Virtio PCI Card Specification
v0.8.7 DRAFT
-

Rusty Russell <rusty@rustcorp.com.au>
IBM Corporation

2010 April 20.

Chapter 1

Purpose and Description

This document describes the specifications of the “virtio” family of *PCI* devices. These are devices are found in *virtual environments*, yet by design they are not all that different from physical PCI devices, and this document treats them as such. This allows the guest to use standard PCI drivers and discovery mechanisms.

The purpose of virtio and this specification is that virtual environments and guests should have a straightforward, efficient, standard and extensible mechanism for virtual devices, rather than boutique per-environment or per-OS mechanisms.

Straightforward: Virtio PCI devices use normal PCI mechanisms of interrupts and DMA which should be familiar to any device driver author. There is no exotic page-flipping or COW mechanism: it’s just a PCI device.[1]

Efficient: Virtio PCI devices consist of rings of descriptors for input and output, which are neatly separated to avoid cache effects from both guest and device writing to the same cache lines.

Standard: Virtio PCI makes no assumptions about the environment in which it operates, beyond supporting PCI. In fact the virtio devices specified in the appendices do not require PCI at all: they have been implemented on non-PCI buses.[2]

[1] This lack of page-sharing implies that the implementation of the device (e.g. the hypervisor or host) needs full access to the guest memory. Communication with untrusted parties (i.e. inter-guest communication) requires copying.

[2] The Linux implementation further separates the PCI virtio code from the specific virtio drivers: these drivers are shared with the non-PCI implementations (currently lguest and S/390).

Extensible: Virtio PCI devices contain feature bits which are acknowledged by the guest operating system during device setup. This allows forwards and backwards compatibility: the device offers all the features it knows about, and the driver acknowledges those it understands and wishes to use.

1.1 Virtqueues

The mechanism for bulk data transport on virtio PCI devices is pretentiously called a virtqueue. Each device can have zero or more virtqueues: for example, the network device has one for transmit and one for receive.

Each virtqueue occupies two or more physically-contiguous pages (defined, for the purposes of this specification, as 4096 bytes), and consists of three parts:

Descriptor Table	Available Ring *(padding)*	Used Ring

When the driver wants to send buffers to the device, it puts them in one or more slots in the descriptor table, and writes the descriptor indices into the available ring. It then notifies the device. When the device has finished with the buffers, it writes the descriptors into the used ring, and sends an interrupt.

Chapter 2

Specification

2.1 PCI Discovery

Any PCI device with Vendor ID 0x1AF4, and Device ID 0x1000 through 0x103F inclusive is a virtio device[1]. The device must also have a Revision ID of 0 to match this specification.

The Subsystem Device ID indicates which virtio device is supported by the device. The Subsystem Vendor ID should reflect the PCI Vendor ID of the environment (it's currently only used for informational purposes by the guest).

Subsystem Device ID	Virtio Device	Specification
1	network card	Appendix C
2	block device	Appendix D
3	console	Appendix E
4	entropy source	Appendix F
5	memory ballooning	Appendix G
9	9P transport	-

2.2 Device Configuration

To configure the device, we use the first I/O region of the PCI device. This contains a *virtio header* followed by a *device-specific region.*

There may be different widths of accesses to the I/O region; the "natural" access method for each field in the virtio header must be used (i.e. 32-bit accesses for 32-bit fields, etc), but the device-specific region can be accessed using any width accesses, and should obtain the same results.

[1] The actual value within this range is ignored

Note that this is possible because while the virtio header is PCI (i.e. little) endian, the device-specific region is encoded in the native endian of the guest (where such distinction is applicable).

2.2.1 Device Initialization Sequence

We start with an overview of device initialization, then expand on the details of the device and how each step is preformed.

1. Reset the device. This is not required on initial start up.
2. The ACKNOWLEDGE status bit is set: we have noticed the device.
3. The DRIVER status bit is set: we know how to drive the device.
4. Device-specific setup, including reading the Device Feature Bits, discovery of virtqueues for the device, optional MSI-X setup, and reading and possibly writing the virtio configuration space.
5. The subset of Device Feature Bits understood by the driver is written to the device.
6. The DRIVER_OK status bit is set.
7. The device can now be used (ie. buffers added to the virtqueues)[2]

If any of these steps go irrecoverably wrong, the guest should set the FAILED status bit to indicate that it has given up on the device (it can reset the device later to restart if desired).

We now cover the fields required for general setup in detail.

2.2.2 Virtio Header

The virtio header looks as follows:

Bits	32	32	32	16	16	16	8	8
Read/Write	R	R+W	R+W	R	R+W	R+W	R+W	R
Purpose	Device Features	Guest Features	Queue Address	Queue Size	Queue Select	Queue Notify	Device Status	ISR Status

If MSI-X is enabled for the device, two additional fields immediately follow this header:

[2] Historically, drivers have used the device before steps 5 and 6. This is only allowed if the driver does not use any features which would alter this early use of the device.

Bits	16	16
Read/Write	R+W	R+W
Purpose (MSI-X)	Configuration Vector	Queue Vector

Immediately following these general headers, there may be device-specific headers:

Bits	Device Specific
Read/Write	Device Specific
Purpose	Device Specific...

2.2.2.1 Device Status

The Device Status field is updated by the guest to indicate its progress. This provides a simple low-level diagnostic: it's most useful to imagine them hooked up to traffic lights on the console indicating the status of each device.

The device can be reset by writing a 0 to this field, otherwise at least one bit should be set:

ACKNOWLEDGE (1) Indicates that the guest OS has found the device and recognized it as a valid virtio device.

DRIVER (2) Indicates that the guest OS knows how to drive the device. Under Linux, drivers can be loadable modules so there may be a significant (or infinite) delay before setting this bit.

DRIVER_OK (3) Indicates that the driver is set up and ready to drive the device.

FAILED (128) Indicates that something went wrong in the guest, and it has given up on the device. This could be an internal error, or the driver didn't like the device for some reason, or even a fatal error during device operation. The device must be reset before attempting to re-initialize.

2.2.2.2 Feature Bits

The least significant 31 bits of the first configuration field indicates the features that the device supports (the high bit is reserved, and will be used to indicate the presence of future feature bits elsewhere). The bits are allocated as follows:

0 to 23 Feature bits for the specific device type

24 to 30 Feature bits reserved for extensions to the queue mechanism

For example, feature bit 0 for a network device (i.e. Subsystem Device ID 1) indicates that the device supports checksumming of packets.

The feature bits are *negotiated:* the device lists all the features it understands in the Device Features field, and the guest writes the subset that it understands into the Guest Features field. The only way to renegotiate is to reset the device.

