RDMA Aware Networks Programming User Manual

Rev 1.4

www.mellanox.com

Mellanox TECHNOLOGIES

Mellanox Technologies
350 Oakmead Parkway Suite 100
Sunnyvale, CA 94085
U.S.A.
www.mellanox.com
Tel: (408) 970-3400
Fax: (408) 970-3403

Mellanox Technologies, Ltd.
Beit Mellanox
PO Box 586 Yokneam 20692
Israel
www.mellanox.com
Tel: +972 (0)74 723 7200
Fax: +972 (0)4 959 3245

Document Number: 2865

Table of Contents

Revision History

Table 1 - Revision History

Rev.	Date	Changes
Rev 1.4	Feb. 2013	• Merged Chapter 2 (Introduction to the Programming User Guide) into Chapter 1 • Reformatted sections of Chapter 8:“Programming Examples Using RDMA Verbs”
Rev 1.3	Sep. 2012	• Added new verbs and structures from verbs.h • Added new verbs and structures from rdma_cma.h • Added new verbs and structures from rdma_verbs.h • Added RDMA _CM_EVENTS • Added IBV_EVENTS • Added IBV_WC Status Codes • Added additional programming examples using RDMA Verbs: APM, Multicast and SRQ • Added discussion regarding the differences between RDMA over IB transport versus RoCE
Rev 1.2	Jan. 2010	• Updated Programming Example Appendix A • Added RDMAoE support
Rev 1.1	Oct. 2009	• Integrated Low-Latency-Ethernet API, RDMA_CM, VPI and Multicast code example
Rev 1.0	Mar. 2009	• Reorganized programming example

Glossary

Table 2 - Glossary (Sheet 1 of 4)

Term	Description
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 describes the information about the completed WR (status size etc.)
DMA (Direct Memory Access)	Allowing Hardware to move data blocks directly to and from 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.
Host	A computer platform executing an Operating System which may control one or more network adapters
IB	InfiniBand

Table 2 - Glossary (Sheet 2 of 4)

Term	Description
Join operation	An IB port must explicitly join a multicast group by sending a request to the SA 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 contiguous 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 L_Key and R_Key are created and 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)	An allocated resource that enables remote access after being bound to a specified area within an existing Memory Registration. Each Memory Window has an associated Window Handle, set of access privileges, and current R_Key.
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 of 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)

Table 2 - Glossary (Sheet 3 of 4)

Term	Description
Responder	The side of the connection that will respond to commands from the requestor which may include a request to write to the responder memory or read from the responder memory and finally a command requesting the responder to receive a message.
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. Also note that a RDMA Write with immediate will consume a RR.
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/UC/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. The message size of a UD packet is limited to the Path MTU

Table 2 - Glossary (Sheet 4 of 4)

Term	Description
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 RDMA 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)	A WQE, pronounced “wookie”, is an element in a work queue.
WR (Work Request)	A request which was posted by a user to a work queue.

1 RDMA Architecture Overview

1.1 InfiniBand

InfiniBand (IB) is a high-speed, low latency, low CPU overhead, highly efficient and scalable server and storage interconnect technology. One of the key capabilities of InfiniBand is its support for native Remote Direct Memory Access (RDMA). InfiniBand enables data transfer between servers and between server and storage without the involvement of the host CPU in the data path. InfiniBand uses I/O channels for data communication (up to 16 million per host), where each channel provides the semantics of a virtualized NIC or HCA (security, isolations etc). InfiniBand provides various technology or solution speeds ranging from 10Gb/s (SDR) up to 56Gb/s (FDR) per port, using copper and optical fiber connections. InfiniBand efficiency and scalability have made it the optimal performance and cost/performance interconnect solution for the world's leading high-performance computing, cloud, Web 2.0, storage, database and financial data centers and applications. InfiniBand is a standard technology, defined and specified by the IBTA organization.

1.2 Virtual Protocol Interconnect® (VPI)

The Mellanox Virtual Protocol Interconnect (VPI) architecture provides a high performance, low latency and reliable means for communication among network adapters and switches supporting both InfiniBand and Ethernet semantics. A VPI adapter or switch can be set to deliver either InfiniBand or Ethernet semantics per port. A dual-port VPI adapter, for example, can be configured to one of the following options:

- An adapter (HCA) with two InfiniBand ports
- A NIC with two Ethernet ports
- An adapter with one InfiniBand port and one Ethernet port at the same time

Similarly, a VPI switch can have InfiniBand-only ports, Ethernet-only ports, or a mix of both InfiniBand and Ethernet ports working at the same time.

Mellanox-based VPI adapters and switches support both the InfiniBand RDMA and the Ethernet RoCE solutions.

1.3 RDMA over Converged Ethernet (RoCE)

RoCE is a standard for RDMA over Ethernet that is also defined and specified by the IBTA organization. RoCE provides true RDMA semantics for Ethernet as it does not require the complex and low performance TCP transport (needed for iWARP, for example).

RoCE is the most efficient low latency Ethernet solution today. It requires a very low CPU overhead and takes advantage of Priority Flow Control in Data Center Bridging Ethernet for lossless connectivity. RoCE has been fully supported by the Open Fabrics Software since the release of OFED 1.5.1.

1.4 Comparison of RDMA Technologies

Currently, there are three technologies that support RDMA: InfiniBand, Ethernet RoCE and Ethernet iWARP. All three technologies share a common user API which is defined in this document, but have different physical and link layers.

When it comes to the Ethernet solutions, RoCE has clear performance advantages over iWARP — both for latency, throughput and CPU overhead. RoCE is supported by many leading solutions, and is incorporated within Windows Server software (as well as InfiniBand).

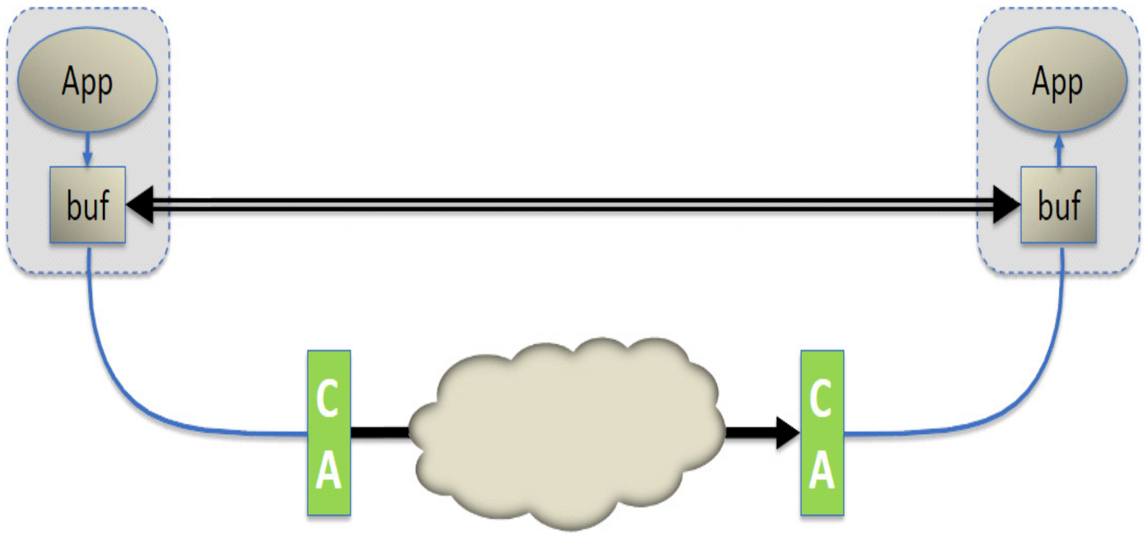
RDMA technologies are based on networking concepts found in a traditional network but there are differences them and their counterparts in IP networks. The key difference is that RDMA provides a messaging service which applications can use to directly access the virtual memory on remote computers. The messaging service can be used for Inter Process Communication (IPC), communication with remote servers and to communicate with storage devices using Upper Layer Protocols (ULPs) such as iSCSI Extensions for RDMA (ISER) and SCSI RDMA Protocol (SRP), Storage Message Block (SMB), Samba, Lustre, ZFS and many more.

RDMA provides low latency through stack bypass and copy avoidance, reduces CPU utilization, reduces memory bandwidth bottlenecks and provides high bandwidth utilization. The key benefits that RDMA delivers accrue from the way that the RDMA messaging service is presented to the application and the underlying technologies used to transport and deliver those messages. RDMA provides Channel based IO. This channel allows an application using an RDMA device to directly read and write remote virtual memory.

In traditional sockets networks, applications request network resources from the operating system through an API which conducts the transaction on their behalf. However RDMA use the OS to establish a channel and then allows applications to directly exchange messages without further OS intervention. A message can be an RDMA Read, an RDMA Write operation or a Send/Receive operation. IB and RoCE also support Multicast transmission.

The IB Link layer offers features such as a credit based flow control mechanism for congestion control. It also allows the use of Virtual Lanes (VLs) which allow simplification of the higher layer level protocols and advanced Quality of Service. It guarantees strong ordering within the VL along a given path. The IB Transport layer provides reliability and delivery guarantees.

The Network Layer used by IB has features which make it simple to transport messages directly between applications' virtual memory even if the applications are physically located on different servers. Thus the combination of IB Transport layer with the Software Transport Interface is better thought of as a RDMA message transport service. The entire stack, including the Software Transport Interface comprises the IB messaging service.



The most important point is that every application has direct access to the virtual memory of devices in the fabric. This means that applications do not need to make requests to an operating system to transfer messages. Contrast this with the traditional network environment where the shared network resources are owned by the operating system and cannot be accessed by a user application. Thus, an application must rely on the involvement of the operating system to move data from the application's virtual buffer space, through the network stack and out onto the wire. Similarly, at the other end, an application must rely on the operating system to retrieve the data on the wire on its behalf and place it in its virtual buffer space.

TCP/IP/Ethernet is a byte-stream oriented transport for passing bytes of information between sockets applications. TCP/IP is lossy by design but implements a reliability scheme using the

Transmission Control Protocol (TCP). TCP/IP requires Operating System (OS) intervention for every operation which includes buffer copying on both ends of the wire. In a byte stream-oriented network, the idea of a message boundary is lost. When an application wants to send a packet, the OS places the bytes into an anonymous buffer in main memory belonging to the operating system and when the byte transfer is complete, the OS copies the data in its buffer into the receive buffer of the application. This process is repeated each time a packet arrives until the entire byte stream is received. TCP is responsible for retransmitting any lost packets due to congestion.

In IB, a complete message is delivered directly to an application. Once an application has requested transport of an RDMA Read or Write, the IB hardware segments the outbound message as needed into packets whose size is determined by the fabric path maximum transfer unit. These packets are transmitted through the IB network and delivered directly into the receiving application's virtual buffer where they are re-assembled into a complete message. The receiving application is notified once the entire message has been received. Thus neither the sending nor the receiving application is involved until the entire message is delivered into the receiving application's buffer.

1.5 Key Components

These are being presented only in the context of the advantages of deploying IB and RoCE. We do not discuss cables and connectors.

Host Channel Adapter

HCAs provide the point at which an IB end node (for example, a server) connects to an IB network. These are the equivalent of the Ethernet (NIC) card but they do much more. HCAs provide address translation mechanism under the control of the operating system which allows an application to access the HCA directly. The same address translation mechanism is the means by which an HCA accesses memory on behalf of a user level application. The application refers to virtual addresses while the HCA has the ability to translate these addresses into physical addresses in order to affect the actual message transfer.

Range Extenders

InfiniBand range extension is accomplished by encapsulating the InfiniBand traffic onto the WAN link and extending sufficient buffer credits to ensure full bandwidth across the WAN.

Subnet Manager

The InfiniBand subnet manager assigns Local Identifiers (LIDs) to each port connected to the InfiniBand fabric and develops a routing table based on the assigned LIDs. The IB Subnet Manager is a concept of Software Defined Networking (SDN) which eliminates the interconnect complexity and enables the creation of very large scale compute and storage infrastructures.

Switches

IB switches are conceptually similar to standard networking switches but are designed to meet IB performance requirements. They implement flow control of the IB Link Layer to prevent packet dropping, and to support congestion avoidance and adaptive routing capabilities, and advanced Quality of Service. Many switches include a Subnet Manager. At least one Subnet Manager is required to configure an IB fabric.

1.6 Support for Existing Applications and ULPs

IP applications are enabled to run over an InfiniBand fabric using IP over IB (IPoIB) or Ethernet over IB (EoIB) or RDS ULPs. Storage applications are supported via iSER, SRP, RDS, NFS, ZFS, SMB and others. MPI and Network Direct are all supported ULPs as well, but are outside the scope of this document.

1.7 References

- IBTA Intro to IB for End Users

  http://members.infinibandta.org/kwspub/Intro_to_IB_for_End_Users.pdf
- Mellanox InfiniBandFAQ_FQ_100.pdf

  http://www.mellanox.com/pdf/whitepapers/InfiniBandFAQ_FQ_100.pdf
- Mellanox WP_2007_IB_Software_and_Protocols.pdf

  http://www.mellanox.com/pdf/whitepapers/WP_2007_IB_Software_and_Protocols.pdf
- Mellanox driver software stacks and firmware are available for download from Mellanox Technologies' Web pages: http://www.mellanox.com

2 RDMA-Aware Programming Overview

The VPI architecture permits direct user mode access to the hardware. Mellanox provides a dynamically loaded library, creating access to 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.

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 section of memory 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.

For both RDMA read and write, the remote side isn't aware that this operation being done (other than the preparation of the permissions and resources).

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 3. RD is not supported by this API.

Table 3 - 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 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.

2.2.3 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 or a RDMA Write with immediate, 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 (even 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)

The GRH is used for routing between subnets. When using RoCE, the GRH is used for routing inside the subnet and therefore is a mandatory. The use of the GRH is mandatory in order for an application to support both IB and RoCE.

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. If GRH is used with 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, SRQ). 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 **7.1.4 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 **7.1.4 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 **7.1.4 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, MW, and SRQ.
   Implemented in programming example by **7.1.4 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 local and remote keys (lkey/rkey) which will later be used to refer to this memory buffer.
Implemented in programming example by **7.1.4 resources_create.**

6. Create a Completion Queue (CQ);

   A CQ contains completed work requests (WR). Each WR will generate a completion queue entry (CQE) that is placed on the CQ. The CQE will specify if the WR was completed successfully or not.
   Implemented in programming example by **7.1.4 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 **7.1.4 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). This provides needed information used by the QP to be able send / receive data.
   Implemented in programming example by **7.1.6 connect_qp, 7.1.7 modify_qp_to_init, 7.1.8 post_receive, 7.1.10 modify_qp_to_rtr, and 7.1.11 modify_qp_to_rts.**

9. Post work requests and poll for completion;

   Use the created QP for communication operations.
   Implemented in programming example by **7.1.12 post_send and 7.1.13 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 **7.1.14 resources_destroy.**

3 VPI Verbs API

This chapter describes the details of the VPI verbs API.

3.1 Initialization

3.1.1 ibv_fork_init

Template:

int **ibv_fork_init**(void)

Input Parameters:

```
None
```

Output Parameters:

```
None
```

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

ibv_fork_init initializes libibverbs' data structures to handle the fork() function safely and avoid data corruption, whether fork() is called explicitly or implicitly such as in system() calls.

It is not necessary to call ibv_fork_init if all parent process threads are always blocked until all child processes end or change address space via an exec() operation.

This function works on Linux kernels supporting the MADV_DONTFORK flag for madvise() (2.6.17 and higher).

Setting the environment variable RDMAV_FORK_SAFE or IBV_FORK_SAFE to any value has the same effect as calling ibv_fork_init().

Setting the environment variable RDMAV_HUGEPAGES_SAFE to any value tells the library to check the underlying page size used by the kernel for memory regions. This is required if an application uses huge pages either directly or indirectly via a library such as libhugetlbfs.

Calling ibv_fork_init() will reduce performance due to an extra system call for every memory registration, and the additional memory allocated to track memory regions. The precise performance impact depends on the workload and usually will not be significant.

Setting RDMAV_HUGEPAGES_SAFE adds further overhead to all memory registrations.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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 error. If the call fails, errno will be set to indicate the reason for the 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.

3.2.7 ibv_node_type_str

Template:

const char ***ibv_node_type_str** (enum ibv_node_type node_type)

Input Parameters:

node_type	**ibv_node_type enum value which may be an HCA, Switch, Router, RNIC or Unknown**

Output Parameters:

none

Return Value:

A constant string which describes the enum value node_type

Description:

ibv_node_type_str returns a string describing the node type enum value, node_type. This value can be an InfiniBand HCA, Switch, Router, an RDMA enabled NIC or unknown

```
enum ibv_node_type {
	IBV_NODE_UNKNOWN		= -1,
	IBV_NODE_CA			= 1,
	IBV_NODE_SWITCH,
	IBV_NODE_ROUTER,
	IBV_NODE_RNIC
	};
```

3.2.8 ibv_port_state_str

Template:

const char ***ibv_port_state_str** (enum ibv_port_state port_state)

Input Parameters:

`port_state` **The enumerated value of the port state**

Output Parameters:

None

Return Value:

A constant string which describes the enum value port_state

Description:

ibv_port_state_str returns a string describing the port state enum value, port_state.

```
enum ibv_port_state {
       IBV_PORT_NOP                          = 0,
       IBV_PORT_DOWN                         = 1,
       IBV_PORT_INIT                         = 2,
       IBV_PORT_ARMED                        = 3,
       IBV_PORT_ACTIVE                       = 4,
       IBV_PORT_ACTIVE_DEFER                 = 5
       };
```

3.3 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.

3.3.1 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 error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_query_device retrieves the various attributes associated with a device. The user should malloc 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 as follows:

```
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 datagram 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

3.3.2 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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	Supported capabilities of this port. There are currently no enumerations/defines declared in verbs.h
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

3.3.3 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 error. If the call fails, errno will be set to indicate the reason for the 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. GID[0] is unique and contains the port's GUID.

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;
};
```

3.3.4 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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 analogous 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.

3.3.5 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) providing a degree of protection from unauthorized access. The user must create at least one PD to use VPI verbs.

3.3.6 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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.

