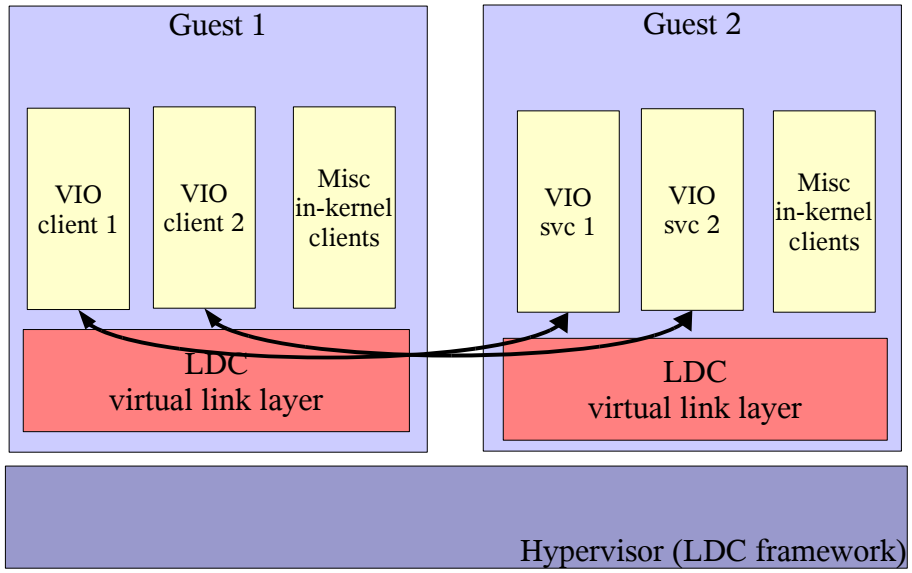
1 Overview

Logical domain channels provide a virtual link layer abstraction that are designed as point-to-point communication channels between logical domains or between a logical domain and an external entity such as a service processor or the Hypervisor itself. Logical domain channels provide an encapsulation protocol onto which higher level transport can be built such as TCP/IP and PPP.



Within a LDom a LDC is instantiated as a single endpoint (unless the LDC loops back to the same LDom). The identity of the owner of the other endpoint is opaque to the LDom - this enables LDCs to be re-connected to other endpoints at will. Conventional attestation protocols may be layered on top of the basic LDC mechanism if the identity of the owner of the other end of a LDC is required. Such attestation is beyond the scope for this document.

Logical Domain Channels (LDC) provide two ways of transferring data between endpoints. A simple packet based transfer mechanism where data is sent in 64-byte packets. The second approach allows clients to export regions of their memory address space with clients at the other end of specified LDC connections. The importing clients can then access the remote memory region by either mapping it into its address space, use an Hypervisor API call to copy data to/from exported memory, or program an IOMMU (or MMU) to directly read/write the memory.

2 sun4v LDC virtual link layer

The Hypervisor Logical Domain Communication (LDC) framework and API provides a simple and fast link-layer mechanism to send packets of up to 64 bytes over a direct connection between two client endpoints.

The goal of any transport is to provide support for connection establishment and termination, flow control, data loss, error reporting and retransmissions when necessary. This section outlines a simple virtual link layer encapsulation protocol implemented as part of the sun4v LDC framework on top of the basic Hypervisor link-level messaging mechanism.

NOTE: The LDC virtual link layer uses network byte ordering for all data transfers.

2.1 Communication overview

- **Packet Based Transfer**

  Data can be transferred out of a virtual machine by encapsulating it into LDC packets or transferring it directly from one domain's memory to another using the Hypervisor shared memory communication support. The link layer protocol will provide client drivers the ability to choose either mechanism for data transfer.

  In the case of the packet based mechanism, the link layer protocol will fragment and reassemble messages as part of the transfer. It will insert additional header information as part of each packet to indicate the start and end of a fragmented data transfer. The actual details of the transfer itself will be invisible to the client driver. It is recommended that this approach be used only for short messages.

