Solaris Open Fabrics User Verbs

Implementation Details

OFUV Project Team

July 30, 2008

ABSTRACT

The OFUV implementation ports the OFA OFED user-space libraries to Solaris and provides the new kernel agent drivers required to utilize the existing Solaris Infiniband hardware provider drivers.

1 Purpose....4

2 Overview....5

3 SOL_UVERBS – OFUV Kernel Agent Driver....6
3.1 Interfaces....7
3.1.1 User/Kernel ABI....7
3.2 Driver Operational Concepts....12
3.2.1 User Objects....12
3.3 In a similar fashion a write reference to a user object is obtained with the following call. 13
3.3.1 User Context....16
3.3.2 Event Processing....17
3.4 Sequence Diagrams....18
3.4.1 SOL_UVERBS Device Open....19
3.4.2 SOL_UVERBS Resource Management....19
3.4.3 SOL_UVERBS Transport Scenario....20
3.5 Source File Breakdown....21
3.6 Known Issues/Limitations....23

LIBIBVERBS – OFED IB Verbs User Space Library....25
3.7 Modifications....25
3.8 Memory-Mapping Support....26

LIBMTHCA – Tavor/Arbel User Space Library....28
3.9 Modifications....28
3.10 Queue Allocation Changes....30

LIBMLX4 – Hermon User Space Library....31
3.11 Modifications....31
3.12 Queue Allocation Changes....32
3.13 User-Space Doorbell Changes....33

1 Purpose

This document is intended to provide implementation details of the various OFUV components that comprise the OFED user space port on Solaris. This document is targeted to an audience that would be involved in the extension or maintenance of the software components.

Sun documentation exists that describes the overall OFUV software architecture for the project, and that documentation should be reviewed prior to this document[1].

An iterative early access release process was utilized during the development phase of this project, with each release adding additional support. This document is based on the EA-2 release of the software, which offers both UD and RC support for the Tavor and Hermon drivers, and RDMA CM support for multicast and the UD transport. It is not anticipated the elements discussed in this document will require change for the EA-3 release, which will add RDMA CM support for the RC transport.

[1] Reference "The Solaris Open Fabrics User Verbs Architecture", by Pramod Gunjikar and Brendan Doyle.

2 Overview

Tasked involved in this project could largely be sorted into four development areas:

- Building of the code and Solaris environment compatibility modifications.
- Enabling the OFED user space verbs infrastructure on Solaris.
- Enabling the OFED user space common RDMA communication management infrastructure on Solaris.
- Enabling the OFED user space MAD infrastructure on Solaris.

This document focuses on the implementation and issues as they relate to the enablement of the OFED user space verb infrastructure (LIBIBVERBS) and the associated kernel agent (SOL_UVERBS). It also briefly touches on the OFED user space hardware specific libraries that provide the device specific verb implementation (LIBMTHCA and LIBMLX4), and a thin user space compatibility library created to handle differences in the Linux and Solaris environments.

The documentation on the build system, both nuances and procedures is documented in the source tree itself and is not covered in this document[2]. The enabling of the OFED user space common RDMA communication management (LIBRDMACM) was largely performed by Sun and is not covered in this document. Finally, the MAD infrastructure is largely independent of the verbs infrastructure and will be presented in separate text, which in all likelihood could ultimately be folded into this document.

The code itself is well documented and should be used to reference the details of individual functions; this document strives to describe operation at a higher level that is not necessarily easily obtained from review of the code.

[2] Genernal build comments may be found in the "top of source tree" in the text file "how_to_build".

3 SOL_UVERBS – OFUV Kernel Agent Driver

The SOL_UVERBS component is the kernel agent providing the translation from the OFED User Verbs library kernel API to the Solaris IB Transport Framework API (IBTF). The driver utilized the Solaris DAPL kernel agent driver as an example of interaction with the IBTF. It merges the capability and semantics of the Linux User Verbs kernel driver that are expected by the OFED user space libraries with the semantics and requirements required of the IBTF. A basic understanding of the IBTF and the Linux User/Kernel interaction is a necessity in realizing many of the nuances of the driver.

The SOL_UVERBS driver is a single instance pseudo device driver that registers itself as a client to the IBTF framework. It then creates a character minor device node for each HCA reported by the IBTF. These device nodes are utilized by LIBIBVERBS to open specific HCA instances. The character device names follow the format of "/devices/ib/sol_uverbs@0:uverbs#", where "#" is the minor number 0..Max-1 and maps to an HCA in the IBTF HCA list. SOL_UVERBS exports device properties for each minor device that may be used to assist in determining the proper character minor device to open.

SOL_UVREBS also creates one additional character minor device node that may only be opened from within the kernel and is provided for the SOL_IB_UCMA kernel module use. This character device name is "/devices/ib/sol_uverbs@0:ucma".

The following is example output for the SOL_UVERBS device nodes:

```
prtconf -v /devices/ib/sol_uverbs@0:uverbs3
sol_uverbs, instance #0
    System software properties:
        name='ddi-forceattach' type=int items=1
            value=00000001
    Driver properties:
        name='device-id' type=int items=1 dev=(217,3)
            value=0000634a
        name='vendor-id' type=int items=1 dev=(217,3)
            value=000015b3
        name='guid' type=int64 items=1 dev=(217,3)
            value=0002c90200258764
        name='device-id' type=int items=1 dev=(217,2)
            value=0000634a
        name='vendor-id' type=int items=1 dev=(217,2)
            value=000015b3
        name='guid' type=int64 items=1 dev=(217,2)
            value=0003ba0001005040
        name='device-id' type=int items=1 dev=(217,1)
            value=00006282
        name='vendor-id' type=int items=1 dev=(217,1)
            value=000015b3
        name='guid' type=int64 items=1 dev=(217,1)
```

```
            value=0002c90200227d18
        name='device-id' type=int items=1 dev=(217,0)
            value=00006278
        name='vendor-id' type=int items=1 dev=(217,0)
            value=000015b3
        name='guid' type=int64 items=1 dev=(217,0)
            value=0002c9020024ad48
        name='abi-version' type=int items=1 dev=(217,0)
            value=00000006
    Hardware properties:
        name='client-guid' type=string items=1
            value='sol_uverbs,0'
    Paths from multipath bus adapters:
        tavor#1 (online)
    Device Minor Nodes:
        dev=(217,0)
            dev_path=/ib/sol_uverbs@0:uverbs0
                spectype=chr type=minor
        dev=(217,1)
            dev_path=/ib/sol_uverbs@0:uverbs1
                spectype=chr type=minor
        dev=(217,2)
            dev_path=/ib/sol_uverbs@0:uverbs2
                spectype=chr type=minor
        dev=(217,3)
            dev_path=/ib/sol_uverbs@0:uverbs3
                spectype=chr type=minor
        dev=(217,16)
            dev_path=/ib/sol_uverbs@0:ucma
                spectype=chr type=minor
```

3.1 Interfaces

SOL_UVERBS implements three interfaces, the OFED user/kernel ABI, and the SOL_IB_UCMA/SOL_UVERBS module interface, and a mechanism to share the list of IBTF HCA's it maintains. The SOL_UVERBS user/kernel ABI implementation is described within this document, the SOL_IB_UCMA/SOL_UVERBS interface is largely driven by the SOL_UCMA implementation. SOL_UVERBS is a consumer of the standard DDI/DDK interfaces and in once case the VFS interface[3].

