multicast group and release any multicast resources. After the join operation completes, any associated QP is automatically attached to the multicast group, and the join context is returned to the user through the private_data field in the rdma_cm_event.

See Also:

rdma_leave_multicast, rdma_bind_addr, rdma_resolve_addr, rdma_create_qp,
rdma_get_cm_event

4.2.6 rdma_leave_multicast

Template:

int **rdma_leave_multicast** (struct rdma_cm_id*id, struct sockaddr*addr)

Input Parameters:

id	Communication identifier associated with the request.
addr	Multicast address identifying the group to leave.

Output Parameters:

none

Description:

Leaves a multicast group and detaches an associated QP from the group.

Notes:

Calling this function before a group has been fully joined results in canceling the join operation. Users should be aware that messages received from the multicast group may stilled be queued for completion processing immediately after leaving a multicast group. Destroying an rdma_cm_id will automatically leave all multicast groups.

See Also:

rdma_join_multicast, rdma_destroy_qp

4.2.7 rdma_create_qp

Template:

int **rdma_create_qp** (struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)

Input Parameters:

`id`	`RDMA identifier.`
`pd`	`protection domain for the QP.`
`qp_init_attr`	`initial QP attributes.`

Output Parameters:

`none`

Description:

Allocate a QP associated with the specified rdma_cm_id and transition it for sending and receiving.

Notes:

The rdma_cm_id must be bound to a local RDMA device before calling this function, and the protection domain must be for that same device. QPs allocated to an rdma_cm_id are automatically transitioned by the librdmacm through their states. After being allocated, the QP will be ready to handle posting of receives. If the QP is unconnected, it will be ready to post sends.

See Also:

rdma_bind_addr, rdma_resolve_addr, rdma_destroy_qp, ibv_create_qp, ibv_modify_qp

4.2.8 rdma_destroy_qp

Template:

void **rdma_destroy_qp** (struct rdma_cm_id *id)

Input Parameters:

`id` RDMA identifier.

Output Parameters:

`none`

Description:

Destroy a QP allocated on the rdma_cm_id.

Notes:

Users must destroy any QP associated with an rdma_cm_id before destroying the ID.

See Also:

rdma_create_qp, rdma_destroy_id, ibv_destroy_qp

4.3 Event Handling Operations

4.3.1 rdma_get_cm_event

Template:

int **rdma_get_cm_event** (struct rdma_event_channel *channel, struct rdma_cm_event **event)

Input Parameters:

`channel`	`Event channel to check for events.`
`event`	`Allocated information about the next communication event.`

Description:

Retrieves a communication event. If no events are pending, by default, the call will block until an event is received.

Notes:

The default synchronous behavior of this routine can be changed by modifying the file descriptor associated with the given channel. All events that are reported must be acknowledged by calling rdma_ack_cm_event. Destruction of an rdma_cm_id will block until related events have been acknowledged.

Event Data

Communication event details are returned in the rdma_cm_event structure. This structure is allocated by the rdma_cm and released by the rdma_ack_cm_event routine. Details of the rdma_cm_event structure are given below.

`id`	`The rdma_cm identifier associated with the event. If the event type is RDMA_CM_EVENT_CONNECT_REQUEST, then this references a new id for that communication.`
`listen_id`	`For RDMA_CM_EVENT_CONNECT_REQUEST event types, this references the corresponding listening request identifier.`
`event`	`Specifies the type of communication event which occurred. See EVENT TYPES below.`
`status`	`Returns any asynchronous error information associated with an event. The status is zero unless the corresponding operation failed.`
`param`	`Provides additional details based on the type of event. Users should select the conn or ud subfields based on the rdma_port_space of the rdma_cm_id associated with the event. See UD EVENT DATA and CONN EVENT DATA below.`

UD Event Data

Event parameters related to unreliable datagram (UD) services:

RDMA_PS_UDP and RDMA_PS_IPOIB. The UD event data is valid for RDMA_CM_EVENT_ESTABLISHED and RDMA_CM_EVENT_MULTICAST_JOIN events, unless stated otherwise.

private_data	References any user-specified data associated with RDMA_CM_EVENT_CONNECT_REQUEST or RDMA_CM_EVENT_ESTABLISHED events. The data referenced by this field matches that specified by the remote side when calling rdma_connect or rdma_accept. This field is NULL if the event does not include private data. The buffer referenced by this pointer is deallocated when calling rdma_ack_cm_event.
private_data_len	The size of the private data buffer. Users should note that the size of the private data buffer may be larger than the amount of private data sent by the remote side. Any additional space in the buffer will be zeroed out.
ah_attr	Address information needed to send data to the remote end-point(s). Users should use this structure when allocating their address handle.
qp_num	QP number of the remote endpoint or multicast group.
qkey	QKey needed to send data to the remote endpoint(s).

Conn Event Data

Event parameters related to connected QP services: RDMA_PS_TCP. The connection related event data is valid for RDMA_CM_EVENT_CONNECT_REQUEST and RDMA_CM_EVENT_ESTABLISHED events, unless stated otherwise.

private_data	References any user-specified data associated with the event. The data referenced by this field matches that specified by the remote side when calling rdma_connect or rdma_accept. This field is NULL if the event does not include private data. The buffer referenced by this pointer is deallocated when calling rdma_ack_cm_event.
private_data_len	The size of the private data buffer. Users should note that the size of the private data buffer may be larger than the amount of private data sent by the remote side. Any additional space in the buffer will be zeroed out.
responder_resources	The number of responder resources requested of the recipient. This field matches the initiator depth specified by the remote node when calling rdma_connect and rdma_accept.
initiator_depth	The maximum number of outstanding RDMA read/atomic operations that the recipient may have outstanding. This field matches the responder resources specified by the remote node when calling rdma_connect and rdma_accept.
flow_control	Indicates if hardware level flow control is provided by the sender.
retry_count For	RDMA_CM_EVENT_CONNECT_REQUEST events only, indicates the number of times that the recipient should retry send operations.
rnr_retry_count	The number of times that the recipient should retry receiver not ready (RNR) NACK errors.
srq	Specifies if the sender is using a shared-receive queue.
qp_num	Indicates the remote QP number for the connection.

Event Types

The following types of communication events may be reported.

RDMA_CM_EVENT_ADDR_RESOLVED
Address resolution (rdma_resolve_addr) completed successfully.

RDMA_CM_EVENT_ADDR_ERROR
Address resolution (rdma_resolve_addr) failed.

RDMA_CM_EVENT_ROUTE_RESOLVED
Route resolution (rdma_resolve_route) completed successfully.

RDMA_CM_EVENT_ROUTE_ERROR
Route resolution (rdma_resolve_route) failed.

RDMA_CM_EVENT_CONNECT_REQUEST
Generated on the passive side to notify the user of a new connection request.

RDMA_CM_EVENT_CONNECT_RESPONSE
Generated on the active side to notify the user of a successful response to a connectionrequest. It is only generated on rdma_cm_id's that do not have a QP associated with them.

RDMA_CM_EVENT_CONNECT_ERROR
Indicates that an error has occurred trying to establish or a connection. May be generated on the active or passive side of a connection.

RDMA_CM_EVENT_UNREACHABLE
Generated on the active side to notify the user that the remote server is not reachable or unable to respond to a connection request.

RDMA_CM_EVENT_REJECTED
Indicates that a connection request or response was rejected by the remote end point.

RDMA_CM_EVENT_ESTABLISHED
Indicates that a connection has been established with the remote end point.

RDMA_CM_EVENT_DISCONNECTED
The connection has been disconnected.

RDMA_CM_EVENT_DEVICE_REMOVAL
The local RDMA device associated with the rdma_cm_id has been removed. Upon receiving this event, the user must destroy the related rdma_cm_id.

