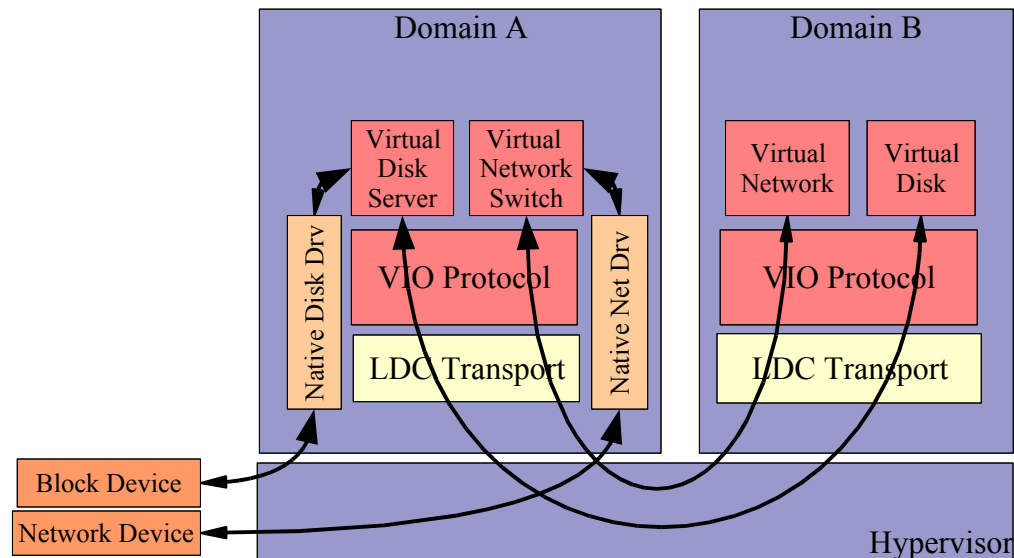


1.1 Virtual IO communication protocol

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.



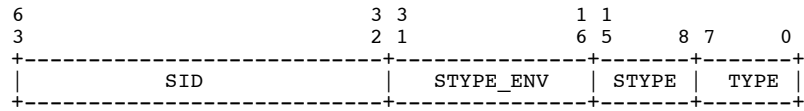
1.1.1 VIO data transfer

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

35 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:



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

45 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

50 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.

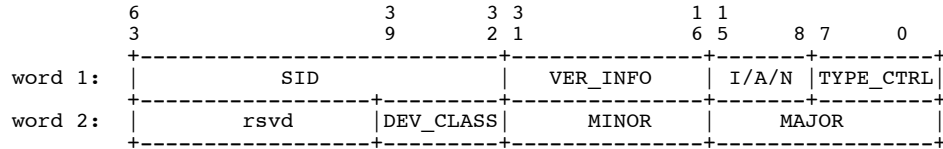
55 If the device class is recognized and the version major/minor numbers are acceptable then the receiving endpoint responds back with a CTRL/ACK/VER_INFO message leaving all the parameters unchanged. It also stores the sender's SID for use in future message exchanges.

60 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 CTRL/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.

65

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:



70

The currently supported devices types are listed below:

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

75

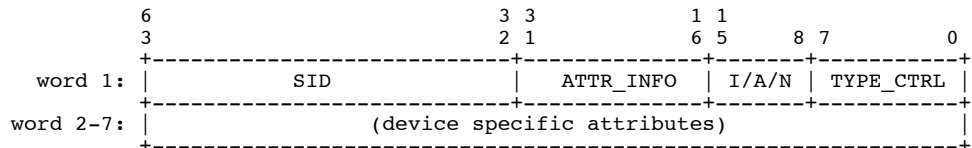
NOTE: Irrespective of what state the receiving endpoint believes the channel to be in, receipt of a CTRL/INFO/VER_INFO message at any time will cause the endpoint to reset any internal state it may be maintaining for that channel and restart the handshake.

1.1.3.2 Attribute exchange

80

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.

85

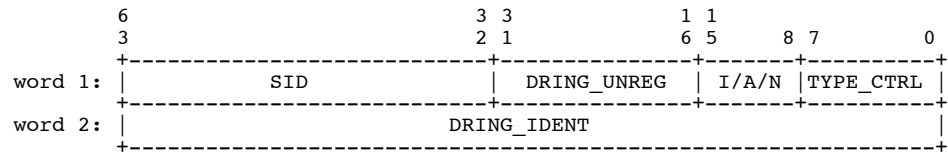


1.1.3.3 Descriptor ring registration

90

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.

ring, the DRING_DATA requests will be NACKed.