In particular, new fields in the device configuration header are indicated by offering a feature bit, so the guest can check before accessing that part of the configuration space.

This allows for forwards and backwards compatibility: if the device is enhanced with a new feature bit, older guests will not write that feature bit back to the Guest Features field and it can go into backwards compatibility mode. Similarly, if a guest is enhanced with a feature that the device doesn't support, it will not see that feature bit in the Device Features field and can go into backwards compatibility mode (or, for poor implementations, set the FAILED Device Status bit).

2.2.2.3 Configuration/Queue Vectors

When MSI-X capability is present and enabled in the device (through standard PCI configuration space) 4 bytes at byte offset 20 are used to map configuration change and queue interrupts to MSI-X vectors. In this case, the ISR Status field is unused, and device specific configuration starts at byte offset 24 in virtio header structure. When MSI-X capability is not enabled, device specific configuration starts at byte offset 20 in virtio header.

Writing a valid MSI-X Table entry number, 0 to 0x7FF, to one of Configuration/Queue Vector registers, *maps* interrupts triggered by the configuration change/selected queue events respectively to the corresponding MSI-X vector. To disable interrupts for a specific event type, unmap it by writing a special NO_VECTOR value:

```
/* Vector value used to disable MSI for queue */
#define VIRTIO_MSI_NO_VECTOR            0xffff
```

Reading these registers returns vector mapped to a given event, or NO_VECTOR if unmapped. All queue and configuration change events are unmapped by default.

Note that mapping an event to vector might require allocating internal device resources, and might fail. Devices report such failures by returning the NO_VECTOR value when the relevant Vector field is read. After mapping an event to vector, the driver must verify success by reading the Vector field value: on success, the previously written value is returned, and on failure, NO_VECTOR is returned. If a mapping failure is detected, the driver can retry mapping with fewervectors, or disable MSI-X.

2.3 Virtqueue Configuration

As a device can have zero or more virtqueues for bulk data transport (for example, the network driver has two), the driver needs to configure them as part of the device-specific configuration.

This is done as follows, for each virtqueue a device has:

1. Write the virtqueue index (first queue is 0) to the Queue Select field.
2. Read the virtqueue size from the Queue Size field, which is always a power of 2. This controls how big the virtqueue is (see below). If this field is 0, the virtqueue does not exist.
3. Allocate and zero virtqueue in contiguous physical memory, on a 4096 byte alignment. Write the physical address, divided by 4096 to the Queue Address field.[3]
4. Optionally, if MSI-X capability is present and enabled on the device, select a vector to use to request interrupts triggered by virtqueue events. Write the MSI-X Table entry number corresponding to this vector in Queue Vector field. Read the Queue Vector field: on success, previously written value is returned; on failure, NO_VECTOR value is returned.

The Queue Size field controls the total number of bytes required for the virtqueue according to the following formula:

```
#define ALIGN(x) (((x) + 4095) & ~4095)
static inline unsigned vring_size(unsigned int qsz)
{
     return ALIGN(sizeof(struct vring_desc)*qsz + sizeof(u16)*(2 + qsz))
          + ALIGN(sizeof(struct vring_used_elem)*qsz);
}
```

This currently wastes some space with padding, but also allows future extensions. The virtqueue layout structure looks like this (qsz is the Queue Size field, which is a variable, so this code won't compile):

```
struct vring {
    /* The actual descriptors (16 bytes each) */
    struct vring_desc desc[qsz];

    /* A ring of available descriptor heads with free-running index. */
    struct vring_avail avail;
```

[3] The 4096 is based on the x86 page size, but it's also large enough to ensure that the separate parts of the virtqueue are on separate cache lines.

```
    // Padding to the next 4096 boundary.
    char pad[];

    // A ring of used descriptor heads with free-running index.
    struct vring_used used;
};
```

2.3.1 A Note on Virtqueue Endianness

Note that the *endian* of these fields and everything else in the virtqueue is the native endian of the guest, not little-endian as PCI normally is. This makes for simpler guest code, and it is assumed that the host already has to be deeply aware of the guest endian so such an "endian-aware" device is not a significant issue.

2.3.2 Descriptor Table

The descriptor table refers to the buffers the guest is using for the device. The addresses are physical addresses, and the buffers can be chained via the next field. Each descriptor describes a buffer which is read-only or write-only, but a chain of descriptors can contain both read-only and write-only buffers.

No descriptor chain may be more than 2^32 bytes long in total.

```
struct vring_desc {
    /* Address (guest-physical). */
    u64 addr;
    /* Length. */
    u32 len;
/* This marks a buffer as continuing via the next field. */
#define VRING_DESC_F_NEXT   1
/* This marks a buffer as write-only (otherwise read-only). */
#define VRING_DESC_F_WRITE      2
/* This means the buffer contains a list of buffer descriptors. */
#define VRING_DESC_F_INDIRECT   4
    /* The flags as indicated above. */
    u16 flags;
    /* Next field if flags & NEXT */
    u16 next;
};
```

The number of descriptors in the table is specified by the Queue Size field for this virtqueue.Indirect Descriptors

Some devices benefit by concurrently dispatching a large number of large requests. The VIRTIO_RING_F_INDIRECT_DESC feature can be used to allow this (see 3). To increase ring capacity it is possible to store a table of *indirect descriptors* anywhere in memory, and insert a descriptor in main virtqueue (with flags&INDIRECT on) that refers to memory buffer containing this *indirect descriptor table*; fields *addr* and *len* refer to the indirect table address and length in bytes, respectively. The indirect table layout structure looks like this (len is the length of the descriptor that refers to this table, which is a variable, so this code won't compile):

```
struct indirect_descriptor_table {
    /* The actual descriptors (16 bytes each) */
    struct vring_desc desc[len / 16];
};
```

The first indirect descriptor is located at start of the indirect descriptor table (index 0), additional indirect descriptors are chained by next field. An indirect descriptor without next field (with flags&NEXT off) signals the end of the indirect descriptor table, and transfers control back to the main virtqueue. An indirect descriptor can not refer to another indirect descriptor table (flags&INDIRECT must be off). A single indirect descriptor table can include both read-only and write-only descriptors; write-only flag (flags&WRITE) in the descriptor that refers to it is ignored.

2.3.3 Available Ring

The available ring refers to what descriptors we are offering the device: it refers to the head of a descriptor chain. The "flags" field is currently 0 or 1: 1 indicating that we do not need an interrupt when the device consumes a descriptor from the available ring. This interrupt suppression is merely an optimization; it may not suppress interrupts entirely.

