Virtual I/O Device (VIRTIO) Version 1.0

Committee Specification 02

18 January 2015

Specification URIs

This version:
http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/tex/ (Authoritative)
http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/virtio-v1.0-cs02.pdf
http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/virtio-v1.0-cs02.html

Previous version:
http://docs.oasis-open.org/virtio/virtio/v1.0/cs01/tex/ (Authoritative)
http://docs.oasis-open.org/virtio/virtio/v1.0/cs01/virtio-v1.0-cs01.pdf
http://docs.oasis-open.org/virtio/virtio/v1.0/cs01/virtio-v1.0-cs01.html

Latest version:
http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf
http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html

Technical Committee:
OASIS Virtual I/O Device (VIRTIO) TC

Chairs:
Rusty Russell (rusty@au.ibm.com), IBM
Michael S. Tsirkin (mst@redhat.com), Red Hat

Editors:
Rusty Russell (rusty@au.ibm.com), IBM
Michael S. Tsirkin (mst@redhat.com), Red Hat
Cornelia Huck (cornelia.huck@de.ibm.com), IBM
Pawel Moll (pawel.moll@arm.com), ARM

Additional artifacts:
This prose specification is one component of a Work Product that also includes:

- Example Driver Listing:
  http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/listings/

Related work:
This specification replaces or supersedes:

- Virtio PCI Card Specification Version 0.9.5:
  http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf

Abstract:

This document describes the specifications of the “virtio” family of devices. These devices are found in virtual environments, yet by design they look like physical devices to the guest within the virtual machine - and this document treats them as such. This similarity allows the guest to use standard 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.

Status:

This document was last revised or approved by the Virtual I/O Device (VIRTIO) TC on the above date. The level of approval is also listed above. Check the “Latest version” location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=virtio#technical.

Technical Committee members should send comments on this specification to the Technical Committee’s email list. Others should send comments to the Technical Committee by using the “Send A Comment” button on the Technical Committee’s web page at https://www.oasis-open.org/committees/virtio/.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (https://www.oasis-open.org/committees/virtio/ipr.php).

Citation format:

When referencing this specification the following citation format should be used:

[VIRTIO-v1.0]

Virtual I/O Device (VIRTIO) Version 1.0. Edited by Rusty Russell, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. 18 January 2015. OASIS Committee Specification 02. http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/virtio-v1.0-cs02.html. Latest version: http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html.

Notices

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name ”OASIS” is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

1 Introduction

This document describes the specifications of the “virtio” family of devices. These devices are found in virtual environments, yet by design they look like physical devices to the guest within the virtual machine - and this document treats them as such. This similarity allows the guest to use standard 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 devices use normal bus 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 normal device.[1]

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

Standard: Virtio makes no assumptions about the environment in which it operates, beyond supporting the bus to which device is attached. In this specification, virtio devices are implemented over MMIO, Channel I/O and PCI bus transports [2], earlier drafts have been implemented on other buses not included here.

Extensible: Virtio 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 Normative References

[RFC2119]	Bradner S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt
[S390 PoP]	z/Architecture Principles of Operation, IBM Publication SA22-7832, http://publibfi.boulder.ibm.com/epubs/pdf/dz9zr009.pdf, and any future revisions
[S390 Common I/O]	ESA/390 Common I/O-Device and Self-Description, IBM Publication SA22-7204, http://publibfp.dhe.ibm.com/cgi-bin/bookmgr/BOOKS/dz9ar501/CCONTENTS, and any future revisions
[PCI]	Conventional PCI Specifications, http://www.pcisig.com/specifications/conventional/, PCI-SIG
[PCIe]	PCI Express Specifications http://www.pcisig.com/specifications/pciexpress/, PCI-SIG
[IEEE 802]	IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture, http://standards.ieee.org/about/get/802/802.html, IEEE

---

[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 virtio transport code from the specific virtio drivers: these drivers are shared between different transports.

