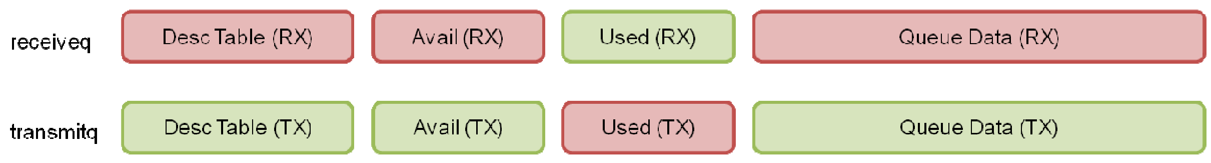
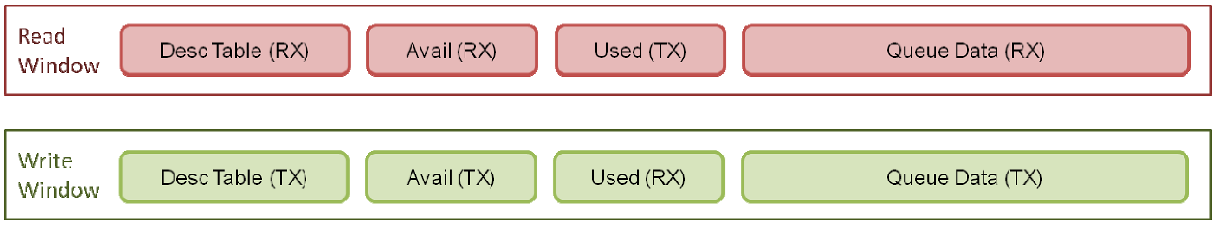
Peer shared memory communication device (virtio-peer) GENERAL OVERVIEW

Virtqueue View



Window View



The Virtio Peer shared memory communication device (**virtio-peer**) is a virtual device which allows high performance low latency guest to guest communication. It uses a new queue extension feature tentatively called **VIRTIO_F_WINDOW** which indicates that descriptor tables, available and used rings and Queue Data reside in physical memory ranges called **Windows**, each identified with an unique identifier called **WindowID**.

Each queue is configured to belong to a specific WindowID, and during queue identification and configuration, the Physical Guest Addresses in the queue configuration fields are to be considered as offsets in octets from the start of the corresponding Window.

For example for PCI, in the virtio_pci_common_cfg structure these fields are affected:
le64 queue_desc;
le64 queue_avail;
le64 queue_used;

For **MMIO** this includes instead these MMIO Device layout fields are affected:
QueueDescLow, QueueDescHigh
QueueAvailLow, QueueAvailHigh
QueueUsedLow, QueueUsedHigh

For **PCI** a new virtio_pci_cap of cfg type VIRTIO_PCI_CAP_WINDOW_CFG is defined.
It contains the following fields:

```
struct virtio_pci_window_cap {
   struct virtio_pci_cap cap;
}
```

This configuration structure is used to identify the existing Windows, their WindowIDs, ranges and flags. The WindowID is read from the cap.bar field. The Window starting physical guest address is calculated by starting from the contents of the PCI BAR register with index WindowID, plus the cap.offset. The Window size is read from the cap.length field.

XXX TODO XXX describe also the new MMIO registers here.

Virtqueue discovery:

We are faced with two main options with regards to virtqueue discovery in this model.

OPTION1: The simplest option is to make the previous fields **read-only** when using Windows, and have the virtualization environment / hypervisor provide the starting addresses of the descriptor table, avail ring and used rings, possibly allowing more flexibility on the Queue Data.

OPTION2: The other option is to have the guest completely in control of the allocation decisions inside its write Window, including the virtqueue data structures starting addresses inside the Window, and provide a simple virtqueue peer initialization mechanism.

The virtio-peer device is the simplest device implementation which makes use of the Window feature, containing only two virtqueues.

In addition to the Desc Table and Rings, these virtqueues also contain **Queue Data** areas inside the respective Windows.

It uses two Windows, one for data which is **read-only** for the driver (**read Window**), and a separate one for data which is **read-write** for the driver (**write Window**).

In the Descriptor Table of each virtqueue, the field
    le64 **addr**;
is added to the Queue Data address of the corresponding Window to obtain the physical guest address of a buffer.
A value of **length** in a descriptor which exceeds the Queue Data area is invalid, and its use will cause undefined behavior.

The driver must consider the Desc Table, Avail Ring and Queue Data area of the receiveq as read-only, and the Used Ring as read-write.
The Desc Table, Avail Ring and Queue Data of the receiveq will be therefore allocated inside the read Window, while the Used ring will be allocated in the write Window.