The "idx" field indicates where we would put the *next* descriptor entry (modulo the ring size). This starts at 0, and increases.

```
struct vring_avail {
#define VRING_AVAIL_F_NO_INTERRUPT      1
   u16 flags;
   u16 idx;
   u16 ring[qsz]; /* qsz is the Queue Size field read from device */
};
```

2.3.4 Used Ring

The used ring is where the device returns buffers once it is done with them. The flags field can be used by the device to hint that no notification is necessary when the guest adds to the *available* ring (the flag is kept here because this is the only part of the virtqueue written by the device).

Each entry in the ring is a pair: the head entry of the descriptor chain describing the buffer (this matches an entry placed in the available ring by the guest earlier), and the total of bytes written into the buffer. The latter is extremely useful for guests using untrusted buffers: if you do not know exactly how much has been written by the device, you usually have to zero the buffer to ensure no data leakage occurs.

```
/* u32 is used here for ids for padding reasons. */
struct vring_used_elem {
    /* Index of start of used descriptor chain. */
    u32 id;
    /* Total length of the descriptor chain which was used (written to) */
    u32 len;
};

struct vring_used {
#define VRING_USED_F_NO_NOTIFY  1
    u16 flags;
    u16 idx;
    struct vring_used_elem ring[qsz];
};
```

2.3.5 Helpers for Managing Virtqueues

The Linux Kernel Source code contains the definitions above and helper routines in a more usable form, in include/linux/virtio_ring.h. This was explicitly licensed by IBM under the (3-clause) BSD license so that it can be freely used by all other projects, and is reproduced (with slight variation to remove Linux assumptions) in Appendix A.

2.4 Device Operation

There are two parts to device operation: supplying new buffers to the device, and processing used buffers from the device. As an example, the virtio network device has two virtqueues: the transmit virtqueue and the receive virtqueue. The driver adds outgoing (read-only) packets are added to the transmit virtqueue, and then frees them after they are used. Similarly, incoming (write-only) buffers are added to the receive virtqueue, and processed after they are used.

2.4.1 Supplying Buffers to The Device

Actual transfer of buffers from the guest OS to the device operates as follows:

1. Place the buffer(s) into free descriptor(s).
    (a) If there are no free descriptors, the guest may choose to notify the device even if notifications are suppressed (to reduce latency).[4]
2. Place the id of the buffer in the next ring entry of the available ring.
3. The steps (1) and (2) may be performed repeatedly if batching is possible.
4. A memory barrier should be executed to ensure the device sees the updated descriptor table and available ring before the next step.
5. The available "idx" field should be increased by the number of entries added to the available ring.
6. A memory barrier should be executed to ensure that we update the idx field before checking for notification suppression.
7. If notifications are not suppressed, the device should be notified of the new buffers.

Note that the above code does not take precautions against the available ring buffer wrapping around: this is not possible since the ring buffer is the same size as the descriptor table, so step (1) will prevent such a condition.

In addition, the maximum queue size is 32768 (it must be a power of 2 which fits in 16 bits), so the 16-bit "idx" value can always distinguish between a full and empty buffer.

Here is a description of each stage in more detail.

2.4.1.1 Placing Buffers Into The Descriptor Table

A buffer consists of zero or more read-only physically-contiguous elements followed by zero or more physically-contiguous write-only elements (it must have at least one element). This algorithm maps it into the descriptor table:

1. for each buffer element, `b`:
    (a) Get the next free descriptor table entry, `d`
    (b) Set `d.addr` to the physical address of the start of `b`
    (c) Set `d.len` to the length of `b`.
    (d) If `b` is write-only, set `d.flags` to VRING_DESC_F_WRITE, otherwise 0.

[4] The Linux drivers do this only for read-only buffers: for write-only buffers, it is assumed that the driver is merely trying to keep the receive buffer ring full, and no notification of this expected condition is necessary.

(e) If there is a buffer element after this:

   i. Set `d.next` to the index of the next free descriptor element.

   ii. Set the VRING_DESC_F_NEXT bit in `d.flags`.

In practice, the d.next fields are usually used to chain free descriptors, and a separate count kept to check there are enough free descriptors before beginning the mappings.

2.4.1.2 Updating The Available Ring

The head of the buffer we mapped is the first `d` in the algorithm above. A naive implementation would do the following:

```
avail->ring[avail->idx % qsz] = head;
```

However, in general we can add many descriptors before we update the "idx" field (at which point they become visible to the device), so we keep a counter of how many we've added:

```
avail->ring[(avail->idx + added++) % qsz] = head;
```

2.4.1.3 Updating The Index Field

Once the idx field of the virtqueue is updated, the device will be able to access the descriptor entries we've created and the memory they refer to. This is why a memory barrier is generally used before the idx update, to ensure it sees the most up-to-date copy.

The idx field always increments, and we let it wrap naturally at 65536:

```
avail->idx += added;
```

2.4.1.4 Notifying The Device

Device notification occurs by writing the 16-bit virtqueue index of this virtqueue to the Queue Notify field of the virtio header in the first I/O region of the PCI device. This can be expensive, however, so the device can suppress such notifications if it doesn't need them. We have to be careful to expose the new idx value *before* checking the suppression flag: it's OK to notify gratuitously, but not to skip a notification. So again, we use a memory barrier here before reading the flags.

If the VRING_USED_F_NOTIFY flag is not set, we go ahead and write to the PCI configuration space.

2.4.2 Receiving Used Buffers From The Device

Once the device has used a buffer (read from or written to it, or parts of both, depending on the nature of the virtqueue and the device), it sends an interrupt, following an algorithm very similar to the algorithm used for the driver to send the device a buffer:

1. Write the head descriptor number to the next field in the used ring.
2. Update the used ring idx.
3. If the VRING_AVAIL_F_NO_INTERRUPT flag is not set in avail->flags:
   (a) If MSI-X capability is disabled:
      i. Set the lower bit of the ISR Status field for the device.
      ii. Send the appropriate PCI interrupt for the device.

   (b) If MSI-X capability is enabled:
      i. Request the appropriate MSI-X interrupt message for the device, Queue Vector field sets the MSI-X Table entry number.
      ii. If Queue Vector field value is NO_VECTOR, no interrupt message is requested for this event.