RDMA_CM_EVENT_MULTICAST_JOIN
The multicast join operation (rdma_join_multicast) completed successfully.

RDMA_CM_EVENT_MULTICAST_ERROR
An error either occurred joining a multicast group, or, if the group had already been joined, on an existing group. The specified multicast group is no longer accessible and should be rejoined, if desired.

RDMA_CM_EVENT_ADDR_CHANGE
The network device associated with this ID through address resolution changed its HW address, eg

following of bonding failover. This event can serve as a hint for applications who want the links used for their RDMA sessions to align with the network stack.

RDMA_CM_EVENT_TIMEWAIT_EXIT
The QP associated with a connection has exited its timewait state and is now ready to be re-used. After a QP has been disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state has completed, the rdma_cm will report this event.

See Also:

rdma_ack_cm_event, rdma_create_event_channel, rdma_resolve_addr, rdma_resolve_route, rdma_connect, rdma_listen, rdma_join_multicast, rdma_destroy_id, rdma_event_str

4.3.2 rdma_ack_cm_event

Template:

int **rdma_ack_cm_event** (struct rdma_cm_event *event)

Input Parameters:

`event` Event to be released.

Description:

Free a communication event. All events which are allocated by rdma_get_cm_event must be released, there should be a one-to-one correspondence between successful gets and acks. This call frees the event structure and any memory that it references.

See Also:

rdma_get_cm_event, rdma_destroy_id

4.3.3 rdma_event_str

Template:

char ***rdma_event_str** (enumrdma_cm_event_type event)

Input Parameters:

`event` Asynchronous event.

Description:

Returns a string representation of an asynchronous event.

See Also:

rdma_get_cm_event

Appendix A. Programming Example

(Send, Receive, RDMA Read, RDMA Write)

