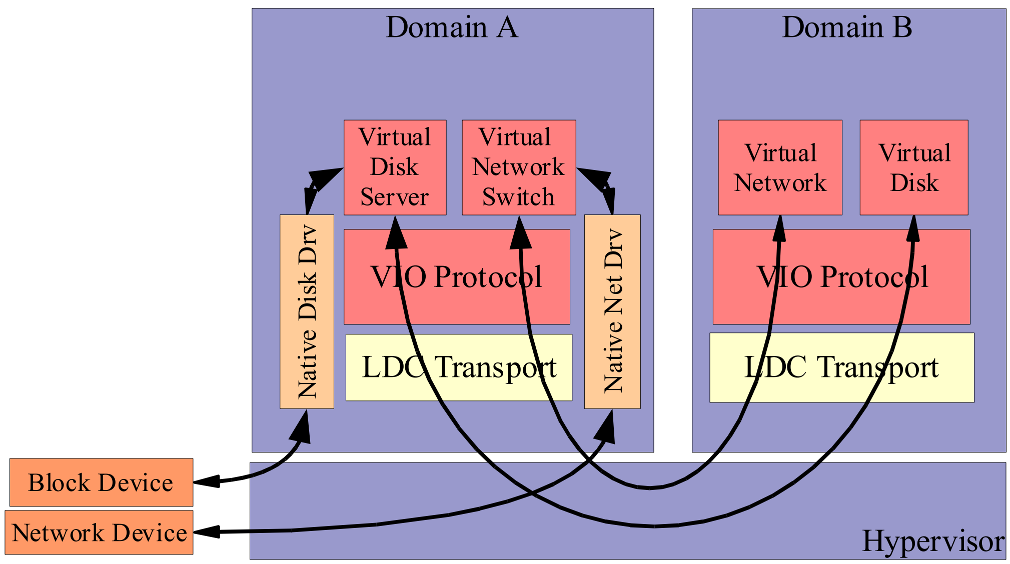
1.1 Virtual IO communication protocol

Virtual devices, clients and/or services, at the most basic level rely on the underlying Hypervisor LDC framework (FWARC/2005/733) and LDC transport layer (FWARC/2006/140) to transfer data. Since both these layers only provide a basic communication mechanism, VIO devices will first go through a basic handshake procedure to agree on transmission properties for the channel, before meaningful data can be exchanged between the two channel endpoints. As part of the handshake they will negotiate a common version, device attributes, data transfer type, and if necessary shared memory descriptor ring information. Following a successful handshake, the devices can send and receive data. All VIO devices use the LDC *unreliable* transport mode for all communication.

The figure below shows two logical domains with VIO device clients and services communicating with each other using the VIO protocol and layered on top of the underlying LDC framework. Domain A has exclusive access to local physical devices through native device drivers and exports access to these devices over the LDC connection to domain B.



1.1.1 VIO data transfer

VIO devices will transfer data either using packet mode by storing the data in LDC datagrams or sharing the data using the shared memory capability of the Hypervisor. A VIO device that uses packet mode, will use either a single LDC datagram packet or use the fragmentation-reassembly capabilities of the LDC transport layer to packetize and transfer larger messages. The Hypervisor shared memory support allows guests to share memory regions in their address space with another guest at the other end of a channel (FWARC/2006/184). This capability allows VIO client drivers to share segments of memory with a VIO client or service so that data can be transferred efficiently and much faster, instead of transferring data over the channel by packetizing each transfer.

Like conventional IO devices, the virtual IO devices that use the Hypervisor shared memory infrastructure for data transfer, will setup and use descriptor rings. The descriptor ring is a contiguous circular ring buffer that IO devices use to queue requests, receive responses and transfer associated data. VIO devices that use shared memory will either share their descriptor rings or send the descriptors as in-band messages. The subsequent sections describe the content of control and data packets, the transfer protocol and the structure of the descriptor rings used by VIO devices. It also specifies the device specific content of the LDC packets and descriptors for virtual network and disk devices.

1.1.2 VIO device message tag

All packets exchanged by VIO devices over a channel will use a common message tag as the header for the message. The message tag uniquely identifies the session, the type and subtype of the message. The subtype envelope contains message specific meta-data. All packets sent/received by VIO devices will specify all message tag fields and no field is optional. The format of the message tag along with values for the *type*, *subtype* and *subtype_env* fields are shown below:

```
6                           3 3               1 1
3                           2 1               6 5       8 7      0
+---------------------------+---------------+-------+-------+
|            SID            |   STYPE_ENV   | STYPE |  TYPE |
+---------------------------+---------------+-------+-------+

 Messages Types:                              Sub-Message Types:
 VIO_TYPE_CTRL  0x01                  VIO_SUBTYPE_INFO      0x01
 VIO_TYPE_DATA  0x02                  VIO_SUBTYPE_ACK       0x02
 VIO_TYPE_ERR   0x04                  VIO_SUBTYPE_NACK      0x04

        Sub-Type Envelope :
        if type = VIO_TYPE_CTRL (0x0000 - 0x003f)
               VIO_VER_INFO           0x0001
               VIO_ATTR_INFO          0x0002
               VIO_DRING_REG          0x0003
               VIO_DRING_UNREG        0x0004
               VIO_RDX                0x0005
               (reserved)             0x0006 – 0x003f

        if type = VIO_TYPE_DATA (0x0040 - 0x007f)
               VIO_PKT_DATA           0x0040
               VIO_DESC_DATA          0x0041
               VIO_DRING_DATA         0x0042
               (reserved)             0x0043 – 0x007f

        if type = VIO_TYPE_ERR (0x0080 - 0x00ff)
               (reserved)             0x0080 – 0x00ff

        device class specific sub-type envelopes
               VNET_xxx                      0x0100 - 0x01ff
               VDSK_xxx                      0x0200 - 0x02ff
               (reserved)                    0x0300 - 0xffff
```

1.1.3 VIO device peer-to-peer handshake

For VIO devices, both the server and/or client has to successfully complete a handshake before data transfer can commence. The handshake can be initiated by either parties. In the description below each message sent or received is specified using the format *<type>*/ *<subtype>* / *<subtype_env>*.

1.1.3.1 Version negotiation

A handshake is initiated by one peer sending a CTRL/INFO/VER_INFO to the other endpoint. This message consists of a '*dev_class'* field identifying the type of the sending device, and a *'major/minor'* pair which specify the protocol version (the protocol version will determine the type and amount of data that will be expected to be exchanged in later phases of the handshake). It also sets the session ID (*sid)* to a random value by setting it to the lower 32-bits of the CPU tick. The client will send a new session ID with each version negotiation request. The session ID corresponding to the accepted version gets used as part of each message sent as part of the session.

If the device class is recognized and the version major/minor numbers are acceptable then the receiving endpoint responds back with a CTRL/ACK/VER_INFO message leaving all the parameters unchanged. It also stores the sender's SID for use in future message exchanges.

If the major version is not supported, then the peer sends back a CTRL/NACK/VER_INFO message containing the next lower major version it supports. If it does not support any lower major numbers, it will NACK with the version major and minor values set to zero. The initiating endpoint can then if it wishes send another CRTL/INFO/VER_INFO message either with the major number it received from its peer, if it is acceptable, or with its next lower choice of version. If the major version is supported but not at the specified minor version level, the receiver will ACK back with a lower supported minor version number.

