AIX Version 7.1

Technical Reference: Communications, Volume 3

SC23-6777-00

AIX Version 7.1

Technical Reference: Communications, Volume 3

SC23-6777-00

Note

Before using this information and the product it supports, read the information in "Notices" on page 53.

First Edition (September 2010)

This edition applies to AIX Version 7.1 and to all subsequent releases and modifications until otherwise indicated in new editions.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to Information Development, Department 04XA-905-6B013, 11501 Burnet Road, Austin, Texas 78758-3400. To send comments electronically, use this commercial Internet address: pserinfo@us.ibm.com. Any information that you supply may be used without incurring any obligation to you.

© Copyright IBM Corporation 2010.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 2010

About this document

This book provides experienced C programmers with complete detailed information about programming with OFED (Open Fabrics Enterprise Distribution) verbs over iWARP/RNIC fabrics in AIX®.

To use the book effectively, you should be familiar with commands, system calls, subroutines, file formats, and special files. This publication is also available on the documentation CD that is shipped with the operating system.

This book is part of the six-volume technical reference set, AIX Version 6.1 Technical Reference, that provides information on system calls, kernel extension calls, and subroutines in the following volumes:

- *AIX Version 6.1 Technical Reference: Base Operating System and Extensions Volume 1 and AIX Version 6.1 Technical Reference: Base Operating System and Extensions Volume 2* **provide information on system calls, subroutines, functions, macros, and statements associated with base operating system runtime services.**
- *AIX Version 6.1 Technical Reference: Communications Volume 1 and AIX Version 6.1 Technical Reference: Communications Volume 2* **provide information on entry points, functions, system calls, subroutines, and operations related to communications services.**
- *AIX Version 6.1 Technical Reference: Kernel and Subsystems Volume 1 and AIX Version 6.1 Technical Reference: Kernel and Subsystems Volume 2* **provide information about kernel services, device driver operations, file system operations, subroutines, the configuration subsystem, the communications subsystem, the low function terminal (LFT) subsystem, the logical volume subsystem, the M-audio capture and playback adapter subsystem, the printer subsystem, the SCSI subsystem, and the serial DASD subsystem.**

Highlighting

The following highlighting conventions are used in this book:

Bold	Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects.
Italics	Identifies parameters whose actual names or values are to be supplied by the user.
`Monospace`	Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type.

Case-sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command `is not found`. Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

© Copyright IBM Corp. 2010

Related Publications

- Operating system and device management
- Networks and communication management
- AIX Version 6.1 General Programming Concepts: Writing and Debugging Programs
- AIX Version 6.1 Communications Programming Concepts
- AIX Version 6.1 Kernel Extensions and Device Support Programming Concepts
- AIX Version 6.1 Files Reference

Technical Reference: Communications, Volume 3

Experienced C programmers can find complete detailed information about programming with OFED (Open Fabrics Enterprise Distribution) verbs over iWARP/RNIC fabrics in AIX.

To use the information effectively, you must be familiar with commands, system calls, subroutines, file formats, and special files. This topic is also available on the documentation CD that is shipped with the operating system.

To view or download the PDF version of this topic, select Technical Reference: Communications, Volume 3.

Downloading Adobe Reader: You need Adobe® Reader installed on your system to view or print this PDF. You can download a free copy from the Adobe website (www.adobe.com/products/acrobat/readstep.html).

Introduction

The **Technical Reference: Communications, Volume 3** book enables you to get started with OFED RNIC verbs programming over Chelsio RDMA RNIC fabrics in the AIX operating system. It enables applications that require high throughput and low latency to use the RDMA feature to the best.

Scope

The scope of this document is to give information on how to get started with the programming of OFED verbs over Chelsio RDMA iWARP/RNIC fabrics in the AIX operating system. The OFED programming enables applications that require high throughput and low latency to take advantage of RDMA.

Note: The verb layer for OFED verbs are common for iWARP, the InfiniBand architecture, and verbs derived from the InfiniBand architecture. Many InfiniBand terms are used but a few are not implemented for iWARP.

Terms

You can find the list of terms and their full forms used in the Technical Reference: Communications, Volume 3 book.

CM	Communications Manager
iWARP	Internet Wide Area RDMA Protocol also known as RDMA over Ethernet
RDMA	Remote Direct Memory Access
RNIC	RDMA Network Interface Controller (RNIC) - A network I/O adapter or embedded controller with iWARP, and Verbs functionality
OFED	Open Fabrics Enterprise Distribution
VERB	Abstract definition of functionality; not an API

Hardware considerations

AIX platforms support RDMA Network Interface Controller (RNIC) with 10 Gbps Internet Wide Area RDMA Protocol (iWARP), also called as RDMA over IP or ethernet.

© Copyright IBM Corp. 2010

Software considerations

The AIX OFED Verbs is based on the OFED 1.4 code of OpenFabrics Alliance. Currently, only iWARP/RNIC is supported. The 32-bit, and 64-bit user applications are supported. The following libraries are linked with the application.

- Librdmacm
- Libibverbs

Libraries

Librdmacm

The **librdmacm** library provides the connection management functionality and a generic RDMA set of CM interfaces that runs over iWARP .

A single `/dev/rdma/rdma_cm` device node is used by the user space to communicate with the kernel, regardless of the number of adapters or ports present.

Applications that wish to run over any RDMA device must use this library.

Libibverbs

Libibverbs is a library that enables user-space processes to use RDMA **verbs**.

Libibverbs is a library that enables user-space processes to use RDMA **verbs** as described in the InfiniBand Architecture Specification (see, http://www.infinibandta.org) and the RDMA Protocol Verbs Specification (see, http://tools.ietf.org/html/draft-hilland-rddp-verbs-00).

Several `/dev/rdma/uverbsN` character device nodes are used to handle communication between the library **libibverbs** and the kernel **ib_uverbs** layer. There is one such device per RNIC adapter registered with OFED core (uverbs1, uverbs2, etc). The library writes commands corresponding to the verb to execute on the appropriate device.

Configuration

A file named `cxgb3.driver` must exist in the directory `/etc/libibverbs.d`, which enables you to use the driver for the Chelsio T3 Ethernet adapter 10 GB iWARP, by default.

The `cxgb3.driver` file must contain the following code:

```
# cat /etc/libibverbs.d/cxgb3.driver
driver cxgb3
```

Use the environment variable **IBV_CONFIG_DIR** to use another directory than the `/etc/libibverbs.d/` directory.

Commands

ibv_devices command

Lists the RDMA devices available for use from user space.

ibv_devinfo command

Prints information about RNIC devices available for use from user space.

Syntax

```
ibv_devinfo [-v] { [-d<dev>] [-i<port>] } | [-l]
```

Flags

-d *dev*	Uses RDMA device <*dev*> (default first device found).
-i *port*	Uses port <*port*> of RDMA device (default all ports).
-l	Prints only the RDMA devices name.
-v	Prints all the attributes of the RDMA device(s).

ofedctrl command

Loads and unloads the kernel extension, **ofed_core**.

Syntax

```
ofedctrl { [-k <kernext-name>] -l | u | q } | -h
```

Flags

-k *kernext-name*	Specifies the kernel extension path. The default is `/usr/lib/drivers/ofed_core`.
-l	Loads the kernext.
-u	Unloads the kernext.
-q	Indicates whether the kernext is loaded or not.
-h	Specifies the usage.

rping command

Tests the RDMA CM connection by using the RDMA ping-pong test.

Syntax

```
rping -s [-v] [-V] [-d] [-P] [-a address] [-p port] [-C message_count] [-S message_size]
```

```
rping -c [-v] [-V] [-d] -a address [-p port] [-C message_count] [-S message_size]
```

Description

The **rping** command establishes a reliable RDMA connection between two nodes using the **librdmacm** library. Optionally the **rping** command also performs RDMA transfers between the nodes, then disconnects.

Flags

-s	Runs as the server.
-c	Runs as the client.
-a *address*	Specifies the network address to bind the connection to, on the server and specifies the server address to connect to, on the client.
-p	Specifies the port number for the listening server.
-v	Displays the ping data.
-V	Validates the ping data.
-d	Displays the debug information.
-C *message_count*	Specifies the number of messages to transfer over each connection. By default, the value is infinite.
-S *message_size*	Specifies the size of each message transferred, in bytes. By default, the value is 100.
-P	Runs the server in persistent mode. This allows multiple **rping** clients to connect to a single server instance and the server will run until the instance is killed.

Communication Manager Overview

The communication manager (RDMA_CM) is used to setup reliable connection data transfers.

The communication manager provides an RDMA transport neutral interface for establishing connections. The API is based on sockets, but adapted for queue pair (QP) based semantics: communication is over a specific RDMA device, and data transfers are message based.

The RDMA CM via the **librdmacm** library provides only the communication management (connection setup and teardown) portion of an RDMA API. It works in conjunction with the verbs API via the **libibverbs** library for data transfers.

Resources (objects) operated on by Verbs

You can find the list of resources and their descriptions operated on by verbs.

Completion Queue (CQ):
A queue (FIFO) which contains CQEs. Associated with a queue pair, they are used to receive completion notifications and events.

Completion Queue Entry (CQE):
An entry in the CQ that describes the information about the completed WR (status, size, etc.)

Event Channel:
Used to report communication events. 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.

Memory Region (MR):
A set of memory buffers that are already registered with access permissions. These buffers require registration in order for the network adapter to make use of them.

Protection Domain (PD):
Protection domains enable a client to associate multiple resources, such as queue pairs, and memory regions, within a domain of trust. The client can then grant access rights for sending/receiving data within the protection domain to others that are on the RDMA fabric.

Queue Pair (QP):
Queue pairs (QPs) contain a send queue, for sending outbound messages and requesting RDMA operations, and a receive queue for receiving incoming messages or immediate data.

Scatter /Gather Elements (SGE):
An entry to a pointer to a full or a part of a local registered memory block. The element holds 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 Work Request (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.

Work Queue (WQ):
Send Queue or Receive Queue.

Work Queue Element (WQE):
An element in a work queue.

Work Request (WR):
A request that was posted by a user to a work queue.

Available communication operations

Send / Send with immediate

The send operation enables you to send data to the receive queue of a remote QP. The receiver must have previously posted a receive buffer to receive the data. The sender does not have any control over where the data resides in the remote host.

Optionally, an immediate 4 byte value is 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.

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

RDMA read
The **RDMA read** operation reads a memory region from the remote host. You must specify the remote virtual address and a local memory address where the read information is copied. Prior to performing the 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 to the remote host.

RDMA write / RDMA write with immediate
The **RDMA write** operation is similar to the **RDMA read** operation, 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 do notify the remote host of the immediate value.

Atomic Operations
The atomic operations are not supported by the iWARP specifications.

The InfiniBand architecture supports these operations.

Note: The InfiniBand architecture is not supported now.

Transport modes
Transport modes supports only Reliable Connection (RC).

The available transport modes are:
- 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 a order.
- An RC connection is very similar to a TCP connection.