3.1.1 User/Kernel ABI

The OFED header files are used to define the user-kernel ABI and consist of both a user space implementation and a kernel space implementation. Two files are utilized since the packaging and delivery of user and kernel source and/or binaries may occur independently. The SOL_UVERBS driver exports a device property named "abi-version" that may be examined by the consumer and used to determine the appropriateness of the user/kernel ABI compatibility. The property has the following format:

```
name='abi-version' type=int items=1 dev=(217,0)
```

[3] The VFS interface is utilized in a fashion similar to the OFED user verb kernel driver to facilitate event delivery of asynchronous and completion events.

```
value=00000006
```

If the user/kernel ABI revision are compatible, the consumer opens the desired character minor device for use. At the time of the open SOL_UVERBS creates unique minor device number beyond the range of all possible HCA/UCMA minor devices, and uses that to identify the caller's user context on subsequent calls into the driver.

The consumer then uses the device "write" entry point to invoke commands associated with the user/kernel ABI. The user commands are defined in "/usr/include/infiniband/kern-abi.h"; the kernel definition is defined in "compat/kernel/ib_user_verbs.h". Where implementation specific information may be required, the commands allow for an opaque data area to be passed transparently between the underlying kernel device specific channel interface implementation and the device specific user space consumer of the ABI (e.g. the HERMON kernel driver may pass HERMON specific variables associated with a command to the device specific consumer entity LIBMLX4).

This ABI has been extended with support for two commands not required by the OFED implementation, IB_USER_VERBS_CMD_QUERY_GID and IB_USER_VERBS_CMD_QUERY_PKEY. In the OFED kernel implementation the underlying hardware specific driver exposes the device's GID table and PKEY table entries via the sys file system. In the absence of the sys file system, the consumer utilizes the SOL_UVERB driver to read the values from the underlying hardware specific driver.

3.1.1.1 Kernel Command Summary

The following is a brief overview of the supported user-kernel ABI command. For the complete description of the processing associated with each command please reference the "C" source code function block headers for reach command.

IB_USER_VERBS_CMD_GET_CONTEXT - Returns the user context information that was allocated when the character driver was opened. Included in this is the file descriptor number to use for handling of asynchronous events.

IB_USER_VERBS_CMD_QUERY_DEVICE - Initiates an ibt_query_hca() operation on the underlying HCA device and converts the results to the OFED definitions. Note that the device firmware revision is not returned by the IBTF implementation and is set 0 in the attributes returned by SOL_UVERBS.

IB_USER_VERBS_CMD_QUERY_PORT - Initiates an ibt_query_hca_ports() operation on the underlying HCA device and port and converts the result to the OFED HCA port attributes. Note that the port's active link width and speed, as well as the physical state are not returned by the IBTF port attributes structure and as such are invalid here. The link width and speed are set to 4X and SDR defaults, while the port physical state is left at 0 in the attributes returned by SOL_UVERBS.

IB_USER_VERBS_CMD_ALLOC_PD – Creates a protection domain user object and associates it with the underlying protection domain returned from ibt_alloc_pd(). The opaque channel interface protection domain data is returned to the caller.

IB_USER_VERBS_CMD_DEALLOC_PD - De-allocates a protection domain via the ibt_free_pd() call and releases the user protection domain object.

IB_USER_VERBS_CMD_REG_MR – Allocates a memory region user object and places a reference on the associated protection domain use object. The associated memory region is obtained using the ibt_register_mr() call.

IB_USER_VERBS_CMD_DEREG_MR - De-register a memory region via the ibt_deregister_mr() call and cleans up the user objects by releasing the reference to the associate protection domain, and frees the memory region user object.

IB_USER_VERBS_CMD_CREATE_COMP_CHANNEL - Create a completion channel user object and allocates a VNODE to be associated with the new event file. The VNODE is assigned to file descriptor in the caller's file descriptor table and then returned to the caller to be used for reading of completion queue events from user space.

IB_USER_VERBS_CMD_CREATE_CQ – Allocates a completion queue user object and if a completion channel is specified it looks up and places a reference on the appropriate completion channel file descriptor and user object. An underlying CQ is allocated via the ibt_alloc_cq() call. The private data for the IBT CQ is set the associated user object that will be used in handling of events. The opaque channel interface completion queue data is returned to the caller.

IB_USER_VERBS_CMD_RESIZE_CQ – The specified completion queue is resized via the ibt_resize_cq() call. The new opaque channel interface completion queue data is returned to the caller.

IB_USER_VERBS_CMD_POLL_CQ – Although a generic kernel implementation is provided via the ibt_poll_cq() call, all of the supported OFED hardware user libraries poll the completion queue in an OS Bypass manner and will not utilize this call.

IB_USER_VERBS_CMD_REQ_NOTIFY_CQ – Although a generic kernel implementation is provided via the ibt_enable_cq_notify() call, all of the supported OFED hardware user libraries arm the completion queue in an OS Bypass manner and will not utilize this call.

IB_USER_VERBS_CMD_DESTROY_CQ - Destroys the completion queue via the ibt_free_cq(), and releases resource associated with the completion queue user object. All potentially queued completion and asynchronous events are

released.

IB_USER_VERBS_CMD_CREATE_QP – Allocates a queue pair user object and creates the underlying queue pair via the ibt_alloc_qp() call. The private data for the IBT QP is set to the user object and will be used in handling of asynchronous events. The opaque channel interface queue pair data is returned to the caller. Note that only UD and RC queue pairs are supported.

IB_USER_VERBS_CMD_QUERY_QP – Queries the queue pair associated with the user object specified via the ibt_query_qp() call. The data is converted to the OFED format and returned to the caller.

IB_USER_VERBS_CMD_MODIFY_QP – Modifies the state/attributes of the IBT QP associated with the user object queue pair specified using the ibt_modify_qp() call. The actual modify operation may be disabled, in which case the actual ibt_modify_qp() call is not made. The intent is to allow the SOL_IB_UCMA module to disable modify operations on a queue pair that will be brought up by the Solaris CM. Note that only UD and RC queue pairs are supported.