The guest interrupt handler should:

1. If MSI-X capability is disabled: read the ISR Status field, which will reset it to zero. If the lower bit is zero, the interrupt was not for this device. Otherwise, the guest driver should look through the used rings of each virtqueue for the device, to see if any progress has been made by the device which requires servicing.
2. If MSI-X capability is enabled: look through the used rings of each virtqueue mapped to the specific MSI-X vector for the device, to see if any progress has been made by the device which requires servicing.

```
while (vq->last_seen_used != vring->used.idx) {
    struct vring_used_elem *e = vring.used->ring[vq->last_seen_used%vsz];
    process_buffer(e);
    vq->last_seen_used++;
}
```

2.4.3 Dealing With Configuration Changes

Some virtio PCI devices can change the device configuration state, as reflected in the virtio header in the PCI configuration space. In this case:

1. If MSI-X capability is disabled: an interrupt is delivered and the second highest bit is set in the ISR Status field to indicate that the driver should re-examine the configuration space.Note that a single interrupt can indicate both that one or more virtqueue has been used and that the configuration space has changed: even if the config bit is set, virtqueues must be scanned.

2. If MSI-X capability is enabled: an interrupt message is requested. The Configuration Vector field sets the MSI-X Table entry number to use. If Configuration Vector field value is NO_VECTOR, no interrupt message is requested for this event.

Chapter 3

Creating New Device Types

Various considerations are necessary when creating a new device type:

How Many Virtqueues?

It is possible that a very simple device will operate entirely through its configuration space, but most will need at least one virtqueue in which it will place requests. A device with both input and output (eg. console and network devices described here) need two queues: one which the driver fills with buffers to receive input, and one which the driver places buffers to transmit output.

What Configuration Space Layout?

Configuration space is generally used for rarely-changing or initialization-time parameters. But it is a limited resource, so it might be better to use a virtqueue to update configuration information (the network device does this for filtering, otherwise the table in the config space could potentially be very large).

Note that this space is generally the guest's native endian, rather than PCI's little-endian.

What Device Number?

Currently device numbers are assigned quite freely: a simple request mail to the author of this document or the Linux virtualization mailing list[1] will be sufficient to secure a unique one.

[1] https://lists.linux-foundation.org/mailman/listinfo/virtualization

Meanwhile for experimental drivers, use 65535 and work backwards.

How many MSI-X vectors?

Using the optional MSI-X capability devices can speed up interrupt processing by removing the need to read ISR Status register by guest driver (which might be an expensive operation), reducing interrupt sharing between devices and queues within the device, and handling interrupts from multiple CPUs. However, some systems impose a limit (which might be as low as 256) on the total number of MSI-X vectors that can be allocated to all devices. Devices and/or device drivers should take this into account, limiting the number of vectors used unless the device is expected to cause a high volume of interrupts. Devices can control the number of vectors used by limiting the MSI-X Table Size or not presenting MSI-X capability in PCI configuration space. Drivers can control this by mapping events to as small number of vectors as possible, or disabling MSI-X capability altogether.

Message Framing

The descriptors used for a buffer should not effect the semantics of the message, except for the total length of the buffer. For example, a network buffer consists of a 10 byte header followed by the network packet. Whether this is presented in the ring descriptor chain as (say) a 10 byte buffer and a 1514 byte buffer, or a single 1524 byte buffer, or even three buffers, should have no effect.

In particular, no implementation should use the descriptor boundaries to determine the size of any header in a request.[2]

Device Improvements

Any change to configuration space, or new virtqueues, or behavioural changes, should be indicated be negotiation of a new feature bit. This establishes clarity[3] and avoids future expansion problems.

Clusters of functionality which are always implemented together can use a single bit, but if one feature makes sense without the others they should not be gratuitously grouped together to conserve feature bits. We can always extend the spec when the first person needs more than 24 feature bits for their device.

[2] The current qemu device implementations mistakenly insist that the first descriptor cover the header in these cases exactly, so a cautious driver should arrange it so.

[3] Even if it does mean documenting design or implementation mistakes!

Nomenclature

PCI Peripheral Component Interconnect; a common device bus. See http://en.wikipedia.org/wiki/Peripheral Component Interconnect

virtualized Environments where access to hardware is restricted (and often emulated) by a hypervisor.

Appendix A: virtio_ring.h

```
#ifndef VIRTIO_RING_H
#define VIRTIO_RING_H
/* An interface for efficient virtio implementation.
 *
 * This header is BSD licensed so anyone can use the definitions
 * to implement compatible drivers/servers.
 *
 * Copyright 2007, 2009, IBM Corporation
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. Neither the name of IBM nor the names of its contributors
 *    may be used to endorse or promote products derived from this software
 *    without specific prior written permission.
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ``AS IS''
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL IBM OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 */
```

```
/* This marks a buffer as continuing via the next field. */
#define VRING_DESC_F_NEXT       1
/* This marks a buffer as write-only (otherwise read-only). */
#define VRING_DESC_F_WRITE      2

/* The Host uses this in used->flags to advise the Guest: don't kick me
 * when you add a buffer.  It's unreliable, so it's simply an
 * optimization.  Guest will still kick if it's out of buffers. */
#define VRING_USED_F_NO_NOTIFY  1
/* The Guest uses this in avail->flags to advise the Host: don't
 * interrupt me when you consume a buffer.  It's unreliable, so it's
 * simply an optimization.  */
#define VRING_AVAIL_F_NO_INTERRUPT      1

/* Virtio ring descriptors: 16 bytes.
 * These can chain together via "next". */
struct vring_desc {
        /* Address (guest-physical). */
        uint64_t addr;
        /* Length. */
        uint32_t len;
        /* The flags as indicated above. */
        uint16_t flags;
        /* We chain unused descriptors via this, too */
        uint16_t next;
};

struct vring_avail {
        uint16_t flags;
        uint16_t idx;
        uint16_t ring[];
};

/* u32 is used here for ids for padding reasons. */
struct vring_used_elem {
        /* Index of start of used descriptor chain. */
        uint32_t id;
        /* Total length of the descriptor chain which was written to. */
        uint32_t len;
};

struct vring_used {
        uint16_t flags;
        uint16_t idx;
        struct vring_used_elem ring[];
};
```

```
struct vring {
        unsigned int num;

        struct vring_desc *desc;
        struct vring_avail *avail;
        struct vring_used *used;
};

/* The standard layout for the ring is a continuous chunk of memory which
 * looks like this.  We assume num is a power of 2.
 *
 * struct vring {
 *      // The actual descriptors (16 bytes each)
 *      struct vring_desc desc[num];
 *
 *      // A ring of available descriptor heads with free-running index.
 *      __u16 avail_flags;
 *      __u16 avail_idx;
 *      __u16 available[num];
 *
 *      // Padding to the next align boundary.
 *      char pad[];
 *
 *      // A ring of used descriptor heads with free-running index.
 *      __u16 used_flags;
 *      __u16 used_idx;
 *      struct vring_used_elem used[num];
 * };
 * Note: for virtio PCI, align is 4096.
 */
static inline void vring_init(struct vring *vr, unsigned int num, void *p,
                              unsigned long align)
{
        vr->num = num;
        vr->desc = p;
        vr->avail = p + num*sizeof(struct vring_desc);
        vr->used = (void *)(((unsigned long)&vr->avail->ring[num]
                              + align -1)
                            & ~(align - 1));
}

static inline unsigned vring_size(unsigned int num, unsigned long align)
{
        return ((sizeof(struct vring_desc)*num + sizeof(uint16_t)*(2+num)
                + align - 1) & ~(align - 1))
                + sizeof(uint16_t)*2 + sizeof(struct vring_used_elem)*num;
}
#endif /* VIRTIO_RING_H */
```