Similarly, if the '*dev_class*' is unrecognized, the receiver will respond back with CTRL/NACK/VER_INFO with the parameters unchanged and the handshake is deemed to have failed. The format of the version exchange packet to shown below:

```
          6                         3        3 3              1 1
          3                         9        2 1              6 5     8 7      0
          +----------------------------+---------------+-------+---------+
word 1:   |           SID              |   VER_INFO    | I/A/N |TYPE_CTRL|
          +-----------------+---------+---------------+-------+---------+
word 2:   |       rsvd      |DEV_CLASS|     MINOR     |      MAJOR      |
          +-----------------+---------+---------------+-----------------+
```

The currently supported devices types are listed below:

VDEV_NETWORK	0x1
VDEV_NETWORK_SWITCH	0x2
VDEV_DISK	0x3
VDEV_DISK_SERVER	0x4

NOTE: Irrespective of what state the receiving endpoint believes the channel to be in, receipt of a CTRL/INFO/VER_INFO message at any time will cause the endpoint to reset any internal state it may be maintaining for that channel and restart the handshake.

1.1.3.2 Attribute exchange

Following the initial version negotiation phase, VIO device clients/services will exchange device specific attribute information, depending on the device class and the agreed upon API version. Each attribute information packet is of the type CTRL/INFO/ATTR_INFO and contains parameters like transfer mode, maximum transfer size, and other device specific attributes. A ACK response is an acknowledgment by the peer that it will use these attributes in future transfer. A NACK response is an indication of mismatched attributes. It is up to the particular device class whether it restarts the handshake or exchanges other attributes. The device specific section for virtual disk and network devices contains more information about the exchanged attributes.

```
            6                           3 3             1 1
            3                           2 1             6 5     8 7          0
            +----------------------------+--------------+-------+-----------+
  word 1:   |           SID              |  ATTR_INFO   | I/A/N | TYPE_CTRL |
            +----------------------------+--------------+-------+-----------+
word 2-7:   |                   (device specific attributes)                |
            +---------------------------------------------------------------+
```

1.1.3.3 Descriptor ring registration

Most virtual devices will use the shared memory capabilities of the Hypervisor LDC framework to send and receive data. Like conventional IO devices, the virtual IO devices will use descriptor rings to keep track of all transactions being performed by the device. Prior to using a descriptor ring, and following version negotiation, and other device specific attribute exchange, VIO clients will register shared descriptor ring information with its channel peer.

```
              6                                3 3          1 1
              3                                2 1          6 5     8 7        0
              +--------------------------------+-----------+-------+----------+
   word 1:    |              SID               | DRING_REG | I/A/N |TYPE_CTRL |
              +--------------------------------+-----------+-------+----------+
   word 2:    |                         DRING_IDENT                           |
              +--------------------------------+------------------------------+
   word 3:    |         DESCRIPTOR_SIZE        |        NUM_DESCRIPTORS       |
              +--------------------------------+-----------+------------------+
   word 4:    |            NCOOKIES            | reserved  |     OPTIONS      |
              +--------------------------------+-----------+------------------+
 word 5-n:    |              (LDC_TRANSPORT_COOKIE * NCOOKIES)                |
              +---------------------------------------------------------------+
```

A VIO client will register a descriptor ring by sending a CTRL/INFO/DRING_REG message to its peer. The message will contain information about the number of descriptors in the ring, the descriptor size, the LDC transport cookie(s) associated with the descriptor ring memory and the number of cookies. The *options* field allows certain VIO clients to specify descriptor ring properties that describe its intended use. The supported values in v1.0 of the VIO protocol are:

```
VIO_TX_DRING          0x1   /* Tx descriptor ring */
VIO_RX_DRING          0x2   /* Rx descriptor ring */
```

On receiving the registration message, the receiver will ACK the message, and in the ACK provide the sender an unique *dring_ident.* The *dring_ident* will be used by the sender to either unregister the ring or refer to the descriptor ring during data transfer. A NACK to this message from the receiving end is regarded as a fatal error and the entire session is deemed to have failed and a new session has to be established by re-initiating a handshake. The *dring_ident* field is not used in the registration message and only used during the ACK.

- ***LDC transport cookie:***

A LDC transport cookie (*LDC_TRANSPORT_COOKIE)* is 16-bytes in size and consists of *cookie_addr* and *cookie_size* fields. The *cookie_addr* field corresponds to the Hypervisor LDC shared memory cookie for each page (see FWARC/2006/184) and the *cookie_size* corresponds to the actual number of bytes that is shared within the page pointed to by the cookie. If the descriptor ring memory segment spans multiple pages, an unique transport cookie is used to refer to each page within the segment. The format of the LDC transport cookie is shown below:

```
6
3                                                              0
+--------------------------------------------------------------+
|            HV shared memory cookie (cookie_addr)             |
+--------------------------------------------------------------+
|                         cookie_size                          |
+--------------------------------------------------------------+
```

When two or more successive pages in the descriptor ring memory segment are stored in consecutive entries in the LDC map table, a single transport cookie can be used refer to all these page entries. The *cookie_addr* in this case will still point to first page in the set, but the *cookie_size* will correspond to the size spanning all consecutive entries.

A VIO device might typically share multiple descriptor rings with its peer and can choose to register all descriptor rings with its peer at the time of the initial handshake or at any point after data transfer has commenced. If a device intends to do all its data transfer using descriptor rings, it will have to register at least one descriptor ring before data transfer can commence.

A VIO client can unregister a descriptor ring by sending a CTRL/INFO/DRING_UNREG message to its peer. It will specify the *dring_ident* it received from the peer at the time of registration. The peer will ACK a successful unregister request and NACK the request if the *dring_ident* specified is invalid. If subsequent data transfers refer to an unregistered descriptor ring, the DRING_DATA requests will be NACKd.

```
          6                           3 3             1 1
          3                           2 1             6 5     8 7        0
          +---------------------------+---------------+-------+----------+
 word 1:  |          SID              |  DRING_UNREG  | I/A/N |TYPE_CTRL |
          +---------------------------+---------------+-------+----------+
 word 2:  |                        DRING_IDENT                           |
          +--------------------------------------------------------------+
```

1.1.3.4 Handshake completion

After successful completion of all negotiations and required information exchange, an endpoint will send a RDX message to its peer to indicate that it can now receive data from it. An endpoint initiates this by sending a CTRL/INFO/RDX message to the receiving end. The receiver acknowledges the message by sending CTRL/ACK/RDX. Because LDC connections are duplex, each endpoint has to send a RDX message to its peer before data transfer can commence in both directions. When a RDX is sent by an endpoint, the endpoint is explicitly enabling a simplex communication path, whereby it announces that it can now receive data from its peer. It is VIO device specific whether they require the establishment of a duplex connection before data transfer can commence. There is no payload associated with a RDX message and they are not NACKed.

```
          6                           3 3        1 1
          3                           2 1        6 5        8 7          0
          +---------------------------+----------+----------+------------+
 word 1:  |          SID              |   RDX    | INFO/ACK |  TYPE_CTRL |
          +---------------------------+----------+----------+------------+
```

Once the channel has been established (indicated by the receipt of a RDX message) in either simplex or duplex mode further informational messages may be sent by the initiating endpoint or requested by the receiving endpoint as time goes by. The content and effect these messages have on the session is device specific. These messages are also regarded as in-band notifications.

1.1.4 VIO data transfer modes

VIO devices can send data to their peers over a channel using different transfer modes. During the handshake, each device will specify to its peer the transfer mode (*xfer_mode*) it intends to use as part of the attribute info message. The device specific attribute message format specifies the location of the *xfer_mode* field in the message. The supported transfer modes in versions 1.0 and 1.1 of the VIO protocol are:

```
VIO_PKT_MODE        0x1   /* packet based transfer */
VIO_DESC_MODE       0x2   /* in-band descriptors */
VIO_DRING_MODE      0x3   /* descriptor rings */
```

In version 1.2, the VIO protocol will allow concurrent use of the different transfer modes, specifically packet based transfer and descriptor ring modes. In order to do this, the *xfer_mode* field in the attribute info message will be changed to a bit mask with the following values:

```
VIO_PKT_MODE        0x1   /* packet based transfer */
VIO_DESC_MODE       0x2   /* in-band descriptors */
VIO_DRING_MODE      0x4   /* descriptor rings */
```

In version 1.2, the virtual network and switch clients will use the packet transfer mode in addition to the descriptor ring mode (xfer_mode=0x5) to send high priority ethernet frames as data packets for faster out-of-band processing.