IB_USER_VERBS_CMD_DESTROY_QP – Frees the IBT queue pair via the ibt_free_qp() call and releases queue pair user object resources. Any asynchronous events pending for this queue pair will be released.

IB_USER_VERBS_CMD_ATTACH_MCAST – Verifies the specified queue pair user object is not already a member of the indicated multicast group, and if not joins the IBT QP to the group via the ibt_attach_mcq() call.

IB_USER_VERBS_CMD_DETACH_MCAST - Detaches the IBT queue pair associated with the specified queue pair user object from the indicated multicast group.

IB_USER_VERBS_CMD_CREATE_SRQ – Allocates a shared receive queue user object and creates the underlying shared receive queue via the ibt_alloc_srq() call. The private data for the IBT SRQ is set to the user object and will be used in handling of asynchronous events. The opaque channel interface shared receive queue data is returned to the caller.

IB_USER_VERBS_CMD_MODIFY_SRQ - Modification of shared receive queue; this function is not supported.

IB_USER_VERBS_CMD_QUERY_SRQ - Returns the attributes of the IBT shared receive queue associated with the specified shared receive queue user object.

IB_USER_VERBS_CMD_DESTROY_SRQ - Destroys the IBT shared receive queue via the ibt_free_srq() call and releases the shared receive queue user object resources. Any asynchronous events pending for this shared receive queue will be released.

IB_USER_VERBS_CMD_QUERY_GID – Queries the underlying IBT HCA and port via the ibt_query_hca_ports() call and returns the GID for the index specified to the caller. This command is an extension to the base OFED user-kernel ABI; in the OFED user verbs implementation the GID values would be read directly from the Linux sys file system.

IB_USER_VERBS_CMD_QUERY_PKEY – Queries the underlying IBT HCA and port via the ibt_query_hca_ports() call and returns the PKEY for the index specified to the caller. This command is an extension to the base OFED user-kernel ABI; in the OFED user verbs implementation the PKEY values would be read directly from the Linux sys file system

The following group of commands does not require a kernel implementation on Solaris since all hardware specific kernel driver/user library implementations allow these operations to be performed in an OS Bypass mode.

IB_USER_VERBS_CMD_POST_SEND

IB_USER_VERBS_CMD_POST_RECV

IB_USER_VERBS_CMD_POST_SRQ_RECV

IB_USER_VERBS_CMD_CREATE_AH

IB_USER_VERBS_CMD_DESTROY_AH

The following commands are intended for XRC use and are not supported by SOL_UVERBS in the EA-2 release.

IB_USER_VERBS_CMD_CREATE_XRC_SRQ

IB_USER_VERBS_CMD_OPEN_XRC_DOMAIN

IB_USER_VERBS_CMD_CLOSE_XRC_DOMAIN

IB_USER_VERBS_CMD_CREATE_XRC_RCV_QP

IB_USER_VERBS_CMD_MODIFY_XRC_RCV_QP

IB_USER_VERBS_CMD_QUERY_XRC_RCV_QP

IB_USER_VERBS_CMD_REG_XRC_RCV_QP

IB_USER_VERBS_CMD_UNREG_XRC_RCV_QP

3.2 Driver Operational Concepts

3.2.1 User Objects

The SOL_UVERBS driver implements the concept of user objects to keep track of and

manage resources allocated on the behalf of a user space consumer. Each user object maintains a reference count for handling dependencies and a read/write mutex to provide appropriate serialization where required. User objects exist for the following resources.

- User Context
- Protection Domain
- Address Handle
- Memory Region
- Completion Queue
- Queue Pair
- Shared Receive Queue
- User Event Files

User objects may also contain variables that facilitate the semantics and/or validation of an object; for example a user queue pair object may contain a list of multicast groups of which it has attached.

Each class of user objects are maintained in separate tables that are similar to a Linux IDR implementation. The tables resize dynamically and facilitate quick lookup using the user space object handle as the identifier. Each user context maintains a list of user object identifiers to facilitate cleanup and removal of user objections upon abnormal process exit.

The user object programming API is fully documented in sol_uverbs.h, and there is a very repetitive pattern throughout the code in how the objects are used. In general creation of a user object consists of allocating memory for the user object and then invoking the user object initialization function. For example to create a new queue pair user object, initialize it, and then hold a write lock on the object the code would follow the pattern below.

```
uqp = kmem_zalloc(sizeof(*uqp), KM_SLEEP);
if (uqp == NULL) {
        ...
}
uverbs_uobj_init(&uqp->uobj, cmd->user_handle,
                  SOL_UVERBS_UQP_UOBJ_TYPE);
rw_enter(&uqp->uobj.uo_lock, RW_WRITER);
```

Additional object specific type and instance initialization would continue and if applicable the associated IBT resource would be set. The object would then be

added to the appropriate user object table.

```
if (uverbs_uobj_add(&uverbs_uqp_uo_tbl, &uqp->uobj) != 0) {
        ...
}
```

Finally once the object initialization is complete, the object valid flag ("uo_live") is set and the write lock released.

```
uqp->uobj.uo_live = 1;
rw_exit(&uqp->uobj.uo_lock);
```

The valid flag is used to make an object visible, when not set a search in the user object table will return as if the object does not exist. The specifically handles the case of a user object read lock request following a user object write lock request that will invalidate the user object.

The initialization of a user object sets the reference count to 1. Several routines are available that allow a user object to be looked up and the appropriate lock taken (read/write) and optionally a reference added. For example the following would lookup the queue pair user object associated with the user handle specified, establishing a read lock that also adds a reference to the object.

```
uqp = uverbs_uobj_get_uqp_read(cmd->qp_handle);
```

When finished the following call would release a single reference to the user object, and frees the object should only a single reference be held to the object.

```
uverbs_uobj_put(&uqp->uobj);
```

3.3 In a similar fashion a write reference to a user object is obtained with the following call.

```
uqp = uverbs_uobj_get_uqp_write(cmd->qp_handle);
```

Finally, to remove an object from the associated user object table the following would be used once a write lock had been taken. The object valid indicator is cleared and the "put" call releases the reference and lock associated with the user object write lookup. The user object is then removed from the table and then the initial reference set when the user object was initialized is removed and the object is freed.

```
uqp->uo_live = 0;
uverbs_uobj_put(&uqp->uobj);
uverbs_uobj_remove(&uverbs_uqp_uo_tbl, &uqp->uobj);
uverbs_uobj_deref(&uqp->uobj, uverbs_uobj_free);
```

Note that once the "uo_live" flag is cleared, the object is no longer visible. Code

serialized on gaining access to this object will have it's referenced released and return as though the object does not exist.