Appendix A. Programming Example
(Send, Receive, RDMA Read, RDMA Write)

```
/*
 * BUILD COMMAND:
 *   gcc -Wall -I/usr/local/ofed/include -O2 -o RDMA_RC_example -L/usr/local/ofed/lib64 -L/usr/local/ofed/lib -lib-
verbs RDMA_RC_example.c
 *
 * Copyright (c) 2009 Mellanox Technologies. All rights reserved.
 *
 * This software is available to you under a choice of one of two
 * licenses. You may choose to be licensed under the terms of the GNU
 * General Public License (GPL) Version 2, available from the file
 * COPYING in the main directory of this source tree, or the
 * OpenIB.org BSD license below:
 *
 *     Redistribution and use in source and binary forms, with or
 *     without modification, are permitted provided that the following
 *     conditions are met:
 *
 *      - Redistributions of source code must retain the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer.
 *
 *      - Redistributions in binary form must reproduce the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer in the documentation and/or other materials
 *        provided with the distribution.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 */

/******************************************************************************
*
*          RDMA Aware Networks Programming Example
*
*  This code demonstrates how to perform the following operations using the
*  VPI Verbs API:
*
*          Send
```

```
*          Receive
*          RDMA Read
*          RDMA Write
*
******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdint.h>
#include <inttypes.h>
#include <endian.h>
#include <byteswap.h>
#include <getopt.h>
#include <sys/time.h>
#include <arpa/inet.h>
#include <infiniband/verbs.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

/* poll CQ timeout in millisec (2 seconds) */
#define MAX_POLL_CQ_TIMEOUT 2000
#define MSG      "SEND operation      "
#define RDMAMSGR "RDMA read operation "
#define RDMAMSGW "RDMA write operation"
#define MSG_SIZE (strlen(MSG) + 1)

#if __BYTE_ORDER == __LITTLE_ENDIAN
static inline uint64_t htonll(uint64_t x) { return bswap_64(x); }
static inline uint64_t ntohll(uint64_t x) { return bswap_64(x); }
#elif __BYTE_ORDER == __BIG_ENDIAN
static inline uint64_t htonll(uint64_t x) { return x; }
static inline uint64_t ntohll(uint64_t x) { return x; }
#else
#error __BYTE_ORDER is neither __LITTLE_ENDIAN nor __BIG_ENDIAN
#endif

/* structure of test parameters */
struct config_t
{
 const char          *dev_name;          /* IB device name */
 char                *server_name;       /* server host name */
 u_int32_t           tcp_port;           /* server TCP port */
 int                 ib_port;            /* local IB port to work with */
 int                 gid_idx;            /* gid index to use */
};

/* structure to exchange data which is needed to connect the QPs */
struct cm_con_data_t
{
 uint64_t        addr;           /* Buffer address */
 uint32_t        rkey;           /* Remote key */
 uint32_t        qp_num;         /* QP number */
```

```
    uint16_t            lid;                    /* LID of the IB port */
    uint8_t             gid[16];                /* gid */
} __attribute__ ((packed));

/* structure of system resources */
struct resources
{
    struct ibv_device_attr          /* Device attributes */
    device_attr;
    struct ibv_port_attr            port_attr;          /* IB port attributes */
    struct cm_con_data_t            remote_props;       /* values to connect to remote side */
    struct ibv_context              *ib_ctx;            /* device handle */
    struct ibv_pd                   *pd;                /* PD handle */
    struct ibv_cq                   *cq;                /* CQ handle */
    struct ibv_qp                   *qp;                /* QP handle */
    struct ibv_mr                   *mr;                /* MR handle for buf */
    char                            *buf;               /* memory buffer pointer, used for RDMA and send
                                                        ops */
    int                             sock;               /* TCP socket file descriptor */
};

struct config_t config =
{
    NULL,                       /* dev_name */
    NULL,                       /* server_name */
    19875,                      /* tcp_port */
    1,                          /* ib_port */
    -1                          /* gid_idx */
};


/******************************************************************************
  Socket operations

  For simplicity, the example program uses TCP sockets to exchange control
  information.  If a TCP/IP stack/connection is not available, connection manager
  (CM) may be used to pass this information.  Use of CM is beyond the scope of
  this example

******************************************************************************/


/******************************************************************************
* Function: sock_connect
*
* Input
*   servername    URL of server to connect to (NULL for server mode)
*   port          port of service
*
* Output
*   none
*
* Returns
*   socket (fd) on success, negative error code on failure
*
* Description
*   Connect a socket.  If servername is specified a client connection will be
```

```
*   initiated to the indicated server and port.  Otherwise listen on the
*   indicated port for an incoming connection.
*
******************************************************************************/

static int sock_connect(const char *servername, int port)
{
 struct addrinfo                *resolved_addr = NULL;
 struct addrinfo                *iterator;
 char                           service[6];
 int                            sockfd = -1;
 int                            listenfd = 0;
 int                            tmp;
   struct addrinfo hints =
   {
      .ai_flags = AI_PASSIVE,
      .ai_family = AF_INET,
      .ai_socktype = SOCK_STREAM
   };

   if (sprintf(service, "%d", port) < 0)
      goto sock_connect_exit;

   /* Resolve DNS address, use sockfd as temp storage */

   sockfd = getaddrinfo(servername, service, &hints, &resolved_addr);

   if (sockfd < 0)
   {
      fprintf(stderr, "%s for %s:%d\n", gai_strerror(sockfd), servername, port);
      goto sock_connect_exit;
   }

   /* Search through results and find the one we want */

   for (iterator = resolved_addr; iterator ; iterator = iterator->ai_next)
   {
      sockfd = socket(iterator->ai_family, iterator->ai_socktype, iterator->ai_protocol);

      if (sockfd >= 0)
      {
         if (servername)
            /* Client mode.  Initiate connection to remote */
            if((tmp=connect(sockfd, iterator->ai_addr, iterator->ai_addrlen)))
            {
               fprintf(stdout, "failed connect \n");
               close(sockfd);
               sockfd = -1;
            }
         else
         {
            /* Server mode.  Set up listening socket an accept a connection */
            listenfd = sockfd;
            sockfd = -1;
            if(bind(listenfd, iterator->ai_addr, iterator->ai_addrlen))
```

```
                goto sock_connect_exit;
            listen(listenfd, 1);
            sockfd = accept(listenfd, NULL, 0);
        }
    }
}

sock_connect_exit:

    if(listenfd)
        close(listenfd);

    if(resolved_addr)
        freeaddrinfo(resolved_addr);

    if (sockfd < 0)
    {
        if(servername)
            fprintf(stderr, "Couldn't connect to %s:%d\n", servername, port);
        else
        {
            perror("server accept");
            fprintf(stderr, "accept() failed\n");
        }
    }

return sockfd;
}


/******************************************************************************
* Function: sock_sync_data
*
* Input
 * sock                 socket to transfer data on
 * xfer_size            size of data to transfer
 * local_data           pointer to data to be sent to remote
*
* Output
*   remote_data         pointer to buffer to receive remote data
*
* Returns
*   0 on success, negative error code on failure
*
* Description
*   Sync data across a socket.  The indicated local data will be sent to the
*   remote.  It will then wait for the remote to send it's data back.  It is
*   assumed that the two sides are in sync and call this function in the proper
*   order.  Chaos will ensue if they are not.  :)
*
*   Also note this is a blocking function and will wait for the full data to be
*   received from the remote.
*
******************************************************************************/
```

```
int sock_sync_data(int sock, int xfer_size, char *local_data, char *remote_data)
{
   int                 rc;
   int                 read_bytes = 0;
   int                 total_read_bytes = 0;

   rc = write(sock, local_data, xfer_size);
   if(rc < xfer_size)
      fprintf(stderr, "Failed writing data during sock_sync_data\n");
   else
      rc = 0;

   while(!rc && total_read_bytes < xfer_size)
   {
      read_bytes = read(sock, remote_data, xfer_size);
      if(read_bytes > 0)
         total_read_bytes += read_bytes;
      else
         rc = read_bytes;
   }

   return rc;
}


/******************************************************************************