Connection Establishment through RDMA_CM

The RDMA CM only provides the communication management (connection setup and teardown) portion of an RDMA API. It works in conjunction with the verbs API defined by the **libibverbs** library. The **libibverbs** library provides the interfaces required to send and receive data.

Client Operation
The client operation section provides a general overview of the basic operation for the active, or client side of communication.

A general connection flow is described in the following:

rdma_create_event_channel
Creates a channel to receive events.

rdma_create_id
Allocates an rdma_cm_id that is conceptually similar to a socket.

rdma_resolve_addr
Obtains a local RDMA device to reach the remote address.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_ADDR_RESOLVED event.

rdma_ack_cm_event
Acknowledges the received event.

rdma_create_qp
Allocates a QP for the communication.

rdma_resolve_route
Determines the route to the remote address.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_ROUTE_RESOLVED event.

rdma_ack_cm_event
Acknowledges the received event.

rdma_connect
Connects to the remote server.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_ESTABLISHED event.

rdma_ack_cm_event
Acknowledges the received event.

Performs data transfer over the connection.

rdma_disconnect
Tears down the connection.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_DISCONNECTED event.

rdma_ack_cm_event
Acknowledges the event.

rdma_destroy_qp
Destroys the QP.

rdma_destroy_id
Releases the rdma_cm_id.

rdma_destroy_event_channel
Releases the event channel.

Note: The example shows the client initiating the disconnect, but either side of a connection might initiate the disconnect process.

Server Operation

This section provides a general overview of the basic operation for the passive, or server, side of communication.

A general connection flow would be:

rdma_create_event_channel
Creates a channel to receive events.

rdma_create_id
Allocates an rdma_cm_id that is conceptually similar to a socket.

rdma_bind_addr
Sets the local port number to listen on.

rdma_listen
Begins listening for connection requests.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_CONNECT_REQUEST event with a new rdma_cm_id.

rdma_create_qp
Allocates a QP for the communication on the new rdma_cm_id.

rdma_accept
Accepts the connection request.

rdma_ack_cm_event
Acknowledges the event.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_ESTABLISHED event.

rdma_ack_cm_event
Acknowledges the event.

Performs the data transfer over the connection.

rdma_get_cm_event
Waits for the RDMA_CM_EVENT_DISCONNECTED event.

rdma_ack_cm_event
Acknowledges the event.

rdma_disconnect
Tears down the connection.

rdma_destroy_qp
Destroys the QP.

rdma_destroy_id
Releases the connected rdma_cm_id.

rdma_destroy_id
Releases the listening rdma_cm_id.

rdma_destroy_event_channel
Releases the event channel.

Open sources connection setup application examples

The best way to start the OFED programming is to go through the **libibverbs** and **librdmacm** man pages along with some code examples and doing some runs. Specifically, the **rping** command example that uses both **libibverbs** and **librdmacm** for the connected service.

Rping

The **rping** command sets an RDMA CM connection and does an RDMA ping-pong test.

You can find more information on the **rping** command in Open Source OpenFabrics Alliance OFED 1.4 at http://www.openfabrics.org/

An example using RDMA_CM module

You can find a simple example presented to the OFED community during the LinuxConf.eu 2007.

You can find a simple example presented to the OFED community during the LinuxConf.eu 2007 at http://www.digitalvampire.org/rdma-tutorial-2007/

Client (active) example

An example where the client is active.

```
/*
 * build:
 * cc -o client client.c -lrdmacm -libverbs
 *
 * usage:
 * client <servername> <val1> <val2>
 *
 * connects to server, sends val1 via RDMA write and val2 via send,
 * and receives val1+val2 back from the server.
 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <arpa/inet.h>

#include <rdma/rdma cma.h>
enum   {
        RESOLVE TIMEOUT MS        = 5000,
};
struct pdata {
        uint64_t buf va;
        uint32_t buf rkey;
};

int main(int argc, char   *argv[ ])
{
       struct pdata         server pdata;
       struct rdma_event channel   *cm_channel;
       struct rdma_cm_id       *cm_id;
       struct rdma_cm_event     *event;
       struct rdma_conn_param    conn_param = { };
       struct ibv_pd        *pd;
       struct ibv_comp_channel    *comp_chan;
       struct ibv_cq        *cq;
       struct ibv_cq        *evt_cq;
       struct ibv_mr        *mr;
       struct ibv_qp_init_attr    qp attr = { };
       struct ibv_sge        sge;
       struct ibv_send_wr      send_wr = { };
       struct ibv_send_wr       *bad send wr;
       struct ibv_recv_wr      recv wr = { };
       struct ibv_recv_wr      *bad recv wr;
       struct ibv_wc        wc;
       void             *cq context;
       struct addrinfo        *res, *t;
       struct addrinfo        hints =   {
                          .ai_family    = AF INET,
                          .ai_socktype  = SOCK STREAM
                     };
   int             n;
   uint32_t           *buf;
   int             err;
   /* Set up RDMA CM structures */
   cm_channel = rdma_create_event_channel();
   if (!cm_channel)  return 1;
   err = rdma_create_id(cm_channel, &cm_id, NULL, RDMA_PS_TCP);
   if (err)
     return err;
   n = getaddrinfo(argv[1], "20079", &hints, &res);
   if (n < 0)
     return   1;

   /* Resolve server address and route */
```

```
for (t = res; t; t = t->ai next) {
  err = rdma_resolve_addr(cm_id, NULL, t->ai_addr, RESOLVE_TIMEOUT_MS);
  if (!err)
    break;
}
if (err)
  return err;
err = rdma_get_cm_event(cm_channel, &event);
if (err)
  return err;
if (event->event != RDMA_CM_EVENT_ADDR_RESOLVED)
  return 1;
rdma_ack_cm_event(event);
err = rdma_resolve_route(cm_id, RESOLVE_TIMEOUT_MS);
if (err)
  return err;
err = rdma_get_cm_event(cm_channel, &event);
if (err)
  return err;
if (event->event != RDMA_CM_EVENT_ROUTE_RESOLVED)
  return 1;
rdma_ack_cm_event(event);

/* Create verbs objects now that we know which device to use */
pd = ibv_alloc_pd(cm_id->verbs);
if (!pd)
  return 1;
comp chan = ibv_create_comp_channel(cm_id->verbs);
if (!comp_chan)
  return 1;
cq = ibv_create_cq(cm_id->verbs, 2,NULL, comp_chan, 0);
if (!cq)
  return 1;

if (ibv_req_notify_cq(cq,  0))
  return 1;

buf = calloc(2, sizeof (uint32_t));
if (!buf)
  return 1;
mr = ibv_reg_mr(pd, buf,2 * sizeof(uint32_t), IBV_ACCESS_LOCAL_ WRITE);
if (!mr)
  return 1;
qp_attr.cap.max  send_wr = 2;
qp_attr.cap.max  send_sge = 1;
qp_attr.cap.max  recv_wr = 1;
qp_attr.cap.max  recv_sge = 1;
qp_attr.send_cq           = cq;
qp_attr.recv_cq           = cq;
qp_attr.qp_type           = IBV_QPT_RC;
err = rdma_create_qp(cm_id, pd, &qp_attr);
if (err)
  return err;
conn_param.initiator_depth = 1;
conn_param.retry_count = 7;

/* Connect to server */
err = rdma_connect(cm_id, &conn_param);
if (err)
  return err;
err = rdma_get_cm_event(cm_channel,&event);
if (err)
  return err;
if (event->event != RDMA_CM_EVENT_ESTABLISHED)
  return 1;
memcpy(&server_pdata, event->param.conn.private_data, sizeof server_pdata);
rdma_ack_cm_event(event);
```

```
   /* Prepost receive */
   sge.addr    = (uintptr_t) buf;
   sge.length  = sizeof (uint32_t);
   sge.lkey    = mr->lkey;
   recv_wr.wr_id = 0;
   recv_wr.sg_list = &sge;
   recv_wr.num_sge = 1;
   if (ibv_post_recv(cm_id->qp, &recv_wr, &bad_recv_wr))
     return 1;

   /* Write/send two integers to be added */
   buf[0] = strtoul(argv[2], NULL, 0);
   buf[1] = strtoul(argv[3], NULL, 0);
   printf("%d + %d = ", buf[0], buf[1]);
   buf[0]  = htonl(buf[0]);
   buf[1]  = htonl(buf[1]);

   sge.addr    = (uintptr_t) buf;
   sge.length  = sizeof (uint32_t);
   sge.lkey    =  mr->lkey;
   send_wr.wr_id  = 1;
   sendwr.opcode  = IBV_WR_RDMA_WRITE;
   send _wr.sg_list   = &sge;
   send_wr.num_sge  = 1;
   send_wr.wr.rdma.rkey = ntohl(server_pdata.buf_rkey);
   send_wr.wr.rdma.remote_addr = ntohll(server_pdata.buf_va);
   if (ibv post send(cm id->qp, &send_wr,  &bad_send_wr))
     return 1;
   sge.addr    = (uintptr_t) buf + sizeof (uint32_t);
   sge.length  = sizeof (uint32_t);
   sge.lkey    = mr->lkey;
   send_wr.wr_id                = 2;
   send_wr.opcode                = IBV_WR_SEND;
   send_wr.send_flags            = IBV_SEND_SIGNALED;
   send_wr.sg_list               =&sge;
   send_wr.num_sge              = 1;
   if (ibv_post_send(cm_id->qp, &send_wr,&bad_send_wr))
     return 1;

   /* Wait for receive completion */
   while (1) {           if (ibv_get_cq_event(comp_chan,&evt_cq, &cq_context))
     return 1;
   if (ibv_req_notify_cq(cq, 0))
     return 1;
   if (ibv_poll_cq(cq, 1, &wc) != 1)
     return 1;
   if (wc.status != IBV_WC_SUCCESS)
     return 1;
   if (wc.wr_id == 0) {
     printf("%d\n", ntohl(buf[0]));
     return 0;
   }
    }
    return 0;
  }
```