The following headers describe the basic user operations. Note that defines exist to facilitate the invocation for specific user object types (reference sol_uverbs.h).

```
/*
 * Function:
 *      uverbs_uobj_ref
 * Input:
 *      uobj        - Pointer to the user object
 * Ouput:
 *      None
 * Returns:
 *      None
 * Description:
 *      Place a reference on the specified user object.
 */
static void uverbs_uobj_ref(uverbs_uobj_t *uobj)

/*
 * Function:
 *      uverbs_uobj_deref
 * Input:
 *      uobj        - Pointer to the user object
 *      free_func   - Pointer to release function, called if the
 *                    last reference is removed for the user object.
 * Ouput:
 *      None
 * Returns:
 *      None
 * Description:
 *      Remove a reference to a user object.  If a free function
 *      was specified and the last reference is released, then the
 *      free function is invoked to release the user object.
 */
static void uverbs_uobj_deref(uverbs_uobj_t *uobj,
                              void (*free_func)(uverbs_uobj_t *uobj))

/*
 * Function:
 *      uverbs_uobj_get_read
 * Input:
 *      tbl         - Pointer to the user object management table to
 *                    be used in the lookup.
 *      uo_id       - The ID to object mapping, assigned to the user
 *                    object at addition to the table.
 * Ouput:
 *      None
 * Returns:
 *      A pointer to the user object associated with uo_id or NULL
 *      if the entry does not exist.
 * Description:
 *      Lookup a user object and place a reference on it.  Acquires
 *      the object with a READ lock.  The reference and lock should
```

```
 *      be released using the uverbs_uobj_put() call.
 */
static uverbs_uobj_t* uverbs_uobj_get_read(uverbs_uobj_table_t *tbl,
                                           int uo_id)

/*
 * Function:
 *      uverbs_uobj_get_write
 * Input:
 *      tbl          - Pointer to the user object management table to
 *                     be used in the lookup.
 *      uo_id        - The ID to object mapping, assigned to the user
 *                     object at addition to the table.
 * Ouput:
 *      None
 * Returns:
 *      A pointer to the user object associated with uo_id or NULL
 *      if the entry does not exist.
 * Description:
 *      Lookup a user object and place a reference on it.  Acquires
 *      the object with a WRITE lock.  The reference and lock should
 *      be released using the uverbs_uobj_put() call.
 */
static uverbs_uobj_t* uverbs_uobj_get_write(uverbs_uobj_table_t *tbl,
                                            int uo_id)

/*
 * Function:
 *      uverbs_uobj_add
 * Input:
 *      uo_tbl  - A pointer to the user object resource management
 *                table to which the object should be added.
 *      uobj    - A pointer to the user object to be added; a reference
 *                should exist on this object prior to addition, and
 *                the object should be removed prior to all references
 *                being removed.
 * Output:
 *      uobj    - The user object "uo_id" is updated and should be
 *                used in subsequent lookup operations.
 * Returns:
 *      DDI_SUCCESS on success, else error code.
 * Description:
 *      Add a user object to the specified user object resource
 *      management table.
 */
int uverbs_uobj_add(uverbs_uobj_table_t *uo_tbl, uverbs_uobj_t *uobj)

/*
 * Function:
 *      uverbs_uobj_remove
 * Input:
 *      uo_tbl  - A pointer to the user object resource management
 *                table from which the object should be removed.
 *      uobj    - A pointer ot the user object to be removed.
 * Output:
 *      None
 * Returns:
```

```
 *      A pointer to the user object that was removed on success,
 *      otherwise NULL.
 * Description:
 *      Remove a user object from the specified user resource
 *      management table.
 */
uverbs_uobj_t* uverbs_uobj_remove(uverbs_uobj_table_t *uo_tbl,
                                  uverbs_uobj_t *uobj)
```

3.3.1 User Context

A user context is a special user object created and initialized for each consumer device open. A new SOL_UVERBS minor device is cloned, with the minor number mapping back to the user objects handle. On subsequent calls into the SOL_UVERBS driver to process user-kernel ABI commands, the user context handle is extracted and used to lookup the user object user context. This in turn places a read lock on the user context object and increments it's reference count.

An asynchronous event file user object is created as well during the creation of the user context and the associated file descriptor set in the callers file descriptor table. The descriptor is returned to the caller when it executes an IB_USER_VERBS_CMD_GET_CONTEXT command. The user context object and the asynchronous event file object each place a reference on the other; this is used to handle the race that can occur between the consumer's closing of the SOL_UVERBS device file descriptor and the closing of the SOL_UVERBS asynchronous event file descriptor.

```
/*
 * User Context.
 *
 * A user context is created when a user process opens a specific minor
 * device.  The context maintains a list of resources created by this
 * user that allows the resources to be cleaned up on user close.
 */
typedef struct uverbs_uctxt_uobj {
        uverbs_uobj_t             uobj;
        kmutex_t                 lock;
        uverbs_module_context_t *mod_ctxt;
        sol_uverbs_hca_t         *hca;       /* ptr to specific hca */

        /*
         * List of user resource objects created by this context.  The
         * objects themselves live in the associated object table, and
         * the code should use the table to access and use resources.
         * Any objects that remain in these list will be destroyed at
         * user close to free the associated resources.
         *
         * The user context "lock" should be held when invoking
         * routines to manipulate the lists.
         */
        genlist_t                pd_list;
        genlist_t                mr_list;
```

```
        genlist_t                cq_list;
        genlist_t                qp_list;
        genlist_t                srq_list;
        genlist_t                ah_list;

        /*
         * Event filesystem interfaces for IB asyncrhonous events
         * and completion events.
         */
        uverbs_ufile_uobj_t     *comp_evfile;
        uverbs_ufile_uobj_t     *async_evfile;
} uverbs_uctxt_uobj_t;
```

The completion event file user object is set when the consumer executes an IB_USER_VERB_CMD_CREATE_COMP_CHANNEL command.

When the SOL_UVERBS device file descriptor is closed, the user context object is removed from the associated management table and the references to it and the reference it holds on the associated event files released. Depending on if the event files have been closed the object may not be immediately freed, but will be freed when the last event file reference is released.

3.3.2 Event Processing

The SOL_UVERBS driver maps IBT asynchronous events and IBT completion notification events to the associated user object for delivery to user space consumers. Delivery of events is performed via two unique file descriptors and the associated read and poll operations. The first, the event file used for asynchronous events is created when the user context object is created as part of the consumer open of SOL_UVERBS. The second, the completion notification event file is specifically created when the consumer requests creation of a completion channel via the IB_USER_VERB_CMD_CREATE_COMP_CHANNEL.

SOL_UVERBS asynchronous event processing maps affiliated asynchronous events back to the associated user object and extracts the appropriate user handle to pass to the consumer with the event. Unaffiliated asynchronous events are passed to all consumers utilizing the underlying IBT HCA.