 End of socket operations
******************************************************************************/


/******************************************************************************
* Function: poll_completion
*
* Input
*   res                 pointer to resources structure
*
* Output
*   none
*
* Returns
*   0 on success, 1 on failure
*
* Description
*   Poll the completion queue for a single event.  This function will continue to
*   poll the queue until MAX_POLL_CQ_TIMEOUT milliseconds have passed.
*
******************************************************************************/

static int poll_completion(struct resources *res)
{
 struct ibv_wc          wc;
 unsigned long          start_time_msec;
 unsigned long          cur_time_msec;
```

```
struct timeval          cur_time;
int                     poll_result;
int                     rc = 0;

  /* poll the completion for a while before giving up of doing it .. */
  gettimeofday(&cur_time, NULL);
  start_time_msec = (cur_time.tv_sec * 1000) + (cur_time.tv_usec / 1000);

  do
  {
    poll_result = ibv_poll_cq(res->cq, 1, &wc);
    gettimeofday(&cur_time, NULL);
    cur_time_msec = (cur_time.tv_sec * 1000) + (cur_time.tv_usec / 1000);
  } while ((poll_result == 0) && ((cur_time_msec - start_time_msec) < MAX_POLL_CQ_TIMEOUT));

  if(poll_result < 0)
  {
    /* poll CQ failed */
    fprintf(stderr, "poll CQ failed\n");
    rc = 1;
  }
  else if (poll_result == 0)
      {
       /* the CQ is empty */
       fprintf(stderr, "completion wasn't found in the CQ after timeout\n");
         rc = 1;
       }
      else
      {
       /* CQE found */
       fprintf(stdout, "completion was found in CQ with status 0x%x\n", wc.status);

       /* check the completion status (here we don't care about the completion opcode */
       if (wc.status != IBV_WC_SUCCESS)
           {
           fprintf(stderr, "got bad completion with status: 0x%x, vendor syndrome: 0x%x\n", wc.status,
             wc.vendor_err);
           rc = 1;
           }
      }

  return rc;
}

/******************************************************************************
* Function: post_send
*
* Input
*   res      pointer to resources structure
*   opcode   IBV_WR_SEND, IBV_WR_RDMA_READ or IBV_WR_RDMA_WRITE
*
* Output
*   none
*
```

```
* Returns
*   0 on success, error code on failure
*
* Description
*   This function will create and post a send work request
******************************************************************************/

static int post_send(struct resources *res, int opcode)
{
 struct ibv_send_wr          sr;
 struct ibv_sge              sge;
 struct ibv_send_wr          *bad_wr = NULL;
 int                         rc;


   /* prepare the scatter/gather entry */
   memset(&sge, 0, sizeof(sge));

   sge.addr = (uintptr_t)res->buf;
   sge.length = MSG_SIZE;
   sge.lkey = res->mr->lkey;

   /* prepare the send work request */
   memset(&sr, 0, sizeof(sr));

   sr.next = NULL;
   sr.wr_id = 0;
   sr.sg_list = &sge;
   sr.num_sge = 1;
   sr.opcode = opcode;
   sr.send_flags = IBV_SEND_SIGNALED;

   if(opcode != IBV_WR_SEND)
   {
      sr.wr.rdma.remote_addr = res->remote_props.addr;
      sr.wr.rdma.rkey = res->remote_props.rkey;
   }

   /* there is a Receive Request in the responder side, so we won't get any into RNR flow */
   rc = ibv_post_send(res->qp, &sr, &bad_wr);
   if (rc)
      fprintf(stderr, "failed to post SR\n");
   else
   {
      switch(opcode)
      {
         case IBV_WR_SEND:
            fprintf(stdout, "Send Request was posted\n");
            break;

         case IBV_WR_RDMA_READ:
            fprintf(stdout, "RDMA Read Request was posted\n");
            break;
```

```
            case IBV_WR_RDMA_WRITE:
                fprintf(stdout, "RDMA Write Request was posted\n");
                break;

            default:
                fprintf(stdout, "Unknown Request was posted\n");
                break;
        }
    }

    return rc;
}

/******************************************************************************
* Function: post_receive
*
* Input
*   res    pointer to resources structure
*
* Output
*   none
*
* Returns
*   0 on success, error code on failure
*
* Description
*
******************************************************************************/

static int post_receive(struct resources *res)
{
 struct ibv_recv_wr              rr;
 struct ibv_sge                  sge;
 struct ibv_recv_wr              *bad_wr;
 int                             rc;

    /* prepare the scatter/gather entry */
    memset(&sge, 0, sizeof(sge));
    sge.addr = (uintptr_t)res->buf;
    sge.length = MSG_SIZE;
    sge.lkey = res->mr->lkey;

    /* prepare the receive work request */
    memset(&rr, 0, sizeof(rr));

    rr.next = NULL;
    rr.wr_id = 0;
    rr.sg_list = &sge;
    rr.num_sge = 1;

    /* post the Receive Request to the RQ */
    rc = ibv_post_recv(res->qp, &rr, &bad_wr);
    if (rc)
        fprintf(stderr, "failed to post RR\n");
```

```
    else
        fprintf(stdout, "Receive Request was posted\n");

    return rc;
}

/******************************************************************************
* Function: resources_init
*
* Input
*   res    pointer to resources structure
*
* Output
*   res    is initialized
*
* Returns
*   none
*
* Description
*   res is initialized to default values
******************************************************************************/
static void resources_init(struct resources *res)
{
    memset(res, 0, sizeof *res);
    res->sock = -1;
}

/******************************************************************************
* Function: resources_create
*
* Input
*   res    pointer to resources structure to be filled in
*
* Output
*   res    filled in with resources
*
* Returns
*   0 on success, 1 on failure
*
* Description
*
*  This function creates and allocates all necessary system resources.  These
*  are stored in res.
*****************************************************************************/

static int resources_create(struct resources *res)
{
    struct ibv_device **dev_list = NULL;
    struct ibv_qp_init_attr qp_init_attr;
    struct ibv_device *ib_dev = NULL;
     size_t     size;
     int        i;
     int        mr_flags = 0;
     int        cq_size = 0;
```

```
 int            num_devices;
 int            rc = 0;

/* if client side */
if (config.server_name)
{
    res->sock = sock_connect(config.server_name, config.tcp_port);
    if (res->sock < 0)
    {
        fprintf(stderr, "failed to establish TCP connection to server %s, port %d\n",
            config.server_name, config.tcp_port);
        rc = -1;
        goto resources_create_exit;
    }
}
else
{
    fprintf(stdout, "waiting on port %d for TCP connection\n", config.tcp_port);

    res->sock = sock_connect(NULL, config.tcp_port);
    if (res->sock < 0)
    {
        fprintf(stderr, "failed to establish TCP connection with client on port %d\n",
            config.tcp_port);
        rc = -1;
        goto resources_create_exit;
    }
}

fprintf(stdout, "TCP connection was established\n");

fprintf(stdout, "searching for IB devices in host\n");

/* get device names in the system */
dev_list = ibv_get_device_list(&num_devices);
if (!dev_list)
{
    fprintf(stderr, "failed to get IB devices list\n");
    rc = 1;
    goto resources_create_exit;
}

/* if there isn't any IB device in host */
if (!num_devices)
{
    fprintf(stderr, "found %d device(s)\n", num_devices);
    rc = 1;
    goto resources_create_exit;
}

fprintf(stdout, "found %d device(s)\n", num_devices);

/* search for the specific device we want to work with */
for (i = 0; i < num_devices; i ++)
```

```
{
    if(!config.dev_name)
    {
        config.dev_name = strdup(ibv_get_device_name(dev_list[i]));
        fprintf(stdout, "device not specified, using first one found: %s\n", config.dev_name);
    }
    if (!strcmp(ibv_get_device_name(dev_list[i]), config.dev_name))
    {
        ib_dev = dev_list[i];
        break;
    }
}

/* if the device wasn't found in host */
if (!ib_dev)
{
    fprintf(stderr, "IB device %s wasn't found\n", config.dev_name);
    rc = 1;
    goto resources_create_exit;
}

/* get device handle */
res->ib_ctx = ibv_open_device(ib_dev);
if (!res->ib_ctx)
{
    fprintf(stderr, "failed to open device %s\n", config.dev_name);
    rc = 1;
    goto resources_create_exit;
}

/* We are now done with device list, free it */

ibv_free_device_list(dev_list);
dev_list = NULL;
ib_dev = NULL;


/* query port properties  */
if (ibv_query_port(res->ib_ctx, config.ib_port, &res->port_attr))
{
    fprintf(stderr, "ibv_query_port on port %u failed\n", config.ib_port);
    rc = 1;
    goto resources_create_exit;
}

/* allocate Protection Domain */
res->pd = ibv_alloc_pd(res->ib_ctx);
if (!res->pd)
{
    fprintf(stderr, "ibv_alloc_pd failed\n");
    rc = 1;
    goto resources_create_exit;
}
```

```
/* each side will send only one WR, so Completion Queue with 1 entry is enough */
cq_size = 1;
res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, NULL, 0);
if (!res->cq)
{
   fprintf(stderr, "failed to create CQ with %u entries\n", cq_size);