Appendix B: Reserved Feature Bits

Currently there are three device-independent feature bits defined:

VIRTIO_F_NOTIFY_ON_EMPTY (24) Negotiating this feature indicates that the driver wants an interrupt if the device runs out of available descriptors on a virtqueue, even though interrupts are suppressed using the VRING_AVAIL_F_NO_INTERRUPT flag. An example of this is the networking driver: it doesn't need to know every time a packet is transmitted, but it does need to free the transmitted packets a finite time after they are transmitted. It can avoid using a timer if the device interrupts it when all the packets are transmitted.

VIRTIO_F_RING_INDIRECT_DESC (28) Negotiating this feature indicates that the driver can use descriptors with the VRING_DESC_F_INDIRECT flag set, as described in 2.3.2.

VIRTIO_F_BAD_FEATURE(30) This feature should never be negotiated by the guest; doing so is an indication that the guest is faulty[4]

[4] An experimental virtio PCI driver contained in Linux version 2.6.25 had this problem, and this feature bit can be used to detect it.

Appendix C: Network Device

The virtio network device is a virtual ethernet card, and is the most complex of the devices supported so far by virtio. It has enhanced rapidly and demonstrates clearly how support for new features should be added to an existing device. Empty buffers are placed in one virtqueue for receiving packets, and outgoing packets are enqueued into another for transmission in that order. A third command queue is used to control advanced filtering features.

Configuration

Subsystem Device ID 1

Virtqueues 0:receiveq. 1:transmitq. 2:controlq[5]

Feature bits

- **VIRTIO_NET_F_CSUM (0)** Device handles packets with partial checksum
- **VIRTIO_NET_F_GUEST_CSUM (1)** Guest handles packets with partial checksum
- **VIRTIO_NET_F_MAC (5)** Device has given MAC address.
- **VIRTIO_NET_F_GSO (6)** (Deprecated) device handles packets with any GSO type.[6]
- **VIRTIO_NET_F_GUEST_TSO4 (7)** Guest can receive TSOv4.
- **VIRTIO_NET_F_GUEST_TSO6 (8)** Guest can receive TSOv6.
- **VIRTIO_NET_F_GUEST_ECN (9)** Guest can receive TSO with ECN.
- **VIRTIO_NET_F_GUEST_UFO (10)** Guest can receive UFO.
- **VIRTIO_NET_F_HOST_TSO4 (11)** Device can receive TSOv4.

[5] Only if VIRTIO_NET_F_CTRL_VQ set

[6] It was supposed to indicate segmentation offload support, but upon further investigation it became clear that multiple bits were required.

VIRTIO_NET_F_HOST_TSO6 (12) Device can receive TSOv6.

VIRTIO_NET_F_HOST_ECN (13) Device can receive TSO with ECN.

VIRTIO_NET_F_HOST_UFO (14) Device can receive UFO.

VIRTIO_NET_F_MRG_RXBUF (15) Guest can merge receive buffers.

VIRTIO_NET_F_STATUS (16) Configuration status field is available.

VIRTIO_NET_F_CTRL_VQ (17) Control channel is available.

VIRTIO_NET_F_CTRL_RX (18) Control channel RX mode support.

VIRTIO_NET_F_CTRL_VLAN (19) Control channel VLAN filtering.

Device configuration layout Two configuration fields are currently defined. The mac address field always exists (though is only valid if VIRTIO_NET_F_MAC is set), and the status field only exists if VIRTIO_NET_F_STATUS is set. Only one bit is currently defined for the status field: VIRTIO_NET_S_LINK_UP.

```
#define VIRTIO_NET_S_LINK_UP     1

struct virtio_net_config {
    u8 mac[6];
    u16 status;
};
```

Device Initialization

1. The initialization routine should identify the receive and transmission virtqueues.

2. If the VIRTIO_NET_F_MAC feature bit is set, the configuration space "mac" entry indicates the "physical" address of the the network card, otherwise a private MAC address should be assigned. All guests are expected to negotiate this feature if it is set.

3. If the VIRTIO_NET_F_CTRL_VQ feature bit is negotiated, identify the control virtqueue.

4. If the VIRTIO_NET_F_STATUS feature bit is negotiated, the link status can be read from the bottom bit of the "status" config field. Otherwise, the link should be assumed active.

5. The receive virtqueue should be filled with receive buffers. This is described in detail below in “Setting Up Receive Buffers”.

6. A driver can indicate that it will generate checksumless packets by negotating the VIRTIO_NET_F_CSUM feature. This “checksum offload” is a common feature on modern network cards.

7. If that feature is negotiated, a driver can use TCP or UDP segmentation offload by negotiating the VIRTIO_NET_F_HOST_TSO4 (IPv4 TCP), VIRTIO_NET_F_HOST_TSO6 (IPv6 TCP) and VIRTIO_NET_F_HOST_UFO (UDP fragmentation) features. It should not send TCP packets requiring segmentation offload which have the Explicit Congestion Notification bit set, unless the VIRTIO_NET_F_HOST_ECN feature is negotiated.[7]

8. The converse features are also available: a driver can save the virtual device some work by negotiating these features.[8] The VIRTIO_NET_F_GUEST_CSUM feature indicates that partially checksummed packets can be received, and if it can do that then the VIRTIO_NET_F_GUEST_TSO4, VIRTIO_NET_F_GUEST_TSO6, VIRTIO_NET_F_GUEST_UFO and VIRTIO_NET_F_GUEST_ECN are the input equivalents of the features described above. See “Receiving Packets” below.

Device Operation

Packets are transmitted by placing them in the transmitq, and buffers for incoming packets are placed in the receiveq. In each case, the packet itself is preceeded by a header:

```
struct virtio_net_hdr {
#define VIRTIO_NET_HDR_F_NEEDS_CSUM    1
        u8 flags;
#define VIRTIO_NET_HDR_GSO_NONE        0
#define VIRTIO_NET_HDR_GSO_TCPV4       1
#define VIRTIO_NET_HDR_GSO_UDP           3
#define VIRTIO_NET_HDR_GSO_TCPV6       4
#define VIRTIO_NET_HDR_GSO_ECN      0x80
        u8 gso_type;
        u16 hdr_len;
        u16 gso_size;
        u16 csum_start;
        u16 csum_offset;
/* Only if VIRTIO_NET_F_MRG_RXBUF: */
        u16 num_buffers
```

[7] This is a common restriction in real, older network cards.