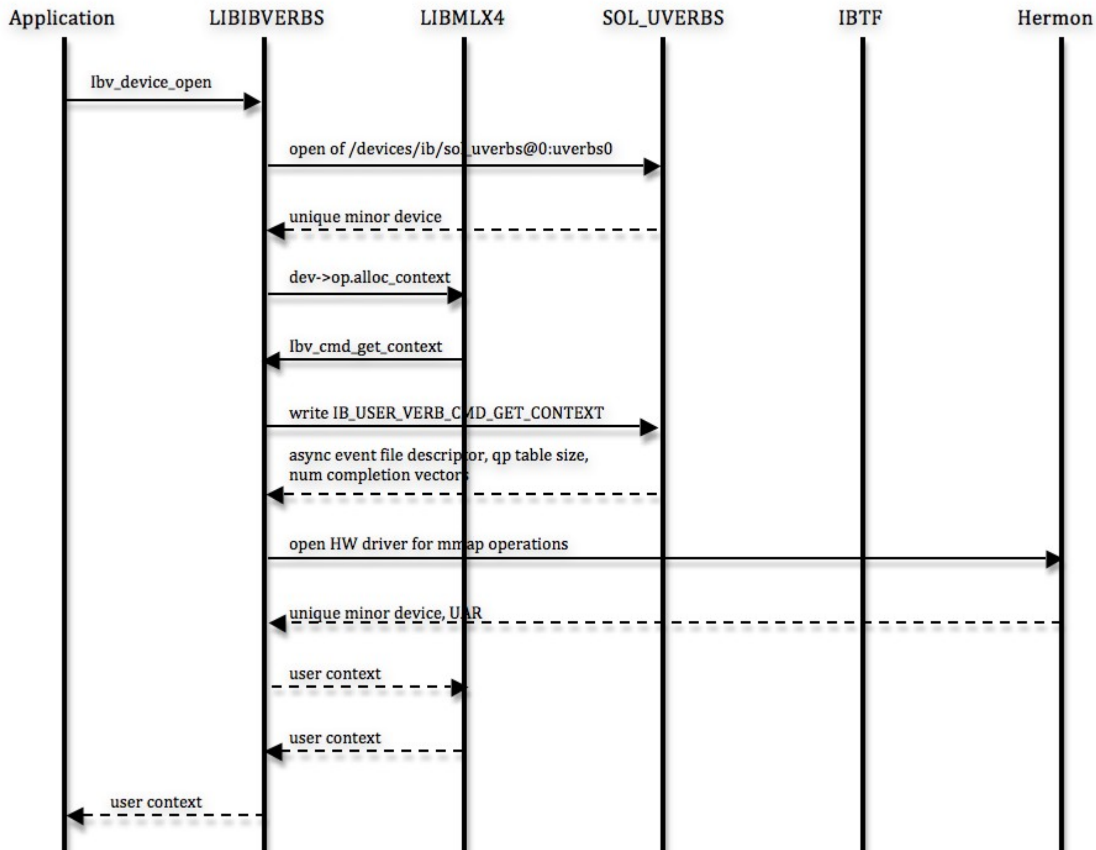
SOL_UVERBS completion notification event processing maps completion notification events back to the appropriate completion queue user object and extracts the appropriate handle to pass to the consumer with the event.

3.4 Sequence Diagrams

While the SOL_UVERBS code path taken with the individual execution of user-kernel ABI commands is easily understood from review of the source code and comments, it is not necessarily apparent how sequence of operations are performed in normal operation. The following sequence diagrams are meant to give a better general understanding of how operations are sequenced, but are not meant to be exhaustive.

3.4.1 SOL_UVERBS Device Open

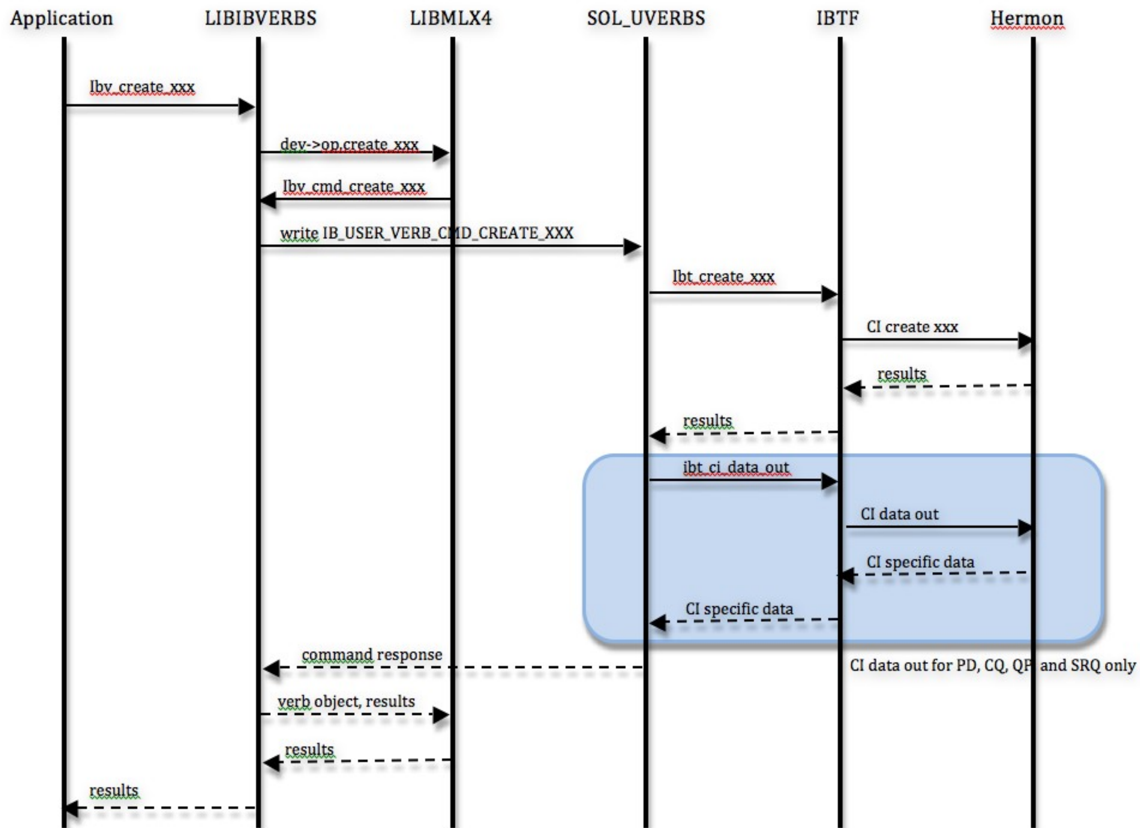
The OFED User Verbs library maintains a list of known IB devices and provides this list of devices to it's consumers via ibv_get_device_list(). Applications register with a specific device in the list by invoking the OFED User Verbs library routine ibv_open_device(); that routine will open the user verbs character device node associated with the device of interest (e.g. mlx4_0). This scenario depicts the flow of operations associated with opening the device, which in turn creates the appropriate user context for the actors in the scenario.