Server (passive) example

```
/*
 * build:
 * cc -o server server.c -lrdmacm -libverbs
 *
 * usage:
 * server
 *
 * waits for client to connect, receives two integers, and sends their
 * sum back to the client.
 */
#include <stdlib.h>
#include <stdint.h>
#include <arpa/inet.h>
#include <rdma/rdma cma.h>
enum   {
        RESOLVE_TIMEOUT_MS         = 5000,
};

struct pdata {
        uint64_t         buf_va;
        uint32_t         buf_rkey;
};
int main(int argc, char   *argv[ ])
{
        struct pdata        rep_pdata;
        struct rdma_event_channel   *cm_channel;
        struct rdma_cm_id      *listen_id;
        struct rdma_cm_id      *cm_id;
        struct cm_event       *event;
        struct rdma_conn_param    conn_param = { };
        struct ibv_pd        *pd;
        struct ibv_comp_channel    *comp_chan;
        struct ibv_cq        *cq;
        struct ibv_cq        *evt_cq;
        struct ibv_mr        *mr;
        struct qp_init_attr     qp_attr = { };
        struct ibv_sge        sge;
        struct ibv_send_wr      send_wr = { };
        struct ibv_send_wr      *bad_send_wr;
        struct ibv_recv_wr      recv_wr = { };
        struct recv_wr        *bad_recv_wr;
        struct ibv_wc        wc;
        void             *cq_context;
        struct sockaddr_in      sin;
        uint32_t          *buf;
        int             err;

         /* Set up RDMA CM structures */
        cm_channel = rdma_create_event_channel();
        if (!cm_channel)
                 return 1;
        err = rdma_create_id(cm_channel,&listen_id, NULL, RDMA_PS_TCP);
        if (err)
                 return err;
        sin.sin_family = AF_INET;
        sin.sin_port     = htons(20079);
        sin.sin_addr.s_addr = INADDR_ANY;
                                                         7
         /* Bind to local port and listen for connection request */
        err = rdma_bind_addr(listen_id, (struct sockaddr   *) &sin);
        if (err)
               return 1;
        err = rdma_listen(listen_id,  1);
        if (err)
               return 1;
        err = rdma_get_cm_event(cm_channel, &event);
        if (err)
               return err;
        if (event->event != RDMA_CM_EVENT_CONNECT_REQUEST)
               return 1;
        cm_id = event->id;
        rdma_ack_cm_event(event);

        /* Create verbs objects now that we know which device to use */
        pd = ibv_alloc_pd(cm_id->verbs);
        if (!pd)
```

```
                return 1;
        comp_chan = ibv_create_comp_channel(cm_id->verbs);
        if (!comp_chan)
                return 1;
        cq = ibv_create_cq(cm_id->verbs,   2,    NULL, comp_chan, 0);
        if (!cq)
                return 1;
        if (ibv_req_notify_cq(cq, 0))
                return 1;
        buf = calloc(2, sizeof (uint32_t));
        if (!buf)
                return 1;
        mr = ibv_reg_mr(pd, buf, 2 * sizeof (uint32_t),
                    IBV_ACCESS_LOCAL_WRITE |
                    IBV_ACCESS_REMOTE_READ |
                    IBV_ACCESS_REMOTE_WRITE);
        if (!mr)
                 return 1;

      qp_attr.cap.max_send_wr = 1;
      qp_attr.cap.max_send_sge = 1;
      qp_attr.cap.max_recv_wr = 1;
      qp_attr.cap.max_recv_sge = 1;
      qp_attr.send_cq = cq;
      qp_attr.recv_cq = cq;
      qp_attr.qp_type = IBV_QPT_RC;

      err = rdma_create_qp(cm_id, pd, &qp_attr);
      if (err)
            return err;

       /* Post receive before accepting connection */
       sge.addr    = (uintptr_t) buf + sizeof (uint32_t);
       sge.length  = sizeof (uint32_t);
       sge.lkey    = mr->lkey;
       recv_wr.sg_list =  &sge;
       recv_wr.num_sge    = 1;
       if (ibv_post_recv(cm_id->qp, &recv_wr,  &bad_recv_wr))
             return 1;
       rep_pdata.buf_va = htonll((uintptr_t) buf);
       rep_pdata.buf_rkey = htonl(mr->rkey);
       conn_param.responder_resources = 1;
       conn_param.private_data          = &rep_pdata;
       conn_param.private_data_len = sizeof rep_pdata;

       /* Accept connection */
       err = rdma_accept(cm_id,     &conn_param);
       if (err)
               return 1;
       err = rdma_get_cm_event(cm_channel, &event);
       if (err)
                return err;
       if (event->event != RDMA_CM_EVENT_ESTABLISHED)
              return 1;
       rdma_ack_cm_event(event);

        /* Wait for receive completion */
       if (ibv_get_cq_event(comp_chan,  &evt_cq, &cq_context))
             return 1;
       if (ibv_req_notify_cq(cq,  0))
             return 1;
       if (ibv_poll_cq(cq, 1, &wc)    < 1)
             return 1;
       if (wc.status != IBV_WC_SUCCESS)
             return 1;

      /* Add two integers and send reply back */
```

```
        buf[0]  = htonl(ntohl(buf[0])  +  ntohl(buf[1]));
        sge.addr    = (uintptr_t) buf;
        sge.length  = sizeof (uint32_t);
        sge.lkey    = mr->lkey;

        send_wr.opcode = IBV_WR_SEND;
        send_wr.send_flags = IBV_SEND_SIGNALED;
        send_wr.sg_list    = &sge;
        send_wr.num_sge = 1;
        if (ibv_post_send(cm id->qp,     &send_wr, &bad_send_wr))
              return 1;

         /* Wait for send completion */
         if (ibv_get_cq_event(comp_chan,     &evt_cq, &cq_context))
              return 1;
         if (ibv_poll_cq(cq, 1, &wc)    < 1)
              return 1;
         if (wc.status != IBV_WC_SUCCESS)
              return 1;
         ibv_ack_cq_events(cq,   2);
              return  0;
  }
```

Verbs API

If an AIX application needs to determine which verbs API to use (OFED iWARP/RNIC verbs or AIX IB verbs) to communicate with a specific destination, here is an example in pseudo-code, to test the result of the **rdma_resolve_addr** command on the required remote address to know if OFED iWARP/RNIC verbs can be used.

The code returns:

- 0, if the destination is reachable using OFED iWARP/RNIC verbs.
- An error, if the communication with the destination cannot be established through an RNIC device, but might be possible with the InfiniBand architecture.

```
/*The following check_ofed_verbs_support routine does:
/*     - Call rdma_create_event_channel to open a channel event     */
/*     - Calls rdma_create_id() to get a cm_id             */
/*     - And then calls rdma_resolve_addr()             */
/*     - Get the communication event                */
/*     - Returns the event status:                 */
/*        0: OK                         */
/*        error: NOK  output device may be not a RNIC device     */
/*     - Calls rdma_destroy_id() to delete the cm_id created       */
/*     - Call rdma_destroy_event_channel  to close a channel event    */

   int check_ofed_verbs_support (struct sockaddr *remoteaddr)
   {
      struct rdma_event channel *cm_channel;
      struct rdma_cm_id *cm_id;
      int ret=0;
      cm_channel = rdma_create_event_channel();
      if (!cm_channel)  {
        fprintf(stderr,"rdma_create_event_channel error\n");
        return -1;
      }
      ret = rdma_create_id(cm_channel, &cm_id, NULL, RDMA_PS_TCP);
      if (ret) {
        fprintf(stderr,"rdma_create_id: %d\n", ret);
        rdma_destroy_event_channel(cm_channel);
        return(ret);
      }
      ret = rdma_resolve_addr(cm_id, NULL, remoteaddr, RESOLVE_TIMEOUT_MS);
      if (ret) {
        fprintf(stderr,"rdma_resolve_addr: %d\n", ret);
        goto out;
      }
      ret = rdma_get_cm_event(cm_channel, &event);
      if (ret) {
        fprintf(stderr," rdma_get_cm_event() failed\n");
        goto out;
      }
      ret = event->status;
      rdma_ack_cm_event(event);
   out:
      rdma_destroy_id(cm_id);
      rdma_destroy_event_channel(cm_channel);
      return(ret);
   }
```

Functions (Verbs)

Librdmacm Library

The API user space is described in the `/usr/include/rdma/rdma_cma.h` file.

Manual pages have been created to describe the various interfaces and test programs that are available. For a full list of interfaces and test programs, refer to the rdma_cm manual page.

Returned Error Rules

The **librdmacm** functions return 0 to indicate success, and a negative value to indicate failure.

If a function operates asynchronously, a return value of 0 means that the operation was successfully started. The operation might still return an error. You must check the status of the related event. If the return value is -1, then **errno** can be examined for additional information of the failure. If the return value is < -1, then additional error reasons can be obtained by comparing the returned value with the values listed in `include/sys/errno.h`.

=0	Success
= -1	Error - see `include/sys/errno*.h` for **errno**
< -1	Error - see `include/sys/errno*.h`
-ENOSYS	Non-supported verbs

Supported Verbs

You can find a list of supported verbs.

Event Channel Operations:

rdma_create_event_channel:

Opens a channel that is used to report communication events.

Syntax

```
#include <rdma/rdma_cma.h>
struct rdma_event_channel *rdma_create_event_channel(void);
```

Description

The **rdma_create_event_channel** function reports the asynchronous events through event channels. Each event channel maps to a file descriptor.

Notes:

- Event channels are used to direct all events on an **rdma_cm_id**. You might require multiple event channels when you are managing a large number of connections or CM ids.
- All created event channels must be destroyed by calling the **rdma_destroy_event_channel** function. You must call the **rdma_get_cm_event** function to retrieve events on an event channel.

Parameters

void	No arguments.

Return Value

The **rdma_create_event_channel** function returns 0 on success, and NULL if the request fails.

rdma_destroy_event_channel:

Closes an event communication channel.

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_event_channel(struct rdma_event_channel *channel);
```

Description

The **rdma_destroy_event_channel** function releases all resources associated with an event channel and closes the associated file descriptor.

Note: All rdma_cm_ids associated with the event channel must be destroyed, and all returned events must be acknowledged before calling this function.

Parameters

channel	Specifies the communication channel to be destroyed.

Return Value

There is no return value.

Connection Manager (CM) ID Operations:

rdma_create_id:

Allocates a communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_id(struct rdma_event_channel *channel, struct rdma_cm_id **id, void *context, enum rdma_port_space ps);
```

Description

The **rdma_create_id** function creates an identifier that is used to track communication information.

Notes:

- The rdma_cm_ids are conceptually equivalent to a socket for RDMA communication. The difference is that the RDMA communication requires explicit binding to a specified RDMA device before communicating, and most operations are asynchronous in nature.

- You must release the rdma_cm_id by calling the **rdma_destroy_id** function.

Port Spaces: RDMA_PS_TCP provides reliable, connection-oriented QP. Unlike TCP, the RDMA port space provides stream based communication.

Parameters

channel	Specifies the communication channel that the events associated with the allocated rdma_cm_id are reported on.
id	Specifies a reference where the allocated communication identifier will be returned.
context	Indicates the user specified context associated with the rdma_cm_id.
ps	Specifies the RDMA port space.

Return Values

The **rdma_create_id** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the channel or id parameter is NULL or unable to query RDMA device.
-EPROTONOSUPPORT	ps is not RDMA_PS_TCP.
-ENOMEM	There is not enough memory to allocate the id by malloc.
-ENODATA	The write operation on channel->fd failed.
-ENODEV	Unable to get the RDMA device .

rdma_destroy_id:

Releases a communication identifier.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_destroy_id(struct rdma_cm_id *id);
```

Description

The **rdma_destroy_id** function destroys the specified rdma_cm_id and cancels any outstanding asynchronous operation.

Note: You must free any associated QP with the rdma_cm_id before calling the **rdma_destroy_id** routine and acknowledge an related events.

Parameters

id	Specifies the communication identifier to destroy.

Return Values

The **rdma_destroy_id** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the channel or id parameter is NULL.
-ENODATA	The write operation on id->channel->fd failed.

rdma_migrate_id:

Moves an rdma_cm_id to a new event channel.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_migrate_id(struct rdma_cm_id *id, struct rdma_event_channel *channel);
```

Description

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

Notes:

- You must not poll for events on the rdma_cm_id's current event channel or run any other routines on the rdma_cm_id while migrating between channels.
- The **rdma_migrate_id** operation stops if there are any unacknowledged events on the current event channel.