- **Shared Memory Access**

  The shared memory access mechanism allows a client driver to make sections of its memory visible to other domains. This support is build on top of the underlying Hypervisor infrastructure for setting up memory map tables to share memory segments. Client drivers will use the interface to obtain a cookie associated with the memory they want to expose. The client can then send the cookie to a client driver in a remote domain using the packet based transfer. The receiving client can then request its LDC framework to consume the cookie and map the remote domain's memory into its address space. Once the mapping is completed, clients can read, write these shared memory regions and also setup DMA operations to directly transfer data into or out of domain buffers.

  A slight modification to the direct memory map is the copy option, where the data is copied in to or out of the buffers that have been exposed by a virtual device client or server via a Hypervisor API. In this approach, when a virtual device wants to send data, either the device client or server will first copy the data from the exporter's memory to a local memory buffer.

  Both methods of data transfer is provided because all virtual machine client may not allow shared memory communication either due to technology limitations or security concerns.

- **Protocol modes**

  Clients of the LDC mechanism can either be clients that implement sophisticated transport layer like capabilities i.e. virtual ethernet with a TCP/IP stack, or a simple client with no special transport capability like the FMA daemon or a virtual console device.

These clients have different reliability requirements on the underlying virtual link layer protocol. The virtual link layer protocol will meet the requirements of either type of client by implementing three different types of data transfer protocol.

- *Raw mode*

  The raw virtual link layer protocol protocol does not add any overhead by appending any headers and sends only 64-byte packets at a time. It has no support for session management, message fragmentation and re-assembly, or retransmissions. It provides a very thin layer over the Hypervisor interface and mostly passes through read and write requests to the Hypervisor.

- *Unreliable mode*

  The unreliable link layer protocol will implement a communication mechanism that will include support of connection establishment via a simple handshake protocol. It will also implement support for negotiating a session and detecting session termination. It will only implement support to detect either lost or out-of-order packets, and not reassemble out of order packets and only stitch together packets received in order. The unreliable mode also supports fragmentation and reassembly of LDC datagrams. Clients of this link layer mechanism will need to implement their own error detection mechanism and do the required retransmission.

- *Reliable mode*

  The reliable link layer protocol implements all the support encompassed within the unreliable link layer protocol. In addition, it implements support for detecting out-of-order packets and packet loss and acknowledges received packets. The primary distinction of reliable mode is to provide an error detection capability via packet ACKs and NACKs. It does not provide a retry or retransmission capability provided in higher level protocols. Higher level uses of reliable mode will need to build their own retry and retansmission machinery.

2.2 LDC link layer API

In Solaris, the LDC virtual link layer support is implemented as a sun4v misc module (`/platform/sun4v/kernel/misc/sparcv9/ldc`) that driver writers will include to make use of the underlying functionality. All interfaces to establish communication, send and receive data, specify channels options, and register callbacks are outlined below. The LDC API interfaces are designed to be called by threads in both the user and kernel context.

2.2.1 Core channel interfaces

- **Channel attribute structure:**

  At the time of initializing the channel for use, the client passes as argument the channel attribute structure. The contents of the structure uniquely identify the client and allow the client to specify The structure has the following format:

```
typedef struct ldc_attr {
        ldc_dev_t  devclass;   /* device class */
        uint64_t   instance;   /* device class inst */
        ldc_mode_t mode;       /* channel mode */
        uint64_t   mtu;        /* Pkt xfer MTU */
```

```
} ldc_attr_t;
```

The '*mode*' field allows clients to set link layer mode for the channel. The valid values are:

```
typedef enum {
      LDC_OPT_RAW,            /* Raw (no link protocol) */
      LDC_OPT_UNRELIABLE,     /* Unreliable mode */
      LDC_OPT_RELIABLE        /* Reliable mode */
} ldc_mode_t;
```

The '*devclass*' field allows a client to provide information to a transport the type of service it is providing or seeking or the type of device it is.

```
typedef enum {
      LDC_DEV_GENERIC,        /* Generic device */
      LDC_DEV_BLK,            /* Block device */
      LDC_DEV_BLK_SVC,        /* Block device service */
      LDC_DEV_NT,             /* Network device */
      LDC_DEV_NT_SVC,         /* Network device service */
      LDC_DEV_SERIAL          /* Serial device */
} ldc_dev_t;
```

- **Channel initialization:**

```
int ldc_init(uint64_t id, ldc_attr_t *attr, ldc_handle_t
*hdl)
  if successful, returns 0,
  else, returns
        EINVAL - invalid channel
        ENOMEM - error allocating memory
        EADDRINUSE - channel already initialized
```

Allocates memory for the internal channel structure, queues and initializes it. The attribute *attr* field specifies the length of the incoming and outgoing queues and link layer mode for the channel. Returns an unique channel handle to the caller.

```
int ldc_fini(ldc_handle_t hdl)
    if successful, returns 0,
    else, returns
          EINVAL - invalid channel
          EBUSY - channel is open
```

Deallocates the internal channel structure and frees queue memory.