The IBTF context is created when the SOL_UVERBS driver attaches and opens the HCA devices. That context is shared among all consumers of the SOL_UVERBS services.

3.4.2 SOL_UVERBS Resource Management

The OFED User Verbs API is well documented in the MAN pages delivered with the distribution. The majority of the API deals with the creation and destruction of HCA resources and the following scenario demonstrates the typical flow to allocate or release a resource.



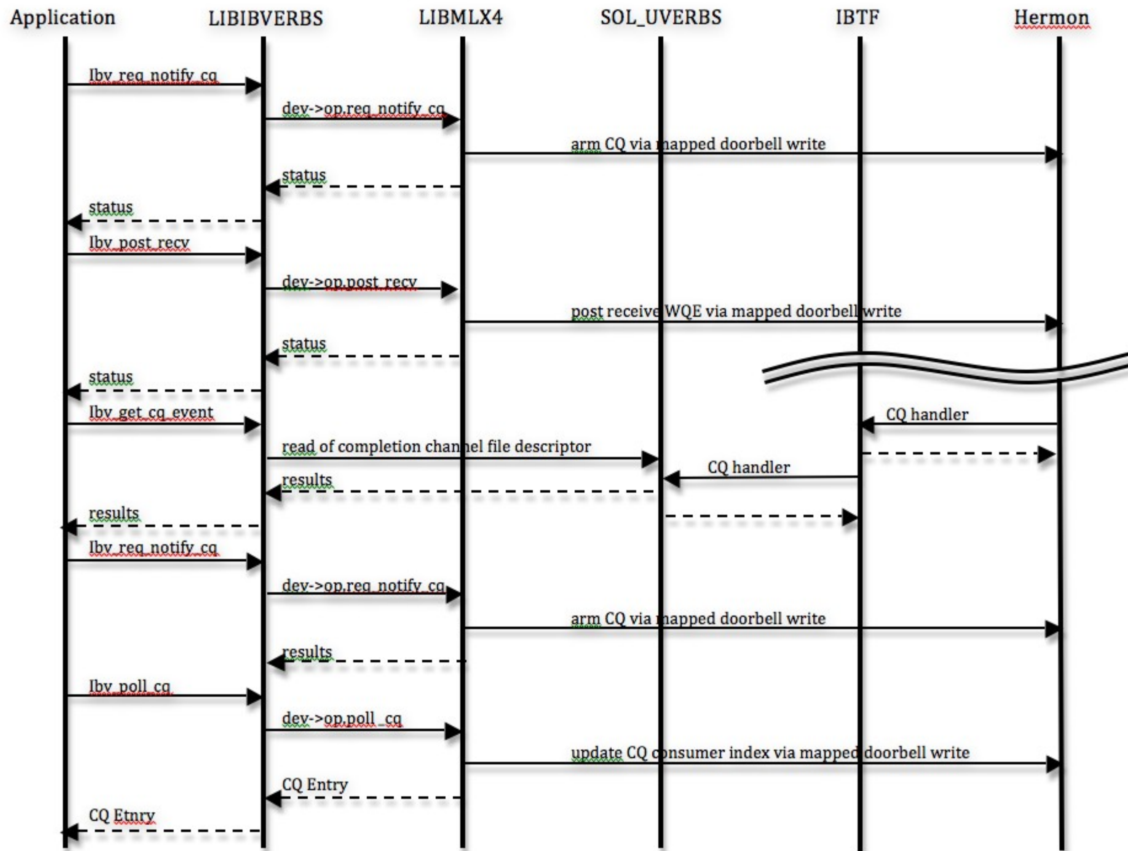
The Channel Interface data out is opaque data passed through the SOL_UVERBS and IBTF. It is used to pass implementation specific data between the Solaris hardware kernel driver and the hardware specific user space library. In this case, the Hermon driver and LIBMLX4. CI data out is provided for protection domains, completion queues, queue pairs, and shared receive queues, and is most commonly used to pass the offset and length to be used to map memory allocated by the kernel driver into the callers address space.

3.4.3 SOL_UVERBS Transport Scenario

This scenario depicts a sequence of a user space consumer performing simple transport operations using completion queue event notifications do determine the presence of completion queue entries. To simplify the scenario it is assumed that the consumer has previously allocated all of the required device resources (protection domain, buffer memory regions, completion event channel, completion queues and queue pair) and brought the QP to the RTS state.

Notice that once the QP is at this state the SOL_UVERBS driver has very little interaction and is only required when the consumer is using CQ events as in this example. That is if the consumer were to do CQ polling, the SOL_UVERBS driver would be completely bypassed in this example; LIBMLX4 would only communicate directly with the underlying hardware using doorbells mapped into user address space.

When using event notification as in this scenario, when the SOL_UVERBS driver IBTF CQ handler is invoked it maps the notification back to the associated user space CQ object and associated context and queues the event. If a caller is blocked waiting for an event it will be awoken.



3.5 Source File Breakdown

The SOL_UVERBS driver functionality is grouped in several source/header files based on related functionality.

sol_uverbs.h: Function prototypes and data types for SOL_UVERBS driver state, user objects, event files, as well as functions for managing user objects.

sol_uverbs_comp.h: Function prototypes for completion queue related operations.

sol_uverbs_event.h: Function prototypes associated with event processing.

sol_uverbs_hca.h: Function prototypes and data structures that define the interface that allows other modules to share the IBTF HCA information and client handle owned by SOL_UVERBS.

sol_uverbs_qp.h: Function prototypes associated with queue pair operations.

sol_uverbs.c: Main driver entry points as well as implementations for the following commands:

IB_USER_VERBS_CMD_GET_CONTEXT

IB_USER_VERBS_CMD_QUERY_DEVICE

IB_USER_VERBS_CMD_QUERY_PORT

IB_USER_VERBS_CMD_ALLOC_PD

IB_USER_VERBS_CMD_DEALLOC_PD

IB_USER_VERBS_CMD_REG_MR

IB_USER_VERBS_CMD_DEREG_MR

IB_USER_VERBS_CMD_CREATE_AH

IB_USER_VERBS_CMD_DESTROY_AH

IB_USER_VERBS_CMD_CREATE_COMP_CHANNEL

sol_uverbs_comp.c: IBT completion queue event handler and implementations for the following commands:

IB_USER_VERBS_CMD_CREATE_CQ

IB_USER_VERBS_CMD_RESIZE_CQ

IB_USER_VERBS_CMD_POLL_CQ

IB_USER_VERBS_CMD_REQ_NOTIFY_CQ

IB_USER_VERBS_CMD_DESTROY_CQ

sol_uverbs_event.c: Implementation of the functions required to support the event processing required by both asynchronous and completion event delivery.