Parameters

id	Specifies the communication identifier to migrate.
channel	Specifies the new event channel for the rdma_cm_id events.

Return Values

The **rdma_migrate_id** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the channel or id parameter is NULL.
-ENODATA	The write operation on channel->fd failed.

rdma_bind_addr:

Binds an RDMA identifier to a source address.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_bind_addr(struct rdma_cm_id *id, struct sockaddr *addr);
```

Description

The **rdma_bind_addr** function associates a source address with an rdma_cm_id. The address might be a wildcard value. If an rdma_cm_id is bound to a local address, the identifier is also bound to a local RDMA device.

Notes:

- The **rdma_bind_addr** routine is called before calling the **rdma_listen** routine to bind to a specific port number. It might also be called on the active side of a connection before calling the **rdma_resolve_addr** routine to bind to a specific address.
- If the **rdma_bind_addr** routine is used to bind to port 0, the rdma_cm selects an available port that can be retrieved with **rdma_get_src_port**.

Parameters

id	Specifies the RDMA identifier.
addr	Specifies the local address information. Wildcard values are permitted.

Return Values

The **rdma_bind_addr** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the id parameter is NULL or the family is a Bad Protocol family.
-ENODATA	The write operation on id->channel->fd failed.
-ENOMEM	The memory is not enough to allocate by malloc.

rdma_resolve_addr:

Resolves the destination and optional source addresses.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_resolve_addr(struct rdma_cm_id *id, struct sockaddr *src_addr, struct sockaddr *dst_addr, int timeout_ms);
```

Description

The **rdma_resolve_addr** function resolves the destination and optional source addresses from IP address to an RDMA address. If successful, the specified rdma_cm_id is bound to a local device.

Notes:

- The **rdma_resolve_addr** routine 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, and the situation is same as if **rdma_bind_addr** was 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.
- The **rdma_resolve_addr** routine is run from the active side of a connection, before calling **rdma_resolve_route** and **rdma_connect**.

Parameters

id	Specifies the RDMA identifier.
src_addr	Specifies the source address information and this parameter might be NULL.
dst_addr	Specifies the destination address information.
timeout_ms	Specifies the time of resolution.

Return Values

The **rdma_resolve_addr** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the id parameter is NULL or the family is a Bad Protocol family.
-ENODATA	The write operation on id->channel->fd failed.

rdma_resolve_route:

Resolves the route information required to establish a connection.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_resolve_route(struct rdma_cm_id *id, int timeout_ms);
```

Description

The **rdma_resolve_route** function resolves an RDMA route to the destination address in order to establish a connection. The destination address must have already been resolved by calling the **rdma_resolve_addr** subroutine.

Note: The **rdma_resolve_route** routine is called on the client side of a connection, after calling the **rdma_resolve_addr** routine, but before calling the **rdma_connect** routine.

Parameters

id	Specifies the RDMA identifier.
timeout_ms	Specifies the time of resolution.

Return Values

The **rdma_resolve_route** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the id parameter is NULL.
-ENODATA	The write operation on id->channel->fd failed.

rdma_connect:

Initiates an active connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

Description

The **rdma_connect** function initiates a connection request to a remote destination.

Note: You must have resolved a route to the destination address by calling **rdma_resolve_route** before calling the **rdma_connect** routine.

Connection Properties

The following properties are used to configure the communication specified by the *conn_param* parameter when connecting or establishing a datagram communication.

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. **private_data** might be NULL if it is not required.

private_data_len:
Specifies the size of the user-controlled data buffer.

responder_resources:
The maximum number of outstanding RDMA read operations that the local side accepts from the remote side. Applies only to RDMA_PS_TCP. The **responder_resources** 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 operations that the local side has to the remote side. Applies only to RDMA_PS_TCP. The **initiator_depth** 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 the hardware flow control is available. The **flow_control** value is exchanged with the remote peer and is not used to configure the QP. Applies only to RDMA_PS_TCP, and is specific to the InfiniBand architecture.

retry_count:
The maximum number of times that a data transfer operation must be tried on the connection when an error occurs. The **retry_count** setting controls the number of times to retry send RDMA, and atomic operations when time outs occur. Applies only to RDMA_PS_TCP, and is specific to the InfiniBand architecture.

rnr_retry_count:
The maximum number of times that a send operation from the remote peer is tried on a connection after receiving a **receiver not ready** (RNR) error. RNR errors are generated when a send request arrives before a buffer is posted to receive the incoming data. Applies only to RDMA_PS_TCP, and is specific to the InfiniBand architecture.

srq: Specifies if the QP associated with the connection is using a shared receive queue. The **srq** field is ignored by the library if a QP is created on the rdma_cm_id. Applies only to RDMA_PS_TCP and is currently not supported.

qp_num:
Specifies the QP number associated with the connection. The **qp_num** field is ignored by the library if a QP is created on the rdma_cm_id. Applies only to RDMA_PS_TCP.

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

Parameters

id	Specifies the RDMA identifier.
conn_param	Specifies the connection parameters.

Return Values

The **rdma_connect** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	• If the id or conn_param parameter is NULL • If the parameter conn_param->responder_resources is bigger than the local RDMA device attribute **max_qp_rd_atom**, and remote RDMA device attribute **max_qp_init_rd_atom** • If the parameter onn_param->initiator_depth is bigger than the local RDMA device attribute **max_qp_init_rd_atom**, and remote RDMA device attribute **max_qp_rd_atom**
-EPROTONOSUPPORT	id->ps is not RDMA_PS_TCP.
-ENODATA	The write operation on id->channel->fd failed.

rdma_listen:

Listens for incoming connection requests.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_listen(struct rdma_cm_id *id, int backlog);
```

Description

The **rdma_listen** function initiates a listen for incoming connection requests lookup. The listen operation is restricted to the locally bound source addresses.

Notes:

- You must have bound the rdma_cm_id to a local address by calling **rdma_bind_addr** before calling the **rdma_listen** routine.
- If the rdma_cm_id is bound to a specific IP address, the listen operation is restricted to that address and the associated RDMA device.
- If the rdma_cm_id is bound to an RDMA port number only, the listen operation occurs across all RDMA devices.

Parameters

id	Specifies the RDMA identifier.
backlog	Specifies the backlog of incoming connection requests.

Return Values

The **rdma_listen** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the id parameter is NULL.
-ENODATA	The write operation on id->channel->fd failed.
-ENOMEM	There is not enough space to allocate by malloc.
-ENODEV	Unable to get an RDMA device.

rdma_accept:

Accepts a connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_accept(struct rdma_cm_id *id, struct rdma_conn_param *conn_param);
```

Description

The **rdma_accept** function is called from the listening side to accept a connection lookup request.

Notes:

- The **rdma_accept** routine is not called on a listening rdma_cm_id, unlike the socket accept routine. Instead, after calling **rdma_listen**, you have to wait for a connection request event to occur.
- Connection request events gives you a newly created rdma_cm_id, similar to a new socket, but the rdma_cm_id is bound to a specific RDMA device. The **rdma_accept** routine is called on the new rdma_cm_id.

Connection Properties

See, the **rdma_connect** routine.

Parameters

id	Specifies the connection identifier associated with the request.
conn_param	Specifies the information required to establish the connection.

Return Values

The **rdma_accept** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	The error occurs: • If the id, conn_param, id->qp, or id->qp->context parameter is NULL. • If conn_param->responder_resources is bigger than the local RDMA device attribute, **max_qp_rd_atom**, and remote RDMA device attribute, **max_qp_init_rd_atom**. • If conn_param->initiator_depth is bigger than the local RDMA device attribute, **max_qp_init_rd_atom**, and remote RDMA device attribute, **max_qp_rd_atom**.
-EPROTONOSUPPORT	The id->ps is not **RDMA_PS_TCP**.
-ENODATA	The write operation on id->channel->fd failed.

rdma_reject:

Rejects a connection request.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_reject(struct rdma_cm_id *id, const void *private_data, uint8_t private_data_len);
```

Description

The **rdma_reject** function is called from the listening side to reject a connection lookup request.

Note: After receiving a connection request event, you might call **rdma_reject** to reject the request. If the underlying RDMA transport supports private data in the reject message, the specified data is passed to the remote side.

Parameters

id	Specifies the connection identifier associated with the request.
private_data	Specifies the optional private data to send with the reject message.
private_data_len	Specifies the size of *private_data* to send, in bytes.

Return Values

The **rdma_reject** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	The error occurs if the id is NULL.
-ENODATA	The write operation on id->channel->fd failed.

rdma_disconnect:

Disconnects a connection.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_disconnect(struct rdma_cm_id *id);
```

Description

The **rdma_disconnect** function disconnects a connection and transitions any associated QP to the error state that will flush posted work requests to the completion queue. This routing might be called by both the client and server side of a connection. After successfully disconnection, an RDMA_CM_EVENT_DISCONNECTED event is generated on both sides of the connection.

Parameters

id	Specifies the connection identifier associated with the request.

Return Values

The **rdma_disconnect** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	The error occurs if the id or id->qp is NULL or bad transport type.
-ENODATA	The write operation on id->channel->fd failed.

rdma_get_src_port:

Returns the local port number of a bound rdma_cm_id.

Syntax

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_src_port(struct rdma_cm_id *id)
```

Description

The **rdma_get_src_port** function returns the local port number for an rdma_cm_id that has been bound to a local address.

Parameters

id	Specifies the connection identifier associated with the request.

Return Values

The **rdma_get_src_port** function returns the following values:

local port number	On success.
0	The error occurs if the id is NULL.

rdma_get_dst_port:

Returns the remote port number of a bound rdma_cm_id.

Syntax

```
#include <rdma/rdma_cma.h>
uint16_t rdma_get_dst_port(struct rdma_cm_id *id)
```

Description

The **rdma_get_dst_port** function returns the remote port number for an rdma_cm_id that has been bound to a remote address.

Parameters

id	Specifies the connection identifier associated with the request.

Return Values

The **rdma_get_dst_port** function returns the following values:

local port number	On success.
0	The error occurs if the id is NULL.

rdma_get_local_addr:

Returns the local IP address of a bound rdma_cm_id.

Syntax

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_local_addr(struct rdma_cm_id *id)
```

Description

The **rdma_get_local_addr** function returns the local IP address for an rdma_cm_id that is bound to a local device.

Parameters

id	Specifies the RDMA identifier.

Return Values

The **rdma_get_local_addr** function returns the following values:

local port number	On success.
0	The error occurs if the id is NULL.

rdma_get_peer_addr:

Returns the remote IP address of a bound rdma_cm_id.

Syntax

```
#include <rdma/rdma_cma.h>
struct sockaddr *rdma_get_peer_addr(struct rdma_cm_id *id)
```

Description

The **rdma_get_peer_addr** function returns the remote IP address associated with an rdma_cm_id.

Parameters

id	Specifies the RDMA identifier.

Return Values

The **rdma_get_peer_addr** function returns the following values:

local port number	On success.
0	The error occurs if the id is NULL.

Event Handling Operations:

rdma_get_cm_event:

Retrieves the next pending communication event.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_get_cm_event(struct rdma_event_channel *channel, struct rdma_cm_event **event);
```

Description

The **rdma_get_cm_event** function retrieves a communication event. If no events are pending, by default, the call blocks until an event is received.