- **Connection establishment:**

```
int ldc_open(ldc_handle_t hdl)
     if successful, returns 0,
     else, returns
            EINVAL - invalid handle
            EFAULT - channel already open
            ENOMEM - error allocating memory
            EIO - internal error
```

Registers the incoming and outgoing queues, the interrupt cookie for the LDC channel's interrupt handler and the target CPU for the interrupt with the Hypervisor. If successful, the channel is in OPEN state. The API will return with an error if there is a memory allocation failure. This is to ensure that the calling thread will never go to sleep in the case of inadequate memory.

```
int ldc_close(ldc_handle_t hdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid handle
           EFAULT - channel is not open
           EBUSY - channel in use
           EIO - internal error
```

Unregisters incoming and outgoing queues with Hypervisor. Also unregisters the interrupt cookie and target CPU. Channel is restored to INIT state.

```
int ldc_status(ldc_handle_t hdl, ldc_status_t *status);
    if successful, returns 0
    else, returns
           EINVAL - invalid handle / status arg
```

Returns the current channel status in arg 'status'. Status values are:

```
typedef enum {
      LDC_INIT,         /* Channel is initialized */
      LDC_OPEN,         /* Channel open */
      LDC_READY,        /* Channel link is ready */
      LDC_UP            /* Channel UP */
} ldc_status_t;
```

```
int ldc_up(ldc_handle_t hdl);
    if successful, returns 0,
```

```
                EINVAL - invalid handle
                EIO - internal error
                ECONNREFUSED - unable to initiate handshake
```

Initiates a handshake with the peer endpoint. This interface is non-blocking and on return does not guarantee that the channel is in UP state. The client will need to subsequently call the ldc_status() interface to determine whether the channel is in UP state.

```
int ldc_down(ldc_handle_t hdl);
    if successful, returns 0,
    else, returns
            EINVAL - invalid handle
            EBUSY - cannot reset channel
            EIO - internal error
```

Flushes all internal Rx and Tx queues and switches channel to either OPEN or READY state. The state depends on whether the peer has configured its Rx queues .

- **Event notification**

  This event notification APIs specified in this section is used by LDC clients to register callbacks and enable/disable callbacks when necessary.

```
int ldc_reg_callback(ldc_handle_t hdl,
          (*func)(uint64_t event, caddr_t callback_arg),
          caddr_t callback_arg);
    if successful, returns 0
    else, returns
            EINVAL - invalid handle
            EINVAL - callback already registered
            EBUSY - callback in progress
```

Registers a callback for the channel.

```
int ldc_unreg_callback(ldc_handle_t hdl);
    if successful, returns 0
    else, returns
           EINVAL - invalid handle
           EINVAL - no callback registered
           EBUSY - callback in progress
```

Removes the callback registered for the channel.

- The list of supported callback events are:

```
LDC_EVT_DOWN    0x1         Channel Down, (Status = OPEN)
LDC_EVT_READY   0x2         Channel Reset, (Status = READY)
```

```
LDC_EVT_UP      0x4         Channel Up, (Status = UP)
LDC_EVT_READ    0x8         Channel has data for read
LDC_EVT_WRITE   0x10        Channel has space for write
```

```
int ldc_set_cb_mode(ldc_handle_t hdl, boolean_t enable);
    if successful, returns 0,
    else, returns
            EINVAL - invalid handle
            EIO - unable to disable callbacks
```

Enables or disables callbacks. In callbacks are disabled, packets are deposited in the Rx queue as long as the queue is not full. Incoming pkts can be read but client callback will not be invoked.