1.1.4.1 Packet based transfer

```
             6                         3 3          1 1
             3                         2 1          6 5     8 7      0
             +-------------------------+------------+-------+----------+
   word 1:   |          SID            |  PKT_DATA  | I/A/N |TYPE_DATA |
             +-------------------------+------------+-------+----------+
   word 2:   |                       SEQ_NO                             |
             +----------------------------------------------------------+
 word 3-7:   |                    DATA_PAYLOAD                          |
             +----------------------------------------------------------+
```

As discussed in the earlier section, VIO packets always consist of a generic message tag header and a sequence id (which is incremented with each packet sent). Additionally, if a VIO device intends to use packet mode for sending data, it can use up to 40 bytes of a LDC datagram without using LDC transport's packet fragmentation capability. Larger transfers will require the use of the fragmentation-reassembly support provided by the underlying LDC transport. The format of a LDC packet containing data is shown above.

1.1.4.2 Descriptor rings

As mentioned in the earlier section, a descriptor ring is a contiguous circular ring buffer VIO devices use to queue requests, receive responses and transfer associated data. Each descriptor in the ring holds request and response parameters specific to the particular device along with opaque cookies that point to the page(s) of memory that are being shared for reading and/or writing. The descriptor ring will utilize Hypervisor shared memory support, so that clients at both ends of the channel can modify the contents of the descriptor(s).

Each VIO client will specify that it intends to use descriptor rings, as part of the attribute info exchange. It will also specify whether or not it intends to share the descriptors using shared memory or send each descriptor as an in-band message. If it shares the descriptor ring using shared memory, it will register at least one descriptor ring with its peer at the other end.

Each entry in a descriptor ring consists of a common descriptor ring entry header and the descriptor payload as shown in the figure below. The descriptor payload consists of fields that are device class specific and are discussed in more detail in sec 1.1.6 and 1.1.7.

```
 6
 3                                        9  8   7          0
+----------------------------------------+---+-----------+
|                reserved                | A |   DSTATE  |
+----------------------------------------+---+-----------+
|                (descriptor payload)                    |
+--------------------------------------------------------+
```

The descriptor *dstate* specifies the state of the the descriptor. The valid state values are:

```
VIO_DESC_FREE         0x1
VIO_DESC_READY        0x2
VIO_DESC_ACCEPTED     0X3
VIO_DESC_DONE         0x4
```

Initially when a descriptor ring is allocated, all entries in the ring are marked with value of VIO_DESC_FREE. When a client queues one or more requests, it will change the flags value for the corresponding descriptor(s) to VIO_DESC_READY. It will then send a message to its peer requesting it to process the descriptors. The client that is processing the descriptor will first change the state to VIO_DESC_ACCEPTED, acknowledging receipt of the request and prior to processing the request. On completing the request, it will update the descriptor with its response and change the value of the flag to VIO_DESC_DONE. The client that initiated the request, will take the appropriate action after seeing the request as been marked as VIO_DESC_DONE and then change it to VIO_DESC_FREE. If the state of a descriptor transitions to an unexpected state, the behavior is undefined. A VIO device under these circumstances, might either reset the session and restart the handshake, or send an error message to its peer.

When the requesting client updates one or more descriptors and marks them as ready for processing, it will send a DATA/INFO/DRING_DATA message to its peer at the other end of the channel. The message will contain the *dring_ident* the requester received at the time of registering the descriptor ring. It also specifies the start and end index corresponding to the descriptors that have been updated. If *end* index value specified is -1, the receiver will process all descriptors starting with the *start* index and continue until it does not find a descriptor marked VIO_DESC_READY. The receiver at this point will send an implicit ACK to the sender to let it know that it is done processing all requests. Subsequently, if the sender marks additional entries as VIO_DESC_READY, it will re-initiate processing by sending another DRING_DATA request.

If the start and end index, either overlap with requests sent earlier or correspond to descriptors not in `VIO_DESC_READY` state, the request will be NACKed by the receiver.

```
           6                         3 3          1 1
           3                         2 1          6 5     8 7          0
           +-------------------------+------------+-------+------------+
  word 1:  |          SID            | DRING_DATA | I/A/N | TYPE_DATA  |
           +-------------------------+------------+-------+------------+
  word 2:  |                       SEQ_NO                              |
           +-----------------------------------------------------------+
  word 3:  |                     DRING_IDENT                           |
           +-------------------------+---------------------------------+
  word 4:  |        END_IDX          |            START_IDX            |
           +-------------------------+--------------------+------------+
  word 5:  |                 reserved                     | PROC_STATE |
           +----------------------------------------------+------------+
```

The requester can also request an explicit acknowledgment from the client processing the request (to track progress) by setting the *(A)cknowledge* field in the descriptor. The client, after processing the descriptor (changes state as `VIO_DESC_DONE`), will send a DATA/ACK/DRING_DATA message with the *dring_ident* for this descriptor ring and *end_idx* equal to this descriptor.

When the requester sends requests with an *end_idx* = -1, the *proc_state* field in the ACK/NACK message, is used by the receiver to indicate its current processing state. The valid *proc_state* field values are:

```
VIO_DP_ACTIVE        0x1   /* active processing req */
VIO_DP_STOPPED       0x2   /* stopped processing req */
```

If the receiver continues to process requests or is waiting for more descriptors to be marked VIO_DESC_READY, it will ACK with *proc_state* set to `VIO_DP_ACTIVE`. Instead, if the receiver stops after processing the last ACK/NACK, and is waiting for an explicit DATA/INFO/DRING_DATA message, it will set the *proc_state* set to `VIO_DP_STOPPED`. The *proc_state* value is then used by the requester to determine when the receiver's state, and accordingly sends an explicit DRING_DATA message when more requests are queued.

It is not always necessary that clients need to register a shared descriptor ring to make use of the HV shared memory infrastructure. A simpler client can still use the shared memory capabilities and instead of sharing the descriptor ring, it will send the descriptor itself as in-band data. The `DESC_HANDLE` in the pkt is an opaque handle that corresponds to the descriptor in the sender's ring. The content of the in-band descriptor packet is shown below:

```
            6                              3 3           1 1
            3                              2 1           6 5     8 7      0
           +-----------------------------+------------+-------+----------+
  word 1:  |            SID              |  DESC_DATA | I/A/N |TYPE_DATA |
           +-----------------------------+------------+-------+----------+
  word 2:  |                           SEQ_NO                            |
           +-------------------------------------------------------------+
  word 3:  |                         DESC_HANDLE                         |
           +-------------------------------------------------------------+
           |                    (descriptor payload)                     |
           +-------------------------------------------------------------+
```

In case of both a DRING_DATA and DESC_DATA message, if the receiver gets a data packet out of order (as indicated by a non-consecutive sequence number) then it will NACK the packet and will not process any further data packets from this client. If there are no errors the receiver will ACK the receipt of descriptor ring or descriptor data packets if there is an explicit request by the sender to ACK a data packet by setting the (*A)cknowledge* bit in the descriptor.

Implementation Note: Upon receipt of a NACK, the sending client can either try to recover or stop sending data and return to initial state and restart the channel negotiation again.

1.1.5 Virtual IO Dynamic Device Service (DDS)

Virtual IO devices following the initial handshake, send and receive data using the packet and/or descriptor based modes as described in the earlier sections. This forms the under pinnings of the virtual IO data transfer infrastructure in a LDoms environment. While compelling for a variety of application workloads, virtualized I/O still does not provide high performance I/O capabilities that certain I/O oriented workloads require. The Hybrid I/O model provides the opportunity to share device resources across multiple client domains with better granularity while overcoming the performance bottlenecks of virtualized I/O.