Notes:

- You can modify the file descriptor associated with the given channel and change the default synchronous behavior of the **rdma_get_cm_event** routine.
- All events that are reported must be acknowledged by calling **rdma_ack_cm_event**.
- Destruction of an rdma_cm_id is blocked until related events are acknowledged.

Parameters

channel	Specifies the event channel to check for events.
event	Specifies the allocated information about the next communication event.

Return Values

0	On success.
-1	Error, see **errno**.
-EINVAL	The error occurs if channel or event is NULL.
-ENODATA	The write operation on channel->fd failed.
-ENOMEM	There is not enough space to allocate by malloc.
-ENODEV	Unable to get RDMA device

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	Specifies the rdma_cm identifier associated with the event. If the event type is RDMA_CM_EVENT_CONNECT_REQUEST, then the *id* references a new id for that communication.
listen_id	Specifies the corresponding listening request identifier for the RDMA_CM_EVENT_CONNECT_REQUEST event types.
event	Specifies the type of communication event that occurred. See Event Types.
status	Returns any asynchronous error information associated with an event. The status is zero unless the corresponding operation fails.
param	Provides additional details based on the type of event. You must select the *conn* subfield based on the **rdma_port_space** of the rdma_cm_id associated with the event. See Conn Event Data.

Conn Event Data

Event parameters are related to the 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.

private_data	References any user-specified data associated with the event. The data referenced by this field matches the value specified by the remote side when calling **rdma_connect** or **rdma_accept**. The *private_data* 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	Specifies the size of the private data buffer. You must note that the size of the private data buffer might be larger than the amount of private data sent by the remote side. Any additional space in the buffer is zeroed out.
responder_resources	Specifies the number of responder resources requested of the recipient. The *responder_resources* field matches the initiator depth specified by the remote node when calling **rdma_connect** and **rdma_accept**.
initiator_dept	Specifies the maximum number of outstanding RDMA read operations that the recipient might have. The *initiator_dept* field matches the responder resources specified by the remote node when calling **rdma_connect** and **rdma_accept**.
flow_control	Indicates if the hardware level flow control is provided by the sender (specific to the InfiniBand architecture).
retry_count	Indicates the number of times that the recipient must retry a send operation that is specific to RDMA_CM_EVENT_CONNECT_REQUEST events (specific to the InfiniBand architecture).
rnr_retry_count	Indicates the number of times that the recipient must retry receiver not ready (RNR) NACK errors (specific to the InfiniBand architecture).
srq	Specifies if the sender is using a shared-receive queue. Currently the field is not supported.
qp_num	Indicates the remote QP number for the connection.

Event Types

The following types of communication events might be reported.

RDMA_CM_EVENT_ADDR_RESOLVED	Indicates the address resolution (rdma_resolve_addr) completed successfully.
RDMA_CM_EVENT_ADDR_ERROR	Indicates that the address resolution (rdma_resolve_addr) failed.
RDMA_CM_EVENT_ROUTE_RESOLVED	Indicates that the route resolution (rdma_resolve_route) completed successfully.
RDMA_CM_EVENT_ROUTE_ERROR	Indicates that the route resolution (rdma_resolve_route) failed.
RDMA_CM_EVENT_CONNECT_REQUEST	Indicates that there is a new connection request on the passive side.
RDMA_CM_EVENT_CONNECT_RESPONSE	Indicates that the there is a successful response to a connection request on the active side. It is only generated on rdma_cm_ids that do not have a QP associated with them.
RDMA_CM_EVENT_CONNECT_ERROR	Indicates that an error has occurred trying to establish a connection. Might be generated on the active or passive side of a connection.
RDMA_CM_EVENT_UNREACHABLE	Indicates that the remote server is not reachable or unable to respond to a connection request on the active side.

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 is established with the remote end point.
RDMA_CM_EVENT_DISCONNECTED	Indicates that the connection is disconnected.
RDMA_CM_EVENT_DEVICE_REMOVAL	Indicates that the local RDMA device associated with the rdma_cm_id is removed. Upon receiving this event, you must destroy the related rdma_cm_id.
RDMA_CM_EVENT_TIMEWAIT_EXIT	Indicates that the QP associated with a connection has exited its timewait state and is now ready to be reused. After a QP is disconnected, it is maintained in a timewait state to allow any in flight packets to exit the network. After the timewait state is complete, the **rdma_cm** reports this event.

rdma_ack_cm_event:

Frees a communication event.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_ack_cm_event(struct rdma_cm_event *event);
```

Description

All events that are allocated by **rdma_get_cm_event** must be released. There must be a one-to-one correspondence between successful gets and acks. The **rdma_ack_cm_event** call frees the event structure and any memory that it references.

Parameters

event	Specifies the event to be released.

Return Values

The **rdma_ack_cm_event** function returns the following values:

0	On success.
-EINVAL	If *event* is NULL.

rdma_event_str:

Returns a string representation of an RDMA CM event.

Syntax

```
#include <rdma/rdma_cma.h>
const char *rdma_event_str(enum rdma_cm_event_type event);
```

Description

The **rdma_event_str** routine returns a string representation of an asynchronous event.

Parameters

event	Specifies an asynchronous event.

Return Values

The **rdma_event_str** function returns the following values:

A string representation	On known events.
UNKNOWN EVENT	On unknown events.

Queue Pair Management:

rdma_create_qp:

Allocates a QP.

Syntax

```
#include <rdma/rdma_cma.h>
int rdma_create_qp(struct rdma_cm_id *id, struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr);
```

Description

The **rdma_create_qp** function allocates a QP associated with a 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 the **rdma_create_qp** 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 is ready to handle posting of receives. If the QP is unconnected, it will be ready to post sends.

Parameters

id	Specifies the communication identifier to destroy.
pd	Specifies the protection domain for the QP.
qp_init_attr	Specifies the initial QP attributes.

Return Values

The **rdma_create_qp** function returns the following values:

0	On success.
-1	Error, see **errno**.
-EINVAL	If the *id*, *pd*, or *qp_init_attr* parameter is NULL or bad parameter in **ibv_qp_init_attr** such as, **cap.max_inline_data** limited to 64 for Chelsio Boards.
-ENOMEM	There is not enough space to allocate by malloc.

rdma_destroy_qp:

Releases a QP.

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_destroy_qp(struct rdma_cm_id *id);
```

Description

The **rdma_destroy_qp** function destroys a QP allocated on the rdma_cm_id.

Note: You must destroy any QP associated with an rdma_cm_id before destroying the ID.

Parameters

id	Specifies the RDMA identifier.

Return Value

There is no return value.

Device Management:

rdma_get_devices:

Gets a list of RDMA devices that are available.

Syntax

```
#include <rdma/rdma_cma.h>
struct ibv_context **rdma_get_devices(int *num_devices);
```

Description

The **rdma_get_devices** function returns a NULL-terminated array of open RDMA devices. You can use this routine to allocate resources on specific RDMA devices that will be shared across multiple rdma_cm_ids.

Note: The returned array must be released by calling the **rdma_free_devices** routine. Devices remain opened while the **librdmacm** library is loaded.

Parameters

num_devices	Specifies the number of devices returned if the value is not NULL.

Return Values

A NULL-terminated array	On success
NULL	On failure

rdma_free_devices:

Frees the list of devices returned by the **rdma_get_devices** routine.

Syntax

```
#include <rdma/rdma_cma.h>
void rdma_free_devices(struct ibv_context **list);
```

Description

The **rdma_free_devices** function frees the device array returned by the **rdma_get_devices** routine.

Parameters

list	Specifies the list of devices returned from the **rdma_get_devices** routine.

Return Value

There is no return value.

Verbs not supported by librdmacm

You can find the list of verbs that are not supported by the **librdmacm** library.

The following is the list of not supported verbs.

rdma_notify	Notifies the **librdmacm** library of an asynchronous event.
rdma_join_multicast	Joins a multicast group.
rdma_leave_multicast	Leaves a multicast group.
rdma_set_option	Sets options for an rdma_cm_id.

Libibverbs

You can find information about the libibverbs library in the `/usr/include/rdma/verbs.h` file delivered with the libibverbs library sources.

See chapters 10 and 11 of the InfiniBand specifications. Man pages are also created to describe the various interfaces and test programs available. For a full list, you can refer to the **verbs** man page.

Returned Error Rules

Most commands return 0 on success. The commands return NULL, -1, or the value of the **errno** variable that indicates the reason of failure. The commands return ENOSYS when the verb is not supported.

Supported Verbs

You can find a list of supported verbs.

Device Management:

ibv_get_device_list, ibv_free_device_list:

Gets and releases the list of available RDMA devices.

Syntax

```
#include <rdma/verbs.h>
struct ibv_device **ibv_get_device_list(int *num_devices);
void ibv_free_device_list(struct ibv_device **list);
```

Description

ibv_get_device_list() returns a NULL-terminated array of RDMA devices currently available. The argument *num_devices* is optional and if it is NULL, it is set to the number of devices returned in the array.

ibv_free_device_list() frees the array of devices list returned by **ibv_get_device_list()**.

Note: Client code must open all the devices it intends to use with **ibv_open_device()** before calling **ibv_free_device_list()**. Once the **ibv_free_device_list()** function frees the array, The system will be able to use only the open devices and the pointers to unopened devices will no longer be valid.

Output Parameters

num_devices	(Optional) If not null, the number of devices returned in the array will be stored here.

Return Value

ibv_get_device_list() returns the array of available RDMA devices, or NULL if the request fails.

ibv_free_device_list() returns no value.

ibv_get_device_name:

Gets the RDMA device's name.

Syntax

```
#include <rdma/verbs.h>
const char *ibv_get_device_name(struct ibv_device *device);
```

Description

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

Parameters

device	struct **ibv_device** for desired device.

Return Value

ibv_get_device_list() returns a pointer to the device name char string on success, and NULL if the request fails.

ibv_get_device_guid:

Returns string describing the **event_type**, **node_type**, and **port_state** enum values.

Syntax

```
#include <rdma/verbs.h>
uint64_t ibv_get_device_guid(struct ibv_device *device);
```

Description

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

Parameters

device	struct **ibv_device** for the desired device.

Return Value

The **ibv_get_device_guid** function returns **uint64_t** on success, and **0** on failure.

If device=NULL, operation open, or write failed on the OFED admin device */dev/rdma/ofed_adm*.

ibv_open_device, ibv_close_device:

Opens, and closes an RDMA device context.