- **Point to Point data transfer**

  The LDC virtual link layer framework allows clients to send data to its peer at the other end of the connection in one or more LDC datagram packets. The *ldc_write* and *ldc_read* calls are used to send data embedded in 64-byte datagram packets. The *ldc_chkq* call is used to check the receive queue for pending packets.

```
int ldc_write(ldc_handle_t hdl, caddr_t buf, size_t *len);
    if successful, returns 0,
    else, returns
           EINVAL - invalid handle
           EMSGSIZE - msg too big
           EIO - internal error (channel is reset)
           EWOULDBLOCK - no space / queues are full
           ECONNRESET - channel is not UP / lost connection
```

Send 'len' bytes from buffer 'buf'. On return the 'len' contains the number of bytes written. In the case of a fragmented transfer, either if the sender's transmit queues become full, the write will return EAGAIN and the 'len' argument will be reset to zero (partial transmissions are not done). The LDC framework will wait and retry a couple of times before returning EAGAIN. The sender will receive a callback notification when more space becomes available in the transmit queue.

```
int ldc_read(ldc_handle_t hdl, caddr_t buf, size_t *len);
    if successful, returns 0,
    else, returns
           EINVAL - invalid handle
           EIO - Error during read (lost or missing pkts)
```

```
                ENOBUFS - Not enough buffer space
                ETIMEDOUT - Timed out waiting for fragments
                ECONNRESET - channel is not UP / lost connection
```

Read up to 'len' bytes into buffer 'buf'. On return, 'len' contains the number of bytes read. The LDC framework will return ENOBUFS if the buffer passed is not big enough to hold the incoming data. If an error is encountered during the read, the call will return back with an error and return a value of zero in *len*. The client has to do repeated reads until *ldc_read* returns no error and data (i.e. length of zero).

```
int ldc_chkq(ldc_handle_t hdl, boolean_t *has_data);
    if successful, returns 0,
    else, returns
            EINVAL - invalid handle
            ECONNRESET - Channel is not in UP state
```

Check the receive queue for pending data. Returns true if there is pending data to be read.

2.2.2 Shared memory interfaces

The shared memory framework allows a domain to share regions of its address space with domains at the other end of a channel for either mapping the memory into its address space or copy data to or from the exported memory via a fast copy implemented in the Hypervisor. The basic semantics and API for shared memory support is modeled after the DDI DMA infrastructure.

- **Memory handle (de)allocation and (un)binding**

```
int ldc_mem_alloc_handle(ldc_handle_t hdl,
                                ldc_mem_handle_t *mhdl);
    if successful, returns 0,
    else, returns
            EINVAL - invalid handle
            ENOMEM - Cannot allocate memory handle
```

Allocates a LDC memory handle that can be bound to a buffer for export purposes.

```
int ldc_mem_free_handle(ldc_mem_handle_t mhdl);
    if successful, returns 0,
    else, returns
            EINVAL - invalid mem handle or handle is bound
```

Free a previously allocated LDC memory handle.

```
int ldc_mem_bind_handle(ldc_mem_handle_t mhdl, caddr_t vaddr,
```

```
                    size_t len, uint8_t type, uint8_t perm,
                    ldc_mem_cookie_t *cookie, uint32_t *ccount);
        if successful, returns 0
        else, returns
              EINVAL - invalid mem handle or handle is bound
              EINVAL - addr not 8-byte aligned
              ENOMEM - too many exported pages
```

Bind a memory handle to a virtual address. The real addresses for the pages backing the virtual address range are then shared with the peer. Returns pointer to the first map table *ldc_mem_cookie* and the total number of cookies associated with this virtual address. The client will need to acquire the remaining cookies using the *ldc_mem_nextcookie()* call.