A new control message type will be added in VIO protocol versions 1.3 and higher to support the Hybrid IO model. The new Dynamic Device Service (DDS) control message, with a subtype envelope value of VIO_DDS_INFO, will provide virtual IO devices and services the ability to exchange and share physical device resource information with their peers.

```
VIO_DDS_INFO       0x6         /* DDS information */
```

Each DDS control message will allow a device to share or reclaim a resource, or change the properties of a resource. A peer on receiving a CTRL/INFO/DDS_INFO message, will take necessary action and then either ACK or NACK the message depending on whether the requested operation was successful or not.

Each VIO_DDS_INFO message, in addition to the VIO msg header, includes a DDS message header consisting of a DDS *class*, *subclass*, and *request_id* fields. Though the format of the DDS message header itself is generic to the VIO protocol, the DDS message class and sub-class values are specified by the virtual network or disk devices. The DDS request ID in the header will used to correlate the INFO requests with ACK and NACK responses. The DDS msg format is shown below:

```
              6                             3 3           1 1
              3                             2 1           6 5      8 7        0
             +-----------------------------+------------+--------+----------+
    word 1:  |            SID              |  DDS_INFO  | I/A/N  |TYPE_CTRL |
             +-----------------------------+------------+--------+----------+
    word 2:  |      DDS_REQUEST_ID         |  reserved  |SUBCLASS|   CLASS  |
             +-----------------------------+---------------------+----------+
  word 3-7:  |                   (dds message payload)                      |
             +--------------------------------------------------------------+
```

Device specific class and subclass values, including contents of the DDS message is discussed in section 1.1.7.5. The class value ranges reserved for various VIO device classes is specified below:

```
DDS_GENERIC_XXX    0x0 - 0xf        /* Generic DDS class */
DDS_VNET_XXX       0x10 – 0x1f      /* vNet DDS class */
DDS_VDSK_XXX       0x20 – 0x2f      /* vDisk DDS class */
reserved           0x30 – 0xff      /* reserved */
```

1.1.6 Virtual Disk specific data

In the protocol outlined above, the attribute exchange and descriptor payload contents are undefined and left to be specified by the VIO devices. This section describes the contents of these packets for use by both the virtual disk client and server to exchange data. The vDisk client, following an attribute exchange, will send to the server block disk read and write requests, in addition to disk control requests. The server will export each block device over an unique channel, and accept requests from the client, once a session has been established.

1.1.6.1 Attribute information

During the initial handshake, as part of the CTRL/INFO/ ATTR_INFO message, the virtual disk server and client exchange information about the transfer protocol and the physical device itself. The format of the attribute contents is shown below:

```
            6                        3 3    2 2        1 1
            3                        2 1    4 3        6 5     8 7      0
            +------------------------+------+----------+-------+---------+
   word 1:  |          SID           |     ATTR_INFO   | I/A/N |TYPE_CTRL|
            +------------------------+------+----------+-------+---------+
   word 2:  |    VDISK_BLOCK_SIZE    | rsvd | VD_MTYPE |VD_TYPE|XFER_MODE|
            +------------------------+------+----------+-------+---------+
   word 3:  |                         OPERATIONS                         |
            +------------------------------------------------------------+
   word 4:  |                         VDISK_SIZE                         |
            +------------------------------------------------------------+
   word 5:  |                         MAX_XFER_SZ                        |
            +------------------------------------------------------------+
```

The vDisk client will provide the server with the transfer mode (*xfer_mode)* and the requested maximum transfer size (*max_xfer_sz)* it intends to use for sending disk requests to the server.

The *vdisk_block_size* is specified in bytes. The vdisk_size and max_xfer_sz are specified in multiples of the vdisk_block size.

For version 1.0 of the vDisk protocol the client's request must set *vdisk_block_size* to the minimum block size the client wishes to handle, and specify the *max_xfer_size*. If the server cannot support the requested *vdisk_block_size* or *max_xfer_sz* requested by the client, but can support a lower size, it will specify its *vdisk_block_size* and/or a lower *max_xfer_sz* in its ACK. If the client has no minimum block size requirement it may use the value of 0 as its requested vdisk_block_size, in this case the *max_xfer_size* in the client's attribute request to the server is interpreted as being specified in bytes. Either client or server may simply reset the LDC connection if they fail to agree on communication attributes.

For version 1.1 of the vDisk protocol, the vDisk server can set *vdisk_size* to -1 if it can not obtain the size at the time of the handshake. This can happen when the underlying disk has been reserved by another system. Under these circumstances, the vDisk client can retrieve the size at a later time, after the completion of the handshake, using the VD_OP_GET_CAPACITY operation.

If either client or server cannot support the specified transfer mode, the connection will be reset and the handshake may be restarted. The server in its ACK message will also provide the vdisk type (*vd_type*), *vdisk_block_size and vdisk_size* to the client. The supported types are:

```
VD_DISK_TYPE_SLICE   0x1   /* slice in blk device */
VD_DISK_TYPE_DISK    0x2   /* entire blk device */
```

All other disk types are reserved and for version 1.0 of the vdisk protocol should be considered as an error.

Only in protocol versions 1.1 and higher of the vdisk protocol, the server in its ACK message will provide the client the *vdisk_size* (specified as a multiple of the block size), and the vdisk media type (*vdisk_mtype*). The supported vdisk media types are:

```
VD_MEDIA_TYPE_FIXED  0x1   /* Fixed device */
VD_MEDIA_TYPE_CD     0x2   /* CD device */
VD_MEDIA_TYPE_DVD    0x3   /* DVD device */
```

All other disk media types are reserved and for version 1.1 of the vdisk protocol should be considered as an error.

Both these fields are *reserved* and not available in version 1.0 of the vdisk protocol. Clients should use the disk geometry information (see section 1.1.5.11) to compute the vdisk size.

The *operations* field is a bit-mask specifying all the disk operations supported by the server, where each bit position, if set, corresponds to the operation command supported by the server. The list of supported operations encodings is described in section 1.1.6.2.

1.1.6.2 vDisk descriptors

Virtual disk clients will send their disk requests by queueing them in descriptors as part of a shared descriptor ring.

As requests are initiated only by the client, and the buffers pointed to by each descriptor are used for both writing and reading disk blocks, the vDisk client will register the descriptor ring as both a Tx and Rx ring. In the case of descriptor rings that are not shared, the virtual disk client will send the requests as in-band descriptor messages.

The descriptor payload is formatted as follows:

```
 6                             3 3   2 2   1 1
 3                             2 1   4 3   6 5     8 7       0
+-----------------------------+-----+-----+-------+---------+
|                          REQ_ID                           |
+-----------------------------+-----------+-------+---------+
|           STATUS            | reserved  | SLICE |OPERATION|
+-----------------------------+-----------+-------+---------+
|                          OFFSET                           |
+-----------------------------------------------------------+
|                           SIZE                            |
+-----------------------------+-----------------------------+
|          reserved           |          NCOOKIES           |
+-----------------------------+-----------------------------+
|                  LDC_COOKIE * NCOOKIES                    |
+-----------------------------------------------------------+
```

The payload contains the *operation* being performed.

The *offset* field specifies the relative disk block address when doing a block read or write operation to the disk. This corresponds to the block offset from the start of the disk, or the disk slice as appropriate. It is specified in terms of the vdisk_block_size received from the server.

The *size* field specifies the number of blocks being read or written when doing a

VD_OP_BREAD or VD_OP_BWRITE operation. In the case where the *vdisk_block_size* in the client's attribute request is zero the *size* is interpreted as being specified in bytes.