3.3.7 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 entries (CQE). Each Queue Pair (QP) has an associated send and receive 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 parameter 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 unnecessary. 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.

3.3.8 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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. The CQ may (or may not) contain completions when it is being resized thus, it can be resized during work with the CQ.

3.3.9 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, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

ibv_destroy_cq frees a completion queue (CQ). This command will fail if there is any queue pair (QP) that still has the specified CQ associated with it.

3.3.10 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).

3.3.11 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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.

3.4 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) or shared receive queues (SRQ) and address handles (AH).

3.4.1 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 memory 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 even allowing different permissions or PDs to be assigned to the same memory, depending on user requirements.

Access flags may be bitwise or one of the following enumerations:

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;
};
```

3.4.2 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, -1 on error. If the call fails, errno will be set to indicate
the reason for the 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.

3.4.3 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 = 2, IBV_QPT_UC, IBV_QPT_UD, IBV_QPT_XRC, IBV_QPT_RAW_PACKET = 8, IBV_QPT_RAW_ETH = 8
sq_sig_all	If this value is set to 1, all send 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 inline data on the send queue.

3.4.4 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, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_destroy_qp frees a queue pair (QP).

3.4.5 ibv_create_srq

Template:

struct **ibv_srq *ibv_create_srq**(struct ibv_pd *pd, struct ibv_srq_init_attr *srq_init_attr)

Input Parameters:

pd	The protection domain associated with the shared receive queue (SRQ)
srq_init_attr	A list of initial attributes required to create the SRQ

Output Parameters:

ibv_srq__attr	Actual values of the struct are set

Return Value:

A pointer to the created SRQ or NULL on failure

Description:

ibv_create_srq creates a shared receive queue (SRQ). srq_attr->max_wr and srq_attr->max_sge are read to determine the requested size of the SRQ, and set to the actual values allocated on return. If ibv_create_srq succeeds, then max_wr and max_sge will be at least as large as the requested values.

struct ibv_srq is defined as follows:

```
struct ibv_srq {
        struct ibv_context     *context;struct ibv_context from ibv_open_device
        void                   *srq_context;
        struct ibv_pd          *pd;             Protection domain
        uint32_t                handle;
        pthread_mutex_t         mutex;
        pthread_cond_t          cond;
        uint32_t                events_completed;

}
```

struct ibv_srq_init_attr is defined as follows:

```
struct ibv_srq_init_attr
{
       void                                    *srq_context;
       struct ibv_srq_attr                     attr;
};
```

srq_context	struct ibv_context from ibv_open_device
attr	An ibv_srq_attr struct defined as follows:

struct ibv_srq_attr is defined as follows:

```
struct ibv_srq_attr
{
      uint32_t                              max_wr;
      uint32_t                              max_sge;
      uint32_t                              srq_limit;
};
```

max_wr	Requested maximum number of outstanding WRs in the SRQ
max_sge	Requested number of scatter elements per WR
srq_limit;	The limit value of the SRQ (irrelevant for ibv_create_srq)

3.4.6 ibv_modify_srq

Template:

int **ibv_modify_srq** (struct ibv_srq *srq, struct ibv_srq_attr *srq_attr, int srq_attr_mask)

Input Parameters:

`srq`	`The SRQ to modify`
`srq_attr`	`Specifies the SRQ to modify (input)/the current values of the selected SRQ attributes are returned (output)`
`srq_attr_mask`	`A bit-mask used to specify which SRQ attributes are being modified`

Output Parameters:

`srq_attr`	`The struct ibv_srq_attr is returned with the updated values`

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

ibv_modify_srq modifies the attributes of the SRQ srq using the attribute values in srq_attr based on the mask srq_attr_mask. srq_attr is an ibv_srq_attr struct as defined above under the verb ibv_create_srq. The argument srq_attr_mask specifies the SRQ attributes to be modified. It is either 0 or the bitwise OR of one or more of the flags:

IBV_SRQ_MAX_WR	Resize the SRQ
IBV_SRQ_LIMIT	Set the SRQ limit

If any of the attributes to be modified is invalid, none of the attributes will be modified. Also, not all devices support resizing SRQs. To check if a device supports resizing, check if the IBV_DEVICE_SRQ_RESIZE bit is set in the device capabilities flags.

Modifying the SRQ limit arms the SRQ to produce an IBV_EVENT_SRQ_LIMIT_REACHED 'low watermark' async event once the number of WRs in the SRQ drops below the SRQ limit.

3.4.7 ibv_destroy_srq

Template:

int **ibv_destroy_srq**(struct ibv_srq *srq)

Input Parameters:

`srq`	**`The SRQ to destroy`**

Output Parameters:

`none`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_destroy_srq destroys the specified SRQ. It will fail if any queue pair is still associated with this SRQ.

3.4.8 ibv_open_xrc_domain

Template:

struct **ibv_xrc_domain** *ibv_open_xrc_domain(struct ibv_context *context, int fd, int oflag)

Input Parameters:

`context`	`struct ibv_context from ibv_open_device`
`fd`	`The file descriptor to be associated with the XRC domain`
`oflag`	`The desired file creation attributes`

Output Parameters:

`A file descriptor associated with the opened XRC domain`

Return Value:

`A reference to an opened XRC domain or NULL`

Description:

ibv_open_xrc_domain opens an eXtended Reliable Connection (XRC) domain for the RDMA device context. The desired file creation attributes oflag can either be 0 or the bitwise OR of O_CREAT and O_EXCL. If a domain belonging to the device named by the context is already associated with the inode, then the O_CREAT flag has no effect. If both O_CREAT and O_XCL are set, open will fail if a domain associated with the inode already exists. Otherwise a new XRC domain will be created and associated with the inode specified by fd.

Please note that the check for the existence of the domain and creation of the domain if it does not exist is atomic with respect to other processes executing open with fd naming the same inode.

If fd equals -1, then no inode is associated with the domain, and the only valid value for oflag is O_CREAT.

Since each ibv_open_xrc_domain call increments the xrc_domain object's reference count, each such call must have a corresponding ibv_close_xrc_domain call to decrement the xrc_domain object's reference count.

3.4.9 ibv_create_xrc_srq

Template:

struct ibv_srq ***ibv_create_xrc_srq**(struct ibv_pd *pd,
struct ibv_xrc_domain *xrc_domain,
struct ibv_cq *xrc_cq,
struct ibv_srq_init_attr *srq_init_attr)

Input Parameters:

pd	**The protection domain associated with the shared receive queue**
xrc_domain	**The XRC domain**
xrc_cq	**The CQ which will hold the XRC completion**
srq_init_attr	**A list of initial attributes required to create the SRQ (described above)**

Output Parameters:

ibv_srq_attr	Actual values of the struct are set

Return Value:

A pointer to the created SRQ or NULL on failure

Description:

ibv_create_xrc_srq creates an XRC shared receive queue (SRQ) associated with the protection domain pd, the XRC domain domain_xrc and the CQ which will hold the completion xrc_cq

struct ibv_xrc_domain is defined as follows:

```
struct ibv_xrc_domain {
        struct ibv_context     *context;struct ibv_context from ibv_open_device
        uint64_t                handle;
}
```

3.4.10 ibv_close_xrc_domain

Template:

int **ibv_close_xrc_domain**(struct ibv_xrc_domain *d)

Input Parameters:

`d`	**`A pointer to the XRC domain the user wishes to close`**

Output Parameters:

`none`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_close_xrc_domain closes the XRC domain, d. If this happens to be the last reference, then the XRC domain will be destroyed. This function decrements a reference count and may fail if any QP or SRQ are still associated with the XRC domain being closed.

3.4.11 ibv_create_xrc_rcv_qp

Template:

int ib**v_create_xrc_rcv_qp**(struct ibv_qp_init_attr *init_attr, uint32_t *xrc_rcv_qpn)

Input Parameters:

`init_attr`	`The structure to be populated with QP information`
`xrc_rcv_qpn`	`The QP number associated with the receive QP to be created`

Output Parameters:

`init_attr`	`Populated with the XRC domain information the QP will be associated with`
`xrc_rcv_qpn`	`The QP number associated with the receive QP being created`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_create_xrc_rcv_qp creates an XRC queue pair (QP) to serve as a receive side only QP and returns the QP number through xrc_rcv_qpn. This number must be passed to the remote (sender) node. The remote node will use xrc_rcv_qpn in ibv_post_send when it sends messages to an XRC SRQ on this host in the same xrc domain as the XRC receive QP.

The QP with number xrc_rcv_qpn is created in kernel space and persists until the last process registered for the QP called ibv_unreg_xrc_rcv_qp, at which point the QP is destroyed. The process which creates this QP is automatically registered for it and should also call ibv_unreg_xrc_rcv_qp at some point to unregister.

Any process which wishes to receive on an XRC SRQ via this QP must call ibv_reg_xrc_rcv_qp for this QP to ensure that the QP will not be destroyed while they are still using it.

Please note that because the QP xrc_rcv_qpn is a receive only QP, the send queue in the init_attr struct is ignored.

3.4.12 ibv_modify_xrc_rcv_qp

Template:

int **ibv_modify_xrc_rcv_qp**(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num, struct ibv_qp_attr *attr, int attr_mask)

Input Parameters:

xrc_domain	The XRC domain associated with this QP
xrc_qp_num	The queue pair number to identify this QP
attr	The attributes to use to modify the XRC receive QP
attr_mask	The mask to use for modifying the QP attributes

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_modify_xrc_rcv_qp modifies the attributes of an XRC receive QP with the number xrc_qp_num which is associated with the attributes in the struct attr according to the mask attr_mask. It then moves the QP through the following transitions: Reset->Init->RTR

At least the following masks must be set (the user may add optional attributes as needed)

Next State	Next State Required attributes
Init	IBV_QP_STATE, IBV_QP_PKEY_INDEX, IBV_QP_PORT, IBV_QP_ACCESS_FLAGS
RTR	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

Please note that if any attribute to modify is invalid or if the mask as invalid values, then none of the attributes will be modified, including the QP state.

3.4.13 ibv_reg_xrc_rcv_qp

Template:

int **ibv_reg_xrc_rcv_qp(**struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)

Input Parameters:

xrc_domain	The XRC domain associated with the receive QP
xrc_qp_num	The number associated with the created QP to which the user process is to be registered

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_reg_xrc_rcv_qp registers a user process with the XRC receive QP whose number is xrc_qp_num associated with the XRC domain xrc_domain.

This function may fail if the number xrc_qp_num is not the number of a valid XRC receive QP (for example if the QP is not allocated or it is the number of a non-XRC QP), or the XRC receive QP was created with an XRC domain other than xrc_domain.

3.4.14 ibv_unreg_xrc_rcv_qp

Template:

int **ibv_unreg_xrc_rcv_qp(**struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num)

Input Parameters:

xrc_domain	The XRC domain associated with the XRC receive QP from which the user wishes to unregister
xrc_qp_num	The QP number from which the user process is to be unregistered

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_unreg_xrc_rcv_qp unregisters a user process from the XRC receive QP number xrc_qp_num which is associated with the XRC domain xrc_domain. When the number of user processes registered with this XRC receive QP drops to zero, the QP is destroyed.

3.4.15 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 (UD), the AH is associated with a work request (WR).

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

3.4.16 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, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_destroy_ah frees an address handle (AH). Once an AH is destroyed, it can't be used anymore in UD QPs

3.5 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.

3.5.1 ibv_modify_qp

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, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_modify_qp this verb changes QP attributes and one of those attributes may be the QP state. Its name is a bit of a misnomer, since you cannot 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
```

3.5.2 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 should be posted before the QP can be transitioned to the RTR state.

3.5.3 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.

3.5.4 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.

3.6 Active Queue Pair Operations

A QP can be queried staring at the point it was created and 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.

3.6.1 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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**.

3.6.2 ibv_query_srq

Template:

int **ibv_query_srq**(struct ibv_srq *srq, struct ibv_srq_attr *srq_attr)

Input Parameters:

`srq`	`The SRQ to query`
`srq_attr`	`The attributes of the specified SRQ`

Output Parameters:

`srq_attr`	`The struct ibv_srq_attr is returned with the attributes of the specified SRQ`

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

ibv_query_srq returns the attributes list and current values of the specified SRQ. It returns the attributes through the pointer srq_attr which is an ibv_srq_attr struct described above under ibv_create_srq. If the value of srq_limit in srq_attr is 0, then the SRQ limit reached ('low watermark') event is not or is no longer armed. No asynchronous events will be generated until the event is re-armed.

3.6.3 ibv_query_xrc_rcv_qp

Template:

int **ibv_query_xrc_rcv_qp**(struct ibv_xrc_domain *xrc_domain, uint32_t xrc_qp_num,
struct ibv_qp_attr *attr, int attr_mask,
struct ibv_qp_init_attr *init_attr)

Input Parameters:

`xrc_domain`	`The XRC domain associated with this QP`
`xrc_qp_num`	`The queue pair number to identify this QP`
`attr`	`The ibv_qp_attr struct in which to return the attributes`
`attr_mask`	`A mask specifying the minimum list of attributes to retrieve`
`init_attr`	`The ibv_qp_init_attr struct to return the initial attributes`

Output Parameters:

`attr`	`A pointer to the struct containing the QP attributes of interest`
`init_attr`	`A pointer to the struct containing initial attributes`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_query_xrc_rcv_qp retrieves the attributes specified in attr_mask for the XRC receive QP with the number xrc_qp_num and domain xrc_domain. It returns them through the pointers attr and init_attr.

The attr_mask specifies a minimal list to retrieve. Some RDMA devices may return extra attributes not requested. Attributes are valid if they have been set using the ibv_modify_xrc_rcv_qp. The exact list of valid attributes depends on the QP state. Multiple ibv_query_xrc_rcv_qp calls may yield different returned values for these attributes: qp_state, path_mig_state, sq_draining, ah_attr (if automatic path migration (APM) is enabled).

3.6.4 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, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

ibv_post_recv posts a linked list of WRs to a queue pair's (QP) receive queue. At least one receive buffer should be posted to the receive queue to transition the QP to RTR. Receive buffers are consumed as the remote peer executes Send, Send with Immediate and RDMA Write with Immediate operations. Receive buffers are **NOT** used for other 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**

3.6.5 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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 completion queue entry (CQE) 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) - this is a bitwise OR of the flags. See the details 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 meaningful 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**.

3.6.6 ibv_post_srq_recv

Template:

int **ibv_post_srq_recv**(struct ibv_srq *srq, struct ibv_recv_wr *recv_wr, struct ibv_recv_wr **bad_recv_wr)

Input Parameters:

`srq`	`The SRQ to post the work request to`
`recv_wr`	`A list of work requests to post on the receive queue`

Output Parameters:

`bad_recv_wr`	`pointer to first rejected WR`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_post_srq_recv posts a list of work requests to the specified SRQ. It stops processing the WRs from this list at the first failure (which can be detected immediately while requests are being posted), and returns this failing WR through the bad_recv_wr parameter.

The buffers used by a WR can only be safely reused after WR the request is fully executed and a work completion has been retrieved from the corresponding completion queue (CQ).

If a WR is being posted to a UD QP, the Global Routing Header (GRH) of the incoming message will be placed in the first 40 bytes of the buffer(s) in the scatter list. If no GRH is present in the incoming message, then the first 40 bytes will be undefined. This means that in all cases for UD QPs, the actual data of the incoming message will start at an offset of 40 bytes into the buffer(s) in the scatter list.

3.6.7 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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 there is already a CQE in that CQ, an event won't be generated for this event. If the solicited_only flag is set, then only CQEs for WRs 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**.

3.6.8 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, -1 on error. If the call fails, errno will be set to indicate the reason for the 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 i**bv_ack_cq_events** operation. Since the i**bv_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.

3.6.9 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.

3.6.10 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	IBV_WC_SEND,
	IBV_WC_RDMA_WRITE,
	IBV_WC_RDMA_READ,
	IBV_WC_COMP_SWAP,
	IBV_WC_FETCH_ADD,
	IBV_WC_BIND_MW,
	IBV_WC_RECV= 1 << 7,
	IBV_WC_RECV_RDMA_WITH_IMM
vendor_err	vendor 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

3.6.11 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. T**his 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, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

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.

When the function **ibv_init_ah_from_wc** completes, the ah_attr will be filled in and the 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.

3.6.12 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.

3.6.13 ibv_attach_mcast

Template:

int **ibv_attach_mcast**(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)

Input Parameters:

`qp`	`QP to attach to the multicast group`
`gid`	`The multicast group GID`
`lid`	`The multicast group LID in host byte order`

Output Parameters:

`none`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_attach_mcast attaches the specified QP, qp, to the multicast group whose multicast group GID is gid, and multicast LID is lid.

Only QPs of Transport Service Type IBV_QPT_UD may be attached to multicast groups.

In order to receive multicast messages, a join request for the multicast group must be sent to the subnet administrator (SA), so that the fabric's multicast routing is configured to deliver messages to the local port.

If a QP is attached to the same multicast group multiple times, the QP will still receive a single copy of a multicast message.

3.6.14 ibv_detach_mcast

Template:

int **ibv_detach_mcast**(struct ibv_qp *qp, const union ibv_gid *gid, uint16_t lid)

Input Parameters:

```
qp      QP to attach to the multicast group
gid     The multicast group GID
lid     The multicast group LID in host byte order
```

Output Parameters:

```
none
```

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

ibv_detach_mcast detaches the specified QP, qp, from the multicast group whose multicast group GID is gid, and multicast LID is lid.