The *type* argument specifies whether the memory is being exported for direct map or copy access (or map via a shadow page) only.

```
LDC_SHADOW_MAP    0x1   /* export for copy access */
LDC_DIRECT_MAP    0x2   /* export for mapped access */
LDC_IO_MAP        0x4   /* export for IO access */
```

The *perm* argument specifies read, write and execute permissions for the exported pages.

```
LDC_MEM_R         0x1
LDC_MEM_W         0x2
LDC_MEM_X         0x4
LDC_MEM_RWX       (LDC_MEM_R|LDC_MEM_W|LDC_MEM_X)
LDC_MEM_RW        (LDC_MEM_R|LDC_MEM_W)
```

```
int ldc_mem_unbind_handle(ldc_mem_handle_t mhdl);
     if successful, returns 0,
     else, returns
           EINVAL - invalid memory handle or not bound
```

Unbind the virtual memory region associated with the specified memory handle. All associated cookies are freed and the corresponding RA space is no longer exported.

```
int ldc_mem_nextcookie(ldc_mem_handle_t mhdl,
                       ldc_mem_cookie_t *cookie);
     if successful, returns 0,
     else, returns
           EINVAL - invalid memory handle
```

Return the next cookie associated with the specified memory handle.

- **LDC Memory copy**

```
int ldc_mem_copy(ldc_handle_t handle, caddr_t vaddr,
                 size_t *size, ldc_mem_cookie_t cookie,
                 uint64_t off, uint64_t  direction);
    if successful, returns 0,
    else, returns
           EINVAL - invalid handle
           EINVAL - vaddr is not 8-byte aligned
           EIO - channel not UP
           EACCES - invalid permissions or copy not permitted
```

Copy *size* amount of data either from or to the client specified virtual address space to or from the exported memory associated with the cookie. The *off* argument corresponds to the relative offset from the base of the exported memory pointed to by the cookie.

```
#define LDC_COPY_IN     0x0
#define LDC_COPY_OUT    0x1
```

The *direction* argument accepts values `LDC_COPY_IN` or `LDC_COPY_OUT` determines whether the data is read from or written to exported memory.

- **LDC direct memory map**

```
int ldc_mem_map(ldc_mem_handle_t mhdl,
                ldc_mem_cookie_t *cookie, uint32_t ccount,
                uint8_t mtype, caddr_t *raddr);
    if successful, returns 0,
    else, returns
           EINVAL - invalid memory handle
           EINVAL - channel not UP, handle not bound
           EACCES - invalid permissions or map not permitted
```

Map memory associated with the cookies into the caller's address space. If the *type* is set to `LDC_SHADOW_MAP`, and *raddr* is NULL, the LDC framework allocates a shadow region of memory corresponding to the exported memory. If *raddr* is not NULL, the LDC framework uses this as the shadow memory. If the *type* is set to `LDC_DIRECT_MAP`, the LDC framework will allocate a RA range that maps to the exported memory.

```
int ldc_mem_unmap(ldc_mem_handle_t mhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid memory handle
           EINVAL - channel not open
```

```
                EACCES - invalid permissions or map not permitted
```

Memory associated with the handle is freed from the caller's address space. If the mapping was of type `LDC_SHADOW_MAP` and the real address was provided by the caller as the time of mapping, the caller will need to free the allocated memory following the unmap call.

- **Memory consistency**

  All LDC mapped memory follows entry consistency. All access to memory has to be preceded by a *ldc_mem_acquire( )* call. This guarantees that the local memory is contents are consistent with the contents of the remote exported memory. The *raddr* and *size* refer to the base and bounds of the memory that is being kept consistent.

  Access to memory has to be followed by a call to *ldc_mem_release( )* call. This guarantees that the remote exported memory contents are consistent with the contents of the local memory. The *raddr* and *size* refer to the base and bounds of the memory that is being kept consistent

```
int ldc_mem_acquire(ldc_mem_handle_t mhdl, uint64_t offset,
                            size_t size);
    if successful, returns 0,
    else, returns
           EINVAL - invalid memory handle
           EINVAL - channel not open
           EPERM - invalid access permissions
           EACCES - memory is no longer mapped


int ldc_mem_release(ldc_mem_handle_t mhdl, uint64_t offset,
                            size_t size);
    if successful, returns 0,
    else, returns
           EINVAL - invalid memory handle
           EINVAL - channel not open
           EPERM - invalid access permissions
           EACCES - memory is no longer mapped
```

- **Memory information**

```
int ldc_mem_info(ldc_mem_handle_t mhdl,
                  ldc_mem_info_t *minfo);
    if successful, returns 0,
    else, returns
           EINVAL - invalid memory handle
```