For each client request sent to the server, the server will process the descriptor contents and submit the request to the device. Each virtual disk request is identified by an unique *req_id*. The *operation field* specifies the operation being done on the device. The server will then return the status of the operation in the same descriptor but with the '*status*' field containing the outcome of the operation. The supported values in version 1.0 of the vdisk protocol are:

```
VD_OP_BREAD          0x01         /* Block Read */
VD_OP_BWRITE         0x02         /* Block Write */
VD_OP_FLUSH          0x03         /* Flush disk contents */
VD_OP_GET_WCE        0x04         /* Get W$ status */
VD_OP_SET_WCE        0x05         /* Enable/Disable W$ */
VD_OP_GET_VTOC       0x06         /* Get VTOC */
VD_OP_SET_VTOC       0x07         /* Set VTOC */
VD_OP_GET_DISKGEOM   0X08         /* Get disk geometry */
VD_OP_SET_DISKGEOM   0x09         /* Set disk geometry */
VD_OP_GET_DEVID      0x0b         /* Get device ID */
VD_OP_GET_EFI        0x0c         /* Get EFI */
VD_OP_SET_EFI        0x0d         /* Set EFI */
VD_OP_xxx            0x0e - 0xff /* reserved for 1.0 */
```

In addition, the following values are supported in version 1.1 of the vDisk protocol:

```
VD_OP_SCSICMD        0x0a         /* SCSI control command */
VD_OP_RESET          0x0e         /* Reset disk */
VD_OP_GET_ACCESS     0x0f         /* Get disk access */
VD_OP_SET_ACCESS     0x10         /* Set disk access */
VD_OP_GET_CAPACITY   0x11         /* Get disk capacity */
VD_OP_xxx            0x12 - 0xff /* reserved for 1.1 */
```

As mentioned before, the vDisk server at the time of the initial attribute exchange will specify the bit mask of operations it supports. If the server does not support a required operation, it is up to the specific client implementation to decide whether it returns an error or internally implements the operation. All operations can be optionally implemented by a particular vDisk server implementation. If an operation is supported by the server, the outcome of the operation will be always available in the descriptor ring entry *status* field.

The *ncookies* and *ldc_cookie* fields refer to the segment of memory from/to which data is being read/written. See sec 1.1.3.3 for more information about the LDC transport cookie.

1.1.6.3 Disks and slices

A vdisk server may export either an entire disk device, or a simple slice (or partition) of a disk to a client as configured by the administrator. In the event that an entire disk is exported to a client, it is client policy as to how it determines the partitioning information or re-partitions that whole virtual disk.

To enable a server to potentially mount or examine a disk created by a client, the server may elect to offer the VD_OP_GET/SET_VTOC operations to its client. If the client elects to use these operations to retrieve partition information, the client when it reads or writes to the disk must specify the slice being accessed - in this case the offset field for those transactions is specified relative to the start of the referenced slice (not the start of the disk).

A client is not required to use the VTOC operations, and the server is not required to support them. In either of these events, if the client wishes to use the disk exported by the server it must read (and write - if re-partitioning) its own partition table at some client specific location on the disk.

Attempts to mix reads and writes with get and set VTOC operations to read/manipulate disk partition information have undefined results, and clients are required (though this may only be optionally enforced by the server) to use a consistent approach to discovering or modifying disk partition information.

The *slice* field is currently only used for VD_OP_BREAD and VD_OP_BWRITE. For all other operations it is ignored, and should be set to zero. If the disk served is of type `VD_DISK_TYPE_SLICE` the slice field is treated as reserved; i.e. must be set to zero, and ignored by the consumer. For a VD_DISK_TYPE_DISK the slice field refers to the disk slice or partition on which a specific operation is being done - the field only has meaning for disk servers that export a GET_VTOC service so that clients know which slice corresponds to which partition.

If the vDisk client does not use the VTOC service, it must specify a value of 0xff for the slice field for read and write transactions so that the server knows that the offset specified is the absolute offset relative to the start of a disk. Mixing read and write transactions to specific slices together with absolute disk transactions has undefined results, and clients must not do this. A client must close the disk channel and re-negotiate the vDisk service if it wishes to switch between using slice based access (explicitly passing the value of the *slice* being accessed) and absolute access (where *slice* is 0xff) when the server offers a disk type of VD_DISK_TYPE_DISK.

1.1.6.4 VDisk Block Read command (VD_OP_BREAD)

This command performs a basic read of a block from the device service. The decriptor ring entry for this command contains the offset and number of blocks to read together with the LDC cookies for the data buffers.

Once completed the status field in the descriptor is updated with the completion status of the operation.

1.1.6.5 VDisk Block Write command (VD_OP_BWRITE)

This command performs a basic write of a block from the device service. The decriptor ring entry for this command contains the offset and number of blocks to write together with the LDC cookies for the data buffers.

Once completed the status field in the descriptor is updated with the completion status of the operation.

1.1.6.6 VDisk Flush command (VD_OP_FLUSH)

This command performs a barrier and synchronisation operation with the disk service. There are no additional parameters in the decriptor entry for this command.

Before completing this command, the disk service will ensure that all previously executed write operations are flushed to their respective disk devices, and all previously executed reads are completed and their data returned to the client.

1.1.6.7 VDisk Get Write Cache enablement status (VD_OP_GET_WCE)

This command is used by a virtual disk client to query whether write-caching has been enabled on the disk being exported by the vDisk server. The payload is a single 32 bit unsigned integer. A value of 0 means write caching is not enabled, a value of 1 means write-caching is enabled (a flush operation should be used as a barrier to ensure writes are forced to non-volatile storage). All other values are reserved and have undefined meaning.

1.1.6.8 VDisk Enable/Disable Write Cache (VD_OP_SET_WCE)

This command is used a virtual disk client to enable or disable the write cache on the disk being exported by the vDisk server. The payload is a single 32 bit integer. A value of zero disables write-caching on the server side. A value of 1 enables write caching on the server side. All other values are reserved and are treated as errors by the vDisk server.

1.1.6.9 VDisk Get Volume Table of Contents (VD_OP_GET_VTOC)

This command is used to return information about the table of contents for the disk volume a client is attached to. The successful result of this command includes the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following header format:

```
           6                          3 3              1 1            0
           3                          2 1              6 5            0
           +-------------------------------------------------------------+
   word 0: |                        Volume name                          |
           +---------------------------+----------------+----------------+
   word 1: |         reserved          | num_partitions |  sector_size   |
           +---------------------------+----------------+----------------+
   word 2: |                        ASCII Label                          |
           +-------------------------------------------------------------+
   word 3: |                   ASCII Label continued                     |
           +-------------------------------------------------------------+
```

The volume name is an 8 character ASCII name for the volume.

The ASCII label is a 128 character ASCII label assigned to this disk volume. This is distinct from the actual volume name.

The field sector_size is the size in bytes of each sector of the disk volume.

The field num_partitions is the number of partitions on this disk volume. The header described above is immediately followed by the structure below repeated once for each of the number of partitions specified by the header:

```
             6                          3 3              1 1            0
             3                          2 1              6 5            0
             +---------------------------+----------------+--------------+
  word X+0:  |         reserved          |   perm flags   |ID tag of part|
             +---------------------------+----------------+--------------+
  word X+1:  |               start block number of partition               |
             +-------------------------------------------------------------+
  word X+2:  |                number of blocks in partition                |
             +-------------------------------------------------------------+
```

Reserved fields should be ignored.

1.1.6.10 VDisk Set Volume Table of Contents (VD_OP_SET_VTOC)

This command is used by a virtual disk client to set the table of contents for the disk volume the client is attached to.

The supplied data structure has the same format as for the get VTOC command (VD_OP_GET_VTOC). Reserved fields must be set to zero.