sol_uverbs_hca.c: Implementation of the common IBT HCA list interface, and the special functionality required by SOL_IB_UCMA to disable QP modificaitons.

sol_uverbs_qp.c: Implementations for the following queue pair and shared receive queue commands:

IB_USER_VERBS_CMD_CREATE_QP

IB_USER_VERBS_CMD_QUERY_QP

IB_USER_VERBS_CMD_MODIFY_QP

IB_USER_VERBS_CMD_DESTROY_QP

IB_USER_VERBS_CMD_CREATE_SRQ

IB_USER_VERBS_CMD_QUERY_SRQ

IB_USER_VERBS_CMD_MODIFY_SRQ

IB_USER_VERBS_CMD_DESTROY_SRQ

IB_USER_VERBS_CMD_ATTACH_MCAST

IB_USER_VERBS_CMD_DETACH_MCAST

3.6 Known Issues/Limitations

The following list indicates known issues or limitations of the EA-2 SOL_UVERBS software.

1. Supports UD and RC transports only.
2. The implementation for IB_USER_VERBS_CMD_QUERY_DEVICE, sol_uverbs_query_device(), has insufficient information provided by IBTF to set the returned firmware revision, and QP per Fast Memory Region. Both of these values are set to 0 and it has no impact on functionality provided by EA-2.
3. The implementation for IB_USER_VERBS_CMD_QUERY_PORT, sol_uverbs_query_port(), has insufficient information provided by IBTF to set active link width, active link speed, and physical state of the link. The values are set to “4X”, “2.5 GHz”, and “0” respectively. These values have no impact on functionality provided by EA-2, but utilities they display this information may display inaccurate data.
4. The SOL_UVERBS event delivery mechanism is reliant on use of VFS services. This implementation is patterned off of the Linux design, and requires the least amount of change to user libraries.
5. Shared Receive Queue’s do not work on Hermon for Sparc based architectures; they do work for X86 architectures.

LIBIBVERBS – OFED IB Verbs User Space Library

The LIBIBVERBS provides the OFED IB Verbs user space API to consumers. It utilizes the callback functions provided by the user space hardware specific libraries, the SOL_UVERBS kernel agent, and Linux sys file system replacement to implement the API. The API itself is well documented in the MAN pages delivered with the OFED distribution.

An important goal of the Solaris OFUV project was to limit changes to the OFED user space libraries to facilitate advancing the code as the OFED distribution moves forward. To a large extent this was successful in terms of the LIBIBVERBS code with the exception of the reliance of the code on the Linux sys file system. In Linux, the underlying kernel drivers export both static and dynamic device specific values via the sys file system, which is used by LIBIBVERBS in the implementation of parts of the API. Creation of a fully capable replacement to the sys file system was beyond the immediate scope of this project and subsequently many of the changes to LIBIBVERBS are directly related to retrieving values associated with sys file system reads.

3.7 Modifications

The EA-2 LIBVERBS source code is contained in ofa_user-1.3/libibverbs-1.1.1/src, public header files are located in ofa_user-1.3/libibverbs-1.1.1/include/infiniband and are delivered to the appropriate installed location when built. Changes required to the source are bracketed with an "#ifdef OFA_SOLARIS" compile toggle to help identify the changes when moving them forward to new OFED releases.

1. cmd.c

    - After executing user-kernel command IB_USER_VERBS_CMD_GET_CONTEXT the appropriate Solaris hardware specific kernel driver instance (i.e. Hermon, Tavor, or Arbel) is opened to support mmap operations.

1. device.c

    - When opening SOL_UVERBS kernel agent character device, use the directory appropriate for Solaris, "/devices/ib".

    - Close the Solaris hardware specific kernel driver instance that was opened to support memory-mapping operations.

1. ibverbs.h

    - Undefine HAVE_SYMVER_SUPPORT.

1. include/infiniband/arch.h

- Determine endianess in Solaris specific manner.

1. include/infiniband/kern-abi.h

   - Addition of IB_USER_VERBS_CMD_QUERY_GID and IB_USER_VERBS_CMD_QUERY_PKEY to facilitate reading of values via the kernel agent instead of the sys file system.

   - Modify command responses as required to return IBT channel interface data out where appropriate.

1. include/infiniband/verbs.h

   - Add file descriptor to be used for memory map operations to ibv_context.

1. init.c

   - When determining SOL_UVERBS devices parse the "/devices/ib" directory instead of using the Linux sys file system.

   - Do not execute the routine check_memlock_limit() that is required for Linux.

1. sysfs.c

   - A replacement for ibv_read_sysfs_file() that provides a mechanism to retrieve values normally retrieved via the sys file system. Although adequate for an EA release, this code is a candidate for replacement with a more suitable Linux sys file system replacement.

1. verbs.c

   - Implements the ibv_query_gid() command as a user-kernel command. Normally GIDs would be read via the Linux sys file system. The replacement issues a user-kernel command to the SOL_UVERBS driver to read the specified GID entry.

   - Implements the ibv_query_pkey() command as a user-kernel command. Normally PKEYs would be read via the Linux sys file system. The replacement issues a user-kernel command to the SOL_UVERBS driver to read the PKEY at the specified index.

3.8 Memory-Mapping Support

In the Solaris IBT model, the kernel provider driver returns buffer offset/length pairs to the user-space provider library and then supports the memory mapping operations to map those offsets into the user processes address space. The user verbs code centralizes this change and handles opening the correct Solaris IB kernel provider driver when the device is opened and the user context allocated, and closes that driver when the device is closed. The mmap() file descriptor is added to the

user context and will be available to the user-space provider libraries when they must perform a memory mapping operation.

LIBMTHCA – Tavor/Arbel User Space Library

LIBMTHCA provides the Tavor, Arbel in Tavor compatibility mode, and Arbel memfree OFED hardware specific user provider library implementation. Currently only the non-memfree (Tavor and Arbel in Tavor compatibility mode) modes are supported; the decision was made to implement the Arbel memfree code path upon completion of the Hermon user library. This decision was motivated with the belief that the Arbel memfree operation would require a similar flow to the Hermon user library and the higher priority of the Hermon support.

LIBMTHCA has been modified to support detection the appropriate mode and will take the memfree code path if the underlying hardware is Arbel memfree. The underlying support is in place in LIBIBVERBS as well.

An important goal of the Solaris OFUV project was to limit changes to the OFED user space libraries to facilitate advancing the code as the OFED distribution moves forward. To a large extent this was successful in terms of the LIBMTHCA code. The majority of the changes are related to the kernel vs. user allocations of CQ and QP/SRQ memory. In general, this results in an inversion of the user library logic associated with CQ, QP, and SRQ operations and adds code to extract the kernel parameters and perform the memory-mapping operations required for OS bypass operation.