Syntax

```
#include <rdma/verbs.h>
struct ibv_context *ibv_open_device(struct ibv_device *device);
int ibv_close_device(struct ibv_context *context);
```

Description

The **ibv_open_device()** routine opens the device *device*, and creates a context for further use.

The **ibv_close_device()** routine closes the device context *context*.

Note: The **ibv_close_device()** routine does not release all the resources allocated using the parameter *context*. To avoid resource leaks, you must release all associated resources before closing a context.

Parameter

devices	struct **ibv_device** for the required device.

Return Value

The **ibv_open_device**, and **ibv_close_device** functions return a verbs context that can be used for future operations on the device, on success, and returns NULL if the device=NULL or the open operation fails.

ibv_query_device:

Queries the attributes of an RDMA device.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_device(struct ibv_context *context, struct ibv_device_attr *device_attr)
```

Description

The **ibv_query_device()** routine returns the attributes of the device with context *context*. The parameter device_attr is a pointer to an **ibv_device_attr** struct, as defined in <rdma/verbs.h>.

Note: The maximum values returned by the **ibv_query_device()** function are the upper limits of supported resources by the device. It might not be possible to use these maximum values, since the actual number of any resource that can be created is limited by the machine configuration, the amount of host memory, user permissions, and the amount of resources already in use.

Input Parameter

context	struct ibv_context from **ibv_open_device**.

Output Parameter

device_attr	struct ibv_device_attr containing device attributes.

Return Values

0	On success.
errno	On failure.
EINVAL	If context parameter or *device_attr* is NULL

ibv_query_port:

Queries the attributes of an RDMA port.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_port(struct ibv_context *context, uint8_t port_num, struct ibv_port_attr *port_attr)
```

Description

The **ibv_query_port()** routine returns the attributes of port *port_num* for device context *context* through the pointer *port_attr*. The parameter *port_attr* is an ibv_port_attr struct, as defined in <rdma/verbs.h>.

struct ibv_port_attr:

```
  struct ibv_port_attr {
    enum ibv_port_state      state;  /* Logical port state */
    enum ibv_mtu         max_mtu; /* Max MTU supported by port */
    enum ibv_mtu         active_mtu; /* Actual MTU */
    int             gid_tbl_len; /* Length of source GID table */
    uint32_t          port_cap_flags; /* Port capabilities */
    uint32_t          max_msg_sz; /* Maximum message size */
    uint32_t          bad_pkey_cntr; /* Bad P_Key counter */
    uint32_t          qkey_viol_cntr; /* Q_Key violation counter */
    uint16_t          pkey_tbl_len; /* Length of partition table */
    uint16_t          lid;  /* Base port LID */
    uint16_t          sm_lid;  /* SM LID */
    uint8_t          lmc;  /* LMC of LID */
    uint8_t          max_vl_num; /* Maximum number of VLs */
    uint8_t          sm_sl;  /* SM service level */
    uint8_t          subnet_timeout; /* Subnet propagation delay */
    uint8_t          init_type_reply; /* Type of initialization performed by SM */
    uint8_t          active_width; /* Currently active link width */
    uint8_t          active_speed; /* Currently active link speed */
    uint8_t          phys_state; /* Physical port state */
};
```

Input Parameters

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

Output Parameter

port_attr	struct ibv_port_attr containing port attributes.

Return Values

0	On success.
errno	On failure.
EINVAL	If context parameter or *port_attr* is NULL

ibv_query_pkey:

Queries the P_Key table of an RDMA port.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_pkey(struct ibv_context *context, uint8_t port_num, int index, uint16_t *pkey)
```

Description

The **ibv_query_pkey()** routine returns the P_Key value in the entry *index* of port *port_num* for device context *context* through the pointer *pkey*.

Input Parameters

context	Valid context pointer returned by **ibv_open_device()**.
port_num	Valid port number for the device returned by **ibv_query_device()**.
index	Valid index for port_num from attributes returned by **ibv_query_port()**.

Output Parameter

pkey	Valid pointer to store protection key.

Return Values

0	On success.
-1	If the request fails because, the *context* or *pkey* parameter is NULL or the open or write operation failed on the OFED admin device `/dev/rdma/ofed_adm`.

ibv_query_gid:

Gets GID, which is the NIC's MAC address.

Syntax

```
#include <rdma/verbs.h>
int ibv_query_gid(struct ibv_context *context, uint8_t port_num, int index, union ibv_gid *gid)
```

Description

The **ibv_query_gid()** routine returns the NIC's MAC address in subnet_prefix and 0 in the interface_id.

Input Parameters

context	Specifies the context pointer returned by **ibv_open_device()**.
port_num	Specifies port number for the device returned by **ibv_query_device()**.
index	Specifies index for port_num deduced from attributes returned by **ibv_query_port()**.

Output Parameter

gid	Specifies the pointer to store GID.

Return Values

0	On success.
-1	If the request fails because, the *context* or *gid* parameter is NULL or the open or write operation failed on the OFED admin device `/dev/rdma/ofed_adm`.

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

Queue Pair Management:

ibv_create_qp, ibv_destroy_qp:

Creates or destroys a queue pair (QP).

Syntax

```
#include <rdma/verbs.h>
struct ibv_qp *ibv_create_qp(struct ibv_pd *pd, struct ibv_qp_init_attr *qp_init_attr);
int ibv_destroy_qp(struct ibv_qp *qp)
```

Description

ibv_create_qp() creates a queue pair (QP) associated with the protection domain *pd*. The argument *qp_init_attr* is an ibv_qp_init_attr struct, as defined in `<rdma/verbs.h>`.

```
struct ibv_qp_init_attr {

   void                          *qp_context;         /*Associated context of the QP*/
   struct ibv_cq                 *send_cq;            /*CQ to be associated with the Send Queue (SQ)*/
   struct ibv_cq                 *recv_cq;            /*CQ to be associated with the Receive Queue (RQ)*/
   struct ibv_srq                *srq;                /*Not Supported*/
   struct ibv_qp_cap             cap;                 /*QP capabilities*/
   enum ibv_qp_type              qp_type;             /*QP Transport Service Type: IBV_QPT_RC,*/
   int                           sq_sig_all;          /*If set, each Work Request (WR) submitted to the SQ*/
                                                      /*generates a completion entry */
   struct ibv_xrc_domain         xrc_domain;          /*Not supported*/

struct ibv_qp_cap {

   uint32_t                      max_send_wr;         /*Requested max number of outstanding*/
```

```
                                                        /*WRs in the SQ*/
    uint32_t                    max_recv_wr;            /*Requested max number of outstanding*/
                                                        /*WRs in the RQ*/
    uint32_t                    max_send_sge;           /*Requested max number of scatter/gather*/
                                                        /*(s/g) elements in*/
                                                        /*a WR in the SQ*/
    uint32_t                    max_recv_sge;           /*Requested max number of s/g elements*/
                                                        /*in a WR in the SQ*/
    uint32_t                    max_inline_data;        /*Requested max number of data (bytes)*/
                                                        /*that can be posted*/
                                                        /*inline to the SQ, otherwise 0*/
```

The function **ibv_create_qp()** updates the *qp_init_attr*->`cap struct` with the actual QP values of the QP that was created; the values will be greater than or equal to the values requested. **ibv_destroy_qp()** destroys the QP *qp*.

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

ibv_create_qp() returns a pointer to the created QP on success, or NULL if the request fails.

ibv_destroy_qp() returns 0 on success, or the value of **errno** on failure (which indicates the failure reason).

ibv_modify_qp:

Modifies the attributes of a queue pair (QP).

Syntax

```
#include <rdma/verbs.h>
int ibv_modify_qp(struct ibv_qp *qp, struct ibv_qp_attr *attr, enum ibv_qp_attr_mask attr_mask)
```

Queue pairs (QP) must be taken through an incremental sequence of states prior to use them 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 might now send packets.

The state transitions are accomplished by using the **ibv_modify_qp** command.

Description

The **ibv_modify_qp()** function modifies the attributes of a QP *qp* with the attributes in *attr* according to the mask *attr_mask*. The parameter *attr* is an **ibv_qp_attr** struct, as defined in <rdma/verbs.h>.

The parameter *attr_mask* specifies the QP attributes to be modified. The argument is either 0 or the bitwise OR of one or more of the following flags:

IBV_QP_STATE
Modify qp_state

IBV_QP_CUR_STATE
Set cur_qp_state

IBV_QP_EN_SQD_ASYNC_NOTIFY
Set en_sqd_async_notify

IBV_QP_ACCESS_FLAGS
Set qp_access_flags

IBV_QP_PKEY_INDEX
Set pkey_index

IBV_QP_PORT
Set port_num

IBV_QP_QKEY
Set qkey

IBV_QP_AV
Set ah_attr

IBV_QP_PATH_MTU
Set path_mtu

IBV_QP_TIMEOUT
Set timeout

IBV_QP_RETRY_CNT
Set retry_cnt

IBV_QP_RNR_RETRY
Set rnr_retry

IBV_QP_RQ_PSN
Set rq_psn

IBV_QP_MAX_QP_RD_ATOMIC
Set max_rd_atomic

IBV_QP_ALT_PATH
Set the alternative path via: alt_ah_attr, alt_pkey_index, alt_port_num, alt_timeout

IBV_QP_MIN_RNR_TIMER
Set min_rnr_timer

IBV_QP_SQ_PSN
Set sq_psn

IBV_QP_MAX_DEST_RD_ATOMIC
Set max_dest_rd_atomic

IBV_QP_PATH_MIG_STATE
Set path_mig_state

IBV_QP_CAP
Set cap

IBV_QP_DEST_QPN
Set dest_qp_num

Notes:

- If any of the modify attributes or the modify mask is invalid, none of the attributes are modified (including the QP state).
- Not all devices support resizing QPs. To check if a device supports it, check if the IBV_DEVICE_RESIZE_MAX_WR bit is set in the device capabilities flags.
- Not all devices support alternate paths. To check if a device supports it, check if the IBV_DEVICE_AUTO_PATH_MIG bit is set in the device capabilities flags.
- The following tables indicate for QP Transport Service Type IBV_QPT_RC, the minimum list of attributes that must be changed upon transitioning QP state from Reset --> Init --> RTR --> RTS.

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
RTS	IBV_QP_STATE, IBV_QP_SQ_PSN, IBV_QP_MAX_QP_RD_ATOMIC, IBV_QP_RETRY_CNT, IBV_QP_RNR_RETRY, IBV_QP_TIMEOUT

Input Parameters

qp	Specifies the struct ibv_qp from **ibv_create_qp**.
attr	Specifies the QP attributes.
attr_mask	Specifies the bit mask that defines which attributes within *attr* is set for this call.

Return Values

0	On success.
EINVAL	The error occurs when qp, qp->context, or attr is NULL.

ibv_post_recv:

Posts a list of work requests (WRs) to a receive queue.

Syntax

```
#include <rdma/verbs.h>
int ibv_post_recv(struct ibv_qp *qp, struct ibv_recv_wr *wr, struct ibv_recv_wr **bad_wr)
```

Description

The **ibv_post_recv()** routine posts the linked list of work requests (WRs) starting with *wr* to the receive queue of the queue pair *qp*. The routine stops processing WRs from the list at the first failure that can be detected immediately while requests are being posted, and returns the failing WR through *bad_wr*.

The argument *wr* is an ibv_recv_wr struct, as defined in <rdma/verbs.h>.