1.1.6.11 VDisk Get Disk Geometry (VD_OP_GET_DISKGEOM)

This command is used to return the geometry information about the disk volume a client is attached to. The successful result of this command includes the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following format:

Byte offset	*Size in bytes*	*Field name*	*Description*
0	2	ncyl	Number of data cylinders
2	2	acyl	Number of alternate cylinders
4	2	bcyl	Cylinder offset for fixed head area
6	2	nhead	Number of heads
8	2	nsect	Number of sectors
10	2	intrlv	Interleave factor
12	2	apc	Alternative sectors per cylinder (SCSI only)
14	2	rpm	Revolutions per minute
16	2	pcyl	Number of physical cylinders
18	2	write_reinstruct	Number of sectors to skip for writes
20	2	read_reinstruct	Number of sectors to skip for reads

1.1.6.12 VDisk Set Disk Geometry (VD_OP_SET_DISKGEOM)

This command is used by a virtual disk client to set the geometry information for the disk volume the client is attached to.

The supplied data structure has the same format as the get disk geometry command (VD_OP_GET_DISKGEOM).

1.1.6.13 VDisk SCSI Command (VD_OP_SCSICMD)

This command is used to deliver a SCSI packet to the vDisk server. It is implementation specific as to whether the server passes the received packet directly to a SCSI drive or whether it chooses to simulate the SCSI protocol itself. A server must not advertise this command if it does not support either capability.

The LDC cookie in the descriptor ring should point to the following data structure which describes the command arguments. The same buffer is also used to return the result of the command to the vDisk client.

```
             6              4 4      4 3      3 3      2 2      1 1      0 0      0
             3              8 7      0 9      2 1      4 3      6 5      8 7      0
             +---------------+--------+--------+--------+--------+--------+--------+
word 0:      |    TIMEOUT    |reserved|  CRN   | TPRIO  | TATTR  | SSTAT  | CSTAT  |
             +---------------+--------+--------+--------+--------+--------+--------+
word 1:      |                          OPTIONS                                      |
             +-----------------------------------------------------------------------+
word 2:      |                          CDB LENGTH                                   |
             +-----------------------------------------------------------------------+
word 3:      |                          SENSE LENGTH                                 |
             +-----------------------------------------------------------------------+
word 4:      |                          DATA-IN SIZE                                 |
             +-----------------------------------------------------------------------+
word 5:      |                          DATA-OUT SIZE                                |
             +-----------------------------------------------------------------------+
word 6:      |                          CDB DATA                                     |
  :          :                                                                       :
word I:      |                                                                       |
             +-----------------------------------------------------------------------+
word I+1     |                          SENSE DATA                                   |
  :          :                                                                       :
word J:      |                                                                       |
             +-----------------------------------------------------------------------+
word J+1:    |                          DATA-IN                                      |
  :          :                                                                       :
word K:      |                                                                       |
             +-----------------------------------------------------------------------+
word K+1:    |                          DATA-OUT                                     |
  :          :                                                                       :
word L:      |                                                                       |
             +-----------------------------------------------------------------------+
```

The *cstat* field reports to the vDisk client the SCSI command completion status. SCSI command completion status are described in the SCSI Architecture Model documents[3].

The *sstat* field reports to the vDisk client the SCSI command completion status of the SCSI sense request. SCSI command completion status are described in the SCSI Architecture Model documents[3]. The *sstat* field is defined only if a SCSI sense buffer was provided and if the SCSI command completion status indicates that sense data should be available.

The *tattr* field defines the task attribute of the SCSI command to execute. The possible attributes are:

- 0x00 no task attribute defined
- 0x01 SIMPLE
- 0x02 ORDERED
- 0x03 HEAD OF QUEUE
- 0x04 ACA

Task attributes are defined in the SCSI Architecture Model documents[3]. The vDisk server may ignore the task attribute.

The *tprio* field is a 4-bit value defining the task priority assigned to the SCSI command to execute. The task priority is defined in the SCSI Architecture Model documents[3]. The vDisk server may ignore the task priority.

The *crn* field is a command reference number (CRN). SCSI command reference numbers are defined in the SCSI Architecture Model documents[3]. The vDisk server may ignore the CRN.

The *reserved* field is reserved and should not be used.

The *timeout* field is the time in seconds that the vDisk server should allow for the completion of the command. If it is set to 0 then no timeout is required.

The *options* field is a bitmask specifying options for the SCSI command to execute. The possible bitmask values are:

- 0x01 (CRN)
  This bitmask indicates that a command reference number (CRN) is specified in the request.
- 0x02 (NORETRY)
  This bitmask indicates that the vDisk server should not attempt any retry or other recovery mechanisms if the SCSI command terminates abnormally in any way.

The *Command Descriptor Block (CDB) length* field is set by the vDisk client and indicates the number of bytes available in the *CDB* field.

The *sense length* field is initially set by the vDisk client and indicates the number of bytes available in the *sense* field for storing sense data for SCSI commands returning with a SCSI command completion status indicating that sense data should be available. After the execution of the SCSI command, the vDisk server sets the *sense length* field to the number of bytes effectively returned in the *sense* field, or 0 if no sense data were returned.

The *data-in size* field is initially set by the vDisk client and indicates the number of bytes available for data transfers to the *data-in* field. After the execution of the SCSI command, the vDisk server sets the *data-in size* field to the number of bytes effectively transfered to the *data-in* field, or 0 if no data were transfered.

The *data-out size* field is initially set by the vDisk client and indicates the number of bytes available for data transfers from the *data-out* field. After the execution of the SCSI command, the vDisk server sets the *data-out size* field to the number of bytes effectively transfered from the *data-out* field, or 0 if no data were transfered.

The *CDB* field contains the SCSI Command Descriptor Block (CDB) which defines the SCSI operation to be performed by the vDisk server. The structure of the CDB is part of the SCSI Standart Architecture[3]. The size of the *CDB* field should be equal to the number of bytes indicated by the vDisk client in the *CDB length* field rounded up to a multiple of 8 bytes.

The *sense* field contains sense data for SCSI commands returning with a SCSI command completion status indicating that sense data should be available.. The structure of sense data is described in the SCSI Primary Commands documents[3]. The size of the *sense* field should be equal to the number of bytes indicated by the vDisk client in the *sense length* field rounded up to a multiple of 8 bytes.

The *data-in* field contains command specific information returned by the vDisk server at the time of command completion. The validity of the returned data depends on the SCSI command completion status. The size of the *data-in* field should equal to the number of bytes indicated by the vDisk client in the *data-in size* field rounded up to a multiple of 8 bytes.

The *data-out* field contains command specific information to be sent to the vDisk server. The size of the *data-out* field should be equal to the number of bytes indicated by the vDisk client in the *data-out size* field rounded up to a multiple of 8 bytes.

1.1.6.14 VDisk Get Device ID (VD_OP_GET_DEVID)

Device IDs[1] are persistent unique identifiers for devices in Solaris, and provide a means for identifying a device, independent of device's current name or instance number.

This command is used to return the device ID of a disk volume backing a virtual disk. A successful completion of this command will result in the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following format:

```
              6           4 4             3 3
              3           8 7             2 1                             0
             +-------------+--------------+-------------------------------+
  word 0:    |   reserved  |     type     |            length             |
             +----------------------------+-------------------------------+
word 1-N:    |                          devid                             |
             +------------------------------------------------------------+
```

The field *devid* contains the ID of the disk volume. The field *length* in the request should be set to the size of the buffer allocated by the vdisk client for storing the device ID. The vdisk server will then set it to the size of the returned *devid* in its response. The returned device ID value will be truncated if the provided space is not large enough to store complete ID. The field *type* specifies the type of device ID.

Please refer to PSARC cases 1995/352, 2001/559, 2004/504, for a description of device IDs along and a list of the device ID *type* values.

1.1.6.15 VDisk Get EFI Data (VD_OP_GET_EFI)

This command is used to get EFI data for the disk volume a client is attached to. A successful completion of this command will result in the following data structure with the EFI data in the data field being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring.