[SAM]	SCSI Architectural Model, http://www.t10.org/cgi-bin/ac.pl?t=f&f=sam4r05.pdf
[SCSI MMC]	SCSI Multimedia Commands, http://www.t10.org/cgi-bin/ac.pl?t=f&f=mmc6r00.pdf

1.2 Non-Normative References

[Virtio PCI Draft]	Virtio PCI Draft Specification http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf

1.3 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

1.3.1 Legacy Interface: Terminology

Earlier drafts of this specification (i.e. revisions before 1.0, see e.g. [Virtio PCI Draft]) defined a similar, but different interface between the driver and the device. Since these are widely deployed, this specification accommodates OPTIONAL features to simplify transition from these earlier draft interfaces.

Specifically devices and drivers MAY support:

Legacy Interface is an interface specified by an earlier draft of this specification (before 1.0)

Legacy Device is a device implemented before this specification was released, and implementing a legacy interface on the host side

Legacy Driver is a driver implemented before this specification was released, and implementing a legacy interface on the guest side

Legacy devices and legacy drivers are not compliant with this specification.

To simplify transition from these earlier draft interfaces, a device MAY implement:

Transitional Device a device supporting both drivers conforming to this specification, and allowing legacy drivers.

Similarly, a driver MAY implement:

Transitional Driver a driver supporting both devices conforming to this specification, and legacy devices.

Note: Legacy interfaces are not required; ie. don’t implement them unless you have a need for backwards compatibility!

Devices or drivers with no legacy compatibility are referred to as non-transitional devices and drivers, respectively.

1.4 Structure Specifications

Many device and driver in-memory structure layouts are documented using the C struct syntax. All structures are assumed to be without additional padding. To stress this, cases where common C compilers are known to insert extra padding within structures are tagged using the GNU C __attribute__((packed)) syntax.

For the integer data types used in the structure definitions, the following conventions are used:

u8, u16, u32, u64 An unsigned integer of the specified length in bits.

le16, le32, le64 An unsigned integer of the specified length in bits, in little-endian byte order.

be16, be32, be64 An unsigned integer of the specified length in bits, in big-endian byte order.

2 Basic Facilities of a Virtio Device

A virtio device is discovered and identified by a bus-specific method (see the bus specific sections: 4.1 Virtio Over PCI Bus, 4.2 Virtio Over MMIO and 4.3 Virtio Over Channel I/O). Each device consists of the following parts:

- Device status field
- Feature bits
- Device Configuration space
- One or more virtqueues

2.1 *Device Status* Field

During device initialization by a driver, the driver follows the sequence of steps specified in 3.1.

The *device status* field provides a simple low-level indication of the completed steps of this sequence. It's most useful to imagine it hooked up to traffic lights on the console indicating the status of each device. The following bits are defined:

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.

Note: There could be a significant (or infinite) delay before setting this bit. For example, under Linux, drivers can be loadable modules.

FEATURES_OK (8) Indicates that the driver has acknowledged all the features it understands, and feature negotiation is complete.

DRIVER_OK (4) Indicates that the driver is set up and ready to drive the device.

DEVICE_NEEDS_RESET (64) Indicates that the device has experienced an error from which it can't recover.

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.

2.1.1 Driver Requirements: Device Status Field

The driver MUST update *device status*, setting bits to indicate the completed steps of the driver initialization sequence specified in 3.1. The driver MUST NOT clear a *device status* bit. If the driver sets the FAILED bit, the driver MUST later reset the device before attempting to re-initialize.

The driver SHOULD NOT rely on completion of operations of a device if DEVICE_NEEDS_RESET is set.

Note: For example, the driver can't assume requests in flight will be completed if DEVICE_NEEDS_RESET is set, nor can it assume that they have not been completed. A good implementation will try to recover by issuing a reset.

2.1.2 Device Requirements: Device Status Field

The device MUST initialize *device status* to 0 upon reset.