3.7 Event Handling Operations

3.7.1 ibv_get_async_event

Template:

int **ibv_get_async_event**(struct ibv_context *context, struct ibv_async_event *event)

Input Parameters:

`context`	`struct ibv_context from ibv_open_device`
`event`	`A pointer to use to return the async event`

Output Parameters:

`event`	`A pointer to the async event being sought`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

ibv_get_async_event gets the next asynchronous event of the RDMA device context 'context' and returns it through the pointer 'event' which is an ibv_async_event struct. All async events returned by **ibv_get_async_event** must eventually be acknowledged with **ibv_ack_async_event**.

ibv_get_async_event() is a blocking function. If multiple threads call this function simultaneously, then when an async event occurs, only one thread will receive it, and it is not possible to predict which thread will receive it.

struct ibv_async_event is defined as follows:

```
struct ibv_async_event {
        union {
                struct ibv_cq *cq;         The CQ that got the event
                struct ibv_qp *qp;         The QP that got the event
                struct ibv_srq *srq;       The SRQ that got the event
                int    port_num;           The port number that got the event
        } element;
        enum ibv_event_type                     event_type; Type of event
};
```

One member of the element union will be valid, depending on the event_type member of the structure. event_type will be one of the following events:

QP events:

IBV_EVENT_QP_FATAL	Error occurred on a QP and it transitioned to error state
IBV_EVENT_QP_REQ_ERR	Invalid Request Local Work Queue Error
IBV_EVENT_QP_ACCESS_ERR	Local access violation error
IBV_EVENT_COMM_EST	Communication was established on a QP
IBV_EVENT_SQ_DRAINED	Send Queue was drained of outstanding messages in progress

IBV_EVENT_PATH_MIG	A connection has migrated to the alternate path
IBV_EVENT_PATH_MIG_ERR	A connection failed to migrate to the alternate path
IBV_EVENT_QP_LAST_WQE_REACHED	Last WQE Reached on a QP associated with an SRQ

CQ events:

IBV_EVENT_CQ_ERR	CQ is in error (CQ overrun)

SRQ events:

IBV_EVENT_SRQ_ERR	Error occurred on an SRQ
IBV_EVENT_SRQ_LIMIT_REACHED	SRQ limit was reached

Port events:

IBV_EVENT_PORT_ACTIVE	Link became active on a port
IBV_EVENT_PORT_ERR	Link became unavailable on a port
IBV_EVENT_LID_CHANGE	LID was changed on a port
IBV_EVENT_PKEY_CHANGE	P_Key table was changed on a port
IBV_EVENT_SM_CHANGE	SM was changed on a port
IBV_EVENT_CLIENT_REREGISTER	SM sent a CLIENT_REREGISTER request to a port
IBV_EVENT_GID_CHANGE	GID table was changed on a port

CA events:

IBV_EVENT_DEVICE_FATAL	CA is in FATAL state

3.7.2 ib_ack_async_event

Template:

void **ibv_ack_async_event(**struct ibv_async_event *event)

Input Parameters:

`event`	`A pointer to the event to be acknowledged`

Output Parameters:

`None`

Return Value:

`None`

Description:

All async events that ibv_get_async_event() returns must be acknowledged using ibv_ack_async_event(). To avoid races, destroying an object (CQ, SRQ or QP) will wait for all affiliated events for the object to be acknowledged; this avoids an application retrieving an affiliated event after the corresponding object has already been destroyed.

3.7.3 ibv_event_type_str

Template:

const char ***ibv_event_type_str**(enum ibv_event_type event_type)

Input Parameters:

event_type	ibv_event_type enum value

Output Parameters:

None

Return Value:

A constant string which describes the enum value event_type

Description:

ibv_event_type_str returns a string describing the event type enum value, event_type. event_type may be any one of the 19 different enum values describing different IB events.

```
ibv_event_type {
        IBV_EVENT_CQ_ERR,
        IBV_EVENT_QP_FATAL,
        IBV_EVENT_QP_REQ_ERR,
        IBV_EVENT_QP_ACCESS_ERR,
        IBV_EVENT_COMM_EST,
        IBV_EVENT_SQ_DRAINED,
        IBV_EVENT_PATH_MIG,
        IBV_EVENT_PATH_MIG_ERR,
        IBV_EVENT_DEVICE_FATAL,
        IBV_EVENT_PORT_ACTIVE,
        IBV_EVENT_PORT_ERR,
        IBV_EVENT_LID_CHANGE,
        IBV_EVENT_PKEY_CHANGE,
        IBV_EVENT_SM_CHANGE,
        IBV_EVENT_SRQ_ERR,
        IBV_EVENT_SRQ_LIMIT_REACHED,
        IBV_EVENT_QP_LAST_WQE_REACHED,
        IBV_EVENT_CLIENT_REREGISTER,
        IBV_EVENT_GID_CHANGE,
};
```

4 RDMA_CM API

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`

Return Value:

`A pointer to the created event channel, or NULL if the request fails. On failure, errno will be set to indicate the failure reason.`

Description:

Opens an event 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_ids, users may find it useful to direct events for different cm_ids 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`

Return Value:

`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

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

Creates an identifier that is used to track communication information.

Notes:

rdma_cm_ids 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, connection less 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
```

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

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_migrate_id

Template:

int **rdma_migrate_id**(struct rdma_cm_id *id, struct rdma_event_channel *channel)

Input Parameters:

id	An existing RDMA communication identifier to migrate
channel	The new event channel for rdma_cm_id events

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_migrate_id migrates a communication identifier to a different event channel and moves any pending events associated with the rdma_cm_id to the new channel.

No polling for events on the rdma_cm_id's current channel nor running of any routines on the rdma_cm_id should be done while migrating between channels. rdma_migrate_id will block while there are any unacknowledged events on the current event channel.

If the channel parameter is NULL, then the specified rdma_cm_id will be placed into synchronous operation mode. All calls on the id will block until the operation completes.

4.2.4 rdma_set_option

Template:

int **rdma_set_option**(struct rdma_cm_id *id, int level, int optname, void *optval, size_t optlen)

Input Parameters:

`id`	`RDMA communication identifier`
`level`	`Protocol level of the option to set`
`optname`	`Name of the option to set`
`optval`	`Reference to the option data`
`optlen`	`The size of the option data (optval) buffer`

Output Parameters:

`None`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_set_option sets communication options for an rdma_cm_id. Option levels and details may be found in the enums in the relevant header files.

4.2.5 rdma_create_ep

Template:

int **rdma_create_ep**(struct rdma_cm_id **id, struct rdma_addrinfo *res, struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr)

Input Parameters:

id	A reference where the allocated communication identifier will be returned
res	Address information associated with the rdma_cm_id returned from rdma_getaddrinfo
pd	OPtional protection domain if a QP is associated with the rdma_cm_id
qp_init_attr	Optional initial QP attributes

Output Parameters:

id	The communication identifier is returned through this reference

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure

Description:

rdma_create_ep creates an identifier and optional QP used to track communication information.

If qp_init_attr is not NULL, then a QP will be allocated and associated with the rdma_cm_id, id. If a protection domain (PD) is provided, then the QP will be created on that PD. Otherwise the QP will be allocated on a default PD.

The rdma_cm_id will be set to use synchronous operations (connect, listen and get_request). To use asynchronous operations, rdma_cm_id must be migrated to a user allocated event channel using rdma_migrate_id.

rdm_cm_id must be released after use, using rdma_destroy_ep.

struct rdma_addrinfo is defined as follows:

```
struct rdma_addrinfo {
        int                                         ai_flags;
        int                                         ai_family;
        int                                         ai_qp_type;
        int                                         ai_port_space;
        socklen_t                                   ai_src_len;
        socklen_t                                   ai_dst_len;
        struct sockaddr                             *ai_src_addr;
        struct sockaddr                             *ai_dst_addr;
        char                                        *ai_src_canonname;
        char                                        *ai_dst_canonname;
        size_t                                      ai_route_len;
        void                                        *ai_route;
        size_t                                      ai_connect_len;
```

```
    void                                    *ai_connect;
    struct rdma_addrinfo                    *ai_next;
};
```

ai_flags	Hint flags which control the operation. Supported flags are: RAI_PASSIVE, RAI_NUMERICHOST and RAI_NOROUTE
ai_family	Address family for the source and destination address (AF_INET, AF_INET6, AF_IB)
ai_qp_type	The type of RDMA QP used
ai_port_space	RDMA port space used (RDMA_PS_UDP or RDMA_PS_TCP)
ai_src_len	Length of the source address referenced by ai_src_addr
ai_dst_len	Length of the destination address referenced by ai_dst_addr
*ai_src_addr	Address of local RDMA device, if provided
*ai_dst_addr	Address of destination RDMA device, if provided
*ai_src_canonname	The canonical for the source
*ai_dst_canonname	The canonical for the destination
ai_route_len	Size of the routing information buffer referenced by ai_route.
*ai_route	Routing information for RDMA transports that require routing data as part of connection establishment
ai_connect_len	Size of connection information referenced by ai_connect
*ai_connect	Data exchanged as part of the connection establishment process
*ai_next	Pointer to the next rdma_addrinfo structure in the list

4.2.6 rdma_destroy_ep

Template:

int **rdma_destroy_ep** (struct rdma_cm_id *id)

Input Parameters:

`id`	`The communication identifier to destroy`

Output Parameters:

`None`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure`

Description:

rdma_destroy_ep destroys the specified rdma_cm_id and all associated resources, including QPs associated with the id.

4.2.7 rdma_resolve_addr

Template:

int r**dma_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`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_resolve_addr resolves 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_create_qp, rdma_get_cm_event, rdma_bind_addr, rdma_get_src_port, rdma_get_dst_port, rdma_get_local_addr, rdma_get_peer_addr

4.2.8 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

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_bind_addr associates a source address with an rdma_cm_id. The address may be wild carded. 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

4.2.9 rdma_resolve_route

Template:

int **rdma_resolve_route** (struct rdma_cm_id *id, int timeout_ms)

Input Parameters:

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

Output Parameters:

`None`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_resolve_route resolves an RDMA route to the destination address in order to establish a connection. The destination must already have been resolved by calling rdma_resolve_addr. Thus this function is called on the client side after rdma_resolve_addr but before calling rdma_connect. For InfiniBand connections, the call obtains a path record which is used by the connection.

4.2.10 rdma_listen

Template:

int **rdma_listen**(struct rdma_cm_id *id, int backlog)

Input Parameters:

`id`	`RDMA communication identifier`
`backlog`	`The backlog of incoming connection requests`

Output Parameters:

`None`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_listen initiates a listen for incoming connection requests or datagram service lookup. The listen is restricted to the locally bound source address.

Please note that the rdma_cm_id must already have been bound to a local address by calling rdma_bind_addr before calling rdma_listen. If the rdma_cm_id is bound to a specific IP address, the listen will be restricted to that address and the associated RDMA device. If the rdma_cm_id is bound to an RDMA port number only, the listen will occur across all RDMA devices.

4.2.11 rdma_connect

Template:

int **rdma_connect**(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)

Input Parameters:

```
id              RDMA communication identifier
conn_param      Optional connection parameters
```

Output Parameters:

```
none
```

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

rdma_connect initiates an active connection request. For a connected rdma_cm_id, id, the call initiates a connection request to a remote destination. or an unconnected rdma_cm_id, it initiates a lookup of the remote QP providing the datagram service. The user must already have resolved a route to the destination address by having called rdma_resolve_route or rdma_create_ep before calling this method.

For InfiniBand specific connections, the QPs are configured with minimum RNR NAK timer and local ACK values. The minimum RNR NAK timer value is set to 0, for a delay of 655 ms. The local ACK timeout is calculated based on the packet lifetime and local HCA ACK delay. The packet lifetime is determined by the InfiniBand Subnet Administrator and is part of the resolved route (path record) information. The HCA ACK delay is a property of the locally used HCA. Retry count and RNR retry count values are 3-bit values.

Connections established over iWarp RDMA devices currently require that the active side of the connection send the first message.

struct rdma_conn_param is defined as follows:

```
struct rdma_conn_param {
      const void *private_data;
      uint8_t private_data_len;
      uint8_t responder_resources;
      uint8_t initiator_depth;
      uint8_t flow_control;
      uint8_t retry_count;                  ignored when accepting
      uint8_t rnr_retry_count;
      uint8_t srq;                          ignored if QP created on the rdma_cm_id
      uint32_t qp_num;                      ignored if QP created on the rdma_cm_id
      };
```

Here is a more detailed description of the rdma_conn_param structure members:

```
private_data        References a user-controlled data buffer. The contents of
                    the buffer are copied and transparently passed to the remote
```

	side as part of the communication request. May be NULL if private_data is not required.
private_data_len	Specifies the size of the user-controlled data buffer. Note that the actual amount of data transferred to the remote side is transport dependent and may be larger than that requested.
responder_resources	The maximum number of outstanding RDMA read and atomic operations that the local side will accept from the remote side. Applies only to RDMA_PS_TCP. This value must be less than or equal to the local RDMA device attribute max_qp_rd_atom and remote RDMA device attribute max_qp_init_rd_atom. The remote endpoint can adjust this value when accepting the connection.
initiator_depth	The maximum number of outstanding RDMA read and atomic operations that the local side will have to the remote side. Applies only to RDMA_PS_TCP. This value must be less than or equal to the local RDMA device attribute max_qp_init_rd_atom and remote RDMA device attribute max_qp_rd_atom. The remote endpoint can adjust this value when accepting the connection.
flow_control	Specifies if hardware flow control is available. This value is exchanged with the remote peer and is not used to configure the QP. Applies only to RDMA_PS_TCP.
retry_count	The maximum number of times that a data transfer operation should be retried on the connection when an error occurs. This setting controls the number of times to retry send, RDMA, and atomic operations when timeouts occur. Applies only to RDMA_PS_TCP.
rnr_retry_count	The maximum number of times that a send operation from the remote peer should be retried on a connection after receiving a receiver not ready (RNR) error. RNR errors are generated when a send request arrives before a buffer has been posted to receive the incoming data. Applies only to RDMA_PS_TCP.
srq	Specifies if the QP associated with the connection is using a shared receive queue. This field is ignored by the library if a QP has been created on the rdma_cm_id. Applies only to RDMA_PS_TCP.
qp_num	Specifies the QP number associated with the connection. This field is ignored by the library if a QP has been created on the rdma_cm_id. Applies only to RDMA_PS_TCP.

4.2.12 rdma_get_request

Template:

int **rdma_get_request** (struct rdma_cm_id *listen, struct rdma_cm_id **id)

Input Parameters:

`listen`	`Listening rdma_cm_id`
`id`	`rdma_cm_id associated with the new connection`

Output Parameters:

`id`	`A pointer to rdma_cm_id associated with the request`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_get_request retrieves the next pending connection request event. The call may only be used on listening rdma_cm_ids operating synchronously. If the call is successful, a new rdma_cm_id (id) representing the connection request will be returned to the user. The new rdma_cm_id will reference event information associated with the request until the user calls rdma_reject, rdma_accept, or rdma_destroy_id on the newly created identifier. For a description of the event data, see rdma_get_cm_event.

If QP attributes are associated with the listening endpoint, the returned rdma_cm_id will also reference an allocated QP.

4.2.13 rdma_get_request

Template:

int **rdma_accept**(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)

Input Parameters:

id	RDMA communication identifier
conn_param	Optional connection parameters (described under rdma_connect)

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_accept is called from the listening side to accept a connection or datagram service lookup request.

Unlike the socket accept routine, rdma_accept is not called on a listening rdma_cm_id. Instead, after calling rdma_listen, the user waits for an RDMA_CM_EVENT_CONNECT_REQUEST event to occur. Connection request events give the user a newly created rdma_cm_id, similar to a new socket, but the rdma_cm_id is bound to a specific RDMA device. rdma_accept is called on the new rdma_cm_id.

4.2.14 rdma_reject

Template:

int **rdma_reject**(struct rdma_cm_id *id, const void *private_data, uint8_t private_data_len)

Input Parameters:

id	RDMA communication identifier
private_data	Optional private data to send with the reject message
private_data_len	Size (in bytes) of the private data being sent

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_reject is called from the listening side to reject a connection or datagram service lookup request.

After receiving a connection request event, a user may call rdma_reject to reject the request. The optional private data will be passed to the remote side if the underlying RDMA transport supports private data in the reject message.

4.2.15 rdma_notify

Template:

int **rdma_notify**(struct rdma_cm_id *id, enum ibv_event_type event)

Input Parameters:

`id`	`RDMA communication identifier`
`event`	`Asynchronous event`

Output Parameters:

`None`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_notify is used to notify the librdmacm of asynchronous events which have occurred on a QP associated with the rdma_cm_id, id.

Asynchronous events that occur on a QP are reported through the user's device event handler. This routine is used to notify the librdmacm of communication events. In most cases, use of this routine is not necessary, however if connection establishment is done out of band (such as done through InfiniBand), it is possible to receive data on a QP that is not yet considered connected. This routine forces the connection into an established state in this case in order to handle the rare situation where the connection never forms on its own. Calling this routine ensures the delivery of the RDMA_CM_EVENT_ESTABLISHED event to the application. Events that should be reported to the CM are: IB_EVENT_COMM_EST.

4.2.16 rdma_disconnect

Template:

int **rdma_disconnect**(struct rdma_cm_id *id)

Input Parameters:

```
id                  RDMA communication identifier
```

Output Parameters:

```
None
```

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

rdma_disconnect disconnects a connection and transitions any associated QP to the error state. This action will result in any posted work requests being flushed to the completion queue. rdma_disconnect may be called by both the client and server side of the connection. After successfully disconnecting, an RDMA_CM_EVENT_DISCONNECTED event will be generated on both sides of the connection.

4.2.17 rdma_get_src_port

Template:

uint16_t **rdma_get_src_port**(struct rdma_cm_id *id)

Input Parameters:

`id`	`RDMA communication identifier`

Output Parameters:

`None`

Return Value:

`Returns the 16-bit port number associated with the local endpoint of 0 if the rdma_cm_id, id, is not bound to a port`

Description:

rdma_get_src_port retrieves the local port number for an rdma_cm_id (id) which has been bound to a local address. If the id is not bound to a port, the routine will return 0.

4.2.18 rdma_get_dst_port

Template:

uint16_t **rdma_get_dst_port**(struct rdma_cm_id *id)

Input Parameters:

```
id                  RDMA communication identifier
```

Output Parameters:

```
None
```

Return Value:

```
Returns the 16-bit port number associated with the peer endpoint of 0 if the
rdma_cm_id, id, is not connected
```

Description:

rdma_get_dst_port retrieves the port associated with the peer endpoint. If the rdma_cm_id, id, is not connected, then the routine will return 0.

4.2.19 rdma_get_local_addr

Template:

struct sockaddr ***rdma_get_local_addr**(struct rdma_cm_id *id)

Input Parameters:

`id` RDMA communication identifier

Output Parameters:

None

Return Value:

Returns a pointer to the local sockaddr address of the rdma_cm_id, id. If the id is not bound to an address, then the contents of the sockaddr structure will be set to all zeros

Description:

rdma_get_local_addr retrieves the local IP address for the rdma_cm_id which has been bound to a local device.

4.2.20 rdma_get_peer_addr

Template:

struct sockaddr * **rdma_get_peer_addr** (struct rdma_cm_id *id)

Input Parameters:

```
id                    RDMA communication identifier
```

Output Parameters:

```
None
```

Return Value:

```
A pointer to the sockaddr address of the connected peer. If the rdma_cm_id is not
connected, then the contents of the sockaddr structure will be set to all zeros
```

Description:

rdma_get_peer_addr retrieves the remote IP address of a bound rdma_cm_id.

4.2.21 rdma_get_devices

Template:

struct ibv_context ** **rdma_get_devices** (int *num_devices)

Input Parameters:

num_devices	If non-NULL, set to the number of devices returned

Output Parameters:

num_devices	Number of RDMA devices currently available

Return Value:

`Array of available RDMA devices on success or NULL if the request fails`

Description:

rdma_get_devices retrieves an array of RDMA devices currently available. Devices remain opened while librdmacm is loaded and the array must be released by calling rdma_free_devices.

4.2.22 rdma_free_devices

Template:

void **rdma_free_devices** (struct ibv_context **list)

Input Parameters:

`list` List of devices returned from rdma_get_devices

Output Parameters:

None

Return Value:

None

Description:

rdma_free_devices frees the device array returned by the rdma_get_devices routine.

4.2.23 rdma_getaddrinfo

Template:

int **rdma_getaddrinfo**(char *node, char *service, struct rdma_addrinfo *hints, struct rdma_addrinfo **res)

Input Parameters:

`node`	`Optional: name, dotted-decimal IPv4 or IPv6 hex address to resolve`
`service`	`The service name or port number of the address`
`hints`	`Reference to an rmda_addrinfo structure containing hints about the type of service the caller supports resA  pointer to a linked list of rdma_addrinfo structures containing response information`

Output Parameters:

`res`	`An rdma_addrinfo structure which returns information needed to establish communication`

Return Value:

```
0 on success, -1 on error. If the call fails, errno will be set to indicate
the reason for the failure.
```

Description:

rdma_getaddrinfo provides transport independent address translation. It resolves the destination node and service address and returns information required to establish device communication. It is the functional equivalent of getaddrinfo.

Please note that either node or service must be provided. If hints are provided, the operation will be controlled by hints.ai_flags. If RAI_PASSIVE is specified, the call will resolve address information for use on the passive side of a connection.

The rdma_addrinfo structure is described under the rdma_create_ep routine.

4.2.24 rdma_freeaddrinfo

Template:

void **rdma_freeaddrinfo**(struct rdma_addrinfo *res)

Input Parameters:

`res` The rdma_addrinfo structure to free

Output Parameters:

None

Return Value:

None

Description:

rdma_freeaddrinfo releases the rdma_addrinfo (res) structure returned by the rdma_getaddrinfo routine. Note that if ai_next is not NULL, rdma_freeaddrinfo will free the entire list of addrinfo structures.

4.2.25 rdma_create_qp

Template:

int r**dma_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`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_create_qp allocates a QP associated with the specified rdma_cm_id and transitions 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

4.2.26 rdma_destroy_qp

Template:

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

Input Parameters:

```
id                  RDMA identifier.
```

Output Parameters:

```
none
```

Return Value:

```
none
```

Description:

rdma_destroy_qp destroys 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

4.2.27 rdma_join_multicast

Template:

int r**dma_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`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_join_multicast 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

4.2.28 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`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_leave_multicast 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

4.3 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.

Output Parameters:

none

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

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 MULL 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 connection request. 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.`

Output Parameters:

`none`

Return Value:

`0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.`

Description:

rdma_ack_cm_event frees 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** (enum rdma_cm_event_type event)

Input Parameters:

`event`	`Asynchronous event.`

Output Parameters:

`none`

Return Value:

`A pointer to a static character string corresponding to the event`

Description:

rdma_event_str returns a string representation of an asynchronous event.

See Also:

rdma_get_cm_event

5 RDMA Verbs API

5.1 Protection Domain Operations

5.1.1 rdma_reg_msgs

Template:

struct ibv_mr ***rdma_reg_msgs**(struct rdma_cm_id *id, void *addr, size_t length)

Input Parameters:

id	**A reference to the communication identifier where the message buffer(s) will be used**
addr	**The address of the memory buffer(s) to register**
length	**The total length of the memory to register**

Output Parameters:

ibv_mr	A reference to an ibv_mr struct of the registered memory region

Return Value:

A reference to the registered memory region on success or NULL on failure

Description:

rdma_reg_msgs registers an array of memory buffers for sending or receiving messages or for RDMA operations. The registered memory buffers may then be posted to an rdma_cm_id using rdma_post_send or rdma_post_recv. They may also be specified as the target of an RDMA read operation or the source of an RDMA write request.

The memory buffers are registered with the protection domain associated with the rdma_cm_id. The start of the data buffer array is specified through the addr parameter and the total size of the array is given by the length.

All data buffers must be registered before being posted as a work request. They must be deregistered by calling rdma_dereg_mr.

5.1.2 rdma_reg_read

Template:

struct ibv_mr * **rdma_reg_read**(struct rdma_cm_id *id, void *addr, size_t length)

Input Parameters:

id	**A reference to the communication identifier where the message buffer(s) will be used**
addr	**The address of the memory buffer(s) to register**
length	**The total length of the memory to register**

Output Parameters:

ibv_mr	A reference to an ibv_mr struct of the registered memory region

Return Value:

A reference to the registered memory region on success or NULL on failure. If an error occurs, errno will be set to indicate the failure reason.

Description:

rdma_reg_read Registers a memory buffer that will be accessed by a remote RDMA read operation. Memory buffers registered using rdma_reg_read may be targeted in an RDMA read request, allowing the buffer to be specified on the remote side of an RDMA connection as the remote_addr of rdma_post_read, or similar call.

rdma_reg_read is used to register a data buffer that will be the target of an RDMA read operation on a queue pair associated with an rdma_cm_id. The memory buffer is registered with the protection domain associated with the identifier. The start of the data buffer is specified through the addr parameter, and the total size of the buffer is given by length.

All data buffers must be registered before being posted as work requests. Users must deregister all registered memory by calling the rdma_dereg_mr.

See Also

rdma_cm(7), rdma_create_id(3), rdma_create_ep(3), rdma_reg_msgs(3), rdma_reg_write(3), ibv_reg_mr(3), ibv_dereg_mr(3), rdma_post_read(3)

5.1.3 rdma_reg_write

Template:

struct ibv_mr ***rdma_reg_write**(struct rdma_cm_id *id, void *addr, size_t length)

Input Parameters:

id	**A reference to the communication identifier where the message buffer(s) will be used**
addr	**The address of the memory buffer(s) to register**
length	**The total length of the memory to register**

Output Parameters:

ibv_mr	A reference to an ibv_mr struct of the registered memory region

Return Value:

A reference to the registered memory region on success or NULL on failure. If an error occurs, errno will be set to indicate the failure reason.

Description:

rdma_reg_write registers a memory buffer which will be accessed by a remote RDMA write operation. Memory buffers registered using this routine may be targeted in an RDMA write request, allowing the buffer to be specified on the remote side of an RDMA connection as the remote_addr of an rdma_post_write or similar call.

The memory buffer is registered with the protection domain associated with the rdma_cm_id. The start of the data buffer is specified through the addr parameter, and the total size of the buffer is given by the length.

All data buffers must be registered before being posted as work requests. Users must deregister all registered memory by calling the rdma_dereg_mr.

See Also

rdma_cm(7), rdma_create_id(3), rdma_create_ep(3), rdma_reg_msgs(3), rdma_reg_read(3), ibv_reg_mr(3), ibv_dereg_mr(3), rdma_post_write(3)

5.1.4 rdma_dereg_mr

Template:

int **rdma_dereg_mr**(struct ibv_mr *mr)

Input Parameters:

mr	**A reference to a registered memory buffer**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_dereg_mr deregisters a memory buffer which has been registered for RDMA or message operations. This routine must be called for all registered memory associated with a given rdma_cm_id before destroying the rdma_cm_id.

5.1.5 rdma_create_srq

Template:

int **rdma_create_srq**(struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_srq_init_attr *attr)

Input Parameters:

id	**The RDMA communication identifier**
pd	**Optional protection domain for the shared request queue (SRQ)**
attr	**Initial SRQ attributes**

Output Parameters:

attr	The actual capabilities and properties of the created SRQ are returned through this structure

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_create_srq allocates a shared request queue associated with the rdma_cm_id, id. The id must be bound to a local RMDA device before calling this routine. If the protection domain, pd, is provided, it must be for that same device. After being allocated, the SRQ will be ready to handle posting of receives. If a pd is NULL, then the rdma_cm_id will be created using a default protection domain. One default protection domain is allocated per RDMA device. The initial SRQ attributes are specified by the attr parameter.

If a completion queue, CQ, is not specified for the XRC SRQ, then a CQ will be allocated by the rdma_cm for the SRQ, along with corresponding completion channels. Completion channels and CQ data created by the rdma_cm are exposed to the user through the rdma_cm_id structure. The actual capabilities and properties of the created SRQ will be returned to the user through the attr parameter.

An rdma_cm_id may only be associated with a single SRQ.

5.1.6 rdma_destroy_srq

Template:

void **rdma_destroy_srq**(struct rdma_cm_id *id)

Input Parameters:

`id`	**The RDMA communication identifier whose associated SRQ we wish to destroy.**

Output Parameters:

None

Return Value:

none

Description:

rdma_destroy_srq destroys an SRQ allocated on the rdma_cm_id, id. Any SRQ associated with an rdma_cm_id must be destroyed before destroying the rdma_cm_id, id.

5.2 Active Queue Pair Operations

5.2.1 rdma_post_recvv

Template:

int **rdma_post_recvv(**struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge)

Input Parameters:

id	**A reference to the communication identifier where the message buffer(s) will be posted**
context	**A user-defined context associated with the request**
sgl	**A scatter-gather list of memory buffers posted as a single request**
nsge	**The number of scatter-gather entries in the sgl array**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_recvv posts a single work request to the receive queue of the queue pair associated with the rdma_cm_id, id. The posted buffers will be queued to receive an incoming message sent by the remote peer.

Please note that this routine supports multiple scatter-gather entries. The user is responsible for ensuring that the receive is posted, and the total buffer space is large enough to contain all sent data before the peer posts the corresponding send message. The message buffers must have been registered before being posted, and the buffers must remain registered until the receive completes.

Messages may be posted to an rdma_cm_id only after a queue pair has been associated with it. A queue pair is bound to an rdma_cm_id after calling rdma_create_ep or rdma_create_qp, if the rdma_cm_id is allocated using rdma_create_id.

The user-defined context associated with the receive request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.2 rdma_post_sendv

Template:

int **rdma_post_sendv**(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags)

Input Parameters:

id	**A reference to the communication identifier where the message buffer will be posted**
context	**A user-defined context associated with the request**
sgl	**A scatter-gather list of memory buffers posted as a single request**
nsge	**The number of scatter-gather entries in the sgl array**
flags	**Optional flags used to control the send operation**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_sendv posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the posted buffers will be sent to the remote peer of the connection.

The user is responsible for ensuring that the remote peer has queued a receive request before issuing the send operations. Also, unless the send request is using inline data, the message buffers must already have been registered before being posted. The buffers must remain registered until the send completes.

This routine supports multiple scatter-gather entries.

Send operations may not be posted to an rdma_cm_id or the corresponding queue pair until a connection has been established.

The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.3 rdma_post_readv

Template:

int **rdma_post_readv**(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:

id	**A reference to the communication identifier where the request will be posted**
context	**A user-defined context associated with the request**
sgl	**A scatter-gather list of the destination buffers of the read**
nsge	**The number of scatter-gather entries in the sgl array**
flags	**Optional flags used to control the read operation**
remote_addr	**The address of the remote registered memory to read from**
rkey	**The registered memory key associated with the remote address**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_readv posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the remote memory region at remote_addr will be read into the local data buffers given in the sgl array.

The user must ensure that both the remote and local data buffers have been registered before the read is issued. The buffers must remain registered until the read completes.

Read operations may not be posted to an rdma_cm_id or the corresponding queue pair until a connection has been established.

The user-defined context associated with the read request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.4 rdma_post_writev

Template:

int **rdma_post_writev**(struct rdma_cm_id *id, void *context, struct ibv_sge *sgl, int nsge, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:

id	**A reference to the communication identifier where the request will be posted**
context	**A user-defined context associated with the request**
sgl	**A scatter-gather list of the source buffers of the write**
nsge	**The number of scatter-gather entries in the sgl array**
flags	**Optional flags used to control the write operation**
remote_addr	**The address of the remote registered memory to write into**
rkey	**The registered memory key associated with the remote address**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_writev posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the local data buffers in the sgl array will be written to the remote memory region at remote_addr.

Unless inline data is specified, the local data buffers must have been registered before the write is issued, and the buffers must remain registered until the write completes. The remote buffers must always be registered.

Write operations may not be posted to an rdma_cm_id or the corresponding queue pair until a connection has been established.

The user-defined context associated with the write request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.5 rdma_post_recv

Template:

int **rdma_post_recv**(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr)

Input Parameters:

id	**A reference to the communication identifier where the message buffer will be posted**
context	**A user-defined context associated with the request**
addr	**The address of the memory buffer to post**
length	**The length of the memory buffer**
mr	**A registered memory region associated with the posted buffer**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_recv posts a work request to the receive queue of the queue pair associated with the rdma_cm_id, id. The posted buffer will be queued to receive an incoming message sent by the remote peer.

The user is responsible for ensuring that receive buffer is posted and is large enough to contain all sent data before the peer posts the corresponding send message. The buffer must have already been registered before being posted, with the mr parameter referencing the registration. The buffer must remain registered until the receive completes.

Messages may be posted to an rdma_cm_id only after a queue pair has been associated with it. A queue pair is bound to an rdma_cm_id after calling rdma_create_ep or rdma_create_qp, if the rdma_cm_id is allocated using rdma_create_id.

The user-defined context associated with the receive request will be returned to the user through the work completion request identifier (wr_id) field.

Please note that this is a simple receive call. There are no scatter-gather lists involved here.

5.2.6 rdma_post_send

Template:

int **rdma_post_send**(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags)

Input Parameters:

id	A reference to the communication identifier where the message buffer will be posted
context	A user-defined context associated with the request
addr	The address of the memory buffer to post
length	The length of the memory buffer
mr	Optional registered memory region associated with the posted buffer
flags	Optional flags used to control the send operation

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_send posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the posted buffer will be sent to the remote peer of the connection.

The user is responsible for ensuring that the remote peer has queued a receive request before issuing the send operations. Also, unless the send request is using inline data, the message buffer must already have been registered before being posted, with the mr parameter referencing the registration. The buffer must remain registered until the send completes.

Send operations may not be posted to an rdma_cm_id or the corresponding queue pair until a connection has been established.

The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.7 rdma_post_read

Template:

int **rdma_post_read**(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:

id	A reference to the communication identifier where the request will be posted
context	A user-defined context associated with the request
addr	The address of the local destination of the read request
length	The length of the read operation
mr	Registered memory region associated with the local buffer
flags	Optional flags used to control the read operation
remote_addr	The address of the remote registered memory to read from
rkey	The registered memory key associated with the remote address

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_read posts a work request to the send queue of the queue pair associated with the rdma_cm_id. The contents of the remote memory region will be read into the local data buffer.

For a list of supported flags, see ibv_post_send. The user must ensure that both the remote and local data buffers must have been registered before the read is issued, and the buffers must remain registered until the read completes.

Read operations may not be posted to an rdma_cm_id or the corresponding queue pair until it has been connected.

The user-defined context associated with the read request will be returned to the user through the work completion wr_id, work request identifier, field.

5.2.8 rdma_post_write

Template:

int **rdma_post_write**(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, uint64_t remote_addr, uint32_t rkey)

Input Parameters:

id	**A reference to the communication identifier where the request will be posted**
context	**A user-defined context associated with the request**
addr	**The local address of the source of the write request**
length	**The length of the write operation**
mr	**Optional registered memory region associated with the local buffer**
flags	**Optional flags used to control the write operation**
remote_addr	**The address of the remote registered memory to write into**
rkey	**The registered memory key associated with the remote address**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_write posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the local data buffer will be written into the remote memory region.

Unless inline data is specified, the local data buffer must have been registered before the write is issued, and the buffer must remain registered until the write completes. The remote buffer must always be registered.

Write operations may not be posted to an rdma_cm_id or the corresponding queue pair until a connection has been established.

The user-defined context associated with the write request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.9 rdma_post_ud_send

Template:

int **rdma_post_ud_send**(struct rdma_cm_id *id, void *context, void *addr, size_t length, struct ibv_mr *mr, int flags, struct ibv_ah *ah, uint32_t remote_qpn)

Input Parameters:

id	**A reference to the communication identifier where the request will be posted**
context	**A user-defined context associated with the request**
addr	**The address of the memory buffer to post**
length	**The length of the memory buffer**
mr	**Optional registered memory region associated with the posted buffer**
flags	**Optional flags used to control the send operation**
ah	**An address handle describing the address of the remote node**
remote_qpn	**The destination node's queue pair number**

Output Parameters:

None

Return Value:

0 on success, -1 on error. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_post_ud_send posts a work request to the send queue of the queue pair associated with the rdma_cm_id, id. The contents of the posted buffer will be sent to the specified destination queue pair, remote_qpn.

The user is responsible for ensuring that the destination queue pair has queued a receive request before issuing the send operations. Unless the send request is using inline data, the message buffer must have been registered before being posted, with the mr parameter referencing the registration. The buffer must remain registered until the send completes.

The user-defined context associated with the send request will be returned to the user through the work completion work request identifier (wr_id) field.

5.2.10 rdma_get_send_comp

Template:

int **rdma_get_send_comp**(struct rdma_cm_id *id, struct ibv_wc *wc)

Input Parameters:

id	**A reference to the communication identifier to check for completions**
wc	**A reference to a work completion structure to fill in**

Output Parameters:

wc	A reference to a work completion structure. The structure will contain information about the completed request when routine returns

Return Value:

A non-negative value (0 or 1) equal to the number of completions found on success, or -1 on failure. If the call fails, errno will be set to indicate the reason for the failure.

Description:

rdma_get_send_comp retrieves a completed work request for a send, RDMA read or RDMA write operation. Information about the completed request is returned through the ibv_wc, wc parameter, with the wr_id set to the context of the request. Please see ibv_poll_cq for details on the work completion structure, ibv_wc.

Please note that this call polls the send completion queue associated with the rdma_cm_id, id. If a completion is not found, the call blocks until a request completes. This means, therefore, that the call should only be used on rdma_cm_ids which do not share CQs with other rdma_cm_ids, and maintain separate CQs for sends and receive completions.

5.2.11 rdma_get_recv_comp

Template:

int **rdma_get_recv_comp**(struct rdma_cm_id *id, struct ibv_wc *wc)

Input Parameters:

id	**A reference to the communication identifier to check for completions**
wc	**A reference to a work completion structure to fill in**

Output Parameters:

wc	A reference to a work completion structure. The structure will contain information about the completed request when routine returns

Return Value:

A non-negative value equal to the number of completions found on success, or errno on failure

Description:

rdma_get_recv_comp retrieves a completed work request a receive operation. Information about the completed request is returned through the ibv_wc, wc parameter, with the wr_id set to the context of the request. Please see ibv_poll_cq for details on the work completion structure, ibv_wc.

Please note that this call polls the receive completion queue associated with the rdma_cm_id, id. If a completion is not found, the call blocks until a request completes. This means, therefore, that the call should only be used on rdma_cm_ids which do not share CQs with other rdma_cm_ids, and maintain separate CQs for sends and receive completions.

6 Events

This chapter describes the details of the events that occur when using the VPI API

6.1 IBV Events

6.1.1 IBV_EVENT_CQ_ERR

This event is triggered when a Completion Queue (CQ) overrun occurs or (rare condition) due to a protection error. When this happens, there are no guarantees that completions from the CQ can be pulled. All of the QPs associated with this CQ either in the Read or Send Queue will also get the IBV_EVENT_QP_FATAL event. When this event occurs, the best course of action is for the user to destroy and recreate the resources.

6.1.2 IBV_EVENT_QP_FATAL

This event is generated when an error occurs on a Queue Pair (QP) which prevents the generation of completions while accessing or processing the Work Queue on either the Send or Receive Queues. The user must modify the QP state to Reset for recovery. It is the responsibility of the software to ensure that all error processing is completed prior to calling the modify QP verb to change the QP state to Reset.

If the problem that caused this event is in the CQ of that Work Queue, the appropriate CQ will also receive the IBV_EVENT_CQ_ERR event. In the event of a CQ error, it is best to destroy and recreate the resources

6.1.3 IBV_EVENT_QP_REQ_ERR

This event is generated when the transport layer of the RDMA device detects a transport error violation on the responder side. The error may be caused by the use of an unsupported or reserved opcode, or the use of an out of sequence opcode.

These errors are rare but may occur when there are problems in the subnet or when an RDMA device sends illegal packets.

When this happens, the QP is automatically transitioned to the IBV_QPS_ERR state by the RDMA device. The user must modify the states of any such QPs from the error state to the Reset state for recovery

This event applies only to RC QPs.

6.1.4 IBV_EVENT_QP_ACCESS_ERR

This event is generated when the transport layer of the RDMA device detects a request error violation on the responder side. The error may be caused by

Misaligned atomic request
Too many RDMA Read or Atomic requests

R_Key violation
Length errors without immediate data

These errors usually occur because of bugs in the user code.

When this happens, the QP is automatically transitioned to the IBV_QPS_ERR state by the RDMA device.The user must modify the QP state to Reset for recovery.

This event is relevant only to RC QPs.

6.1.5 IBV_EVENT_COMM_EST

This event is generated when communication is established on a given QP. This event implies that a QP whose state is IBV_QPS_RTR has received the first packet in its Receive Queue and the packet was processed without error.

This event is relevant only to connection oriented QPs (RC and UC QPs). It may be generated for UD QPs as well but that is driver implementation specific.

6.1.6 IBV_EVENT_SQ_DRAINED

This event is generated when all outstanding messages have been drained from the Send Queue (SQ) of a QP whose state has now changed from IBV_QPS_RTS to IBV_QPS_SQD. For RC QPs, this means that all the messages received acknowledgements as appropriate.

Generally, this event will be generated when the internal QP state changes from SQD.draining to SQD.drained. The event may also be generated if the transition to the state IBV_QPS_SQD is aborted because of a transition, either by the RDMA device or by the user, into the IBV_QPS_SQE, IBV_QPS_ERR or IBV_QPS_RESET QP states.

After this event, and after ensuring that the QP is in the IBV_QPS_SQD state, it is safe for the user to start modifying the Send Queue attributes since there aren't are no longer any send messages in progress. Thus it is now safe to modify the operational characteristics of the QP and transition it back to the fully operational RTS state.

6.1.7 IBV_EVENT_PATH_MIG

This event is generated when a connection successfully migrates to an alternate path. The event is relevant only for connection oriented QPs, that is, it is relevant only for RC and UC QPs.

When this event is generated, it means that the alternate path attributes are now in use as the primary path attributes. If it is necessary to load attributes for another alternate path, the user may do that after this event is generated.

6.1.8 IBV_EVENT_PATH_MIG_ERR

This event is generated when an error occurs which prevents a QP which has alternate path attributes loaded from performing a path migration change. The attempt to effect the path migration may have been attempted automatically by the RDMA device or explicitly by the user.

This error usually occurs if the alternate path attributes are not consistent on the two ends of the connection. It could be, for example, that the DLID is not set correctly or if the source port is invalid.CQ The event may also occur if a cable to the alternate port is unplugged.

6.1.9 IBV_EVENT_DEVICE_FATAL

This event is generated when a catastrophic error is encountered on the channel adapter. The port and possibly the channel adapter becomes unusable.

When this event occurs, the behavior of the RDMA device is undetermined and it is highly recommended to close the process immediately. Trying to destroy the RDMA resources may fail and thus the device may be left in an unstable state.

6.1.10 IBV_EVENT_PORT_ACTIVE

This event is generated when the link on a given port transitions to the active state. The link is now available for send/receive packets.

This event means that the port_attr.state has moved from one of the following states

IBV_PORT_DOWN
IBV_PORT_INIT
IBV_PORT_ARMED

to either

IBV_PORT_ACTIVE
IBV_PORT_ACTIVE_DEFER

This might happen for example when the SM configures the port.

The event is generated by the device only if the IBV_DEVICE_PORT_ACTIVE_EVENT attribute is set in the dev_cap.device_cap_flags.

6.1.11 IBV_EVENT_PORT_ERR

This event is generated when the link on a given port becomes inactive and is thus unavailable to send/receive packets.

The port_attr.state must have been in either in either IBV_PORT_ACTIVE or IBV_PORT_ACTIVE_DEFER state and transitions to one of the following states:

IBV_PORT_DOWN
IBV_PORT_INIT
IBV_PORT_ARMED

This can happen when there are connectivity problems within the IB fabric, for example when a cable is accidentally pulled.

This will not affect the QPs associated with this port, although if this is a reliable connection, the retry count may be exceeded if the link takes a long time to come back up.

6.1.12 IBV_EVENT_LID_CHANGE

The event is generated when the LID on a given port changes. This is done by the SM. If this is not the first time that the SM configures the port LID, it may indicate that there is a new SM on the subnet or that the SM has reconfigured the subnet. If the user cached the structure returned from ibv_query_port(), then these values must be flushed when this event occurs.

6.1.13 IBV_EVENT_PKEY_CHANGE

This event is generated when the P_Key table changes on a given port. The PKEY table is configured by the SM and this also means that the SM can change it. When that happens, an IBV_EVENT_PKEY_CHANGE event is generated.

Since QPs use GID table indexes rather than absolute values (as the source GID), it is suggested for clients to check that the GID indexes used by the client's QPs are not changed as a result of this event.

If a user caches the values of the P_Key table, then these must be flushed when the IBV_EVENT_PKEY_CHANGE event is received.

6.1.14 IBV_EVENT_SM_CHANGE

This event is generated when the SM being used at a given port changes. The user application must re-register with the new SM. This means that all subscriptions previously registered from the given port, such as one to join a multicast group, must be reregistered.

6.1.15 IBV_EVENT_SRQ_ERR

This event is generated when an error occurs on a Shared Receive Queue (SRQ) which prevents the RDMA device from dequeuing WRs from the SRQ and reporting of receive completions.

When an SRQ experiences this error, all the QPs associated with this SRQ will be transitioned to the IBV_QPS_ERR state and the IBV_EVENT_QP_FATAL asynchronous event will be generated for them. Any QPs which have transitioned to the error state must have their state modified to Reset for recovery.

6.1.16 IBV_EVENT_SRQ_LIMIT_REACHED

This event is generated when the limit for the SRQ resources is reached. This means that the number of SRQ Work Requests (WRs) is less than the SRQ limit. This event may be used by the user as an indicator that more WRs need to be posted to the SRQ and rearm it.

6.1.17 IBV_EVENT_QP_LAST_WQE_REACHED

This event is generated when a QP which is associated with an SRQ is transitioned into the IBV_QPS_ERR state either automatically by the RDMA device or explicitly by the user. This may have happened either because a completion with error was generated for the last WQE, or the QP transitioned into the IBV_QPS_ERR state and there are no more WQEs on the Receive Queue of the QP.

This event actually means that no more WQEs will be consumed from the SRQ by this QP.

If an error occurs to a QP and this event is not generated, the user must destroy all of the QPs associated with this SRQ as well as the SRQ itself in order to reclaim all of the WQEs associated with the offending QP. At the minimum, the QP which is in the error state must have its state changed to Reset for recovery.

6.1.18 IBV_EVENT_CLIENT_REREGISTER

This event is generated when the SM sends a request to a given port for client reregistration for all subscriptions previously requested for the port. This could happen if the SM suffers a failure and as a result, loses its own records of the subscriptions. It may also happen if a new SM becomes operational on the subnet.

The event will be generated by the device only if the bit that indicates a client reregister is supported is set in port_attr.port_cap_flags.

6.1.19 IBV_EVENT_GID_CHANGE

This event is generated when a GID changes on a given port. The GID table is configured by the SM and this also means that the SM can change it. When that happens, an IBV_EVENT_GID_CHANGE event is generated. If a user caches the values of the GID table, then these must be flushed when the IBV_EVENT_GID_CHANGE event is received.

6.2 IBV WC Events

6.2.1 IBV_WC_SUCCESS

The Work Request completed successfully.

6.2.2 IBV_WC_LOC_LEN_ERR

This event is generated when the receive buffer is smaller than the incoming send. It is generated on the receiver side of the connection.

6.2.3 IBV_WC_LOC_QP_OP_ERR

This event is generated when a QP error occurs. For example, it may be generated if a user neglects to specify responder_resources and initiator_depth values in struct rdma_conn_param before calling rdma_connect() on the client side and rdma_accept() on the server side.

6.2.4 IBV_WC_LOC_EEC_OP_ERR

This event is generated when there is an error related to the local EEC's receive logic while executing the request packet. The responder is unable to complete the request. This error is not caused by the sender.

6.2.5 IBV_WC_LOC_PROT_ERR

This event is generated when a user attempts to access an address outside of the registered memory region. For example, this may happen if the Lkey does not match the address in the WR.

6.2.6 IBV_WC_WR_FLUSH_ERR

This event is generated when an invalid remote error is thrown when the responder detects an invalid request. It may be that the operation is not supported by the request queue or there is insufficient buffer space to receive the request.

6.2.7 IBV_WC_MW_BIND_ERR

This event is generated when a memory management operation error occurs. The error may be due to the fact that the memory window and the QP belong to different protection domains. It may also be that the memory window is not allowed to be bound to the specified MR or the access permissions may be wrong.

6.2.8 IBV_WC_BAD_RESP_ERR

This event is generated when an unexpected transport layer opcode is returned by the responder.

6.2.9 IBV_WC_LOC_ACCESS_ERR

This event is generated when a local protection error occurs on a local data buffer during the process of an RDMA Write with Immediate Data operation sent from the remote node.

6.2.10 IBV_WC_REM_INV_REQ_ERR

This event is generated when the receive buffer is smaller than the incoming send. It is generated on the sender side of the connection. It may also be generated if the QP attributes are not set correctly, particularly those governing MR access.

6.2.11 IBV_WC_REM_ACCESS_ERR

This event is generated when a protection error occurs on a remote data buffer to be read by an RDMA Read, written by an RDMA Write or accessed by an atomic operation. The error is reported only on RDMA operations or atomic operations.

6.2.12 IBV_WC_REM_OP_ERR

This event is generated when an operation cannot be completed successfully by the responder. The failure to complete the operation may be due to QP related errors which prevent the responder from completing the request or a malformed WQE on the Receive Queue.

6.2.13 IBV_WC_RETRY_EXC_ERR

This event is generated when a sender is unable to receive feedback from the receiver. This means that either the receiver just never ACKs sender messages in a specified time period, or it has been disconnected or it is in a bad state which prevents it from responding.

6.2.14 IBV_WC_RNR_RETRY_EXC_ERR

This event is generated when the RNR NAK retry count is exceeded. This may be caused by lack of receive buffers on the responder side.

6.2.15 IBV_WC_LOC_RDD_VIOL_ERR

This event is generated when the RDD associated with the QP does not match the RDD associated with the EEC.

6.2.16 IBV_WC_REM_INV_RD_REQ_ERR

This event is generated when the responder detects an invalid incoming RD message. The message may be invalid because it has in invalid Q_Key or there may be a Reliable Datagram Domain (RDD) violation.

6.2.17 IBV_WC_REM_ABORT_ERR

This event is generated when an error occurs on the responder side which causes it to abort the operation.

6.2.18 IBV_WC_INV_EECN_ERR

This event is generated when an invalid End to End Context Number (EECN) is detected.

6.2.19 IBV_WC_INV_EEC_STATE_ERR

This event is generated when an illegal operation is detected in a request for the specified EEC state.

6.2.20 IBV_WC_FATAL_ERR

This event is generated when a fatal transport error occurs. The user may have to restart the RDMA device driver or reboot the server to recover from the error.

6.2.21 IBV_WC_RESP_TIMEOUT_ERR

This event is generated when the responder is unable to respond to a request within the timeout period. It generally indicates that the receiver is not ready to process requests.

6.2.22 IBV_WC_GENERAL_ERR

This event is generated when there is a transport error which cannot be described by the other specific events discussed here.

6.3 RDMA_CM Events

6.3.1 RDMA_CM_EVENT_ADDR_RESOLVED

This event is generated on the client (active) side in response to rdma_resolve_addr(). It is generated when the system is able to resolve the server address supplied by the client.

6.3.2 RDMA_CM_EVENT_ADDR_ERROR

This event is generated on the client (active) side. It is generated in response to rdma_resolve_addr() in the case where an error occurs. This may happen, for example, if the device cannot be found such as when a user supplies an incorrect device. Specifically, if the remote device has both ethernet and IB interfaces, and the client side supplies the ethernet device name instead of the IB device name of the server side, an RDMA_CM_EVENT_ADDR_ERROR will be generated.

6.3.3 RDMA_CM_EVENT_ROUTE_RESOLVED

This event is generated on the client (active) side in response to rdma_resolve_route(). It is generated when the system is able to resolve the server address supplied by the client.

6.3.4 RDMA_CM_EVENT_ROUTE_ERROR

This event is generated when rdma_resolve_route() fails.

6.3.5 RDMA_CM_EVENT_CONNECT_REQUEST

This is generated on the passive side of the connection to notify the user of a new connection request. It indicates that a connection request has been received.

6.3.6 RDMA_CM_EVENT_CONNECT_RESPONSE

This event may be generated on the active side of the connection to notify the user that the connection request has been successful. The event is only generated on rdma_cm_ids which do not have a QP associated with them.

6.3.7 RDMA_CM_EVENT_CONNECT_ERROR

This event may be generated on the active or passive side of the connection. It is generated when an error occurs while attempting to establish a connection.

6.3.8 RDMA_CM_EVENT_UNREACHABLE

This event is generated on the active side of a connection. It indicates that the (remote) server is unreachable or unable to respond to a connection request.

6.3.9 RDMA_CM_EVENT_REJECTED

This event may be generated on the client (active) side and indicates that a connection request or response has been rejected by the remote device. This may happen for example if an attempt is made to connect with the remote end point on the wrong port.

6.3.10 RDMA_CM_EVENT_ESTABLISHED

This event is generated on both sides of a connection. It indicates that a connection has been established with the remote end point.

6.3.11 RDMA_CM_EVENT_DISCONNECTED

This event is generated on both sides of the connection in response to rdma_disconnect(). The event will be generated to indicate that the connection between the local and remote devices has been disconnected. Any associated QP will transition to the error state. All posted work requests are flushed. The user must change any such QP's state to Reset for recovery.

6.3.12 RDMA_CM_EVENT_DEVICE_REMOVAL

This event is generated when the RDMA CM indicates that the device associated with the rdma_cm_id has been removed. Upon receipt of this event, the user must destroy the related rdma_cm_id.

6.3.13 RDMA_CM_EVENT_MULTICAST_JOIN

This event is generated in response to rdma_join_multicast(). It indicates that the multicast join operation has completed successfully.

6.3.14 RDMA_CM_EVENT_MULTICAST_ERROR

This event is generated when an error occurs while attempting to join a multicast group or on an existing multicast group if the group had already been joined. When this happens, the multicast group will no longer be accessible and must be rejoined if necessary.

6.3.15 RDMA_CM_EVENT_ADDR_CHANGE

This event is generated when the network device associated with this ID through address resolution changes its hardware address. For example, this may happen following bonding fail over. This event may serve to aid applications which want the links used for their RDMA sessions to align with the network stack.

6.3.16 RDMA_CM_EVENT_TIMEWAIT_EXIT

This event is generated when the QP associated with the 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.

7 Programming Examples Using IBV Verbs

This chapter provides code examples using the IBV Verbs

7.1 Synopsis for RDMA_RC Example Using IBV Verbs

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

7.1.1 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.

7.1.2 print_config

Print out configuration information.

7.1.3 resources_init

Clears resources struct.

7.1.4 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.

7.1.5 sock_connect

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

If server, listen for incoming connection on indicated port.

7.1.6 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

7.1.7 modify_qp_to_init

Transition QP to INIT state.

7.1.8 post_receive

Prepare a scatter/gather entry for the receive buffer.

Prepare an RR.

Post the RR.

7.1.9 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.

7.1.10 modify_qp_to_rtr

Transition QP to RTR state.

7.1.11 modify_qp_to_rts

Transition QP to RTS state.

7.1.12 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.

7.1.13 poll_completion

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

7.1.14 resources_destroy

Release/free/deallocate all items in resource struct.

7.2 Code for 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
 *
*/
/******************************************************************************
*
*        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 its 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
******************************************************************************/

/* poll_completion */
/******************************************************************************
* 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 its 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;
}
```

7.3 Synopsis for Multicast Example Using RDMA_CM and IBV Verbs

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. However if RDMA_CM is used, it is required that the network interface will be configured and up (whether it is used over RoCE or over IB).
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.