The returned data structure has the following format:

```
              6
              3                                                           0
             +------------------------------------------------------------+
  word 0:    |                           LBA                              |
             +------------------------------------------------------------+
  word 1:    |                          length                            |
             +------------------------------------------------------------+
word 2-N:    |                         EFI data                           |
             +------------------------------------------------------------+
```

The field *LBA* is the logical block address of the disk volume to get EFI data. Data returned in the EFI data field is determined by the value specified in the LBA field:

- If LBA is equal to 1, then the vdisk server should return the GUID Partition Table Header (GPT).
- If LBA is equal to the PartitionEntryLBA field from the GUID Partition Table Header, then the vdisk server should return the GUID Partition Entry array (aka GPE).

If the EFI data buffer is not large enough to return the request data then the vdisk server should return an error. The field *length* is the maximum number of bytes that can be stored in the data field of the provided structure.

The format of the GUID Partition Table Header and GUID Partition Entry are beyond the scope of this document and are defined in the Extensible Firmware Interface Specification[2].

1.1.6.16 VDisk Set EFI Data (VD_OP_SET_EFI)

This command is used by a virtual disk client to set EFI data for the disk volume the client is attached to. The supplied data structure has the same format as for the get EFI command (VD_OP_GET_EFI).

The value of the LBA field determines the content of the EFI data field and the action taken by the vdisk server.

- If LBA = 1, then the vdisk server should use the contents of the EFI data field to set the GUID Partition Table Header (aka GPT).

- If LBA is equal to the PartitionEntryLBA field from the GUID Partition Table Header, then the vdisk server should the contents of the EFI data field to set the GUID Partition Entry array (aka GPE).

The format of the GUID Partition Table Header and GUID Partition Entry are beyond the scope of this document and are defined in the Extensible Firmware Interface Specification[2].

1.1.6.17 VDisk Reset (VD_OP_RESET)

This command is used by the vDisk client to request the vDisk server to reset the disk or device being exported by it. It is implementation independent as to whether the server physically resets the underlying device or it chooses to only simulate a device reset.

Following a reset, any exclusive access rights or options that might have been set using the VD_OP_SET_ACCESS operation should be cleared in a way similar to receiving a VD_OP_SET_ACCESS operation with the CLEAR option.

In the event of a connection loss between the vDisk client and server, the vDisk server should behave as if it has received a VD_OP_RESET operation. It should clear any exclusive access rights or options set using the VD_OP_SET_ACCESS operation. A vDisk server implementing the disk reset is required to complete the operation prior to reestablishing the connection with the vDisk client.

1.1.6.18 VDisk Get Access (VD_OP_GET_ACCESS)

This command is used by the vDisk client to query whether it has access to the disk being exported by the vDisk server. The response has a payload of a single 64 bit unsigned integer, and may contain the following values:

- 0x00 (DENIED)
  The access to the disk is not allowed.
- 0x01 (ALLOWED)
  The access to the disk is allowed.

1.1.6.19 VDisk Set Access (VD_OP_SET_ACCESS)

This command is used by the vDisk client to request exclusive access to the disk being exported by the vDisk server. The payload is a single 64 bit unsigned integer. It can either contain a value of 0, or a bitmask of the following non-zero values:

- 0x00 (CLEAR)
  The vDisk server should clear any exclusive access rights, and restore non-exclusive, non-preserved access rights. In particular, the vDisk server should relinquish any exclusive access rights that have been acquired with the EXCLUSIVE flag, and disable any mechanism to preserve exclusive access rights enabled with the PRESERVE flag.
- 0x01 (EXCLUSIVE)
  The vDisk server should acquire exclusive access rights to the disk. When the vDisk server has exclusive access rights to the disk then any access to the disk from another host should fail. If another host already has acquired exclusive access rights to the disk then the vDisk server should fail to acquire exclusive access rights.
- 0x02 (PREEMPT)
  The vDisk server can forcefully acquire exclusive access rights to the disk. If another host has already acquired exclusive access rights to the disk, then the vDisk server can preempt the other host and acquire exclusive access rights.
- 0x04 (PRESERVE)
  The vDisk server should try to preserve exclusive access rights to the disk. The vDisk server should try to restore exclusive access rights if exclusive access rights are

broken via random events (for example disk resets). When restoring the exclusive access rights, the vDisk server should not preempt any other host having exclusive access rights to the disk.

The PREEMPT and PRESERVE flags are only valid when the EXCLUSIVE flag is set.

In the event of a connection loss between the vDisk client and server, the vDisk server should perform the equivalent operation to a vDisk Reset Command (VD_OP_RESET) received from the client, and exclusive access rights and options should be cleared.

If the vDisk client still requires exclusive access rights following a connection reset, then it should send a new VD_OP_SET_ACCESS operation to the vDisk server and request exclusive access.

1.1.6.20 VDisk Get Capacity (VD_OP_GET_CAPACITY)

This command is used to get information about the capacity of the disk volume export by the vDisk server. A successful completion of this command will result in the following data structure being returned to the client in the buffer described by the LDC cookie(s) in the descriptor ring:

```
           6                             3 3                             0
           3                             2 1                             0
           +-----------------------------+-----------------------------+
word 0:    |          reserved           |      VDISK_BLOCK_SIZE       |
           +-----------------------------+-----------------------------+
word 1:    |                          VDISK_SIZE                       |
           +-----------------------------------------------------------+
```

The *vdisk_block_size* field contains the length in byte of the logical block of the vDisk. The *vdisk_block_size* should be the same value as the *vdisk_block_size* returned during the initial handshake as part of the attribute exchange.

The *vdisk_size* field contains the size of the vDisk in blocks specified as a multiple of *vdisk_block_size*.

If the vDisk server is unable to obtain the vDisk size, it should set the *vdisk_size* to -1. Under these circumtances, the vDisk client can retry the operation later to check if the size is available.

1.1.7 Virtual network specific data

1.1.7.1 Attribute information

During the initial handshake, as part of the CTRL/INFO/ ATTR_INFO message, the virtual network device will exchange information with the virtual switch and other vNetwork devices about the transfer protocol, its address and MTU. The format of the attribute payload is shown below:

```
          6                          3 3           1 1
          3                          2 1           6 5     8 7       0
          +----------------------------+-----------+---------+---------+
word 1:   |            SID             | ATTR_INFO |  I/A/N  |TYPE_CTRL|
          +----------------------------+-----------+---------+---------+
word 2:   |          reserved          |  ACK_FREQ |ADDR_TYPE|XFER_MODE|
          +----------------------------+-----------+---------+---------+
word 3:   |                           ADDR                             |
          +------------------------------------------------------------+
word 4:   |                           MTU                              |
          +------------------------------------------------------------+
```

The sending client, be it a virtual network device and/or virtual switch will provide its peer with the transfer mode, acknowledgment frequency, address, address type and MTU it intends to use for sending network packets. The peer ACKs the attribute message if it agrees to all the parameters. Currently the only supported address type is:

```
VNET_ADDR_ETHERMAC    0x1   /* Ethernet MAC Address */
```

The *addr* field contains the mac address of the client sending the attribute information.

If VIO version 1.3 or lower is negotiated, it is required that the MTU exchanged by either ends during the attribute exchange matches exactly. If version 1.4 or higher is negotiated, and the MTU received in the ATTR/INFO doesn't match the receiver's MTU, it ACKs with the lower of the two MTUs. All subsequent communication between both ends are required to use the mutually agreed upon MTU.