The device MUST NOT consume buffers or notify the driver before DRIVER_OK.

The device SHOULD set DEVICE_NEEDS_RESET when it enters an error state that a reset is needed. If DRIVER_OK is set, after it sets DEVICE_NEEDS_RESET, the device MUST send a device configuration change notification to the driver.

2.2 Feature Bits

Each virtio device offers all the features it understands. During device initialization, the driver reads this and tells the device the subset that it accepts. The only way to renegotiate is to reset the device.

This allows for forwards and backwards compatibility: if the device is enhanced with a new feature bit, older drivers will not write that feature bit back to the device. Similarly, if a driver is enhanced with a feature that the device doesn't support, it see the new feature is not offered.

Feature bits are allocated as follows:

0 to 23 Feature bits for the specific device type

24 to 32 Feature bits reserved for extensions to the queue and feature negotiation mechanisms

33 and above Feature bits reserved for future extensions.

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

In particular, new fields in the device configuration space are indicated by offering a new feature bit.

2.2.1 Driver Requirements: Feature Bits

The driver MUST NOT accept a feature which the device did not offer, and MUST NOT accept a feature which requires another feature which was not accepted.

The driver SHOULD go into backwards compatibility mode if the device does not offer a feature it understands, otherwise MUST set the FAILED *device status* bit and cease initialization.

2.2.2 Device Requirements: Feature Bits

The device MUST NOT offer a feature which requires another feature which was not offered. The device SHOULD accept any valid subset of features the driver accepts, otherwise it MUST fail to set the FEATURES_OK *device status* bit when the driver writes it.

2.2.3 Legacy Interface: A Note on Feature Bits

Transitional Drivers MUST detect Legacy Devices by detecting that the feature bit VIRTIO_F_VERSION_1 is not offered. Transitional devices MUST detect Legacy drivers by detecting that VIRTIO_F_VERSION_1 has not been acknowledged by the driver.

In this case device is used through the legacy interface.

Legacy interface support is OPTIONAL. Thus, both transitional and non-transitional devices and drivers are compliant with this specification.

Requirements pertaining to transitional devices and drivers is contained in sections named 'Legacy Interface' like this one.

When device is used through the legacy interface, transitional devices and transitional drivers MUST operate according to the requirements documented within these legacy interface sections. Specification text within these sections generally does not apply to non-transitional devices.

2.3 Device Configuration Space

Device configuration space is generally used for rarely-changing or initialization-time parameters. Where configuration fields are optional, their existence is indicated by feature bits: Future versions of this specification will likely extend the device configuration space by adding extra fields at the tail.

Note: The device configuration space uses the little-endian format for multi-byte fields.

Each transport also provides a generation count for the device configuration space, which will change whenever there is a possibility that two accesses to the device configuration space can see different versions of that space.

2.3.1 Driver Requirements: Device Configuration Space

Drivers MUST NOT assume reads from fields greater than 32 bits wide are atomic, nor are reads from multiple fields: drivers SHOULD read device configuration space fields like so:

```
u32 before, after;
do {
        before = get_config_generation(device);
        // read config entry/entries.
        after = get_config_generation(device);
} while (after != before);
```

For optional configuration space fields, the driver MUST check that the corresponding feature is offered before accessing that part of the configuration space.

Note: See section 3.1 for details on feature negotiation.

Drivers MUST NOT limit structure size and device configuration space size. Instead, drivers SHOULD only check that device configuration space is *large enough* to contain the fields necessary for device operation.

Note: For example, if the specification states that device configuration space 'includes a single 8-bit field' drivers should understand this to mean that the device configuration space might also include an arbitrary amount of tail padding, and accept any device configuration space size equal to or greater than the specified 8-bit size.

2.3.2 Device Requirements: Device Configuration Space

The device MUST allow reading of any device-specific configuration field before FEATURES_OK is set by the driver. This includes fields which are conditional on feature bits, as long as those feature bits are offered by the device.