```
struct ibv_recv_wr {
        uint64_t       wr_id;  /* User defined WR ID */
        struct ibv_recv_wr   *next;  /* Pointer to next WR in list, NULL if last WR */
        struct ibv_sge    *sg_list; /* Pointer to the s/g array */
        int          num_sge; /* Size of the s/g array */
};

struct ibv_sge {
        uint64_t        addr;  /* Start address of the local memory buffer */
        uint32_t        length; /* Length of the buffer */
        uint32_t        lkey;  /* Key of the local Memory Region */
};
```

Note: The buffers used by a WR can only be safely reused after the request is complete and a work completion is retrieved from the corresponding completion queue (CQ).

Input Parameters

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

Output Parameter

bad_wr	Specifies the pointer to first rejected WR.

Return Values

0	On success.
errno	On failure.
EINVAL	If qp, qp->context, wr, or wr->sg_list is NULL.

ibv_post_send:

Posts a list of work requests (WRs) to a send queue.

Syntax

```
#include <rdma/verbs.h>
int ibv_post_send(struct ibv_qp *qp, struct ibv_send_wr *wr, struct ibv_send_wr **bad_wr)
```

Description

The **ibv_post_recv()** routine posts the linked list of work requests (WRs) starting with *wr* to the receive queue of the queue pair *qp*. The routine stops processing WRs from the list at the first failure that can be detected immediately while requests are being posted, and returns the failing WR through *bad_wr*.

The argument *wr* is an ibv_send_wr struct, as defined in <rdma/verbs.h>.

RC Transport Service Type supports following opcodes:

OPCODE	**IBV_QPT_RC**
IBV_WR_SEND	Supported
IBV_WR_SEND_WITH_IMM	Supported
IBV_WR_RDMA_WRITE	Supported
IBV_WR_RDMA_WRITE_WITH_IMM	Supported
IBV_WR_RDMA_READ	Supported
IBV_WR_ATOMIC_CMP_AND_SWP	Not supported
IBV_WR_ATOMIC_FETCH_AND_ADD	Not supported

The attribute send_flags describes the properties of the WR . It is either 0 or the bitwise OR of one or more of the following flags:

IBV_SEND_FENCE
: Sets the fence indicator. The IBV_SEND_FENCE flag is valid only for QPs with Transport Service Type IBV_QPT_RC.

IBV_SEND_SIGNALED
: Sets the completion notification indicator. The IBV_SEND_SIGNALED flag is relevant only if QP was created with sq_sig_all=0.

IBV_SEND_SOLICITED
: Sets the solicited event indicator. The IBV_SEND_SOLICITED flag is valid only for Send and RDMA Write with immediate.

IBV_SEND_INLINE
: Sends data in given gather list as inline data in a send WQE. The IBV_SEND_INLINE flag is valid only for Send and RDMA Write. The L_Key is not checked.

Note: The buffers used by a WR can only be safely reused after the request is complete and a work completion is retrieved from the corresponding completion queue (CQ).

Input Parameters

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

Output Parameter

bad_wr	Specifies the pointer to first rejected WR.

Return Values

0	On success.
EINVAL	Error, if qp, qp->context, wr, or wr->sg_list is NULL.
ENOTSUP	Error, if wr->opcode is not one of :IBV_WR_SEND, IBV_WR_RDMA_WRITE, or IBV_WR_RDMA_READ.

Completion Queue Management:

ibv_create_cq, ibv_destroy_cq:

Creates or destroys a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
struct ibv_cq *ibv_create_cq(struct ibv_context *context, int cqe, void *cq_context, struct ibv_comp_channel *channel,
int comp_vector)
int ibv_destroy_cq(struct ibv_cq *cq)
```

Description

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

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

The parameter *cq_context* is a user defined value. If specified during CQ creation, this value is 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 not required. The user simply polls the CQ at regular intervals. However, if you wish to use a pend paradigm, a CC is required. The CC is a mechanism that allows the user to be notified that a new CQE is on the CQ.

The CQ will use the completion vector **comp_vector** for signaling completion events; it must be at least zero and less than `context->num_comp_vectors`.

ibv_destroy_cq() destroys the CQ *cq*.

Notes:
- **ibv_create_cq()** might create a CQ with size greater than or equal to the requested size. Check the *cqe* attribute in the returned CQ for the actual size.
- **ibv_destroy_cq()** fails if any queue pair is still associated with this CQ.

Parameters

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

Return Value

ibv_create_cq() returns a pointer to the CQ, or NULL if the request fails.

ibv_destroy_cq() returns 0 on success, or the value of **errno** on failure (which indicates the failure reason).

ibv_req_notify_cq:

Requests the completion notification on a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
int ibv_req_notify_cq(struct ibv_cq *cq, int solicited_only);
```

Description

The **ibv_req_notify_cq()** routine requests a completion notification on the completion queue (CQ) *cq*.

Upon addition of a new CQ entry (CQE) to *cq*, a completion event is added to the completion channel associated with the CQ. If the argument *solicited_only* is zero, a completion event is generated for any new CQE. If *solicited_only* is non-zero, an event is generated for a new CQE that is considered solicited. A CQE is solicited if it is a receive completion for a message with the Solicited Event header bit set, or if the status is not successful. All other successful receive completions, or any successful send completion is unsolicited.

Note: The request for notification is only once. Only one completion event is generated for each call to **ibv_req_notify_cq()**.

Parameters

cq	Specifies the struct ibv_cq from **ibv_create_cq**.
solicited_only	Notifies only if WR is flagged as solicited.

Return Values

0	On success.
EINVAL	Error, if cq, or cq->context is NULL.

ibv_poll_cq:

Polls a completion queue (CQ).

Syntax

```
#include <rdma/verbs.h>
int ibv_poll_cq(struct ibv_cq *cq, int num_entries, struct ibv_wc *wc)
```

Description

The **ibv_poll_cq()** routine polls the CQ *cq* for work completions and returns the first *num_entries* (or all available completions if the CQ contains fewer than this number) in the array *wc*. The argument *wc* is a pointer to an array of ibv_wc structs, as defined in <rdma/verbs.h>.

```
struct ibv_wc {
        uint64_t         wr_id;    /* ID of the completed Work Request (WR) */
        enum ibv_wc_status     status;   /* Status of the operation */
        enum ibv_wc_opcode     opcode;   /* Operation type specified in the completed WR */
        uint32_t         vendor_err;  /* Vendor error syndrome */
        uint32_t         byte_len;  /* Number of bytes transferred */
        uint32_t         imm_data;  /* Immediate data (in network byte order) */
        uint32_t         qp_num;   /* Local QP number of completed WR */
        uint32_t         src_qp;   /* Source QP number (remote QP number)  */
                     /* of completed WR */
        enum ibv_wc_flags     wc_flags;  /* Flags of the completed WR */
        uint16_t         pkey_index;  /* P_Key index (valid only for GSI QPs) */
        uint16_t         slid;    /* Source LID */
        uint8_t          sl;     /* Service Level */
        uint8_t          dlid_path_bits; /* DLID path bits (not applicable for multicast */
                     /* messages) */
};
 enum ibv_wc_flags         wc_flags;       /* Flags of the completed WR */
```

The attribute **wc_flags** describes the properties of the work completion. It is either 0 or the bitwise OR of one or more of the following flags:

IBV_WC_GRH
GRH is present.

IBV_WC_WITH_IMM
Immediate data value is valid.

Not all **wc** attributes are always valid. If the completion status is other than IBV_WC_SUCCESS, only the **wr_id**, **status**, **qp_num**, and **vendor_err** attributes are valid.

Note: Each polled completion is removed from the CQ and cannot be returned to it. You must consume work completions at a rate that prevents CQ overrun from occurrence. In case of a CQ overrun, the async event **IBV_EVENT_CQ_ERR** is triggered, and the CQ cannot be used.

Input Parameters

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

Output Parameters

wc	Specifies the CQE array.

Return Values

On success, the **ibv_poll_cq()** function returns a non negative value equal to the number of completions found. On failure, a negative value is returned.

-EINVAL	Error, if cq, or cq->context is NULL.

ibv_get_cq_event, ibv_ack_cq_events:

Gets and acknowledges completion queue (CQ) events.

Syntax

```
#include <rdma/verbs.h>
int ibv_get_cq_event(struct ibv_comp_channel *channel, struct ibv_cq **cq, void **cq_context);
void ibv_ack_cq_events(struct ibv_cq *cq, unsigned int nevents);
```

Description

ibv_get_cq_event() waits for the next completion event in the completion event channel *channel*. The argument *cq* is used to return the CQ that caused the event and *cq_context* is used to return the context of the CQ.

ibv_ack_cq_events() acknowledges *nevents* events on the CQ *cq*.

Notes:
- All completion events that **ibv_get_cq_event()** returns must be acknowledged using **ibv_ack_cq_events()**.
- To avoid races, when you destroy a CQ, the CQ waits for the completion of the events. This guarantees a one-to-one correspondence between acknowledgements and successful gets.
- When you call the **ibv_ack_cq_events()** function, it might be relatively expensive in the datapath, since it must take a mutex. Therefore it might be better to amortize this cost by keeping a count of the number of events needing acknowledgement and acknowledging several completion events in one call to **ibv_ack_cq_events()**.

Input Parameters

channel	struct **ibv_comp_channel** from **ibv_create_comp_channel**.

Output Parameters

cq	Pointer to the completion queue (CQ) associated with event.
cq_context	User supplied context set in **ibv_create_cq**.

Return Value

The **ibv_get_cq_event**, and **ibv_ack_cq_events** functions return 0 on success, and -1 if the request fails.

Examples

1. The following code example demonstrates one possible way to work with completion events. It performs the following steps:
   a. Preparation
      1) Creates a CQ.
      2) Requests for notification upon a new (first) completion event.
   b. Completion handling routine
      1) Waits for the completion event and ack it.
      2) Requests for notification upon the next completion event.
      3) Empties the CQ.

      Note: An extra event might be triggered without having a corresponding completion entry in the CQ. This occurs if a completion entry is added to the CQ between requesting for notification and emptying the CQ, and then the CQ is emptied.