   rc = 1;
   goto resources_create_exit;
}

/* allocate the memory buffer that will hold the data */

size = MSG_SIZE;
res->buf = (char *) malloc(size);

if (!res->buf )
{
   fprintf(stderr, "failed to malloc %Zu bytes to memory buffer\n", size);
   rc = 1;
   goto resources_create_exit;
}

memset(res->buf, 0 , size);

/* only in the server side put the message in the memory buffer */
if (!config.server_name)
{
   strcpy(res->buf, MSG);
   fprintf(stdout, "going to send the message: '%s'\n", res->buf);
}
else
   memset(res->buf, 0, size);

/* register the memory buffer */

mr_flags = IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_READ |
    IBV_ACCESS_REMOTE_WRITE ;
res->mr = ibv_reg_mr(res->pd, res->buf, size, mr_flags);
if (!res->mr)
{
   fprintf(stderr, "ibv_reg_mr failed with mr_flags=0x%x\n", mr_flags);
   rc = 1;
   goto resources_create_exit;
}

fprintf(stdout, "MR was registered with addr=%p, lkey=0x%x, rkey=0x%x, flags=0x%x\n",
            res->buf, res->mr->lkey, res->mr->rkey, mr_flags);


/* create the Queue Pair */
memset(&qp_init_attr, 0, sizeof(qp_init_attr));

qp_init_attr.qp_type    = IBV_QPT_RC;
qp_init_attr.sq_sig_all = 1;
```

```
    qp_init_attr.send_cq    = res->cq;
    qp_init_attr.recv_cq    = res->cq;
    qp_init_attr.cap.max_send_wr  = 1;
    qp_init_attr.cap.max_recv_wr  = 1;
    qp_init_attr.cap.max_send_sge = 1;
    qp_init_attr.cap.max_recv_sge = 1;

    res->qp = ibv_create_qp(res->pd, &qp_init_attr);
    if (!res->qp)
    {
        fprintf(stderr, "failed to create QP\n");
        rc = 1;
        goto resources_create_exit;
    }
    fprintf(stdout, "QP was created, QP number=0x%x\n", res->qp->qp_num);

resources_create_exit:

    if(rc)
    {
        /* Error encountered, cleanup */

        if(res->qp)
        {
            ibv_destroy_qp(res->qp);
            res->qp = NULL;
        }

        if(res->mr)
        {
            ibv_dereg_mr(res->mr);
            res->mr = NULL;
        }

        if(res->buf)
        {
            free(res->buf);
            res->buf = NULL;
        }

        if(res->cq)
        {
            ibv_destroy_cq(res->cq);
            res->cq = NULL;
        }

        if(res->pd)
        {
            ibv_dealloc_pd(res->pd);
            res->pd = NULL;
        }

        if(res->ib_ctx)
        {
```

```
        ibv_close_device(res->ib_ctx);
        res->ib_ctx = NULL;
      }

      if(dev_list)
      {
        ibv_free_device_list(dev_list);
        dev_list = NULL;
      }
      if (res->sock >= 0)
      {
        if (close(res->sock))
          fprintf(stderr, "failed to close socket\n");
        res->sock = -1;
      }
    }

    return rc;
}

/******************************************************************************
* Function: modify_qp_to_init
*
* Input
*   qp    QP to transition
*
* Output
*   none
*
* Returns
*   0 on success, ibv_modify_qp failure code on failure
*
* Description
*   Transition a QP from the RESET to INIT state
******************************************************************************/

static int modify_qp_to_init(struct ibv_qp *qp)
{
 struct ibv_qp_attr      attr;
 int                     flags;
 int                     rc;


    memset(&attr, 0, sizeof(attr));

    attr.qp_state = IBV_QPS_INIT;
    attr.port_num = config.ib_port;
    attr.pkey_index = 0;
    attr.qp_access_flags = IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_READ |
       IBV_ACCESS_REMOTE_WRITE;

    flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS;

    rc = ibv_modify_qp(qp, &attr, flags);
```

```
    if (rc)
        fprintf(stderr, "failed to modify QP state to INIT\n");

    return rc;
}

/******************************************************************************
* Function: modify_qp_to_rtr
*
* Input
 *   qp              QP to transition
 *   remote_qpn      remote QP number
 *   dlid            destination LID
 *   dgid            destination GID (mandatory for RoCEE)
*
* Output
*   none
*
* Returns
*   0 on success, ibv_modify_qp failure code on failure
*
* Description
*   Transition a QP from the INIT to RTR state, using the specified QP number
******************************************************************************/

static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid, uint8_t *dgid)
{
 struct ibv_qp_attr        attr;
 int                       flags;
 int                       rc;


    memset(&attr, 0, sizeof(attr));

    attr.qp_state = IBV_QPS_RTR;
    attr.path_mtu = IBV_MTU_256;
    attr.dest_qp_num = remote_qpn;
    attr.rq_psn = 0;
    attr.max_dest_rd_atomic = 1;
    attr.min_rnr_timer = 0x12;
    attr.ah_attr.is_global = 0;
    attr.ah_attr.dlid = dlid;
    attr.ah_attr.sl = 0;
    attr.ah_attr.src_path_bits  = 0;
    attr.ah_attr.port_num = config.ib_port;
    if (config.gid_idx >= 0)
    {
        attr.ah_attr.is_global = 1;
        attr.ah_attr.port_num = 1;
        memcpy(&attr.ah_attr.grh.dgid, dgid, 16);
        attr.ah_attr.grh.flow_label = 0;
        attr.ah_attr.grh.hop_limit = 1;
        attr.ah_attr.grh.sgid_index = config.gid_idx;
        attr.ah_attr.grh.traffic_class = 0;
    }
```

```
    flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN |
        IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER;

    rc = ibv_modify_qp(qp, &attr, flags);
    if (rc)
        fprintf(stderr, "failed to modify QP state to RTR\n");

    return rc;
}

/******************************************************************************
* Function: modify_qp_to_rts
*
* Input
*   qp    QP to transition
*
* Output
*   none
*
* Returns
*   0 on success, ibv_modify_qp failure code on failure
*
* Description
*  Transition a QP from the RTR to RTS state
******************************************************************************/

static int modify_qp_to_rts(struct ibv_qp *qp)
{
 struct ibv_qp_attr    attr;
 int                   flags;
 int                   rc;


    memset(&attr, 0, sizeof(attr));

    attr.qp_state      = IBV_QPS_RTS;
    attr.timeout       = 0x12;
    attr.retry_cnt     = 6;
    attr.rnr_retry     = 0;
    attr.sq_psn        = 0;
    attr.max_rd_atomic = 1;

    flags = IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT |
        IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC;

    rc = ibv_modify_qp(qp, &attr, flags);
    if (rc)
        fprintf(stderr, "failed to modify QP state to RTS\n");

    return rc;
}

/******************************************************************************
```

```
* Function: connect_qp
*
* Input
*   res    pointer to resources structure
*
* Output
*   none
*
* Returns
*   0 on success, error code on failure
*
* Description
*   Connect the QP.  Transition the server side to RTR, sender side to RTS
******************************************************************************/

static int connect_qp(struct resources *res)
{
   struct cm_con_data_t    local_con_data;
   struct cm_con_data_t    remote_con_data;
   struct cm_con_data_t    tmp_con_data;
   int                     rc = 0;
   char                    temp_char;
   union ibv_gid    my_gid;


   if (config.gid_idx >= 0)
   {
      rc = ibv_query_gid(res->ib_ctx, config.ib_port, config.gid_idx, &my_gid);
      if (rc)
      {
         fprintf(stderr, "could not get gid for port %d, index %d\n", config.ib_port, config.gid_idx);
         return rc;
      }
   } else
      memset(&my_gid, 0, sizeof my_gid);


   /* exchange using TCP sockets info required to connect QPs */
   local_con_data.addr = htonll((uintptr_t)res->buf);
   local_con_data.rkey = htonl(res->mr->rkey);
   local_con_data.qp_num = htonl(res->qp->qp_num);
   local_con_data.lid = htons(res->port_attr.lid);
   memcpy(local_con_data.gid, &my_gid, 16);

   fprintf(stdout, "\nLocal LID        = 0x%x\n", res->port_attr.lid);

   if (sock_sync_data(res->sock, sizeof(struct cm_con_data_t), (char *) &local_con_data, (char *) &tmp_con_data) < 0)
   {
      fprintf(stderr, "failed to exchange connection data between sides\n");
      rc = 1;
      goto connect_qp_exit;
   }

   remote_con_data.addr = ntohll(tmp_con_data.addr);
```

```
remote_con_data.rkey = ntohl(tmp_con_data.rkey);
remote_con_data.qp_num = ntohl(tmp_con_data.qp_num);
remote_con_data.lid = ntohs(tmp_con_data.lid);
memcpy(remote_con_data.gid, tmp_con_data.gid, 16);

/* save the remote side attributes, we will need it for the post SR */
res->remote_props = remote_con_data;

fprintf(stdout, "Remote address   = 0x%"PRIx64"\n", remote_con_data.addr);
fprintf(stdout, "Remote rkey = 0x%x\n", remote_con_data.rkey);

fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num);
fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid);
if (config.gid_idx >= 0)
{
   uint8_t *p = remote_con_data.gid;
   fprintf(stdout, "Remote GID =
    %02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x:%02x\n",
    p[0], p[1], p[2], p[3], p[4], p[5], p[6], p[7], p[8], p[9], p[10], p[11], p[12], p[13], p[14], p[15]);
}

/* modify the QP to init */
rc = modify_qp_to_init(res->qp);
if (rc)
{
   fprintf(stderr, "change QP state to INIT failed\n");
   goto connect_qp_exit;