7.3.1 Main

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

7.3.2 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.

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

7.4 Code for Multicast Using RDMA_CM and IBV Verbs

Multicast Code Example

```
/*
 * BUILD COMMAND:
 * gcc -g -Wall -D_GNU_SOURCE -g -O2 -o examples/mckey  examples/mckey.c  -libverbs -lrdmacm
 *
* $Id$
 */

#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;
}
```

8 Programming Examples Using RDMA Verbs

This chapter provides code examples using the RDMA Verbs

8.1 Automatic Path Migration (APM)

```
//*
 * Compile Command:
 * gcc apm.c -o apm -libverbs -lrdmacm
 *
 * Description:
 * This example demonstrates Automatic Path Migration (APM). The basic flow is
 * as follows:
 * 1. Create connection between client and server
 * 2. Set the alternate path details on each side of the connection
 * 3. Perform send operations back and forth between client and server
 * 4. Cause the path to be migrated (manually or automatically)
 * 5. Complete sends using the alternate path
 *
 * There are two ways to cause the path to be migrated.
 * 1. Use the ibv_modify_qp verb to set path_mig_state = IBV_MIG_MIGRATED
 * 2. Assuming there are two ports on at least one side of the connection, and
 *    each port has a path to the other host, pull out the cable of the original
 *    port and watch it migrate to the other port.
 *
 * Running the Example:
 * This example requires a specific IB network configuration to properly
 * demonstrate APM. Two hosts are required, one for the client and one for the
 * server. At least one of these two hosts must have a IB card with two ports.
 * Both of these ports should be connected to the same subnet and each have a
 * route to the other host through an IB switch.
 * The executable can operate as either the client or server application. Start
 * the server side first on one host then start the client on the other host. With default parameters, the
 * client and server will exchange 100 sends over 100 seconds. During that time,
 * manually unplug the cable connected to the original port of the two port
 * host, and watch the path get migrated to the other port. It may take up to
 * a minute for the path to migrated. To see the path get migrated by software,
 * use the -m option on the client side.
 *
 * Server:
 * ./apm -s
 *
 * Client (-a is IP of remote interface):
 * ./apm -a 192.168.1.12
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
```

```
#include <getopt.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameter values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 100
#define DEFAULT_MSG_LENGTH 1000000
#define DEFAULT_MSEC_DELAY 500

/* Resources used in the example */
struct context
{
    /* User parameters */
    int server;
    char *server_name;
    char *server_port;
    int msg_count;
    int msg_length;
    int msec_delay;
    uint8_t alt_srcport;
    uint16_t alt_dlid;
    uint16_t my_alt_dlid;
    int migrate_after;

    /* Resources */
    struct rdma_cm_id *id;
    struct rdma_cm_id *listen_id;
    struct ibv_mr *send_mr;
    struct ibv_mr *recv_mr;
    char *send_buf;
    char *recv_buf;
    pthread_t async_event_thread;
};

/*
 * Function:    async_event_thread
 *
 * Input:
 *      arg    The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      NULL
 *
 * Description:
 *      Reads any Asynchronous events that occur during the sending of data
 *      and prints out the details of the event. Specifically migration
 *      related events.
 */
```

```
static void *async_event_thread(void *arg)
{
    struct ibv_async_event event;
    int ret;

    struct context *ctx = (struct context *) arg;

    while (1) {
        ret = ibv_get_async_event(ctx->id->verbs, &event);
        if (ret) {
            VERB_ERR("ibv_get_async_event", ret);
            break;
        }

        switch (event.event_type) {
        case IBV_EVENT_PATH_MIG:
            printf("QP path migrated\n");
            break;
        case IBV_EVENT_PATH_MIG_ERR:
            printf("QP path migration error\n");
            break;
        default:
            printf("Async Event %d\n", event.event_type);
            break;
        }

        ibv_ack_async_event(&event);
    }

    return NULL;
}

/*
 * Function:    get_alt_dlid_from_private_data
 *
 * Input:
 *      event  The RDMA event containing private data
 *
 * Output:
 *      dlid   The DLID that was sent in the private data
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Takes the private data sent from the remote side and returns the
 *      destination LID that was contained in the private data
 */
int get_alt_dlid_from_private_data(struct rdma_cm_event *event, uint16_t *dlid)
{
    if (event->param.conn.private_data_len < 4) {
        printf("unexpected private data len: %d",
               event->param.conn.private_data_len);
        return -1;
```

```
    }

    *dlid = ntohs(*((uint16_t *) event->param.conn.private_data));
    return 0;
}

/*
 * Function:    get_alt_port_details
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      First, query the device to determine if path migration is supported.
 *      Next, queries all the ports on the device to determine if there is
 *      different port than the current one to use as an alternate port. If so,
 *      copy the port number and dlid to the context so they can be used when
 *      the alternate path is loaded.
 *
 * Note:
 *      This function assumes that if another port is found in the active state,
 *      that the port is connected to the same subnet as the initial port and
 *      that there is a route to the other hosts alternate port.
 */
int get_alt_port_details(struct context *ctx)
{
    int ret, i;
    struct ibv_qp_attr qp_attr;
    struct ibv_qp_init_attr qp_init_attr;
    struct ibv_device_attr dev_attr;

    /* This example assumes the alternate port we want to use is on the same
     * HCA. Ports from other HCAs can be used as alternate paths as well. Get
     * a list of devices using ibv_get_device_list or rdma_get_devices.*/
    ret = ibv_query_device(ctx->id->verbs, &dev_attr);
    if (ret) {
        VERB_ERR("ibv_query_device", ret);
        return ret;
    }

    /* Verify the APM is supported by the HCA */
    if (!(dev_attr.device_cap_flags | IBV_DEVICE_AUTO_PATH_MIG)) {
        printf("device does not support auto path migration!\n");
        return -1;
    }

    /* Query the QP to determine which port we are bound to */
    ret = ibv_query_qp(ctx->id->qp, &qp_attr, 0, &qp_init_attr);
```

```
    if (ret) {
        VERB_ERR("ibv_query_qp", ret);
        return ret;
    }

    for (i = 1; i <= dev_attr.phys_port_cnt; i++) {
        /* Query all ports until we find one in the active state that is
         * not the port we are currently connected to. */

        struct ibv_port_attr port_attr;
        ret = ibv_query_port(ctx->id->verbs, i, &port_attr);
        if (ret) {
            VERB_ERR("ibv_query_device", ret);
            return ret;
        }

        if (port_attr.state == IBV_PORT_ACTIVE) {
            ctx->my_alt_dlid = port_attr.lid;
            ctx->alt_srcport = i;
            if (qp_attr.port_num != i)
                break;
        }
    }

    return 0;
}

/*
 * Function:    load_alt_path
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Uses ibv_modify_qp to load the alternate path information and set the
 *      path migration state to rearm.
 */
int load_alt_path(struct context *ctx)
{
    int ret;
    struct ibv_qp_attr qp_attr;
    struct ibv_qp_init_attr qp_init_attr;

    /* query to get the current attributes of the qp */
    ret = ibv_query_qp(ctx->id->qp, &qp_attr, 0, &qp_init_attr);
    if (ret) {
        VERB_ERR("ibv_query_qp", ret);
        return ret;
```

```
    }

    /* initialize the alternate path attributes with the current path
     * attributes */
    memcpy(&qp_attr.alt_ah_attr, &qp_attr.ah_attr, sizeof (struct ibv_ah_attr));

    /* set the alt path attributes to some basic values */
    qp_attr.alt_pkey_index = qp_attr.pkey_index;
    qp_attr.alt_timeout = qp_attr.timeout;
    qp_attr.path_mig_state = IBV_MIG_REARM;

    /* if an alternate path was supplied, set the source port and the dlid */
    if (ctx->alt_srcport)
        qp_attr.alt_port_num = ctx->alt_srcport;
    else
        qp_attr.alt_port_num = qp_attr.port_num;

    if (ctx->alt_dlid)
        qp_attr.alt_ah_attr.dlid = ctx->alt_dlid;

    printf("loading alt path - local port: %d, dlid: %d\n",
           qp_attr.alt_port_num, qp_attr.alt_ah_attr.dlid);

    ret = ibv_modify_qp(ctx->id->qp, &qp_attr,
                        IBV_QP_ALT_PATH | IBV_QP_PATH_MIG_STATE);
    if (ret) {
        VERB_ERR("ibv_modify_qp", ret);
        return ret;
    }
}

/*
 * Function:    reg_mem
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Registers memory regions to use for our data transfer
 */
int reg_mem(struct context *ctx)
{
    ctx->send_buf = (char *) malloc(ctx->msg_length);
    memset(ctx->send_buf, 0x12, ctx->msg_length);

    ctx->recv_buf = (char *) malloc(ctx->msg_length);
    memset(ctx->recv_buf, 0x00, ctx->msg_length);
```

```
    ctx->send_mr = rdma_reg_msgs(ctx->id, ctx->send_buf, ctx->msg_length);
    if (!ctx->send_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    ctx->recv_mr = rdma_reg_msgs(ctx->id, ctx->recv_buf, ctx->msg_length);
    if (!ctx->recv_mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    return 0;
}

/*
 * Function:    getaddrinfo_and_create_ep
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Gets the address information and creates our endpoint
 */
int getaddrinfo_and_create_ep(struct context *ctx)
{
    int ret;
    struct rdma_addrinfo *rai, hints;
    struct ibv_qp_init_attr qp_init_attr;

    memset(&hints, 0, sizeof (hints));
    hints.ai_port_space = RDMA_PS_TCP;
    if (ctx->server == 1)
        hints.ai_flags = RAI_PASSIVE; /* this makes it a server */

    printf("rdma_getaddrinfo\n");
    ret = rdma_getaddrinfo(ctx->server_name, ctx->server_port, &hints, &rai);
    if (ret) {
        VERB_ERR("rdma_getaddrinfo", ret);
        return ret;
    }

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

    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;
```

```
    printf("rdma_create_ep\n");
    ret = rdma_create_ep(&ctx->id, rai, NULL, &qp_init_attr);
    if (ret) {
        VERB_ERR("rdma_create_ep", ret);
        return ret;
    }

    rdma_freeaddrinfo(rai);

    return 0;
}

/*
 * Function:    get_connect_request
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Wait for a connect request from the client
 */
int get_connect_request(struct context *ctx)
{
    int ret;

    printf("rdma_listen\n");
    ret = rdma_listen(ctx->id, 4);
    if (ret) {
        VERB_ERR("rdma_listen", ret);
        return ret;
    }

    ctx->listen_id = ctx->id;

    printf("rdma_get_request\n");
    ret = rdma_get_request(ctx->listen_id, &ctx->id);
    if (ret) {
        VERB_ERR("rdma_get_request", ret);
        return ret;
    }

    if (ctx->id->event->event != RDMA_CM_EVENT_CONNECT_REQUEST) {
        printf("unexpected event: %s",
               rdma_event_str(ctx->id->event->event));
        return ret;
    }
```

```
    /* If the alternate path info was not set on the command line, get
     * it from the private data */
    if (ctx->alt_dlid == 0 && ctx->alt_srcport == 0) {
        ret = get_alt_dlid_from_private_data(ctx->id->event, &ctx->alt_dlid);
        if (ret) {
            return ret;
        }
    }

    return 0;
}

/*
 * Function:    establish_connection
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Create the connection. For the client, call rdma_connect. For the
 *      server, the connect request was already received, so just do
 *      rdma_accept to complete the connection.
 */
int establish_connection(struct context *ctx)
{
    int ret;
    uint16_t private_data;
    struct rdma_conn_param conn_param;

    /* post a receive to catch the first send */
    ret = rdma_post_recv(ctx->id, NULL, ctx->recv_buf, ctx->msg_length,
                    ctx->recv_mr);
    if (ret) {
        VERB_ERR("rdma_post_recv", ret);
        return ret;
    }

    /* send the dlid for the alternate port in the private data */
    private_data = htons(ctx->my_alt_dlid);

    memset(&conn_param, 0, sizeof (conn_param));
    conn_param.private_data_len = sizeof (int);
    conn_param.private_data = &private_data;
    conn_param.responder_resources = 2;
    conn_param.initiator_depth = 2;
    conn_param.retry_count = 5;
    conn_param.rnr_retry_count = 5;
```

```
    if (ctx->server) {
        printf("rdma_accept\n");
        ret = rdma_accept(ctx->id, &conn_param);
        if (ret) {
            VERB_ERR("rdma_accept", ret);
            return ret;
        }
    }
    else {
        printf("rdma_connect\n");
        ret = rdma_connect(ctx->id, &conn_param);
        if (ret) {
            VERB_ERR("rdma_connect", ret);
            return ret;
        }

        if (ctx->id->event->event != RDMA_CM_EVENT_ESTABLISHED) {
            printf("unexpected event: %s",
                   rdma_event_str(ctx->id->event->event));
            return -1;
        }

        /* If the alternate path info was not set on the command line, get
         * it from the private data */
        if (ctx->alt_dlid == 0 && ctx->alt_srcport == 0) {
            ret = get_alt_dlid_from_private_data(ctx->id->event,
                                                 &ctx->alt_dlid);
            if (ret)
                return ret;
        }
    }

    return 0;
}

/*
 * Function:    send_msg
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Performs an Send and gets the completion
 *
 */
int send_msg(struct context *ctx)
{
    int ret;
```

```
    struct ibv_wc wc;

    ret = rdma_post_send(ctx->id, NULL, ctx->send_buf, ctx->msg_length,
                ctx->send_mr, IBV_SEND_SIGNALED);
    if (ret) {
        VERB_ERR("rdma_send_recv", ret);
        return ret;
    }

    ret = rdma_get_send_comp(ctx->id, &wc);
    if (ret < 0) {
        VERB_ERR("rdma_get_send_comp", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    recv_msg
 *
 * Input:
 *     ctx     The context object
 *
 * Output:
 *     none
 *
 * Returns:
 *     0 on success, non-zero on failure
 *
 * Description:
 *     Waits for a receive completion and posts a new receive buffer
 */
int recv_msg(struct context *ctx)
{
    int ret;
    struct ibv_wc wc;

    ret = rdma_get_recv_comp(ctx->id, &wc);
    if (ret < 0) {
        VERB_ERR("rdma_get_recv_comp", ret);
        return ret;
    }

    ret = rdma_post_recv(ctx->id, NULL, ctx->recv_buf, ctx->msg_length,
                ctx->recv_mr);
    if (ret) {
        VERB_ERR("rdma_post_recv", ret);
        return ret;
    }

    return 0;
}
```

```
/*
 * Function:    main
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *
 */
int main(int argc, char** argv)
{
    int ret, op, i, send_cnt, recv_cnt;
    struct context ctx;
    struct ibv_qp_attr qp_attr;

    memset(&ctx, 0, sizeof (ctx));
    memset(&qp_attr, 0, sizeof (qp_attr));

    ctx.server = 0;
    ctx.server_port = DEFAULT_PORT;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.msec_delay = DEFAULT_MSEC_DELAY;
    ctx.alt_dlid = 0;
    ctx.alt_srcport = 0;
    ctx.migrate_after = -1;

    while ((op = getopt(argc, argv, "sa:p:c:l:d:r:m:")) != -1) {
        switch (op) {
        case 's':
            ctx.server = 1;
            break;
        case 'a':
            ctx.server_name = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
            break;
        case 'c':
            ctx.msg_count = atoi(optarg);
            break;
        case 'l':
            ctx.msg_length = atoi(optarg);
            break;
        case 'd':
            ctx.alt_dlid = atoi(optarg);
            break;
        case 'r':
```

```
            ctx.alt_srcport = atoi(optarg);
            break;
        case 'm':
            ctx.migrate_after = atoi(optarg);
            break;
        case 'w':
            ctx.msec_delay = atoi(optarg);
            break;
        default:
            printf("usage: %s [-s or -a required]\n", argv[0]);
            printf("\t[-s[erver mode]\n");
            printf("\t[-a ip_address]\n");
            printf("\t[-p port_number]\n");
            printf("\t[-c msg_count]\n");
            printf("\t[-l msg_length]\n");
            printf("\t[-d alt_dlid] (requires -r)\n");
            printf("\t[-r alt_srcport] (requires -d)\n");
            printf("\t[-m num_iterations_then_migrate] (client only)\n");
            printf("\t[-w msec_wait_between_sends]\n");
            exit(1);
        }
    }

    printf("mode:       %s\n", (ctx.server) ? "server" : "client");
    printf("address:    %s\n", (!ctx.server_name) ? "NULL" : ctx.server_name);
    printf("port:       %s\n", ctx.server_port);
    printf("count:      %d\n", ctx.msg_count);
    printf("length:     %d\n", ctx.msg_length);
    printf("alt_dlid:   %d\n", ctx.alt_dlid);
    printf("alt_port:   %d\n", ctx.alt_srcport);
    printf("mig_after:  %d\n", ctx.migrate_after);
    printf("msec_wait:  %d\n", ctx.msec_delay);
    printf("\n");

    if (!ctx.server && !ctx.server_name) {
        printf("server address must be specified for client mode\n");
        exit(1);
    }

    /* both of these must be set or neither should be set */
    if (!((ctx.alt_dlid > 0 && ctx.alt_srcport > 0) ||
        (ctx.alt_dlid == 0 && ctx.alt_srcport == 0))) {
        printf("-d and -r must be used together\n");
        exit(1);
    }

    if (ctx.migrate_after > ctx.msg_count) {
        printf("num_iterations_then_migrate must be less than msg_count\n");
        exit(1);
    }

    ret = getaddrinfo_and_create_ep(&ctx);
    if (ret)
        goto out;
```

```
if (ctx.server) {
    ret = get_connect_request(&ctx);
    if (ret)
        goto out;
}

/* only query for alternate port if information was not specified on the
 * command line */
if (ctx.alt_dlid == 0 && ctx.alt_srcport == 0) {
    ret = get_alt_port_details(&ctx);
    if (ret)
        goto out;
}

/* create a thread to handle async events */
pthread_create(&ctx.async_event_thread, NULL, async_event_thread, &ctx);

ret = reg_mem(&ctx);
if (ret)
    goto out;

ret = establish_connection(&ctx);

/* load the alternate path after the connection was created. This can be
 * done at connection time, but the connection must be created and
 * established using all ib verbs */
ret = load_alt_path(&ctx);
if (ret)
    goto out;

send_cnt = recv_cnt = 0;

for (i = 0; i < ctx.msg_count; i++) {
    if (ctx.server) {
        if (recv_msg(&ctx))
            break;

        printf("recv: %d\n", ++recv_cnt);
    }

    if (ctx.msec_delay > 0)
        usleep(ctx.msec_delay * 1000);

    if (send_msg(&ctx))
        break;

    printf("send: %d\n", ++send_cnt);

    if (!ctx.server) {
        if (recv_msg(&ctx))
            break;

        printf("recv: %d\n", ++recv_cnt);
```

```
        }

        /* migrate the path manually if desired after the specified number of
         * sends */
        if (!ctx.server && i == ctx.migrate_after) {
            qp_attr.path_mig_state = IBV_MIG_MIGRATED;
            ret = ibv_modify_qp(ctx.id->qp, &qp_attr, IBV_QP_PATH_MIG_STATE);
            if (ret) {
                VERB_ERR("ibv_modify_qp", ret);
                goto out;
            }
        }
    }

    rdma_disconnect(ctx.id);

out:
    if (ctx.send_mr)
        rdma_dereg_mr(ctx.send_mr);

    if (ctx.recv_mr)
        rdma_dereg_mr(ctx.recv_mr);

    if (ctx.id)
        rdma_destroy_ep(ctx.id);

    if (ctx.listen_id)
        rdma_destroy_ep(ctx.listen_id);

    if (ctx.send_buf)
        free(ctx.send_buf);

    if (ctx.recv_buf)
        free(ctx.recv_buf);

    return ret;
}
```