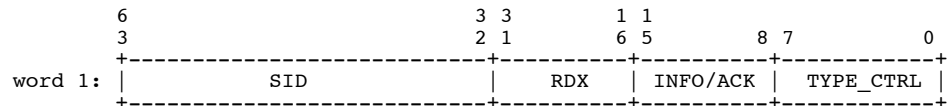


130 1.1.3.4 Handshake completion

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.

135

140



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.

145

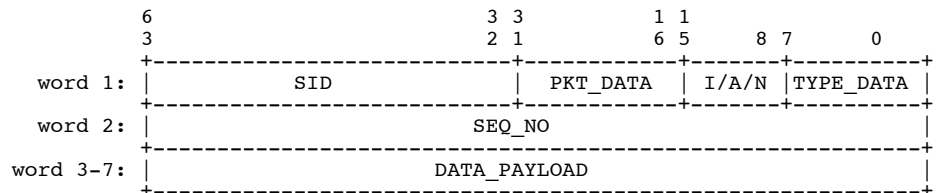
1.1.4 VIO data transfer modes

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 are:

150

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

155 1.1.4.1 Packet based transfer



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.

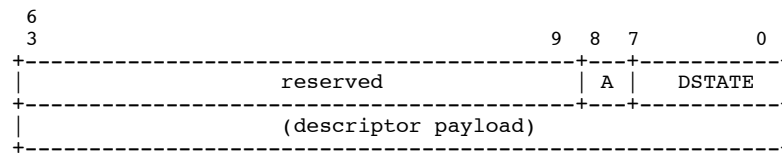
160

1.1.4.2 Descriptor rings

165 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).

170 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.

175 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 1.1.5 and 1.1.6.



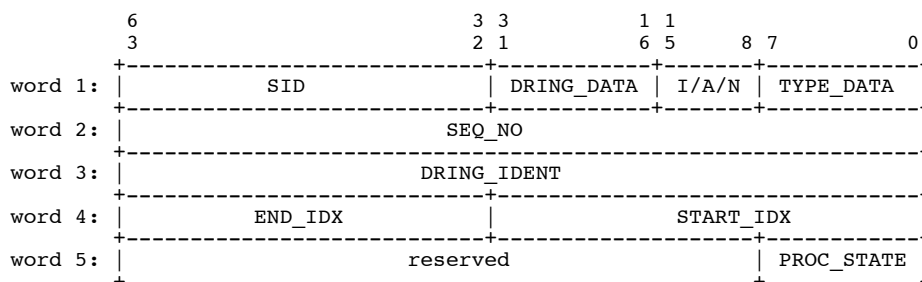
The descriptor *dstate* specifies the state of the the descriptor. The valid state values are:

- VIO_DESC_FREE 0x1
- VIO_DESC_READY 0x2
- 180 VIO_DESC_ACCEPTED 0x3
- VIO_DESC_DONE 0x4

185 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
 190 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.

195 When the requesting client updates one or more descriptors and marks them as ready for
 processing, it will send a DATA/INFO/DRNG_DATA message to its peer at the otherend 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
 200 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.

205 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.



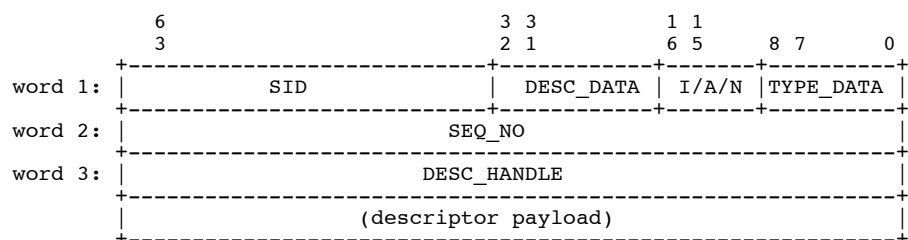
210 The requester can also request an explicit acknowledgment from the client processing the request (to track progress) by setting the *(A)cknowledge* field in the descriptor. The client, after processing the descriptor (changes state as `VIO_DESC_DONE`), will send a `DATA/ACK/DRING_DATA` message with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

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

220 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.

225 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:



230 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.