The driver must consider the Desc Table, Avail Ring and Queue Data area of the transmitq as read-write, and the Used Ring as read-only.
The Desc Table, Avail Ring and Queue Data of the transmitq will be therefore allocated inside the write Window, while the Used Ring will be allocated in the read Window.

Note that in OPTION1, this is done by the hypervisor, while in OPTION2, this is fully under control of the peers (with some hypervisor involvement during initialization).

5.7.1 Device ID

13

5.7.2 Virtqueues

0 receiveq (RX), 1 transmitq (TX)

5.7.3 Feature Bits

Possibly **VIRTIO_F_MULTICAST** (NOT clear yet left out for now)

5.7.4 Device configuration layout

```
struct virtio_peer_config {
   le64 queue_data_offset;
   le32 queue_data_size;
   u8 queue_flags; /* read-only flags*/
   u8 queue_window_idr; /* read-only */
   u8 queue_window_idw; /* read-only */
}
```

The fields above are queue-specific, and are thus selected by writing to the queue selector field in the common configuration structure.
queue_data_offset is the offset from the start of the **Window** of the Queue Data area,
queue_data_size is the size of the Queue Data area.
For the Read Window, the queue_data_offset and queue_data_size are read-only.
For the Write Window, the queue_data_offset and queue_data_size are read-write.

The **queue_flags** if a flag bitfield with the following bits already defined:

(1) = FLAGS_REMOTE : this queue descr, avail, and data is read-only and initialized by the remote peer, while the used ring is initialized by the driver.
If this flag is not set, this queue descr, avail, and data is read-write and initialized by the driver, while the used ring is initialized by the remote peer.

queue_window_idr and queue_window_idw identify the read-window and write-window for this queue (Window IDs).

5.7.5 Device Initialization

Initialization of the virtqueues follows the generic procedure for Virtqueue Initialization with the following modifications.

OPTION1: the driver needs to replace the step
“Allocate and zero” of the data structures and the write to the queue configuration registers with a read from the queue configuration registers to obtain the addresses of the virtqueue data structures.

OPTION 2: for each virtqueue, the driver allocates and zeroes the data structures as usual only for the read-write data structures, while skipping the read-only queue structures, which will be initialized by the peer. The **queue_flags** configuration field can be used to easily determine which fields are to be initialized, and the queue window id registers are used to identify the Windows to use for the data structures.
This feature adds the requirement to enable all virtqueues before the DRIVER_OK (which is already practice in most drivers, this is done as usual by writing 1 to the queue_enable field).

Driver attempts to read back from the queue_enable field for a queue which has not been also enabled by the remote peer will have the device return 0 (disabled) until the remote peer has also initialized its own share of the data structures for the virtqueue as it appears in the remote guest. All the queue configuration fields which still need remote initialization have a reset value of 0.

When the FEATURE BIT is detected, the virtio driver will delay setting of the DRIVER_OK status for the device.

When both peers have enabled the queues by writing 1 to the queue_enable fields, the driver will be notified via a configuration change interrupt (VIRTIO_PCI_ISR_CONFIG).

This will allow the driver to read the necessary queue configuration fields as initialized by the remote peer, and proceed setting the DRIVER_OK status for the device to signal the completion of the initialization steps.

5.7.6 Device Operation

Data is received from the peer on the receive virtqueue.

Data is transmitted to the peer using the transmit virtqueue.

5.7.6.1

OMISSIS

5.7.6.2 Transmitting data

Transmitting a chunk of data of arbitrary size is done by following the steps 3.2.1 to 3.2.1.4. The device will update the used field as described in 3.2.2.

5.7.6.2.1 Packet Transmission Interrupt

OMISSIS

5.7.6.3 Receiving data

Receiving data consists in the driver checking the receiveq available ring to be able to find the receive buffers. The procedure is the one usually performed by the device, involving update of the Used ring and a notification, as described in chapter 3.2.2

5.7.xxx: Additional notes and TODOS

Just a note: the Indirect Descriptors feature (**VIRTIO_RING_F_INDIRECT**) may not compatible with this feature, and thus will not be negotiated by the device.

Notification mechanisms need to be looked at in detail. Mostly we should be able to reuse the existing notification mechanisms, for OPTION2 configuration change we have identified the ISR_CONFIG notification method above.

MMIO needs to be written down.

PCI capabilities need to be checked again, and the fields in CFG_WINDOW in particular.
An alternative could be to extend the pci common configuration structure for the queue-specific extensions, but seems not compatible with multiple features involving similar extensions. Need to consider MMIO, as it's less extensible.

MULTICAST is out of scope of these notes, but seems feasible with some hard work without involving copies by sharing at least the transmit buffer in the producer, but the use case with peers being added and removed dynamically requires a much more complex study. Can this be solved with multiple queues, one for each peer, and configuration change notification interrupts that can disable a queue in the producer when a peer leaves, without taking down the whole device? Would need much more study.