8.2 Multicast Code Example Using RDMA CM

```
/*
 * Compile Command:
 * gcc mc.c -o mc -libverbs -lrdmacm
 *
 * Description:
 * Both the sender and receiver create a UD Queue Pair and join the specified
 * multicast group (ctx.mcast_addr). If the join is successful, the sender must
 * create an Address Handle (ctx.ah). The sender then posts the specified
 * number of sends (ctx.msg_count) to the multicast group. The receiver waits
 * to receive each one of the sends and then both sides leave the multicast
 * group and cleanup resources.
 *
```

```
 * Running the Example:
 * The executable can operate as either the sender or receiver application. It
 * can be demonstrated on a simple fabric of two nodes with the sender
 * application running on one node and the receiver application running on the
 * other. Each node must be configured to support IPoIB and the IB interface
 * (ex. ib0) must be assigned an IP Address. Finally, the fabric must be
 * initialized using OpenSM.
 *
 * Receiver (-m is the multicast address, often the IP of the receiver):
 * ./mc -m 192.168.1.12
 *
 * Sender (-m is the multicast address, often the IP of the receiver):
 * ./mc -s -m 192.168.1.12
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameter values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 4
#define DEFAULT_MSG_LENGTH 64

/* Resources used in the example */
struct context
{
    /* User parameters */
    int sender;
    char *bind_addr;
    char *mcast_addr;
    char *server_port;
    int msg_count;
    int msg_length;

    /* Resources */
    struct sockaddr mcast_sockaddr;
    struct rdma_cm_id *id;
    struct rdma_event_channel *channel;
    struct ibv_pd *pd;
    struct ibv_cq *cq;
    struct ibv_mr *mr;
    char *buf;
    struct ibv_ah *ah;
    uint32_t remote_qpn;
    uint32_t remote_qkey;
```

```
    pthread_t cm_thread;
};

/*
 * Function:    cm_thread
 *
 * Input:
 *      arg     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      NULL
 *
 * Description:
 *      Reads any CM events that occur during the sending of data
 *      and prints out the details of the event
 */
static void *cm_thread(void *arg)
{
    struct rdma_cm_event *event;
    int ret;

    struct context *ctx = (struct context *) arg;

    while (1) {
        ret = rdma_get_cm_event(ctx->channel, &event);
        if (ret) {
            VERB_ERR("rdma_get_cm_event", ret);
            break;
        }

        printf("event %s, status %d\n",
               rdma_event_str(event->event), event->status);

        rdma_ack_cm_event(event);
    }

    return NULL;
}

/*
 * Function:    get_cm_event
 *
 * Input:
 *      channel The event channel
 *      type    The event type that is expected
 *
 * Output:
 *      out_ev  The event will be passed back to the caller, if desired
 *              Set this to NULL and the event will be acked automatically
 *              Otherwise the caller must ack the event using rdma_ack_cm_event
 *
```

```
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Waits for the next CM event and check that is matches the expected
 *      type.
 */
int get_cm_event(struct rdma_event_channel *channel,
                 enum rdma_cm_event_type type,
                 struct rdma_cm_event **out_ev)
{
    int ret = 0;
    struct rdma_cm_event *event = NULL;

    ret = rdma_get_cm_event(channel, &event);
    if (ret) {
        VERB_ERR("rdma_resolve_addr", ret);
        return -1;
    }

    /* Verify the event is the expected type */
    if (event->event != type) {
        printf("event: %s, status: %d\n",
               rdma_event_str(event->event), event->status);
        ret = -1;
    }

    /* Pass the event back to the user if requested */
    if (!out_ev)
        rdma_ack_cm_event(event);
    else
        *out_ev = event;

    return ret;
}

/*
 * Function:    resolve_addr
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Resolves the multicast address and also binds to the source address
 *      if one was provided in the context
 */
int resolve_addr(struct context *ctx)
{
```

```
int ret;
struct rdma_addrinfo *bind_rai = NULL;
struct rdma_addrinfo *mcast_rai = NULL;
struct rdma_addrinfo hints;

memset(&hints, 0, sizeof (hints));
hints.ai_port_space = RDMA_PS_UDP;

if (ctx->bind_addr) {
   hints.ai_flags = RAI_PASSIVE;

   ret = rdma_getaddrinfo(ctx->bind_addr, NULL, &hints, &bind_rai);
   if (ret) {
      VERB_ERR("rdma_getaddrinfo (bind)", ret);
      return ret;
   }
}

hints.ai_flags = 0;

ret = rdma_getaddrinfo(ctx->mcast_addr, NULL, &hints, &mcast_rai);
if (ret) {
   VERB_ERR("rdma_getaddrinfo (mcast)", ret);
   return ret;
}

if (ctx->bind_addr) {
   /* bind to a specific adapter if requested to do so */
   ret = rdma_bind_addr(ctx->id, bind_rai->ai_src_addr);
   if (ret) {
      VERB_ERR("rdma_bind_addr", ret);
      return ret;
   }

   /* A PD is created when we bind. Copy it to the context so it can
    * be used later on */
   ctx->pd = ctx->id->pd;
}

ret = rdma_resolve_addr(ctx->id, (bind_rai) ? bind_rai->ai_src_addr : NULL,
                   mcast_rai->ai_dst_addr, 2000);
if (ret) {
   VERB_ERR("rdma_resolve_addr", ret);
   return ret;
}

ret = get_cm_event(ctx->channel, RDMA_CM_EVENT_ADDR_RESOLVED, NULL);
if (ret) {
   return ret;
}

memcpy(&ctx->mcast_sockaddr,
      mcast_rai->ai_dst_addr,
      sizeof (struct sockaddr));
```

```
    return 0;
}

/*
 * Function:    create_resources
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Creates the PD, CQ, QP and MR
 */
int create_resources(struct context *ctx)
{
    int ret, buf_size;
    struct ibv_qp_init_attr attr;

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

    /* If we are bound to an address, then a PD was already allocated
     * to the CM ID */
    if (!ctx->pd) {
        ctx->pd = ibv_alloc_pd(ctx->id->verbs);
        if (!ctx->pd) {
            VERB_ERR("ibv_alloc_pd", -1);
            return ret;
        }
    }

    ctx->cq = ibv_create_cq(ctx->id->verbs, 2, 0, 0, 0);
    if (!ctx->cq) {
        VERB_ERR("ibv_create_cq", -1);
        return ret;
    }

    attr.qp_type = IBV_QPT_UD;
    attr.send_cq = ctx->cq;
    attr.recv_cq = ctx->cq;
    attr.cap.max_send_wr = ctx->msg_count;
    attr.cap.max_recv_wr = ctx->msg_count;
    attr.cap.max_send_sge = 1;
    attr.cap.max_recv_sge = 1;

    ret = rdma_create_qp(ctx->id, ctx->pd, &attr);
    if (ret) {
        VERB_ERR("rdma_create_qp", ret);
        return ret;
```

```
    }

    /* The receiver must allow enough space in the receive buffer for
     * the GRH */
    buf_size = ctx->msg_length + (ctx->sender ? 0 : sizeof (struct ibv_grh));

    ctx->buf = calloc(1, buf_size);
    memset(ctx->buf, 0x00, buf_size);

    /* Register our memory region */
    ctx->mr = rdma_reg_msgs(ctx->id, ctx->buf, buf_size);
    if (!ctx->mr) {
        VERB_ERR("rdma_reg_msgs", -1);
        return -1;
    }

    return 0;
}

/*
 * Function:    destroy_resources
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Destroys AH, QP, CQ, MR, PD and ID
 */
void destroy_resources(struct context *ctx)
{
    if (ctx->ah)
        ibv_destroy_ah(ctx->ah);

    if (ctx->id->qp)
        rdma_destroy_qp(ctx->id);

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

    if (ctx->mr)
        rdma_dereg_mr(ctx->mr);

    if (ctx->buf)
        free(ctx->buf);

    if (ctx->pd && ctx->id->pd == NULL)
        ibv_dealloc_pd(ctx->pd);
```

```
    rdma_destroy_id(ctx->id);
}

/*
 * Function:    post_send
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Posts a UD send to the multicast address
 */
int post_send(struct context *ctx)
{
    int ret;
    struct ibv_send_wr wr, *bad_wr;
    struct ibv_sge sge;

    memset(ctx->buf, 0x12, ctx->msg_length); /* set the data to non-zero */

    sge.length = ctx->msg_length;
    sge.lkey = ctx->mr->lkey;
    sge.addr = (uint64_t) ctx->buf;

    /* Multicast requires that the message is sent with immediate data
     * and that the QP number is the contents of the immediate data */
    wr.next = NULL;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.opcode = IBV_WR_SEND_WITH_IMM;
    wr.send_flags = IBV_SEND_SIGNALED;
    wr.wr_id = 0;
    wr.imm_data = htonl(ctx->id->qp->qp_num);
    wr.wr.ud.ah = ctx->ah;
    wr.wr.ud.remote_qpn = ctx->remote_qpn;
    wr.wr.ud.remote_qkey = ctx->remote_qkey;

    ret = ibv_post_send(ctx->id->qp, &wr, &bad_wr);
    if (ret) {
        VERB_ERR("ibv_post_send", ret);
        return -1;
    }

    return 0;
}

/*
 * Function:    get_completion
```

```
 *
 * Input:
 *      ctx     The context structure
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Waits for a completion and verifies that the operation was successful
 */
int get_completion(struct context *ctx)
{
    int ret;
    struct ibv_wc wc;

    do {
        ret = ibv_poll_cq(ctx->cq, 1, &wc);
        if (ret < 0) {
            VERB_ERR("ibv_poll_cq", ret);
            return -1;
        }
    }
    while (ret == 0);

    if (wc.status != IBV_WC_SUCCESS) {
        printf("work completion status %s\n",
               ibv_wc_status_str(wc.status));
        return -1;
    }

    return 0;
}

/*
 * Function:    main
 *
 * Input:
 *      argc    The number of arguments
 *      argv    Command line arguments
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Main program to demonstrate multicast functionality.
 *      Both the sender and receiver create a UD Queue Pair and join the
 *      specified multicast group (ctx.mcast_addr). If the join is successful,
 *      the sender must create an Address Handle (ctx.ah). The sender then posts
```

```
 *      the specified number of sends (ctx.msg_count) to the multicast group.
 *      The receiver waits to receive each one of the sends and then both sides
 *      leave the multicast group and cleanup resources.
 */
int main(int argc, char** argv)
{
    int ret, op, i;
    struct context ctx;
    struct ibv_port_attr port_attr;
    struct rdma_cm_event *event;
    char buf[40];

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

    ctx.sender = 0;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.server_port = DEFAULT_PORT;

    // Read options from command line
    while ((op = getopt(argc, argv, "shb:m:p:c:l:")) != -1) {
        switch (op) {
        case 's':
            ctx.sender = 1;
            break;
        case 'b':
            ctx.bind_addr = optarg;
            break;
        case 'm':
            ctx.mcast_addr = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
            break;
        case 'c':
            ctx.msg_count = atoi(optarg);
            break;
        case 'l':
            ctx.msg_length = atoi(optarg);
            break;
        default:
            printf("usage: %s -m mc_address\n", argv[0]);
            printf("\t[-s[ender mode]\n");
            printf("\t[-b bind_address]\n");
            printf("\t[-p port_number]\n");
            printf("\t[-c msg_count]\n");
            printf("\t[-l msg_length]\n");
            exit(1);
        }
    }

    if(ctx.mcast_addr == NULL) {
        printf("multicast address must be specified with -m\n");
        exit(1);
```

```
    }

    ctx.channel = rdma_create_event_channel();
    if (!ctx.channel) {
        VERB_ERR("rdma_create_event_channel", -1);
        exit(1);
    }

    ret = rdma_create_id(ctx.channel, &ctx.id, NULL, RDMA_PS_UDP);
    if (ret) {
        VERB_ERR("rdma_create_id", -1);
        exit(1);
    }

    ret = resolve_addr(&ctx);
    if (ret)
        goto out;

    /* Verify that the buffer length is not larger than the MTU */
    ret = ibv_query_port(ctx.id->verbs, ctx.id->port_num, &port_attr);
    if (ret) {
        VERB_ERR("ibv_query_port", ret);
        goto out;
    }

    if (ctx.msg_length > (1 << port_attr.active_mtu + 7)) {
        printf("buffer length %d is larger then active mtu %d\n",
               ctx.msg_length, 1 << (port_attr.active_mtu + 7));
        goto out;
    }

    ret = create_resources(&ctx);
    if (ret)
        goto out;

    if (!ctx.sender) {
        for (i = 0; i < ctx.msg_count; i++) {
            ret = rdma_post_recv(ctx.id, NULL, ctx.buf,
                                 ctx.msg_length + sizeof (struct ibv_grh),
                                 ctx.mr);
            if (ret) {
                VERB_ERR("rdma_post_recv", ret);
                goto out;
            }
        }
    }

    /* Join the multicast group */
    ret = rdma_join_multicast(ctx.id, &ctx.mcast_sockaddr, NULL);
    if (ret) {
        VERB_ERR("rdma_join_multicast", ret);
        goto out;
    }
```

```
    /* Verify that we successfully joined the multicast group */
    ret = get_cm_event(ctx.channel, RDMA_CM_EVENT_MULTICAST_JOIN, &event);
    if (ret)
        goto out;

    inet_ntop(AF_INET6, event->param.ud.ah_attr.grh.dgid.raw, buf, 40);
    printf("joined dgid: %s, mlid 0x%x, sl %d\n", buf,
           event->param.ud.ah_attr.dlid, event->param.ud.ah_attr.sl);

    ctx.remote_qpn = event->param.ud.qp_num;
    ctx.remote_qkey = event->param.ud.qkey;

    if (ctx.sender) {
        /* Create an address handle for the sender */
        ctx.ah = ibv_create_ah(ctx.pd, &event->param.ud.ah_attr);
        if (!ctx.ah) {
            VERB_ERR("ibv_create_ah", -1);
            goto out;
        }
    }

    rdma_ack_cm_event(event);

    /* Create a thread to handle any CM events while messages are exchanged */
    pthread_create(&ctx.cm_thread, NULL, cm_thread, &ctx);

    if (!ctx.sender)
        printf("waiting for messages...\n");

    for (i = 0; i < ctx.msg_count; i++) {
        if (ctx.sender) {
            ret = post_send(&ctx);
            if (ret)
                goto out;
        }

        ret = get_completion(&ctx);
        if (ret)
            goto out;

        if (ctx.sender)
            printf("sent message %d\n", i + 1);
        else
            printf("received message %d\n", i + 1);
    }

out:
    ret = rdma_leave_multicast(ctx.id, &ctx.mcast_sockaddr);
    if (ret)
        VERB_ERR("rdma_leave_multicast", ret);

    destroy_resources(&ctx);

    return ret;
```

```
}
```