[8] For example, a network packet transported between two guests on the same system may not require checksumming at all, nor segmentation, if both guests are amenable.

```
};
```

The controlq is used to control device features such as filtering.

Packet Transmission

Transmitting a single packet is simple, but varies depending on the different features the driver negotiated.

1. If the driver negotiated VIRTIO_NET_F_CSUM, and the packet has not been fully checksummed, then the virtio_net_hdr's fields are set as follows. Otherwise, the packet must be fully checksummed, and flags is zero.

   - flags has the VIRTIO_NET_HDR_F_NEEDS_CSUM set,
   - csum_start is set to the offset within the packet to begin checksumming, and
   - csum_offset indicates how many bytes after the csum_start the new (16 bit ones' complement) checksum should be placed.[9]

2. If the driver negotiated VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO, and the packet requires TCP segmentation or UDP fragmentation, then the "gso_type" field is set to VIRTIO_NET_HDR_GSO_TCPV4, TCPV6 or UDP. (Otherwise, it is set to VIRTIO_NET_HDR_GSO_NONE). In this case, packets larger than 1514 bytes can be transmitted: the metadata indicates how to replicate the packet header to cut it into smaller packets. The other gso fields are set:

   - hdr_len is a hint to the device as to how much of the header needs to be kept to copy into each packet, usually set to the length of the headers, including the transport header.[10]
   - gso_size is the size of the packet beyond that header (ie. MSS).
   - If the driver negotiated the VIRTIO_NET_F_HOST_ECN feature, the VIRTIO_NET_HDR_GSO_ECN bit may be set in "gso_type" as well, indicating that the TCP packet has the ECN bit set.[11]

[9] For example, consider a partially checksummed TCP (IPv4) packet. It will have a 14 byte ethernet header and 20 byte IP header followed by the TCP header (with the TCP checksum field 16 bytes into that header). csum_start will be 14+20 = 34 (the TCP checksum includes the header), and csum_offset will be 16. The value in the TCP checksum field will be the sum of the TCP pseudo header, so that replacing it by the ones' complement checksum of the TCP header and body will give the correct result.

[10] Due to various bugs in implementations, this field is not useful as a guarantee of the transport header size.

[11] This case is not handled by some older hardware, so is called out specifically in the protocol.

3. If the driver negotiated the VIRTIO_NET_F_MRG_RXBUF feature, the num_buffers field is set to zero.
4. The header and packet are added as one output buffer to the transmitq, and the device is notified of the new entry (see 2.4.1.4).[12]

Packet Transmission Interrupt

Often a driver will suppress transmission interrupts using the VRING_AVAIL_F_NO_INTERRUPT flag (see 2.4.2) and check for used packets in the transmit path of following packets. However, it will still receive interrupts if the VIRTIO_F_NOTIFY_ON_EMPTY feature is negotiated, indicating that the transmission queue is completely emptied.

The normal behavior in this interrupt handler is to retrieve and new descriptors from the used ring and free the corresponding headers and packets.

Setting Up Receive Buffers

It is generally a good idea to keep the receive virtqueue as fully populated as possible: if it runs out, network performance will suffer.

If the VIRTIO_NET_F_GUEST_TSO4, VIRTIO_NET_F_GUEST_TSO6 or VIRTIO_NET_F_GUEST_UFO features are used, the Guest will need to accept packets of up to 65550 bytes long (the maximum size of a TCP or UDP packet, plus the 14 byte ethernet header), otherwise 1514 bytes. So unless VIRTIO_NET_F_MRG_RXBUF is negotiated, every buffer in the receive queue needs to be at least this length [13].

If VIRTIO_NET_F_MRG_RXBUF is negotiated, each buffer must be at least the size of the `struct virtio_net_hdr`.

Packet Receive Interrupt

When a packet is copied into a buffer in the receiveq, the optimal path is to disable further interrupts for the receiveq (see 2.4.2) and process packets until no more are found, then re-enable them.

Processing packet involves:

1. If the driver negotiated the VIRTIO_NET_F_MRG_RXBUF feature, then the “num_buffers” field indicates how many descriptors this packet

[12] Note that the header will be two bytes longer for the VIRTIO_NET_F_MRG_RXBUF case.

[13] Obviously each one can be split across multiple descriptor elements.

is spread over (including this one). This allows receipt of large packets without having to allocate large buffers. In this case, there will be at least “num_buffers” in the used ring, and they should be chained together to form a single packet. The other buffers will *not* begin with a `struct virtio_net_hdr`.

2. If the VIRTIO_NET_F_MRG_RXBUF feature was not negotiated, or the “num_buffers” field is one, then the entire packet will be contained within this buffer, immediately following the `struct virtio_net_hdr`.
3. If the VIRTIO_NET_F_GUEST_CSUM feature was negotiated, the VIRTIO_NET_HDR_F_NEEDS_CSUM bit in the “flags” field may be set: if so, the checksum on the packet is incomplete and the “csum_start” and “csum_offset” fields indicate how to calculate it (see 1).
4. If the VIRTIO_NET_F_GUEST_TSO4, TSO6 or UFO options were negotiated, then the “gso_type” may be something other than VIRTIO_NET_HDR_GSO_NONE, and the “gso_size” field indicates the desired MSS (see 2).Control Virtqueue

The driver uses the control virtqueue (if VIRTIO_NET_F_VTRL_VQ is negotiated) to send commands to manipulate various features of the device which would not easily map into the configuration space.

All commands are of the following form:

```
struct virtio_net_ctrl {
        u8 class;
        u8 command;
        u8 command-specific-data[];
        u8 ack;
};

/* ack values */
#define VIRTIO_NET_OK      0
#define VIRTIO_NET_ERR     1
```

The class, command and command-specific-data are set by the driver, and the device sets the ack byte. There is little it can do except issue a diagnostic if the ack byte is not VIRTIO_NET_OK.

Packet Receive Filtering

If the VIRTIO_NET_F_CTRL_RX feature is negotiated, the driver can send control commands for promiscuous mode, multicast receiving, and filtering of MAC addresses.

Note that in general, these commands are best-effort: unwanted packets may still arrive.