Returns information about the memory handle to the caller. The *ldc_mem_info_t* has the following format:

```
typedef struct ldc_mem_info {
        uint8_t          mtype;      /* map type */
        uint8_t          perm;       /*  RWX  permissions
*/
        caddr_t          vaddr;      /* VA of memseg */
        caddr_t          raddr;      /* RA of memseg */
        ldc_mstatus_t    status;     /* handle status */
} ldc_mem_info_t;
typedef enum {
        LDC_UNBOUND,     /* Mem handle is unbound */
        LDC_BOUND,       /* Mem handle is bound */
        LDC_MAPPED       /* Mem handle is mapped */
} ldc_mstatus_t;
```

- **Channel Information**

```
int ldc_info(ldc_handle_t handle, ldc_info_t *info);
  if successful, returns 0
  else, returns
          EINVAL – invalid channel handle
```

```
Returns certain information about the channel. The ldc_info_t has
the following format:
typedef struct ldc_info {
      uint64_t    direct_map_size_max;
} ldc_info_t;
```

The 'direct_map_size_max' returned, is the size in bytes of the maximum amount of shared memory that can be direct mapped (LDC_DIRECT_MAP) for data transfers on the channel. This value serves as a hint to the client on the total amount of memory that can be directly mapped in. It is recommended that the client use this information, before invoking *ldc_mem_map()* to mapin remote memory exported by a peer. Requests to map more than the direct_map_size_max may fail. It cannot be assumed that a direct map request via *ldc_mem_map()* will be successful until it is invoked, as the availability of direct map space may have changed. The client should rely on *ldc_mem_info()* to know the actual map type after successfully mapping its data buffers using *ldc_mem_map()*.

Descriptor rings

Most virtual devices will use the shared memory capabilities of the LDC framework to send and receive data. Much like conventional IO devices, most virtual IO devices will use descriptor rings to keep track of all transactions being performed by the device. A simple descriptor ring framework layered on top of the shared memory framework described above is available as library service to sun4v virtual devices.

The LDC shared memory framework allows clients to allocate and share a ring with its peer over a LDC connection. A shared descriptor ring can then be mapped by a remote client into its address space. In the remote client cannot map the ring directly, it can mirror the descriptor ring with a shadow copy and use acquire/release semantics when accessing the ring.

A client can map a shared descriptor ring either directly into its address space or mirror it via a shadow copy. The shared access type specified at the time of descriptor ring binding determines whether shadow memory is allocated during a map. In the case the descriptor ring is mirrored and a shadow memory is allocated, explicit calls to acquire and release is necessary. This ensures that shadow copy contents are synchronized when descriptor contents are modified.

NOTE: Because descriptor rings can be directly accessed and modified by a remote client in another OS instance, it is recommended that the owning client does not rely on the contents of the ring for its correct operation. The fields in the ring should only be a subset copy of fields in a local desc ring, and the client should only depend on the contents of the private descriptor fields.

- **Descriptor ring create, (un)bind and destroy**

  When a client wants to allocate a descriptor ring it will make a call to *ldc_mem_dring_create*() with the ring length and the size of the descriptors. The client will then need to bind the descriptor ring to a specific LDC connection and also specify access permissions for the descriptor ring. On return, the API will provide the client a descriptor ring handle.

```
int ldc_mem_dring_create(uint64_t len, size_t dsize,
     ldc_dring_handle_t *dhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid type/perm
           ENOMEM - no memory for descriptor ring
```

  ldc_mem_dring_create() allocates a descriptor ring of the length '*len*' with descriptor size '*dsize*'.

  NOTE: The descriptor size have to be 8-byte aligned and the descriptor ring has to be greater than or equal to 8KB and a power-of-two.