2.3.3 Legacy Interface: A Note on Device Configuration Space endian-ness

Note that for legacy interfaces, device configuration space is generally the guest's native endian, rather than PCI's little-endian. The correct endian-ness is documented for each device.

2.3.4 Legacy Interface: Device Configuration Space

Legacy devices did not have a configuration generation field, thus are susceptible to race conditions if configuration is updated. This affects the block *capacity* (see 5.2.4) and network *mac* (see 5.1.4) fields; when using the legacy interface, drivers SHOULD read these fields multiple times until two reads generate a consistent result.

2.4 Virtqueues

The mechanism for bulk data transport on virtio devices is pretentiously called a virtqueue. Each device can have zero or more virtqueues[1]. Each queue has a 16-bit queue size parameter, which sets the number of entries and implies the total size of the queue.

Each virtqueue consists of three parts:

- Descriptor Table
- Available Ring
- Used Ring

where each part is physically-contiguous in guest memory, and has different alignment requirements.

The memory aligment and size requirements, in bytes, of each part of the virtqueue are summarized in the following table:

Virtqueue Part	Alignment	Size
Descriptor Table	16	$16*$(Queue Size)
Available Ring	2	$6+2*$(Queue Size)
Used Ring	4	$6+8*$(Queue Size)

The Alignment column gives the minimum alignment for each part of the virtqueue.

The Size column gives the total number of bytes for each part of the virtqueue.

Queue Size corresponds to the maximum number of buffers in the virtqueue[2]. Queue Size value is always a power of 2. The maximum Queue Size value is 32768. This value is specified in a bus-specific way.

When the driver wants to send a buffer to the device, it fills in a slot in the descriptor table (or chains several together), and writes the descriptor index into the available ring. It then notifies the device. When the device has finished a buffer, it writes the descriptor index into the used ring, and sends an interrupt.

2.4.1 Driver Requirements: Virtqueues

The driver MUST ensure that the physical address of the first byte of each virtqueue part is a multiple of the specified alignment value in the above table.

2.4.2 Legacy Interfaces: A Note on Virtqueue Layout

For Legacy Interfaces, several additional restrictions are placed on the virtqueue layout:

Each virtqueue occupies two or more physically-contiguous pages (usually defined as 4096 bytes, but depending on the transport) and consists of three parts:

Descriptor Table	Available Ring (…padding…)	Used Ring

---

[1] For example, the simplest network device has one virtqueue for transmit and one for receive.

[2] For example, if Queue Size is 4 then at most 4 buffers can be queued at any given time.

The bus-specific Queue Size field controls the total number of bytes for the virtqueue. When using the legacy interface, the transitional driver MUST retrieve the Queue Size field from the device and MUST allocate the total number of bytes for the virtuqueue according to the following formula:

```
#define ALIGN(x) (((x) + PAGE_SIZE) & ~PAGE_SIZE)
static inline unsigned virtq_size(unsigned int qsz)
{
     return ALIGN(sizeof(struct virtq_desc)*qsz + sizeof(u16)*(3 + qsz))
          + ALIGN(sizeof(u16)*3 + sizeof(struct virtq_used_elem)*qsz);
}
```