Setting Promiscuous Mode

```
#define VIRTIO_NET_CTRL_RX      0
 #define VIRTIO_NET_CTRL_RX_PROMISC       0
 #define VIRTIO_NET_CTRL_RX_ALLMULTI      1
```

The class VIRTIO_NET_CTRL_RX has two commands: VIRTIO_NET_CTRL_RX_PROMISC turns promiscuous mode on and off, and VIRTIO_NET_CTRL_RX_ALLMULTI turns all-multicast receive on and off. The command-specific-data is one byte containing 0 (off) or 1 (on).

Setting MAC Address Filtering

```
struct virtio_net_ctrl_mac {
        u32 entries;
        u8 macs[entries][ETH_ALEN];
};

#define VIRTIO_NET_CTRL_MAC     1
 #define VIRTIO_NET_CTRL_MAC_TABLE_SET          0
```

The device can filter incoming packets by any number of destination MAC addresses.[14] This table is set using the class VIRTIO_NET_CTRL_MAC and the command VIRTIO_NET_CTRL_MAC_TABLE_SET. The command-specific-data is two variable length tables of 6-byte MAC addresses. The first table contains unicast addresses, and the second contains multicast addresses.

VLAN Filtering

If the driver negotiates the VIRTION_NET_F_CTRL_VLAN feature, it can control a VLAN filter table in the device.

```
#define VIRTIO_NET_CTRL_VLAN          2
 #define VIRTIO_NET_CTRL_VLAN_ADD                  0
 #define VIRTIO_NET_CTRL_VLAN_DEL                  1
```

Both the VIRTIO_NET_CTRL_VLAN_ADD and VIRTIO_NET_CTRL_VLAN_DEL command take a 16-bit VLAN id as the command-specific-data.

[14] Since there are no guarentees, it can use a hash filter orsilently switch to allmulti or promiscuous mode if it is given too many addresses.

Appendix D: Block Device

The virtio block device is a simple virtual block device (ie. disk). Read and write requests (and other exotic requests) are placed in the queue, and serviced (probably out of order) by the device except where noted.

Configuration

Subsystem Device ID 2

Virtqueues 0:requestq.

Feature bits

VIRTIO_BLK_F_BARRIER (0) Host supports request barriers.

VIRTIO_BLK_F_SIZE_MAX (1) Maximum size of any single segment is in "size_max".

VIRTIO_BLK_F_SEG_MAX (2) Maximum number of segments in a request is in "seg_max".

VIRTIO_BLK_F_GEOMETRY (4) Disk-style geometry specified in "geometry".

VIRTIO_BLK_F_RO (5) Device is read-only.

VIRTIO_BLK_F_BLK_SIZE (6) Block size of disk is in "blk_size".

VIRTIO_BLK_F_SECTOR_MAX (10) Maximum total sectors in an I/O.

Device configuration layout The capacity of the device (expressed in 512-byte sectors) is always present. The availability of the others all depend on various feature bits as indicated above.

```
struct virtio_blk_config {
        u64 capacity;
        u32 size_max;
        u32 seg_max;
        struct virtio_blk_geometry {
                u16 cylinders;
                u8 heads;
                u8 sectors;
        } geometry;
        u32 blk_size;
        u32 sectors_max;
};
```

Device Initialization

1. The device size should be read from the “capacity” configuration field. No requests should be submitted which goes beyond this limit.

2. If the VIRTIO_BLK_F_BLK_SIZE feature is negotiated, the blk_size field can be read to determine the optimal sector size for the driver to use. This does not effect the units used in the protocol (always 512 bytes), but awareness of the correct value can effect performance.

3. If the VIRTIO_BLK_F_RO feature is set by the device, any write requests will fail.

4. If the VIRTIO_BLK_F_SECTOR_MAX feature is negotiated, the sectors_max field should be read to determine the maximum I/O size for the driver to use. No requests should be submitted which go beyond this limit.

Device Operation

The driver queues requests to the virtqueue, and they are used by the device (not necessarily in order). Each request is of form:

```
struct virtio_blk_req {
#define VIRTIO_BLK_T_IN          0
#define VIRTIO_BLK_T_OUT         1
#define VIRTIO_BLK_T_BARRIER     0x80000000
        u32 type;
        u32 ioprio;
        u64 sector;
        char data[][512];
#define VIRTIO_BLK_S_OK        0
#define VIRTIO_BLK_S_IOERR     1
        u8 status;
};
```

The type of the request is either a read (VIRTIO_BLK_T_IN) or a write (VIRTIO_BLK_T_OUT); the high bit indicates that this request acts as a barrier and that all preceeding requests must be complete before this one, and all following requests must not be started until this is complete.

The ioprio field is a hint about the relative priorities of requests to the device: higher numbers indicate more important requests.

The sector number indicates the offset (multiplied by 512) where the read or write is to occur.

Note that these first three fields are always read-only: the data field is either read-only or write-only, depending on the type of the request. The size of the read or write can be derived from the total size of the request buffer.

The final byte is written by the device: either VIRTIO_BLK_S_OK or VIRTIO_BLK_S_IOERR.

Appendix E: Console Device

The virtio console device is a simple device for data input and output. A device may have one or more ports. Each port has a pair of input and output virtqueues. Moreover, a device has a pair of control IO virtqueues. The control virtqueues are used to communicate information between the device and the driver about ports being opened and closed on either side of the connection, indication from the host about whether a particular port is a console port, port hotunplug, etc. For data IO, one or more empty buffers are placed in the receive queue for incoming data and outgoing characters are placed in the transmit queue.

Configuration

Subsystem Device ID 3

Virtqueues 0:receiveq(port0). 1:transmitq(port0), 2:control receiveq, 3:control transmitq, 4:receiveq(port1), 5:transmitq(port1), ...

Feature bits

- **VIRTIO_CONSOLE_F_SIZE (0)** Configuration cols and rows fields are valid.
- **VIRTIO_CONSOLE_F_MULTIPORT(1)** Device has support for multiple ports; configuration fields nr_ports and max_nr_ports are valid and control virtqueues will be used.

Device configuration layout The size of the console is supplied in the configuration space if the VIRTIO_CONSOLE_F_SIZE feature is set. Furthermore, if the VIRTIO_CONSOLE_F_MULTIPORT feature is set, the maximum number of ports supported by the device and the number of ports currently active can be fetched.

```
struct virtio_console_config {
        u16 cols;
        u16 rows;
        u32 max_nr_ports;
        u32 nr_ports;
};
```

Device Initialization

1. If the VIRTIO_CONSOLE_F_SIZE feature is negotiated, the driver can read the console dimensions from the configuration fields.