235 1.1.5 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`

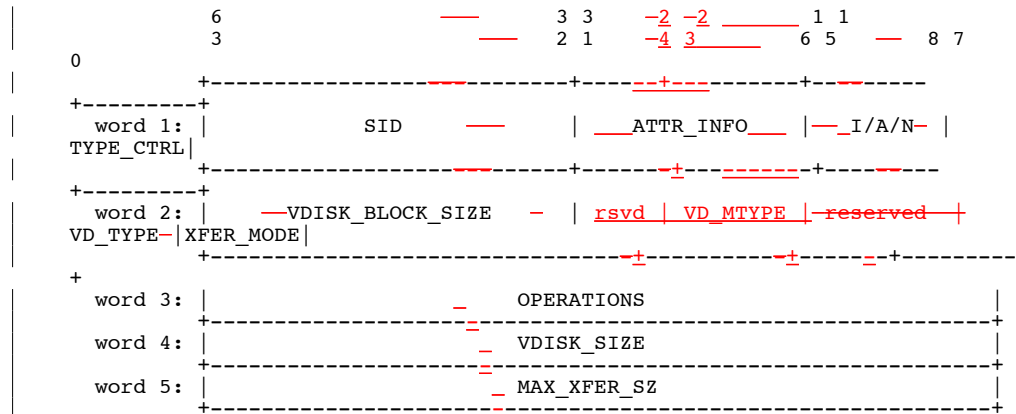
240

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.

1.1.5.1 Attribute information

245

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:



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.

250

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

255

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.

260

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:

265

```

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

```

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

270

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 */

```



```

315 |         VD_OP_SET_DISKGEOM    0x09        /* Set disk geometry */
|         VD_OP_SCSICMD        0x0a        /* SCSI control command */
|         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            0x0be - 0xff    /* reserved */

```

320 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.

325 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.

1.1.5.3 Disks and slices

330 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.

335 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.

340 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.

345 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.

350 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.

1.1.5.4 VDisk Block Read command (VD_OP_BREAD)

360 This command performs a basic read of a block from the device service. The descriptor 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.

365 1.1.5.5 VDisk Block Write command (VD_OP_BWRITE)

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

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

1.1.5.6 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 descriptor entry for this command.

375 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.

1.1.5.7 VDisk Get Write Cache enablement status (VD_OP_GET_WCE)

380 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.

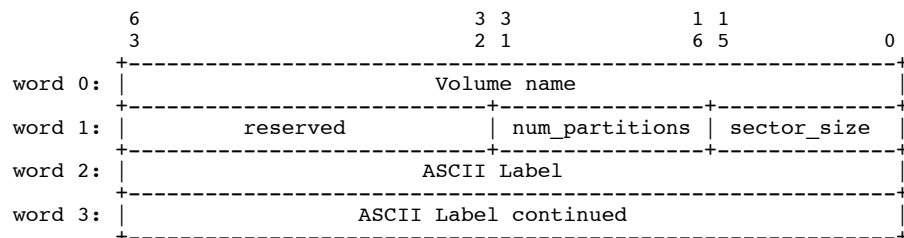
1.1.5.8 VDisk Enable/Disable Write Cache (VD_OP_SET_WCE)

385 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 vDiskserver.

1.1.5.9 VDisk Get Volume Table of Contents (VD_OP_GET_VTOC)

390 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:



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

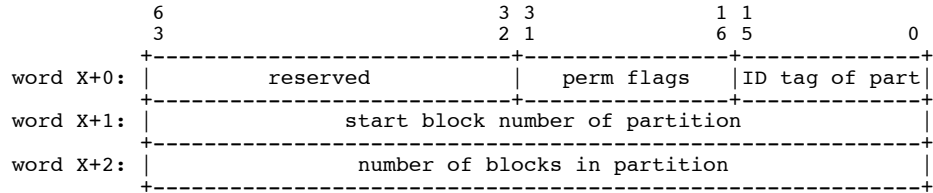
395 The ASCII label is a 128 character ASCII label assigned to this disk volume. This is

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:

400



Reserved fields should be ignored.

1.1.5.10 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.

405

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.

1.1.5.11 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.

410

The returned data structure has the following format:

<i>Byte offset</i>	<i>Size in bytes</i>	<i>Field name</i>	<i>Description</i>
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

1.1.5.12 VDisk Set Disk Geometry (VD_OP_SET_DISKGEOM)

415

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).

1.1.5.13 *VDisk SCSI Command (VD_OP_SCASICMD)*

420 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.

425 A successful completion of this command may result in the returning of a SCSI result packet in the same buffer used to supply the original command packet.

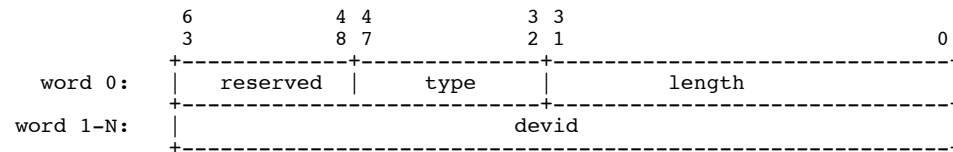
This command option is merely a pass-through capability for supporting SCSI packet transfer between a vDisk client and a vDisk server. SCSI packet formats are beyond the scope of this document.

1.1.5.14 *VDisk Get Device ID (VD_OP_GET_DEVID)*

430 Device IDs¹ 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.

435 The returned data structure has the following format:



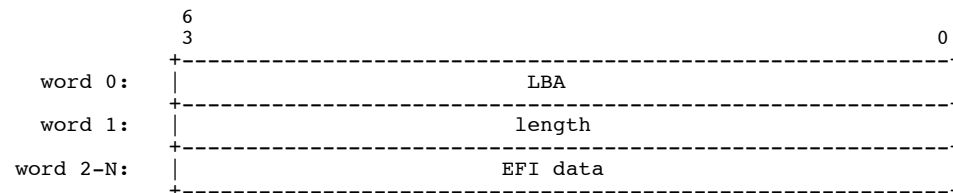
440 The field *devid* contains the ID of the disk volume. The field *length* in the request should be set to the size of the buffer allocated by the vdisk client for storing the device ID. The vdisk server will then set it to the size of the returned *devid* in its response. The returned device ID value 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.

1.1.5.15 *VDisk Get EFI Data (VD_OP_GET_EFI)*

445 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:



450 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).

455 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 Specification².

460 **1.1.5.16 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).

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

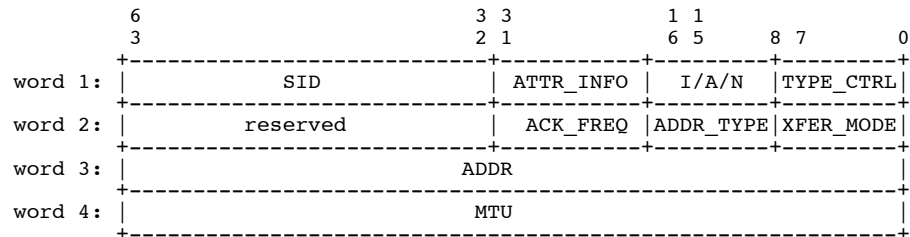
- If LBA = 1, then the vdisk server should use the contents of the EFI data field to set the GUID Partition Table Header (aka GPT).
- If LBA is equal to the PartitionEntryLBA field from the GUID Partition Table Header, then the vdisk server should the contents of the EFI data field to set the GUID Partition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

470 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 Specification².

1.1.6 Virtual network specific data

1.1.6.1 Attribute information

475 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:



480 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 */

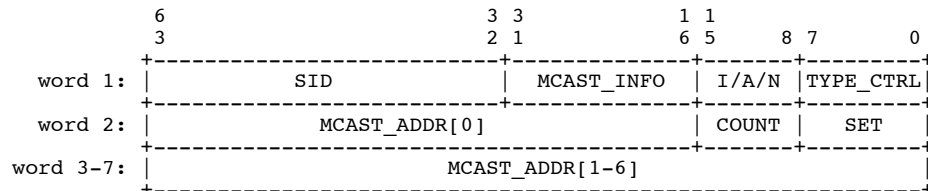
The *addr* field contains the mac address of the client sending the attribute information.

485 **1.1.6.2 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.

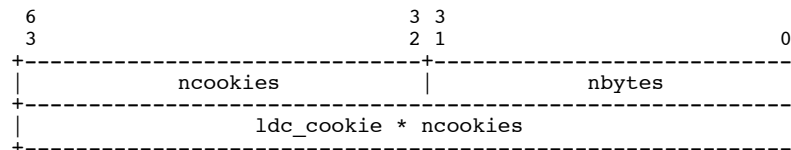
490 VNET_MCAST_INFO 0x101 /* Multicast information */

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.



1.1.6.3 vNet descriptors

500 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:



505 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.

510 | 1.2 References

- 515 | 1. [ARC Cases](#)
 - [FWARC/2005/633 - Project O 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](#)
- 520 | 2. [Extensible Firmware Interface Specification](#)
http://developer.intel.com/technology/efi/main_specification.htm