1.1.7.2 Multicast information

Virtual network devices can set/unset the multicast groups they are interested in to a virtual network switch at any point after a succesful handshake and during normal data transfer. Each packet sent by a vnet device is of type CTRL/INFO/MCAST_INFO.

```
VNET_MCAST_INFO           0x101       /* Multicast information */
```

If the *set* field is equal to '1', then the corresponding mcast addresses are being set by the vnet device, or else the switch assumes that the specified address(es) are being removed. The peer will ACK the info packet if it successfully registered or removed the specified multicast mac addresses. If the multicast address was already set earlier or if the network device tries to unset an address that was not set earlier, the virtual switch will NACK the request. The `MCAST_ADDR` field can contain a max of `VNET_NUM_MCAST=7` multicast addresses, where each address is `ETHERADDRL=6` bytes in length. The *count* field specifies the actual number of multicast addresses in the packet.

```
             6                          3 3            1 1
             3                          2 1            6 5     8 7       0
             +--------------------------+--------------+-------+---------+
    word 1:  |           SID            |  MCAST_INFO  | I/A/N |TYPE_CTRL|
             +--------------------------+--------------+-------+---------+
    word 2:  |              MCAST_ADDR[0]              | COUNT |   SET   |
             +-----------------------------------------+-------+---------+
  word 3-7:  |                      MCAST_ADDR[1-6]                      |
             +-----------------------------------------------------------+
```

1.1.7.3 vNet descriptors

Virtual network and switch device clients that use HV shared memory will send / forward Ethernet frames by specifying the length of the data and the LDC memory cookie(s) corresponding to the page(s) containing the frame in each descriptor. The descriptor payload will be of the following format:

```
 6                              3 3                            0
 3                              2 1                            0
 +------------------------------+-----------------------------+
 |           ncookies           |            nbytes           |
 +------------------------------+-----------------------------+
 |                  ldc_cookie * ncookies                     |
 +------------------------------------------------------------+
```

The *nbytes* field specifies the number of bytes being transmitted. The *ncookies* and *ldc_cookie* fields refer to the segment of memory from/to which data is being read/written. See sec 1.1.3.3 for more information about the LDC transport cookie.

In the current implementation, since each request/payload contained within a descriptor corresponds to an Ethernet frame being transmitted by either a vNet or vSwitch device, the vNet and vSwitch will register the descriptor ring as a transmit ring. Future implementations of the protocol might use the descriptor rings as receive rings.

1.1.7.4 Virtual LAN (VLAN) support

The VIO protocol for virtual network and switch devices will be extended in version 1.3 to include support for virtual LANs (VLANs) as specified by the IEEE 802.1Q[4] specification. A VLAN aware network or switch device will be capable of sending, receiving or switching ethernet frames that contain a vlan tagged header. If a network/switch device negotiates version 1.3 or higher with its peer, the MTU size it specifies in the attribute info message (sec 1.1.7.1) should correspond to the size of a tagged ethernet frame. Similarly, if a peer negotiates version 1.2 or lower, sending/receiving tagged frames can result in undefined behavior including the frames being dropped.

1.1.7.5 Network Device Resource Sharing via DDS

The VIO DDS control message provides the capability to share device resources between VIO device peers. The DDS framework will be primarily used by a vSwitch device to share the underlying physical network device's resources with a vNet device.

All DDS messages for vNet and vSwitch devices will contain a *class* field that uniquely identifies the type of device from which the resources are being shared. In version v1.3 of the VIO protocol, the vNet device will define a new DDS message class DDS_VNET_NIU for sharing the resources of a Niagara-2 NIU device.

```
DDS_VNET_NIU          0x10  /* NIU vNet class */
```

Each DDS message of class VNET_NIU sent by a vSwitch or a vNet will contain a *subclass* field that specifies the requested operation. The DDS subclass values for a VNET_NIU class are:

```
DDS_VNET_ADD_SHARE   0x1   /* Add a device share */
DDS_VNET_DEL_SHARE   0x2   /* Delete a device share */
DDS_VNET_REL_SHARE   0x3   /* Release a device share */
DDS_VNET_MOD_SHARE   0x4   /* Modify a device share */
```

The DDS_VNET_(ADD/DEL/REL)_SHARE messages subclasses are used when adding or deleting a resource to a domain or releasing a resource from a domain.

The ADD_SHARE message is used by the vSwitch device to add a virtual region resource uniquely identified by its *cookie* to a vNet device identified by its *macaddr*. The DEL_SHARE message is similarly used by the vSwitch to remove a virtual region resource that was previously added using the ADD_SHARE operation. The REL_SHARE message is used by the vNet device to inform the vSwitch device that it is no longer using a previously added shared resource. The vSwitch on receiving a REL_SHARE message can reclaim and reassign the resource to another vNet. A vNet device should not attempt to use a resource that it had previously released via the REL_SHARE operation. The message format for the add, delete and release operations is identical and is shown below:

```
           6             4 4           3 3          1 1
           3             8 7           2 1          6 5          8 7       0
           +-------------+-------------+------------+------------+---------+
  word 1:  |            SID            | DDS_INFO   |    INFO    |TYPE_CTRL|
           +---------------------------+------------+------------+---------+
  word 2:  |      DDS_REQUEST_ID       | reserved   |A/D/R_SHARE |VNET_NIU |
           +-------------+-------------+------------+------------+---------+
  word 3:  |  reserved   |                     MACADDR                     |
           +-------------+-------------------------------------------------+
  word 4:  |                           COOKIE                              |
           +---------------------------------------------------------------+
```

The resource modification operation allows a vSwitch device to modify the contents of a shared virtual region. In addition to the macaddr and cookie fields, the message also contains a updated map of TX and RX resources assigned to the virtual region resource. The format of the modify message is shown below:

```
          6                  4 4                  3 3                  1 1                8 7         0
          3                  8 7                  2 1                  6 5
         +--------------------+--------------------+------------------+-----------------+----------+
word 1:  |                 SID                     |    DDS_INFO      |      INFO       |TYPE_CTRL |
         +-----------------------------------------+------------------+-----------------+----------+
word 2:  |           DDS_REQUEST_ID                |    reserved      |   MOD_SHARE     |VNET_NIU  |
         +--------------------+--------------------+------------------+-----------------+----------+
word 3:  |      reserved      |                          MACADDR                                   |
         +--------------------+--------------------------------------------------------------------+
word 4:  |                                        COOKIE                                           |
         +-----------------------------------------------------------------------------------------+
word 5:  |                                        TX_RES_MAP                                       |
         +-----------------------------------------------------------------------------------------+
word 6:  |                                        RX_RES_MAP                                       |
         +-----------------------------------------------------------------------------------------+
```

In addition to the different CTRL/INFO/DDS_INFO request messages, the vNet and vSwitch devices will also ACK and NACK all received DDS requests. The ACK and NACK responses will contain a STATUS field that specify the outcome of the requested operation. The format of the ACK/NACK response message is below:

```
          6                                    3 3              1 1                8 7         0
          3                                    2 1              6 5
         +-----------------------------------------+-------------+-----------------+----------+
word 1:  |                 SID                     |  DDS_INFO   |       A/N       |TYPE_CTRL |
         +-----------------------------------------+-------------+-----------------+----------+
word 2:  |           DDS_REQUEST_ID                |  reserved   |A/D/R/M_SHARE    |VNET_NIU  |
         +-----------------------------------------+-------------+-----------------+----------+
word 3:  |                                       STATUS                                       |
         +------------------------------------------------------------------------------------+
```

The currently defined ACK and NACK status values are:

```
DDS_VNET_SUCCESS      0x0    /* Operation was successful */
DDS_VNET_FAIL         0x1    /* Operation failed */
```

1.2 References

1. ARC Cases
    - FWARC/2005/633 - Project Q Logical Domaining Umbrella
    - FWARC/2006/055 - Domain Services
    - FWARC/2006/074 - sun4v interrupt cookies
    - FWARC/2006/135 - sun4v channel console packets
    - FWARC/2006/140 - sun4v channels transport protocol
    - FWARC/2006/072 – sun4v virtual devices machine description data
    - PSARC/1995/352 – Disk IDs
2. Extensible Firmware Interface Specification
   http://developer.intel.com/technology/efi/main_specification.htm
3. SCSI Standards Architecture
   http://www.t10.org/scsi-3.htm
4. 802.1Q – Virtual LANs
   http://www.ieee802.org/1/pages/802.1Q.html