}

/* let the client post RR to be prepared for incoming messages */
if (config.server_name)
{
   rc = post_receive(res);
   if (rc)
   {
      fprintf(stderr, "failed to post RR\n");
      goto connect_qp_exit;
   }
}


/* modify the QP to RTR */
rc = modify_qp_to_rtr(res->qp, remote_con_data.qp_num, remote_con_data.lid, remote_con_data.gid);
if (rc)
{
   fprintf(stderr, "failed to modify QP state to RTR\n");
   goto connect_qp_exit;
}

rc = modify_qp_to_rts(res->qp);
if (rc)
{
   fprintf(stderr, "failed to modify QP state to RTR\n");
   goto connect_qp_exit;
```

```
    }

    fprintf(stdout, "QP state was change to RTS\n");


    /* sync to make sure that both sides are in states that they can connect to prevent packet loose */
    if (sock_sync_data(res->sock, 1, "Q", &temp_char))  /* just send a dummy char back and forth */
    {
        fprintf(stderr, "sync error after QPs are were moved to RTS\n");
        rc = 1;
    }

   connect_qp_exit:

    return rc;
}

/******************************************************************************
* Function: resources_destroy
*
* Input
*   res    pointer to resources structure
*
* Output
*   none
*
* Returns
*   0 on success, 1 on failure
*
* Description
*   Cleanup and deallocate all resources used
******************************************************************************/

static int resources_destroy(struct resources *res)
{
    int rc = 0;

    if (res->qp)
        if (ibv_destroy_qp(res->qp))
        {
            fprintf(stderr, "failed to destroy QP\n");
            rc = 1;
        }

    if (res->mr)
        if (ibv_dereg_mr(res->mr))
        {
            fprintf(stderr, "failed to deregister MR\n");
            rc = 1;
        }

    if (res->buf)
        free(res->buf);
```

```
    if (res->cq)
      if (ibv_destroy_cq(res->cq))
      {
          fprintf(stderr, "failed to destroy CQ\n");
          rc = 1;
      }

    if (res->pd)
      if (ibv_dealloc_pd(res->pd))
      {
          fprintf(stderr, "failed to deallocate PD\n");
          rc = 1;
      }

    if (res->ib_ctx)
      if (ibv_close_device(res->ib_ctx))
      {
          fprintf(stderr, "failed to close device context\n");
          rc = 1;
      }

    if (res->sock >= 0)
      if (close(res->sock))
      {
          fprintf(stderr, "failed to close socket\n");
          rc = 1;
      }

    return rc;
}

/******************************************************************************
* Function: print_config
*
* Input
*   none
*
* Output
*   none
*
* Returns
*   none
*
* Description
*   Print out config information
******************************************************************************/
static void print_config(void)
{
    fprintf(stdout,     " ------------------------------------------------\n");
    fprintf(stdout,      " Device name          : \"%s\"\n", config.dev_name);
    fprintf(stdout,      " IB port              : %u\n", config.ib_port);
    if (config.server_name)
         fprintf(stdout, " IP                   : %s\n", config.server_name);
    fprintf(stdout,      " TCP port             : %u\n", config.tcp_port);
```

```
    if (config.gid_idx >= 0)
        fprintf(stdout, " GID index            : %u\n", config.gid_idx);
    fprintf(stdout,     " ---------------------------------------------\n\n");
}

/******************************************************************************
* Function: usage
*
* Input
*   argv0    command line arguments
*
* Output
*   none
*
* Returns
*   none
*
* Description
*   print a description of command line syntax
******************************************************************************/

static void usage(const char *argv0)
{
    fprintf(stdout, "Usage:\n");
    fprintf(stdout, "  %s   start a server and wait for connection\n", argv0);
    fprintf(stdout, "  %s <host>   connect to server at <host>\n", argv0);
    fprintf(stdout, "\n");
    fprintf(stdout, "Options:\n");
    fprintf(stdout, "  -p, --port <port>   listen on/connect to port <port> (default 18515)\n");
    fprintf(stdout, "  -d, --ib-dev <dev>   use IB device <dev> (default first device found)\n");
    fprintf(stdout, "  -i, --ib-port <port>   use port <port> of IB device (default 1)\n");
    fprintf(stdout, "  -g, --gid_idx <git index>   gid index to be used in GRH (default not used)\n");
}

/******************************************************************************
* Function: main
*
* Input
*   argc   number of items in argv
*   argv   command line parameters
*
* Output
*   none
*
* Returns
*   0 on success, 1 on failure
*
* Description
*   Main program code
******************************************************************************/

int main(int argc, char *argv[])
{
```

```
    struct resources        res;
    int                     rc = 1;
    char                    temp_char;

/* parse the command line parameters */
while (1)
{
    int c;

    static struct option long_options[] =
    {
      {name = "port",       has_arg = 1,   val = 'p' },
      {name = "ib-dev",     has_arg = 1,   val = 'd' },
      {name = "ib-port",    has_arg = 1,   val = 'i' },
      {name = "gid-idx",    has_arg = 1,   val = 'g' },
      {name = NULL,         has_arg = 0,   val = '\0'}
    };

    c = getopt_long(argc, argv, "p:d:i:g:", long_options, NULL);
    if (c == -1)
        break;

    switch (c)
    {
        case 'p':
            config.tcp_port = strtoul(optarg, NULL, 0);
            break;

        case 'd':
            config.dev_name = strdup(optarg);
            break;

        case 'i':
          config.ib_port = strtoul(optarg, NULL, 0);
          if (config.ib_port < 0)
          {
             usage(argv[0]);
             return 1;
          }
          break;

        case 'g':
          config.gid_idx = strtoul(optarg, NULL, 0);
          if (config.gid_idx < 0)
          {
             usage(argv[0]);
             return 1;
            }
            break;

        default:
            usage(argv[0]);
            return 1;
    }
```

```
}

/* parse the last parameter (if exists) as the server name */
if (optind == argc - 1)
   config.server_name = argv[optind];
else if (optind < argc)
{
   usage(argv[0]);
   return 1;
}

/* print the used parameters for info*/
print_config();

/* init all of the resources, so cleanup will be easy */
resources_init(&res);

/* create resources before using them */
if (resources_create(&res))
{
   fprintf(stderr, "failed to create resources\n");
   goto main_exit;
}

/* connect the QPs */
if (connect_qp(&res))
{
   fprintf(stderr, "failed to connect QPs\n");
   goto main_exit;
}

/* let the server post the sr */
if (!config.server_name)
   if (post_send(&res, IBV_WR_SEND))
   {
      fprintf(stderr, "failed to post sr\n");
      goto main_exit;
   }

/* in both sides we expect to get a completion */
if (poll_completion(&res))
{
    fprintf(stderr, "poll completion failed\n");
    goto main_exit;
}

 /* after polling the completion we have the message in the client buffer too */
if (config.server_name)
   fprintf(stdout, "Message is: '%s'\n", res.buf);
else
{
   /* setup server buffer with read message */
   strcpy(res.buf, RDMAMSGR);
}
```

```
/* Sync so we are sure server side has data ready before client tries to read it */
if (sock_sync_data(res.sock, 1, "R", &temp_char))  /* just send a dummy char back and forth */
{
    fprintf(stderr, "sync error before RDMA ops\n");
    rc = 1;
    goto main_exit;
}


/*  Now the client performs an RDMA read and then write on server.
    Note that the server has no idea these events have occured */

if (config.server_name)
{
    /* First we read contens of server's buffer */

    if (post_send(&res, IBV_WR_RDMA_READ))
    {
        fprintf(stderr, "failed to post SR 2\n");
        rc = 1;
        goto main_exit;
    }

    if (poll_completion(&res))
    {
        fprintf(stderr, "poll completion failed 2\n");
        rc = 1;
        goto main_exit;
    }

    fprintf(stdout, "Contents of server's buffer: '%s'\n", res.buf);

    /* Now we replace what's in the server's buffer */
    strcpy(res.buf, RDMAMSGW);

    fprintf(stdout, "Now replacing it with: '%s'\n", res.buf);

    if (post_send(&res, IBV_WR_RDMA_WRITE))
    {
        fprintf(stderr, "failed to post SR 3\n");
        rc = 1;
        goto main_exit;
    }

    if (poll_completion(&res))
    {
        fprintf(stderr, "poll completion failed 3\n");
        rc = 1;
        goto main_exit;
    }
}

/* Sync so server will know that client is done mucking with it's memory */
```

```
    if (sock_sync_data(res.sock, 1, "W", &temp_char))  /* just send a dummy char back and forth */
    {
        fprintf(stderr, "sync error after RDMA ops\n");
        rc = 1;
        goto main_exit;
    }

    if(!config.server_name)
        fprintf(stdout, "Contents of server buffer: '%s'\n", res.buf);

    rc = 0;

main_exit:
    if (resources_destroy(&res))
    {
        fprintf(stderr, "failed to destroy resources\n");
        rc = 1;
    }

    if(config.dev_name)
        free((char *) config.dev_name);

    fprintf(stdout, "\ntest result is %d\n", rc);

    return rc;
}
```