2. If the VIRTIO_CONSOLE_F_MULTIPORT feature is negotiated, the driver can spawn multiple ports, not all of which may be attached to a console. Some could be generic ports. In this case, the control virtqueues are enabled and according to the nr_ports configuration-space value, the appropriate number of ports and corresponding virtqueues are spawned. After creating and initializing each port, a VIRTIO_CONSOLE_PORT_READY control message is sent to the host for that port so the host can let us know of any additional configuration options set for that port.

3. The receiveq for each port is populated with one or more receive buffers.

Device Operation

1. For output, a buffer containing the characters is placed in the port's transmitq.[15]

2. When a buffer is used in the receiveq (signalled by an interrupt), the contents is the input to the port associated with the virtqueue for which the notification was received.

3. If the driver negotiated the VIRTIO_CONSOLE_F_SIZE feature, a configuration change interrupt may occur. The updated size can be read from the configuration fields.

4. If the driver negotiated the VIRTIO_CONSOLE_F_MULTIPORT feature, a configuration change interrupt may occur that bumps up the value of nr_ports. In this case, ports are hot-added.

5. Changes to ports' state are effected by control messages. Appropriate action is taken on the port indicated in the control message. The layout of the structure of the control buffer and the events associated are:

[15] Because this is high importance and low bandwidth, the current Linux implementation polls for the buffer to be used, rather than waiting for an interrupt, simplifying the implementation significantly.

```
struct virtio_console_control {
        uint32_t id;     /* Port number */
        uint16_t event; /* The kind of control event */
        uint16_t value; /* Extra information for the event */
};

/* Some events for the internal messages (control packets) */
#define VIRTIO_CONSOLE_PORT_READY        0
#define VIRTIO_CONSOLE_CONSOLE_PORT      1
#define VIRTIO_CONSOLE_RESIZE            2
#define VIRTIO_CONSOLE_PORT_OPEN         3
#define VIRTIO_CONSOLE_PORT_NAME         4
#define VIRTIO_CONSOLE_PORT_REMOVE       5
```

Appendix F: Entropy Device

The virtio entropy device supplies high-quality randomness for guest use.

Configuration

Subsystem Device ID 4

Virtqueues 0:requestq.

Feature bits None currently defined

Device configuration layout None currently defined.

Device Initialization

1. The virtqueue is initialized

Device Operation

When the driver requires random bytes, it places the descriptor of one or more buffers in the queue. It will be completely filled by random data by the device.

Appendix G: Memory Balloon Device

The virtio memory balloon device is a primitive device for managing guest memory: the device asks for a certain amount of memory, and the guest supplies it (or withdraws it, if the device has more than it asks for). This allows the guest to adapt to changes in allowance of underlying physical memory. If the feature is negotiated, the device can also be used to communicate guest memory statistics to the host.

Configuration

Subsystem Device ID 5

Virtqueues 0:inflateq. 1:deflateq. 2:statsq.[16]

Feature bits

- **VIRTIO_BALLOON_F_MUST_TELL_HOST (0)** Host must be told before pages from the balloon are used.
- **VIRTIO_BALLOON_F_STATS_VQ (1)** A virtqueue for reporting guest memory statistics is present.

Device configuration layout Both fields of this configuration are always available. Note that they are little endian, despite convention that device fields are guest endian:

```
struct virtio_console_config {
        u32 num_pages;
        u32 actual;
};
```

[16] Only if VIRTIO_BALLON_F_STATS_VQ set

Device Initialization

1. The inflate and deflate virtqueues are identified.
2. If the VIRTIO_BALLOON_F_STATS_VQ feature bit is negotiated:
   (a) Identify the stats virtqueue.
   (b) Add one empty buffer to the stats virtqueue and notify the host.

Device operation begins immediately.

Device Operation

Memory Ballooning The device is driven by the receipt of a configuration change interrupt.

1. The "num_pages" configuration field is examined. If this is greater than the "actual" number of pages, memory must be given to the balloon. If it is less than the "actual" number of pages, memory may be taken back from the balloon for general use.
2. To supply memory to the balloon (aka. inflate):
   (a) The driver constructs an array of addresses of unused memory pages. These addresses are divided by 4096[17] and the descriptor describing the resulting 32-bit array is added to the inflateq.
3. To remove memory from the balloon (aka. deflate):
   (a) The driver constructs an array of addresses of memory pages it has previously given to the balloon, as described above. This descriptor is added to the deflateq.
   (b) If the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is set, the guest may not use these requested pages until that descriptor in the deflateq has been used by the device.
   (c) Otherwise, the guest may begin to re-use pages previously given to the balloon before the device has acknowledged their withdrawl. [18]
4. In either case, once the device has completed the inflation or deflation, the "actual" field of the configuration should be updated to reflect the new number of pages in the balloon.[19]

[17] This is historical, and independent of the guest page size

[18] In this case, deflation advice is merely a courtesy

[19] As updates to configuration space are not atomic, this field isn't particularly reliable, but can be used to diagnose buggy guests.

Memory Statistics

The stats virtqueue is atypical because communication is driven by the device (not the driver). The channel becomes active at driver initialization time when the driver adds an empty buffer and notifies the device. A request for memory statistics proceeds as follows:

1. The device pushes the buffer onto the used ring and sends an interrupt.
2. The driver pops the used buffer and discards it.
3. The driver collects memory statistics and writes them into a new buffer.
4. The driver adds the buffer to the virtqueue and notifies the device.
5. The device pops the buffer (retaining it to initiate a subsequent request) and consumes the statistics.

Memory Statistics Format Each statistic consists of a 16 bit tag and a 64 bit value. Both quantities are represented in the native endian of the guest. All statistics are optional and the driver may choose which ones to supply. To guarantee backwards compatibility, unsupported statistics should be omitted.

```
struct virtio_balloon_stat {
#define VIRTIO_BALLOON_S_SWAP_IN   0
#define VIRTIO_BALLOON_S_SWAP_OUT  1
#define VIRTIO_BALLOON_S_MAJFLT    2
#define VIRTIO_BALLOON_S_MINFLT    3
#define VIRTIO_BALLOON_S_MEMFREE   4
#define VIRTIO_BALLOON_S_MEMTOT    5
        u16 tag;
        u64 val;
} __attribute__((packed));
```

Tags

VIRTIO_BALLOON_S_SWAP_IN The amount of memory that has been swapped in (in bytes).

VIRTIO_BALLOON_S_SWAP_OUT The amount of memory that has been swapped out to disk (in bytes).

VIRTIO_BALLOON_S_MAJFLT The number of major page faults that have occurred.

VIRTIO_BALLOON_S_MINFLT The number of minor page faults that have occurred.

VIRTIO_BALLOON_S_MEMFREE The amount of memory not being used for any purpose (in bytes).

VIRTIO_BALLOON_S_MEMTOT The total amount of memory available (in bytes).