  To share a ring with a peer. the client can bind the ring to a specific LDC connection using the *ldc_mem_dring_bind()* call. The bind call will return back a cookie that can be exchanged with its peer over the LDC connection.

```
int ldc_mem_dring_bind(ldc_handle_t hdl,
                       ldc_dring_handle_t dhdl,
                       uint8_t mtype, uint8_t perm,
                       ldc_mem_cookie_t *dcookie,
                       uint32_t *ccount);
```

```
if successful, returns 0,
else, returns
       EINVAL - invalid handle, or type/perm
       EINVAL - channel closed
```

The 'm*type*' field defines the type of the descriptor ring and had the following values:

```
LDC_SHADOW_MAP    0x1
LDC_DIRECT_MAP    0x2
```

When allocated with type `DIRECT_MAP`, the remote client can map the public descriptor ring into its address space. Alternatively, if the descriptor is created with type `SHADOW_MAP`, the remote client can copy data in and out via the Hypervisor into a shadow copy. The '*perm*' arg determines whether a shared descriptor ring is available for read or write access or both and accepts the following values:

```
LDC_MEM_R         0x1
LDC_MEM_W         0x2
LDC_MEM_RW        (LDC_MEM_R | LDC_MEM_W)
```

- The LDC memory cookie returned can be exchanged with the peer for either mapping directly or access via a shadow copy.

Similarly, a bound descriptor ring can be unbound using the *ldc_mem_dring_unbind()* call.

```
int ldc_mem_dring_unbind(ldc_dring_handle_t dhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid handle
           EACCES - not bound
```

A previously created descriptor ring can be freed by making a call to *ldc_mem_dring_destroy*().

```
int ldc_mem_dring_destroy(ldc_dring_handle_t dhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor handle
```

Before closing a channel, the shared descriptor rings associated with the channel have to be unbound. The client has to explicitly call *ldc_mem_dring_unbind()* to unbind the descriptor rings.

- **Descriptor ring mapping**

Shared descriptor rings can be either directly mapped or mirrored in the importer's address space using the *ldc_mem_dring_map* API. The call determines whether to map the ring or allocate memory for a shadow ring, using the descriptor ring's attributes and the

map type requested by the caller. If the shared descriptor ring was created for copy access, a new local descriptor ring is allocated and marked as the shadow copy of the remote shared ring. If the descriptor ring was created for direct map access, a call is made into the LDC framework to map the ring to the local address space. A previously mapped descriptor ring can be unmapped using the *ldc_mem_dring_unmap* call.

```
int ldc_mem_dring_map(ldc_handle_t handle,
                ldc_mem_cookie_t *cookie, uint32_t ccount,
                uint32_t len, uint32_t dsize, uint8_t mtype,
                ldc_dring_handle_t *dhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor cookie
           EFAULT - cannot map dring

int ldc_mem_dring_unmap(ldc_dring_handle_t dhdl);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor handle
```

In the event a channel associated with a descriptor ring is reset, the access to the mapped descriptor ring is no longer available. The ring owner will need to resend the cookie and request the remote side to remap the descriptor ring when the connection is renegotiated. If a channel is closed, the shared descriptor rings bound to the channel are unbound.

```
int ldc_mem_dring_nextcookie(ldc_dring_handle_t dhdl,
                        ldc_mem_cookie_t *cookie);
     if successful, returns 0,
     else, returns
           EINVAL - invalid descriptor handle
```

Return the next cookie associated with the specified desc ring handle.

- **Descriptor ring information**

  Once a descriptor ring has been created or mapped, clients can obtain the base address, type and access permissions of the descriptor ring using *ldc_mem_dring_info()*.

```
int ldc_mem_dring_info(ldc_dring_handle_t dhdl,
                           ldc_mem_info_t *minfo);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor handle
```

- **Descriptor ring memory access**

Descriptor rings can be either directly mapped or mirrored using a shadow descriptor ring. Like LDC shared memory, descriptor rings also follow entry consistency semantics. All access to a ring is bound by a *ldc_mem_dring_acquire()* and *ldc_mem_dring_release()* operations. This ensures that the contents of the mapped ring is consistent with the remote descriptor ring.