Appendix B. Multicast Code Example

```
/*
 * BUILD COMMAND:
 * gcc -g -Wall -D_GNU_SOURCE -g -O2 -o examples/mckey  examples/mckey.c  -libverbs -lrdmacm
 *
 * Copyright (c) 2005-2007 Intel Corporation.  All rights reserved.
 *
 * This software is available to you under a choice of one of two
 * licenses.  You may choose to be licensed under the terms of the GNU
 * General Public License (GPL) Version 2, available from the file
 * COPYING in the main directory of this source tree, or the
 * OpenIB.org BSD license below:
 *
 *     Redistribution and use in source and binary forms, with or
 *     without modification, are permitted provided that the following
 *     conditions are met:
 *
 *      - Redistributions of source code must retain the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer.
 *
 *      - Redistributions in binary form must reproduce the above
 *        copyright notice, this list of conditions and the following
 *        disclaimer in the documentation and/or other materials
 *        provided with the distribution.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
 * NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS
 * BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN
 * ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
 * CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * $Id$
 */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <byteswap.h>
```

```
#include <unistd.h>
#include <getopt.h>

#include <rdma/rdma_cma.h>

struct cmatest_node
{
        int                     id;
        struct rdma_cm_id       *cma_id;
        int                     connected;
        struct ibv_pd           *pd;
        struct ibv_cq           *cq;
        struct ibv_mr           *mr;
        struct ibv_ah           *ah;
        uint32_t                remote_qpn;
        uint32_t                remote_qkey;
        void                    *mem;
};

struct cmatest
{
        struct rdma_event_channel *channel;
        struct cmatest_node *nodes;
        int conn_index;
        int connects_left;

        struct sockaddr_in6     dst_in;
        struct sockaddr         *dst_addr;
        struct sockaddr_in6     src_in;
        struct sockaddr         *src_addr;
};

static struct cmatest test;
static int connections = 1;
static int message_size = 100;
static int message_count = 10;
static int is_sender;
static int unmapped_addr;
static char *dst_addr;
static char *src_addr;
static enum rdma_port_space port_space = RDMA_PS_UDP;

static int create_message(struct cmatest_node *node)
{
        if (!message_size)
          message_count = 0;

        if (!message_count)
          return 0;

        node->mem = malloc(message_size + sizeof(struct ibv_grh));
        if (!node->mem)
        {
          printf("failed message allocation\n");
```

```
				return -1;
		}
		node->mr = ibv_reg_mr(node->pd, node->mem, message_size + sizeof(struct ibv_grh),
		   IBV_ACCESS_LOCAL_WRITE);
		if (!node->mr)
		{
		   printf("failed to reg MR\n");
		   goto err;
		}
		return 0;
err:
		free(node->mem);
		return -1;
}

static int verify_test_params(struct cmatest_node *node)
{
		struct ibv_port_attr port_attr;
		int ret;

		ret = ibv_query_port(node->cma_id->verbs, node->cma_id->port_num, &port_attr);
		if (ret)
		   return ret;

		if (message_count && message_size > (1 << (port_attr.active_mtu + 7)))
		{
		   printf("mckey: message_size %d is larger than active mtu %d\n", message_size, 1 <<
			  (port_attr.active_mtu + 7));
		   return -EINVAL;
		}

		return 0;
}

static int init_node(struct cmatest_node *node)
{
		struct ibv_qp_init_attr init_qp_attr;
		int cqe, ret;

		node->pd = ibv_alloc_pd(node->cma_id->verbs);
		if (!node->pd)
		{
		   ret = -ENOMEM;
		   printf("mckey: unable to allocate PD\n");
		   goto out;
		}

		cqe = message_count ? message_count * 2 : 2;
		node->cq = ibv_create_cq(node->cma_id->verbs, cqe, node, 0, 0);
		if (!node->cq)
		{
		   ret = -ENOMEM;
		   printf("mckey: unable to create CQ\n");
		   goto out;
```

```
        }

        memset(&init_qp_attr, 0, sizeof init_qp_attr);
        init_qp_attr.cap.max_send_wr = message_count ? message_count : 1;
        init_qp_attr.cap.max_recv_wr = message_count ? message_count : 1;
        init_qp_attr.cap.max_send_sge = 1;
        init_qp_attr.cap.max_recv_sge = 1;
        init_qp_attr.qp_context = node;
        init_qp_attr.sq_sig_all = 0;
        init_qp_attr.qp_type = IBV_QPT_UD;
        init_qp_attr.send_cq = node->cq;
        init_qp_attr.recv_cq = node->cq;
        ret = rdma_create_qp(node->cma_id, node->pd, &init_qp_attr);
        if (ret)
        {
          printf("mckey: unable to create QP: %d\n", ret);
          goto out;
        }

        ret = create_message(node);
        if (ret)
        {
          printf("mckey: failed to create messages: %d\n", ret);
          goto out;
        }
out:
        return ret;
}

static int post_recvs(struct cmatest_node *node)
{
        struct ibv_recv_wr recv_wr, *recv_failure;
        struct ibv_sge sge;
        int i, ret = 0;

        if (!message_count)
          return 0;

        recv_wr.next = NULL;
        recv_wr.sg_list = &sge;
        recv_wr.num_sge = 1;
        recv_wr.wr_id = (uintptr_t) node;

        sge.length = message_size + sizeof(struct ibv_grh);
        sge.lkey = node->mr->lkey;
        sge.addr = (uintptr_t) node->mem;

        for (i = 0; i < message_count && !ret; i++ )
        {
          ret = ibv_post_recv(node->cma_id->qp, &recv_wr, &recv_failure);
          if (ret)
          {
            printf("failed to post receives: %d\n", ret);
            break;
```

```
        }
    }
    return ret;
}

static int post_sends(struct cmatest_node *node, int signal_flag)
{
    struct ibv_send_wr send_wr, *bad_send_wr;
    struct ibv_sge sge;
    int i, ret = 0;

    if (!node->connected || !message_count)
        return 0;

    send_wr.next = NULL;
    send_wr.sg_list = &sge;
    send_wr.num_sge = 1;
    send_wr.opcode = IBV_WR_SEND_WITH_IMM;
    send_wr.send_flags = signal_flag;
    send_wr.wr_id = (unsigned long)node;
    send_wr.imm_data = htonl(node->cma_id->qp->qp_num);

    send_wr.wr.ud.ah = node->ah;
    send_wr.wr.ud.remote_qpn = node->remote_qpn;
    send_wr.wr.ud.remote_qkey = node->remote_qkey;

    sge.length = message_size;
    sge.lkey = node->mr->lkey;
    sge.addr = (uintptr_t) node->mem;

    for (i = 0; i < message_count && !ret; i++)
    {
        ret = ibv_post_send(node->cma_id->qp, &send_wr, &bad_send_wr);
        if (ret)
            printf("failed to post sends: %d\n", ret);
    }
    return ret;
}

static void connect_error(void)
{
    test.connects_left--;
}

static int addr_handler(struct cmatest_node *node)
{
    int ret;

    ret = verify_test_params(node);
    if (ret)
        goto err;

    ret = init_node(node);
    if (ret)
```

```
        goto err;

    if (!is_sender)
    {
        ret = post_recvs(node);
        if (ret)
            goto err;
    }

    ret = rdma_join_multicast(node->cma_id, test.dst_addr, node);
    if (ret)
    {
        printf("mckey: failure joining: %d\n", ret);
        goto err;
    }
    return 0;
err:
    connect_error();
    return ret;
}

static int join_handler(struct cmatest_node *node, struct rdma_ud_param *param)
{
    char buf[40];

    inet_ntop(AF_INET6, param->ah_attr.grh.dgid.raw, buf, 40);
    printf("mckey: joined dgid: %s\n", buf);

    node->remote_qpn = param->qp_num;
    node->remote_qkey = param->qkey;
    node->ah = ibv_create_ah(node->pd, &param->ah_attr);
    if (!node->ah)
    {
        printf("mckey: failure creating address handle\n");
        goto err;
    }

    node->connected = 1;
    test.connects_left--;
    return 0;
err:
    connect_error();
    return -1;
}

static int cma_handler(struct rdma_cm_id *cma_id, struct rdma_cm_event *event)
{
    int ret = 0;

    switch (event->event)