3.9 Modifications

The EA-2 LIBMTHCA source code is contained in ofa_user-1.3/libmthca-1.0.4/src. Changes required to the source are bracketed with an "#ifdef OFA_SOLARIS" compile toggle to help identify the changes when moving them forward to new OFED releases.

1. cq.c

    - Modified poll function WQE index calculations to handle Solaris case where send work queue is placed in front of receive work queue (based on alignment).
    - Modifications to not allocate user space CQE memory.

1. mthca.c

    - Add linkage to the IB device for memfree determination.
    - Handle QP table sizing differences, round up to power of 2 for OFED requirements.
    - Map the User Access Region using the contexts memory map file descriptor.

1. mthca.h

    - Modify the QP context to allow for the RX WQE memory to follow the TX WQE memory. This is required by Solaris kernel and is reduces waste that would otherwise occur as a result of queue alignment constraints.

    - Prototypes for new functions.

1. mthca-abi.h

    - Include sys/ib/adapters/mlnx_umap.h for channel interface data out definitions.

1. qp.c

    - Modify receive WQE index calculations to handle Solaris case where send work queue is placed in front of receive work queue (based on alignment).

    - Modifications to initialize and set parameters associated with Solaris QP kernel allocated memory.

1. srq.c

    - Modifications to initialize and set parameters associated with Solaris SRQ kernel allocated memory.

    - Disable user space allocation of SRQ work queue buffer.

1. verbs.c

    - Extract PD CI data out information and set in user context.

    - Extract CQ CI data out information mapping memory and initializing context as appropriate.

    - Disable user space allocation of CQ buffer.

    - Unmap CQ memory as appropriate.

    - Extract SRQ CI data out information mapping memory and initializing context as appropriate.

    - Unmap CQ memory as appropriate.

    - Extract QP CI data out information mapping memory and initializing context as appropriate.

    - Disable user space allocation of QP buffers.

    - Unmap QP memory as appropriate.

3.10 Queue Allocation Changes

As indicated, the majority of the changes are related to the semantic differences between the IBT and OFED implementations with regard to work and completion queue memory allocation. In the Mellanox hardware specific user library implementations for OFED, queue memory, be it QP/SRQ work queue memory or CQ completion entry memory, is allocated by the user-space library provider. In the Solaris IBT model, the Solaris kernel provider allocates queue memory and returns the offsets and lengths back to the user library that then maps the memory into its user address space.

In addition to the allocation of memory changes, the user library generally inverts the initialization of the memory logic with regard to the sequencing of the call back into the kernel. That is, generally in OFED the initialization of work queue memory is done prior to the call into the kernel as part of the allocation of the memory. In the Solaris implementation, the code is changed to perform the initialization after the call into the kernel and the subsequent mapping of the kernel allocated memory.

LIBMLX4 – Hermon User Space Library

LIBMLX4 provides the Hermon (Connect-X) OFED hardware specific user library implementation in terms of the IB device support.

An important goal of the Solaris OFUV project was to limit changes to the OFED user space libraries to facilitate advancing the code as the OFED distribution moves forward. To a large extent this was successful in terms of the LIBMLX4 code. The majority of the changes are related to the kernel vs. user allocations of CQ and QP/SRQ memory, and the kernel vs. user assignment of doorbells. In general, this results in an inversion of the user library logic associated with CQ, QP, and SRQ operations and adds code to extract the kernel parameters and perform the memory-mapping operations required for OS bypass operation.

3.11 Modifications

The EA-2 LIBMLX4 source code is contained in ofa_user-1.3/libmlx4-1.0/src. Changes required to the source are bracketed with an "#ifdef OFA_SOLARIS" compile toggle to help identify the changes when moving them forward to new OFED releases. When possible, helper functions where modified to account for the Solaris and Linux kernel driver differences without modification of parameter lists.

1. buf.c

    - The Solaris kernel driver allocates CQ, QP, and SRQ buffers, buffer free routines were modified to unmap buffers.

1. dbrec.c

    - Doorbell record management was rewritten to be compatible with the Solaris kernel driver.
    - Mapped regions are reference counted to avoid excessive mmap calls.
    - Doorbell free maintains the old parameter list to reduce changes, but interfaces with the new tables.

1. mlx4.c

    - Handle QP table sizing differences, round up to power of 2 for OFED requirements.
    - Map the User Access Region using the contexts memory map file descriptor.
    - Setting of BlueFlame buffer size and memory-mapping of BlueFlame UAR.
    - Minor changes for initialization of doorbell maintenance structure.

1. mlx4.h

    - Replace the doorbell page list contained in the mlx4_context with the new doorbell list pointer definition.

    - Update/add function prototypes.

1. mlx4-abi.h

    - Include sys/ib/adapters/mlnx_umap.h for channel interface data out definitions.

1. qp.c

    - Modifications to initialize and set parameters associated with Solaris QP kernel allocated memory.

1. srq.c

    - Modifications to initialize and set parameters associated with Solaris SRQ kernel allocated memory.

1. verbs.c

    - Extract PD CI data out information and set in user context.

    - Extract CQ CI data out information mapping memory for completion queue, doorbells, and initializing context as appropriate.

    - Maintain separate doorbell pointer CQ arm and CQ consumer index (they will both reference the same memory mapped region).

    - Extract SRQ CI data out information mapping memory for shared receive queue, doorbell, and initializing context as appropriate.

    - Extract QP CI data out information mapping memory for work queues, doorbells, handling send queue pre-fetch setting, and initializing context as appropriate.

    - Disable XRC calls.

3.12 Queue Allocation Changes

The same queue allocation semantic differences apply for LIBMLX4 and LIBMTHCA. Please reference section 3.10, Queue Allocation Changes, for the description of the LIBMTHCA changes.

3.13 User-Space Doorbell Changes

In the Solaris Hermon driver implementation, the User-Access Region is setup by the kernel provider driver. The LIBMLX4 code includes changes to the management

of doorbells to account for this difference. For each doorbell the LIBMLX4 code will obtain the proper offset and length from the IBT Channel Interface Data Out structure associated with the verb action and map it into the user process address space.

The actual routine will reference count memory offsets that have been mapped into the user process address space and only actually perform a new mmap() operation if the offset and length specified have not been previously mapped. This greatly reduces the number of kernel transitions in normal operation. As doorbells are freed their reference is released and once all references to a particular offset/length have been removed, the associated memory is unmapped from the user process space.

Please reference the file dbrec.c for a complete description of the implementation and note that where possible the API to the doorbell management was not changed, only the implementation. This was done to isolate the changes when possible.