When a client updates its local descriptor ring it notifies the consumer (importing client) of the ring by sending it a message to indicate which entries were updated. The importing client will update its local copy by doing a *ldc_mem_dring_acquire* operation. This will either update the entire local ring or only update the descriptors indicated by the client.

```
int ldc_mem_dring_acquire(ldc_dring_handle_t dhdl,
                uint64_t start, uint64_t end);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor handle
           EINVAL - index is out of range
           EPERM - cannot read remote descriptor ring
           EACCES - descriptor ring is no longer mapped
```

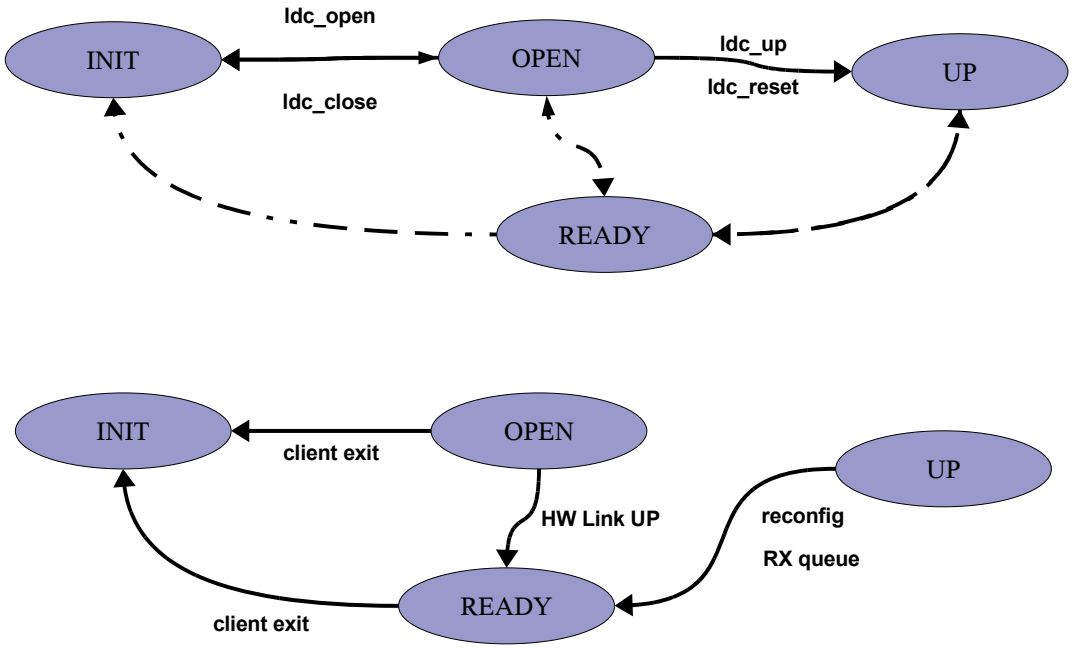
When a client wants to update the remote ring with contents from its local shadow copy it does so with a call to *ldc_mem_dring_release* operation. The operation will copy the range of descriptors bound by the *start* and *end* index from the local to the remote ring.

```
int ldc_mem_dring_release(ldc_dring_handle_t dhdl,
                uint64_t start, uint64_t end);
    if successful, returns 0,
    else, returns
           EINVAL - invalid descriptor handle
           EINVAL - index is out of range
           EPERM - cannot write remote descriptor ring
           EACCES - descriptor ring is no longer mapped
```

NOTE: The acquire and release operations in the case of directly mapped descriptor ring ensure that the remote/local CPU has completed all pending stores via a membar operation.

2.2.3 Virtual Link layer state transition

The figure below show the transition between various states as clients call into the virtual link layer using the interfaces listed below. The states are specified in uppercase. The figure below also shows the transition between various states as a result of asynchronous events or errors. Clients get notified via their interrupt handler.



- **INIT:** Channel is initialized, client has not opened channel. No queues have been registered with the Hypervisor. No callbacks have been registered.
- **OPEN**: Client has opened channel. Internal queues, interrupt number and target CPU have been registered with the Hypervisor. Initial handshake has not occured.
- **READY**: Intermediate state between OPEN and UP. Channel transitions to a READY state when the peer has registered Rx queues with the Hypervisor. LDC framework cannot bring the channel UP unless the state transitions to READY state.
- **UP**: Channel is up and initial handshake has happened. Data transmission can now occur.

2.3 Interface classification

All interfaces specified in this document are considered to be '**Consolidation Private**'

2.4 References

- Project and Architecture docs are available at:

  http://cpubringup.sfbay.sun.com/twiki/bin/view/LDoms/WebHome
- FWARC/2005/633 - LDoms: Project Q Logial Domaining Umbrella
- FWARC/2005/739 - sun4v channelsLDoms