8.3 Shared Received Queue (SRQ)

```
/*
 * Compile Command:
 * gcc srq.c -o srq -libverbs -lrdmacm
 *
 * Description:
 * Both the client and server use an SRQ. A number of Queue Pairs (QPs) are
 * created (ctx.qp_count) and each QP uses the SRQ. The connection between the
 * client and server is established using the IP address details passed on the
 * command line. After the connection is established, the client starts
 * blasting sends to the server and stops when the maximum work requests
 * (ctx.max_wr) have been sent. When the server has received all the sends, it
 * performs a send to the client to tell it to continue. The process repeats
 * until the number of requested number of sends (ctx.msg_count) have been
 * performed.
 *
 * Running the Example:
 * The executable can operate as either the client or server application. It
 * can be demonstrated on a simple fabric of two nodes with the server
 * application running on one node and the client application running on the
 * other. Each node must be configured to support IPoIB and the IB interface
 * (ex. ib0) must be assigned an IP Address. Finally, the fabric must be
 * initialized using OpenSM.
 *
 * Server (-a is IP of local interface):
 * ./srq -s -a 192.168.1.12
 *
 * Client (-a is IP of remote interface):
 * ./srq -a 192.168.1.12
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <rdma/rdma_verbs.h>

#define VERB_ERR(verb, ret) \
        fprintf(stderr, "%s returned %d errno %d\n", verb, ret, errno)

/* Default parameters values */
#define DEFAULT_PORT "51216"
#define DEFAULT_MSG_COUNT 100
#define DEFAULT_MSG_LENGTH 100000
#define DEFAULT_QP_COUNT 4
#define DEFAULT_MAX_WR 64
```

```
/* Resources used in the example */
struct context
{
   /* User parameters */
   int server;
   char *server_name;
   char *server_port;
   int msg_count;
   int msg_length;
   int qp_count;
   int max_wr;

   /* Resources */
   struct rdma_cm_id *srq_id;
   struct rdma_cm_id *listen_id;
   struct rdma_cm_id **conn_id;
   struct ibv_mr *send_mr;
   struct ibv_mr *recv_mr;
   struct ibv_srq *srq;
   struct ibv_cq *srq_cq;
   struct ibv_comp_channel *srq_cq_channel;
   char *send_buf;
   char *recv_buf;
};

/*
 * Function: init_resources
 *
 * Input:
 *     ctx     The context object
 *     rai     The RDMA address info for the connection
 *
 * Output:
 *     none
 *
 * Returns:
 *     0 on success, non-zero on failure
 *
 * Description:
 *     This function initializes resources that are common to both the client
 *     and server functionality.
 *     It creates our SRQ, registers memory regions, posts receive buffers
 *     and creates a single completion queue that will be used for the receive
 *     queue on each queue pair.
 */
int init_resources(struct context *ctx, struct rdma_addrinfo *rai)
{
   int ret, i;
   struct rdma_cm_id *id;

   /* Create an ID used for creating/accessing our SRQ */
   ret = rdma_create_id(NULL, &ctx->srq_id, NULL, RDMA_PS_TCP);
   if (ret) {
```

```
    VERB_ERR("rdma_create_id", ret);
    return ret;
}

/* We need to bind the ID to a particular RDMA device
 * This is done by resolving the address or binding to the address */
if (ctx->server == 0) {
    ret = rdma_resolve_addr(ctx->srq_id, NULL, rai->ai_dst_addr, 1000);
    if (ret) {
        VERB_ERR("rdma_resolve_addr", ret);
        return ret;
    }
}
else {
    ret = rdma_bind_addr(ctx->srq_id, rai->ai_src_addr);
    if (ret) {
        VERB_ERR("rdma_bind_addr", ret);
        return ret;
    }
}

/* Create the memory regions being used in this example */
ctx->recv_mr = rdma_reg_msgs(ctx->srq_id, ctx->recv_buf, ctx->msg_length);
if (!ctx->recv_mr) {
    VERB_ERR("rdma_reg_msgs", -1);
    return -1;
}

ctx->send_mr = rdma_reg_msgs(ctx->srq_id, ctx->send_buf, ctx->msg_length);
if (!ctx->send_mr) {
    VERB_ERR("rdma_reg_msgs", -1);
    return -1;
}

/* Create our shared receive queue */
struct ibv_srq_init_attr srq_attr;
memset(&srq_attr, 0, sizeof (srq_attr));
srq_attr.attr.max_wr = ctx->max_wr;
srq_attr.attr.max_sge = 1;

ret = rdma_create_srq(ctx->srq_id, NULL, &srq_attr);
if (ret) {
    VERB_ERR("rdma_create_srq", ret);
    return -1;
}

/* Save the SRQ in our context so we can assign it to other QPs later */
ctx->srq = ctx->srq_id->srq;

/* Post our receive buffers on the SRQ */
for (i = 0; i < ctx->max_wr; i++) {
    ret = rdma_post_recv(ctx->srq_id, NULL, ctx->recv_buf, ctx->msg_length,
                    ctx->recv_mr);
    if (ret) {
```

```
            VERB_ERR("rdma_post_recv", ret);
            return ret;
        }
    }

    /* Create a completion channel to use with the SRQ CQ */
    ctx->srq_cq_channel = ibv_create_comp_channel(ctx->srq_id->verbs);
    if (!ctx->srq_cq_channel) {
        VERB_ERR("ibv_create_comp_channel", -1);
        return -1;
    }

    /* Create a CQ to use for all connections (QPs) that use the SRQ */
    ctx->srq_cq = ibv_create_cq(ctx->srq_id->verbs, ctx->max_wr, NULL,
                                ctx->srq_cq_channel, 0);
    if (!ctx->srq_cq) {
        VERB_ERR("ibv_create_cq", -1);
        return -1;
    }

    /* Make sure that we get notified on the first completion */
    ret = ibv_req_notify_cq(ctx->srq_cq, 0);
    if (ret) {
        VERB_ERR("ibv_req_notify_cq", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    destroy_resources
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      This function cleans up resources used by the application
 */
void destroy_resources(struct context *ctx)
{
    int i;

    if (ctx->conn_id) {
        for (i = 0; i < ctx->qp_count; i++) {
            if (ctx->conn_id[i]) {
                if (ctx->conn_id[i]->qp &&
                    ctx->conn_id[i]->qp->state == IBV_QPS_RTS) {
```

```
                rdma_disconnect(ctx->conn_id[i]);
            }
            rdma_destroy_qp(ctx->conn_id[i]);
            rdma_destroy_id(ctx->conn_id[i]);
        }
    }

    free(ctx->conn_id);
  }

  if (ctx->recv_mr)
    rdma_dereg_mr(ctx->recv_mr);

  if (ctx->send_mr)
    rdma_dereg_mr(ctx->send_mr);

  if (ctx->recv_buf)
    free(ctx->recv_buf);

  if (ctx->send_buf)
    free(ctx->send_buf);

  if (ctx->srq_cq)
    ibv_destroy_cq(ctx->srq_cq);

  if (ctx->srq_cq_channel)
    ibv_destroy_comp_channel(ctx->srq_cq_channel);

  if (ctx->srq_id) {
    rdma_destroy_srq(ctx->srq_id);
    rdma_destroy_id(ctx->srq_id);
  }
}

/*
 * Function:    await_completion
 *
 * Input:
 *      ctx     The context object
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Waits for a completion on the SRQ CQ
 *
 */
int await_completion(struct context *ctx)
{
  int ret;
  struct ibv_cq *ev_cq;
```

```
    void *ev_ctx;

    /* Wait for a CQ event to arrive on the channel */
    ret = ibv_get_cq_event(ctx->srq_cq_channel, &ev_cq, &ev_ctx);
    if (ret) {
        VERB_ERR("ibv_get_cq_event", ret);
        return ret;
    }

    ibv_ack_cq_events(ev_cq, 1);

    /* Reload the event notification */
    ret = ibv_req_notify_cq(ctx->srq_cq, 0);
    if (ret) {
        VERB_ERR("ibv_req_notify_cq", ret);
        return ret;
    }

    return 0;
}

/*
 * Function:    run_server
 *
 * Input:
 *      ctx     The context object
 *      rai     The RDMA address info for the connection
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Executes the server side of the example
 */
int run_server(struct context *ctx, struct rdma_addrinfo *rai)
{
    int ret, i;
    uint64_t send_count = 0;
    uint64_t recv_count = 0;
    struct ibv_wc wc;
    struct ibv_qp_init_attr qp_attr;

    ret = init_resources(ctx, rai);
    if (ret) {
        printf("init_resources returned %d\n", ret);
        return ret;
    }

    /* Use the srq_id as the listen_id since it is already setup */
    ctx->listen_id = ctx->srq_id;
```

```
ret = rdma_listen(ctx->listen_id, 4);
if (ret) {
   VERB_ERR("rdma_listen", ret);
   return ret;
}

printf("waiting for connection from client...\n");
for (i = 0; i < ctx->qp_count; i++) {
   ret = rdma_get_request(ctx->listen_id, &ctx->conn_id[i]);
   if (ret) {
      VERB_ERR("rdma_get_request", ret);
      return ret;
   }

   /* Create the queue pair */
   memset(&qp_attr, 0, sizeof (qp_attr));

   qp_attr.qp_context = ctx;
   qp_attr.qp_type = IBV_QPT_RC;
   qp_attr.cap.max_send_wr = ctx->max_wr;
   qp_attr.cap.max_recv_wr = ctx->max_wr;
   qp_attr.cap.max_send_sge = 1;
   qp_attr.cap.max_recv_sge = 1;
   qp_attr.cap.max_inline_data = 0;
   qp_attr.recv_cq = ctx->srq_cq;
   qp_attr.srq = ctx->srq;
   qp_attr.sq_sig_all = 0;

   ret = rdma_create_qp(ctx->conn_id[i], NULL, &qp_attr);
   if (ret) {
      VERB_ERR("rdma_create_qp", ret);
      return ret;
   }

   /* Set the new connection to use our SRQ */
   ctx->conn_id[i]->srq = ctx->srq;

   ret = rdma_accept(ctx->conn_id[i], NULL);
   if (ret) {
      VERB_ERR("rdma_accept", ret);
      return ret;
   }
}

while (recv_count < ctx->msg_count) {
   i = 0;
   while (i < ctx->max_wr && recv_count < ctx->msg_count) {
      int ne;

      ret = await_completion(ctx);
      if (ret) {
         printf("await_completion %d\n", ret);
         return ret;
      }
```

```
        do {
            ne = ibv_poll_cq(ctx->srq_cq, 1, &wc);
            if (ne < 0) {
                VERB_ERR("ibv_poll_cq", ne);
                return ne;
            }
            else if (ne == 0)
                break;

            if (wc.status != IBV_WC_SUCCESS) {
                printf("work completion status %s\n",
                       ibv_wc_status_str(wc.status));
                return -1;
            }

            recv_count++;
            printf("recv count: %d, qp_num: %d\n", recv_count, wc.qp_num);

            ret = rdma_post_recv(ctx->srq_id, (void *) wc.wr_id,
                                 ctx->recv_buf, ctx->msg_length,
                                 ctx->recv_mr);
            if (ret) {
                VERB_ERR("rdma_post_recv", ret);
                return ret;
            }

            i++;
        }
        while (ne);
    }

    ret = rdma_post_send(ctx->conn_id[0], NULL, ctx->send_buf,
                         ctx->msg_length, ctx->send_mr, IBV_SEND_SIGNALED);
    if (ret) {
        VERB_ERR("rdma_post_send", ret);
        return ret;
    }

    ret = rdma_get_send_comp(ctx->conn_id[0], &wc);
    if (ret <= 0) {
        VERB_ERR("rdma_get_send_comp", ret);
        return -1;
    }

    send_count++;
    printf("send count: %d\n", send_count);
  }

  return 0;
}

/*
 * Function:    run_client
```

```
 *
 * Input:
 *     ctx     The context object
 *     rai     The RDMA address info for the connection
 *
 * Output:
 *     none
 *
 * Returns:
 *     0 on success, non-zero on failure
 *
 * Description:
 *     Executes the client side of the example
 */
int run_client(struct context *ctx, struct rdma_addrinfo *rai)
{
   int ret, i, ne;
   uint64_t send_count = 0;
   uint64_t recv_count = 0;
   struct ibv_wc wc;
   struct ibv_qp_init_attr attr;

   ret = init_resources(ctx, rai);
   if (ret) {
      printf("init_resources returned %d\n", ret);
      return ret;
   }

   for (i = 0; i < ctx->qp_count; i++) {
      memset(&attr, 0, sizeof (attr));

      attr.qp_context = ctx;
      attr.cap.max_send_wr = ctx->max_wr;
      attr.cap.max_recv_wr = ctx->max_wr;
      attr.cap.max_send_sge = 1;
      attr.cap.max_recv_sge = 1;
      attr.cap.max_inline_data = 0;
      attr.recv_cq = ctx->srq_cq;
      attr.srq = ctx->srq;
      attr.sq_sig_all = 0;

      ret = rdma_create_ep(&ctx->conn_id[i], rai, NULL, &attr);
      if (ret) {
         VERB_ERR("rdma_create_ep", ret);
         return ret;
      }

      ret = rdma_connect(ctx->conn_id[i], NULL);
      if (ret) {
         VERB_ERR("rdma_connect", ret);
         return ret;
      }
   }
```

```
while (send_count < ctx->msg_count) {
    for (i = 0; i < ctx->max_wr && send_count < ctx->msg_count; i++) {
        /* perform our send to the server */
        ret = rdma_post_send(ctx->conn_id[i % ctx->qp_count], NULL,
                             ctx->send_buf, ctx->msg_length, ctx->send_mr,
                             IBV_SEND_SIGNALED);
        if (ret) {
            VERB_ERR("rdma_post_send", ret);
            return ret;
        }

        ret = rdma_get_send_comp(ctx->conn_id[i % ctx->qp_count], &wc);
        if (ret <= 0) {
            VERB_ERR("rdma_get_send_comp", ret);
            return ret;
        }

        send_count++;
        printf("send count: %d, qp_num: %d\n", send_count, wc.qp_num);
    }

    /* wait for a recv indicating that all buffers were processed */
    ret = await_completion(ctx);
    if (ret) {
        VERB_ERR("await_completion", ret);
        return ret;
    }

    do {
        ne = ibv_poll_cq(ctx->srq_cq, 1, &wc);
        if (ne < 0) {
            VERB_ERR("ibv_poll_cq", ne);
            return ne;
        }
        else if (ne == 0)
            break;

        if (wc.status != IBV_WC_SUCCESS) {
            printf("work completion status %s\n",
                   ibv_wc_status_str(wc.status));
            return -1;
        }

        recv_count++;
        printf("recv count: %d\n", recv_count);

        ret = rdma_post_recv(ctx->srq_id, (void *) wc.wr_id,
                             ctx->recv_buf, ctx->msg_length, ctx->recv_mr);
        if (ret) {
            VERB_ERR("rdma_post_recv", ret);
            return ret;
        }
    }
    while (ne);
```

```
    }

    return ret;
}

/*
 * Function:    main
 *
 * Input:
 *      argc    The number of arguments
 *      argv    Command line arguments
 *
 * Output:
 *      none
 *
 * Returns:
 *      0 on success, non-zero on failure
 *
 * Description:
 *      Main program to demonstrate SRQ functionality.
 *      Both the client and server use an SRQ. ctx.qp_count number of QPs are
 *      created and each one of them uses the SRQ. After the connection, the
 *      client starts blasting sends to the server upto ctx.max_wr. When the
 *      server has received all the sends, it performs a send to the client to
 *      tell it that it can continue. Process repeats until ctx.msg_count
 *      sends have been performed.
 */
int main(int argc, char** argv)
{
    int ret, op;
    struct context ctx;
    struct rdma_addrinfo *rai, hints;

    memset(&ctx, 0, sizeof (ctx));
    memset(&hints, 0, sizeof (hints));

    ctx.server = 0;
    ctx.server_port = DEFAULT_PORT;
    ctx.msg_count = DEFAULT_MSG_COUNT;
    ctx.msg_length = DEFAULT_MSG_LENGTH;
    ctx.qp_count = DEFAULT_QP_COUNT;
    ctx.max_wr = DEFAULT_MAX_WR;

    /* Read options from command line */
    while ((op = getopt(argc, argv, "sa:p:c:l:q:w:")) != -1) {
        switch (op) {
        case 's':
            ctx.server = 1;
            break;
        case 'a':
            ctx.server_name = optarg;
            break;
        case 'p':
            ctx.server_port = optarg;
```

```
                break;
            case 'c':
                ctx.msg_count = atoi(optarg);
                break;
            case 'l':
                ctx.msg_length = atoi(optarg);
                break;
            case 'q':
                ctx.qp_count = atoi(optarg);
                break;
            case 'w':
                ctx.max_wr = atoi(optarg);
                break;
            default:
                printf("usage: %s -a server_address\n", argv[0]);
                printf("\t[-s server mode]\n");
                printf("\t[-p port_number]\n");
                printf("\t[-c msg_count]\n");
                printf("\t[-l msg_length]\n");
                printf("\t[-q qp_count]\n");
                printf("\t[-w max_wr]\n");
                exit(1);
            }
        }

        if (ctx.server_name == NULL) {
            printf("server address required (use -a)!\n");
            exit(1);
        }

        hints.ai_port_space = RDMA_PS_TCP;
        if (ctx.server == 1)
            hints.ai_flags = RAI_PASSIVE; /* this makes it a server */

        ret = rdma_getaddrinfo(ctx.server_name, ctx.server_port, &hints, &rai);
        if (ret) {
            VERB_ERR("rdma_getaddrinfo", ret);
            exit(1);
        }

        /* allocate memory for our QPs and send/recv buffers */
        ctx.conn_id = (struct rdma_cm_id **) calloc(ctx.qp_count,
                                                    sizeof (struct rdma_cm_id *));
        memset(ctx.conn_id, 0, sizeof (ctx.conn_id));

        ctx.send_buf = (char *) malloc(ctx.msg_length);
        memset(ctx.send_buf, 0, ctx.msg_length);
        ctx.recv_buf = (char *) malloc(ctx.msg_length);
        memset(ctx.recv_buf, 0, ctx.msg_length);

        if (ctx.server)
            ret = run_server(&ctx, rai);
        else
            ret = run_client(&ctx, rai);
```

```
    destroy_resources(&ctx);
    free(rai);

    return ret;
}
```