    {
    case RDMA_CM_EVENT_ADDR_RESOLVED:
        ret = addr_handler(cma_id->context);
        break;
```

```
        case RDMA_CM_EVENT_MULTICAST_JOIN:
          ret = join_handler(cma_id->context, &event->param.ud);
          break;
        case RDMA_CM_EVENT_ADDR_ERROR:
        case RDMA_CM_EVENT_ROUTE_ERROR:
        case RDMA_CM_EVENT_MULTICAST_ERROR:
          printf("mckey: event: %s, error: %d\n",  rdma_event_str(event->event), event->status);
          connect_error();
          ret = event->status;
          break;
        case RDMA_CM_EVENT_DEVICE_REMOVAL:
          /* Cleanup will occur after test completes. */
          break;
        default:
          break;
        }
        return ret;
}

static void destroy_node(struct cmatest_node *node)
{
        if (!node->cma_id)
          return;

        if (node->ah)
          ibv_destroy_ah(node->ah);

        if (node->cma_id->qp)
          rdma_destroy_qp(node->cma_id);

        if (node->cq)
          ibv_destroy_cq(node->cq);

        if (node->mem)
        {
          ibv_dereg_mr(node->mr);
          free(node->mem);
        }

        if (node->pd)
          ibv_dealloc_pd(node->pd);

        /* Destroy the RDMA ID after all device resources */
        rdma_destroy_id(node->cma_id);
}

static int alloc_nodes(void)
{
        int ret, i;

        test.nodes = malloc(sizeof *test.nodes * connections);
        if (!test.nodes)
        {
          printf("mckey: unable to allocate memory for test nodes\n");
```

```
        return -ENOMEM;
    }
    memset(test.nodes, 0, sizeof *test.nodes * connections);

    for (i = 0; i < connections; i++)
    {
        test.nodes[i].id = i;
        ret = rdma_create_id(test.channel, &test.nodes[i].cma_id, &test.nodes[i], port_space);
        if (ret)
            goto err;
    }
    return 0;
err:
    while (--i >= 0)
        rdma_destroy_id(test.nodes[i].cma_id);
    free(test.nodes);
    return ret;
}

static void destroy_nodes(void)
{
    int i;

    for (i = 0; i < connections; i++)
        destroy_node(&test.nodes[i]);
    free(test.nodes);
}

static int poll_cqs(void)
{
    struct ibv_wc wc[8];
    int done, i, ret;

    for (i = 0; i < connections; i++)
    {
        if (!test.nodes[i].connected)
            continue;

        for (done = 0; done < message_count; done += ret)
        {
            ret = ibv_poll_cq(test.nodes[i].cq, 8, wc);
            if (ret < 0)
            {
                printf("mckey: failed polling CQ: %d\n", ret);
                return ret;
            }
        }
    }
    return 0;
}

static int connect_events(void)
{
    struct rdma_cm_event *event;
```

```
            int ret = 0;

            while (test.connects_left && !ret)
            {
                  ret = rdma_get_cm_event(test.channel, &event);
                  if (!ret)
                  {
                     ret = cma_handler(event->id, event);
                     rdma_ack_cm_event(event);
                  }
            }
            return ret;
}

static int get_addr(char *dst, struct sockaddr *addr)
{
            struct addrinfo *res;
            int ret;

            ret = getaddrinfo(dst, NULL, NULL, &res);
            if (ret)
            {
               printf("getaddrinfo failed - invalid hostname or IP address\n");
               return ret;
            }

            memcpy(addr, res->ai_addr, res->ai_addrlen);
            freeaddrinfo(res);
            return ret;
}

static int run(void)
{
            int i, ret;

            printf("mckey: starting %s\n", is_sender ? "client" : "server");
            if (src_addr)
            {
               ret = get_addr(src_addr, (struct sockaddr *) &test.src_in);
               if (ret)
                  return ret;
            }

            ret = get_addr(dst_addr, (struct sockaddr *) &test.dst_in);
            if (ret)
               return ret;

            printf("mckey: joining\n");
            for (i = 0; i < connections; i++)
            {
               if (src_addr)
               {
                  ret = rdma_bind_addr(test.nodes[i].cma_id, test.src_addr);
                  if (ret)
```

```
            {
                printf("mckey: addr bind failure: %d\n", ret);
                connect_error();
                return ret;
            }
        }

        if (unmapped_addr)
            ret = addr_handler(&test.nodes[i]);
        else
            ret = rdma_resolve_addr(test.nodes[i].cma_id, test.src_addr, test.dst_addr, 2000);
        if (ret)
        {
            printf("mckey: resolve addr failure: %d\n", ret);
            connect_error();
            return ret;
        }
    }

    ret = connect_events();
    if (ret)
        goto out;

    /*
     * Pause to give SM chance to configure switches.  We don't want to
     * handle reliability issue in this simple test program.
     */
    sleep(3);

    if (message_count)
    {
        if (is_sender)
        {
            printf("initiating data transfers\n");
            for (i = 0; i < connections; i++)
            {
                ret = post_sends(&test.nodes[i], 0);
                if (ret)
                goto out;
            }
        }
        else
        {
            printf("receiving data transfers\n");
            ret = poll_cqs();
            if (ret)
                goto out;
        }
    printf("data transfers complete\n");
    }
out:
    for (i = 0; i < connections; i++)
    {
        ret = rdma_leave_multicast(test.nodes[i].cma_id, test.dst_addr);
```

```
            if (ret)
                printf("mckey: failure leaving: %d\n", ret);
        }
        return ret;
}

int main(int argc, char **argv)
{
        int op, ret;


        while ((op = getopt(argc, argv, "m:M:sb:c:C:S:p:")) != -1)
        {
            switch (op)
            {
                case 'm':
                    dst_addr = optarg;
                    break;
                case 'M':
                    unmapped_addr = 1;
                    dst_addr = optarg;
                    break;
                case 's':
                    is_sender = 1;
                    break;
                case 'b':
                    src_addr = optarg;
                    test.src_addr = (struct sockaddr *) &test.src_in;
                    break;
                case 'c':
                    connections = atoi(optarg);
                    break;
                case 'C':
                    message_count = atoi(optarg);
                    break;
                case 'S':
                    message_size = atoi(optarg);
                    break;
                case 'p':
                    port_space = strtol(optarg, NULL, 0);
                    break;
                default:
                    printf("usage: %s\n", argv[0]);
                    printf("\t-m multicast_address\n");
                    printf("\t[-M unmapped_multicast_address]\n"
                           "\t replaces -m and requires -b\n");
                    printf("\t[-s(ender)]\n");
                    printf("\t[-b bind_address]\n");
                    printf("\t[-c connections]\n");
                    printf("\t[-C message_count]\n");
                    printf("\t[-S message_size]\n");
                    printf("\t[-p port_space - %#x for UDP (default), %#x for IPOIB]\n", RDMA_PS_UDP,
                           RDMA_PS_IPOIB);
                    exit(1);
```

```
        }
    }

    test.dst_addr = (struct sockaddr *) &test.dst_in;
    test.connects_left = connections;

    test.channel = rdma_create_event_channel();
    if (!test.channel)
    {
        printf("failed to create event channel\n");
        exit(1);
    }

    if (alloc_nodes())
        exit(1);

    ret = run();

    printf("test complete\n");
    destroy_nodes();
    rdma_destroy_event_channel(test.channel);

    printf("return status %d\n", ret);
    return ret;
}
```