```
cq = ibv_create_cq(ctx, 1, ev_ctx, channel, 0);
if (!cq) {
  fprintf(stderr, "Failed to create CQ\n");
  return 1;
}

/* Request notification before any completion can be created */
if (ibv_req_notify_cq(cq, 0)) {
  fprintf(stderr, "Couldn't request CQ notification\n");
  return 1;
}

.
.
.
/* Wait for the completion event */
if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
  fprintf(stderr, "Failed to get cq_event\n");
  return 1;
}

/* Ack the event */
ibv_ack_cq_events(ev_cq, 1);

/* Request notification upon the next completion event */
if (ibv_req_notify_cq(cq, 0)) {
  fprintf(stderr, "Couldn't request CQ notification\n");
  return 1;
}

/* Empty the CQ: poll all of the completions from the CQ (if any exist) */
do {
  ne = ibv_poll_cq(cq, 1, &wc);
```

```
    if (ne < 0) {
      fprintf(stderr, "Failed to poll completions from the CQ\n");
      return 1;
    }
    if (wc.status != IBV_WC_SUCCESS) {
      fprintf(stderr, "Completion with status 0x%x was found\n", wc.status);
      return 1;
    }
  } while (ne);
```

2. The following code example demonstrates one possible way to work with completion events in nonblocking mode. It performs the following steps:
   a. Sets the completion event channel in the non-blocked mode.
   b. Polls the channel until there it has a completion event.
   c. Gets the completion event and acknowledges it.

```
/* change the blocking mode of the completion channel */
flags = fcntl(channel->fd, F_GETFL);
rc = fcntl(channel->fd, F_SETFL, flags | O_NONBLOCK);
if (rc < 0) {
  fprintf(stderr, "Failed to change file descriptor of completion event channel\n");
  return 1;
}
/*
* poll the channel until it has an event and sleep ms_timeout
* milliseconds between any iteration
*/
my_pollfd.fd = channel->fd;
my_pollfd.events = POLLIN;
my_pollfd.revents = 0;

do {

rc = poll(&my_polfd;, 1, ms_timeout);
  } while (rc == 0);
  if (rc &lt; 0){  fprintf(stderr, "poll failed\n");
  return 1;
  }
 ev_cq = cq;
 /* Wait for the completion event */
 if (ibv_get_cq_event(channel, &ev_cq, &ev_ctx)) {
   fprintf(stderr, "Failed to get cq_event\n");
   return 1;
 }
  /* Ack the event */
 ibv_ack_cq_events(ev_cq, 1);
```

Protection Domain Management:

ibv_alloc_pd, ibv_dealloc_pd:

Allocates or de-allocates a protection domain (PD).

Syntax

```
#include <rdma/verbs.h>
struct ibv_pd *ibv_alloc_pd(struct ibv_context *context)
int ibv_dealloc_pd(struct ibv_pd *pd)
```

Description

ibv_alloc_pd() allocates a PD for the RDMA device context *context*. **ibv_dealloc_pd()** de-allocates the PD *pd*.

Note: **ibv_dealloc_pd()** might fail if any other RDMA resource is still associated with the PD being freed.

Parameters

context	**struct ibv_context** from **ibv_open_device**.

Return Value

ibv_alloc_pd() returns a pointer to the allocated PD, or NULL if the request fails. **ibv_dealloc_pd()** returns 0 on success, or the value of **errno** on failure (which indicates the failure reason).

Memory Region Management:

ibv_reg_mr:

Registers or releases a memory region (MR).

Syntax

```
#include <rdma/verbs.h>
struct ibv_mr *ibv_reg_mr(struct ibv_pd *pd, void *addr,size_t length,enum ibv_access_flags access);
int ibv_dereg_mr(struct ibv_mr *mr);
```

Description

The **ibv_reg_mr()** function registers a memory region (MR) associated with the protection domain *pd*. The MR's starting address is *addr* and its size is *length*. The parameter *access* describes the required memory protection attributes that is either 0 or the bitwise OR of one or more of the following flags:

The attribute **wc_flags** describes the properties of the work completion. It is either 0 or the bitwise OR of one or more of the following flags:

IBV_ACCESS_LOCAL_WRITE
Enables Local Write Access

IBV_ACCESS_REMOTE_WRITE
Enable Remote Write Access

IBV_ACCESS_REMOTE_READ
Enable Remote Read Access

IBV_ACCESS_REMOTE_ATOMIC
Enable Remote Atomic Operation Access (Not supported)

IBV_ACCESS_MW_BIND
Enable Memory Window Binding(Not supported)

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

Note: Local read access is always enabled for the MR.

The **ibv_dereg_mr()** function release the MR *mr*.

Parameters

pd	Specifies the protection domain, struct ibv_pd from ibv_alloc_pd.
addr	Specifies the memory base address.
length	Specifies the length of memory region in bytes.
access	Specifies the access flags.

Return Values

The **ibv_reg_mr()** function returns a pointer to the registered MR on success, and NULL if the request fails. The local key (L_Key) field *lkey* is used as the lkey field of struct **ibv_sge** when posting buffers with ibv_post_* verbs, and the remote key (R_Key) field *rkey* is used by remote processes to perform RDMA operations. The remote process places this rkey as the rkey field of struct ibv_send_wr passed to the **ibv_post_send** function.

The **ibv_dereg_mr()** function returns 0 on success, and the value of errno on failure that indicates the failure reason.

Event Management:

ibv_create_comp_channel, ibv_destroy_comp_channel:

Creates or destroys a completion event channel.

Syntax

```
#include <rdma/verbs.h>
struct ibv_comp_channel *ibv_create_comp_channel(struct ibv_context *context)
int ibv_destroy_comp_channel(struct ibv_comp_channel *channel)
```

Description

ibv_create_comp_channel() creates a completion event channel for the RDMA device context, *context*. 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).

ibv_destroy_comp_channel() destroys the completion event channel, *channel*.

Notes:

- A **completion channel** is an abstraction introduced by **libibverbs** that does not exist in the InfiniBand Architecture verbs specification. A completion channel is essentially file descriptor that is used to deliver completion notifications to a **userspace** process. When a completion event is generated for a completion queue (CQ), the event is delivered via the completion channel attached to that CQ. This might be useful to steer completion events to different threads by using multiple completion channels.
- **ibv_destroy_comp_channel()** fails if any CQs are still associated with the completion event channel being destroyed.

Parameters

context	**struct ibv_context** from **ibv_open_device**.

Return Value

ibv_create_comp_channel() returns a pointer to the created completion event channel, or NULL if the request fails.

ibv_destroy_comp_channel() returns 0 on success, or the value of **errno** on failure (which indicates the failure reason).

ibv_get_async_event, ibv_ack_async_event:

Gets or acknowledges asynchronous events.

Syntax

```
#include <rdma/verbs.h>
int ibv_get_async_event(struct ibv_context *context,  struct ibv_async_event *event);
void ibv_ack_async_event(struct ibv_async_event *event);
```

Description

ibv_get_async_event() waits for the next async event of the RDMA device context, *context* and returns it through the pointer, *event*, which is an **ibv_async_event** struct, as defined in `<rdma/verbs.h>`.

```
struct ibv_async_event {
        union {
                struct ibv_cq *cq;    /* CQ that got the event */
                struct ibv_qp *qp;    /* QP that got the event */
                struct ibv_srq *srq;   /* SRQ that got the event  (Not Supported)*/
                int  port_num;     /* port number that got the event */
        } element;
        enum ibv_event_type     event_type;     /* type of the event */
};
```

The function **ibv_create_qp()** updates the *qp_init_attr*`->cap struct` with the actual QP values of the QP that was created; the values will be greater than or equal to the values requested. **ibv_destroy_qp()** destroys the QP *qp*.

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

QP events	
IBV_EVENT_QP_FATAL	Error occurred on a QP and it transitions 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 is established on a QP.
IBV_EVENT_SQ_DRAINED	Send Queue is drained of outstanding messages in progress.
IBV_EVENT_PATH_MIG	A connection is migrated to an alternate path.
IBV_EVENT_PATH_MIG_ERR	A connection failed to migrate to the alternate path.
CQ events	
IBV_EVENT_CQ_ERR	CQ is in error (CQ overrun).
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 is changed on a port.
IBV_EVENT_PKEY_CHANGE	The **P_Key** table is changed on a port.
CA events	
IBV_EVENT_DEVICE_FATAL	CA is in **FATAL** state.

ibv_ack_async_event() acknowledges the async event, *event*.

Notes:

- All async events that **ibv_get_async_event()** returns must be acknowledged using **ibv_ack_async_event()**. To avoid races, destroying an object (CQor 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.

- The **ibv_get_async_event()** function 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 receives it.

Input Data

`struct ibv_context` **context*	struct **ibv_context** from **ibv_open_device**.
`struct ibv_async_event` **event*	event pointer.

Return Value

ibv_get_async_event() returns 0 on success, and -1 if the request fails.

ibv_ack_async_event() returns no value.

Example

The following code example demonstrates one possible way to work with async events in nonblocking mode. It performs the following steps:

1. Sets the async events queue in non-blocked work mode.
2. Polls the queue until it has an async event.
3. Gets the async event and acknowledges it.

```
/* change the blocking mode of the async event queue */
flags = fcntl(ctx->async_fd, F_GETFL);
rc = fcntl(ctx->async_fd, F_SETFL, flags | O_NONBLOCK);
if (rc &lt; 0) {
       fprintf(stderr, "Failed to change file descriptor of async event queue\n");
       return 1;
}
/*
 * poll the queue until it has an event and sleep ms_timeout
 * milliseconds between any iteration
 */
my_pollfd.fd      = ctx->async_fd;
my_pollfd.events  = POLLIN;
my_pollfd.revents = 0;

do {
        rc = poll(&my_pollfd;,1, ms_timeout);
} while (rc == 0);
if (rc < 0) {
      fprintf(stderr, "poll failed\n");
           return 1;
 }

/* Get the async event */
if (ibv_get_async_event(ctx, &async_event)) {
         fprintf(stderr, "Failed to get async_event\n");
         return 1;
 }
 /* Ack the event */
ibv_ack_async_event(&async_event);
```

ibv_event_type_str(3):

Returns string describing the **event_type**, **node_type**, and **port_state** enum values.

Syntax

```
const char *ibv_event_type_str(enum ibv_event_type event_type);
const char *ibv_node_type_str(enum ibv_node_type node_type);
const char *ibv_port_state_str(enum ibv_port_state port_state);
```

Description

ibv_node_type_str() returns a string describing the node type enum value, *node_type*.

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

ibv_event_type_str() returns a string describing the event type enum value, *event_type*.

Return Value

The **ibv_node_type_str()**, **ibv_port_state_str()**, and **ibv_event_type_str()** functions return a constant string that describes the enum value passed as their argument.

`<<unknown>>` string is passed if the enum value is not known.

Verbs not supported by libibverbs

You can find the list of verbs that are not supported by the **libibverbs** library.

Following are the verbs that are not supported.

ibv_resize_cq	Resizes a completion queue (CQ).
ibv_query_qp	Gets the attributes of a queue pair (QP).
ibv_attach_mcast, **ibv_detach_mcast**	Attaches and detaches a queue pair (QPs) to/from a multicast group.
ibv_fork_init	Initializes libibverbs to support fork().

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 903
11501 Burnet Road
Austin, TX 78758-3400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.
© Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

INFINIBAND, InfiniBand Trade Association, and the INFINIBAND design marks are trademarks and/or service marks of the INFINIBAND Trade Association.

Other company, product, or service names may be trademarks or service marks of others.

Index

© Copyright IBM Corp. 2010

Printed in USA

SC23-6777-00