This wastes some space with padding. When using the legacy interface, both transitional devices and drivers MUST use the following virtqueue layout structure to locate elements of the virtqueue:

```
struct virtq {
        // The actual descriptors (16 bytes each)
        struct virtq_desc desc[ Queue Size ];

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

        // Padding to the next PAGE_SIZE boundary.
        u8 pad[ Padding ];

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

2.4.3 Legacy Interfaces: A Note on Virtqueue Endianness

Note that when using the legacy interface, transitional devices and drivers MUST use the native endian of the guest as the endian of fields and in the virtqueue. This is opposed to little-endian for non-legacy interface as specified by this standard. It is assumed that the host is already aware of the guest endian.

2.4.4 Message Framing

The framing of messages with descriptors is independent of the contents of the buffers. For example, a network transmit buffer consists of a 12 byte header followed by the network packet. This could be most simply placed in the descriptor table as a 12 byte output descriptor followed by a 1514 byte output descriptor, but it could also consist of a single 1526 byte output descriptor in the case where the header and packet are adjacent, or even three or more descriptors (possibly with loss of efficiency in that case).

Note that, some device implementations have large-but-reasonable restrictions on total descriptor size (such as based on IOV_MAX in the host OS). This has not been a problem in practice: little sympathy will be given to drivers which create unreasonably-sized descriptors such as by dividing a network packet into 1500 single-byte descriptors!

2.4.4.1 Device Requirements: Message Framing

The device MUST NOT make assumptions about the particular arrangement of descriptors. The device MAY have a reasonable limit of descriptors it will allow in a chain.

2.4.4.2 Driver Requirements: Message Framing

The driver MUST place any device-writable descriptor elements after any device-readable descriptor elements.

The driver SHOULD NOT use an excessive number of descriptors to describe a buffer.

2.4.4.3 Legacy Interface: Message Framing

Regrettably, initial driver implementations used simple layouts, and devices came to rely on it, despite this specification wording. In addition, the specification for virtio_blk SCSI commands required intuiting field lengths from frame boundaries (see 5.2.6.3 Legacy Interface: Device Operation)

Thus when using the legacy interface, the VIRTIO_F_ANY_LAYOUT feature indicates to both the device and the driver that no assumptions were made about framing. Requirements for transitional drivers when this is not negotiated are included in each device section.

2.4.5 The Virtqueue Descriptor Table

The descriptor table refers to the buffers the driver is using for the device. *addr* is a physical address, and the buffers can be chained via *next*. Each descriptor describes a buffer which is read-only for the device (“device-readable”) or write-only for the device (“device-writable”), but a chain of descriptors can contain both device-readable and device-writable buffers.

The actual contents of the memory offered to the device depends on the device type. Most common is to begin the data with a header (containing little-endian fields) for the device to read, and postfix it with a status tailer for the device to write.

```
struct virtq_desc {
        /* Address (guest-physical). */
        le64 addr;
        /* Length. */
        le32 len;

/* This marks a buffer as continuing via the next field. */
#define VIRTQ_DESC_F_NEXT   1
/* This marks a buffer as device write-only (otherwise device read-only). */
#define VIRTQ_DESC_F_WRITE     2
/* This means the buffer contains a list of buffer descriptors. */
#define VIRTQ_DESC_F_INDIRECT   4
        /* The flags as indicated above. */
        le16 flags;
        /* Next field if flags & NEXT */
        le16 next;
};
```

The number of descriptors in the table is defined by the queue size for this virtqueue: this is the maximum possible descriptor chain length.

Note: The legacy [Virtio PCI Draft] referred to this structure as vring_desc, and the constants as VRING_-DESC_F_NEXT, etc, but the layout and values were identical.

2.4.5.1 Device Requirements: The Virtqueue Descriptor Table

A device MUST NOT write to a device-readable buffer, and a device SHOULD NOT read a device-writable buffer (it MAY do so for debugging or diagnostic purposes).

2.4.5.2 Driver Requirements: The Virtqueue Descriptor Table

Drivers MUST NOT add a descriptor chain over than 2^{32} bytes long in total; this implies that loops in the descriptor chain are forbidden!

2.4.5.3 Indirect Descriptors

Some devices benefit by concurrently dispatching a large number of large requests. The VIRTIO_F_INDIRECT_DESC feature allows this (see A virtio_ring.h). To increase ring capacity the driver can store a table of indirect descriptors anywhere in memory, and insert a descriptor in main virtqueue (with *flags*&VIRTQ_DESC_F_INDIRECT on) that refers to memory buffer containing this indirect descriptor table; *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 virtq_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*. An indirect descriptor without a valid *next* (with *flags*&VIRTQ_DESC_F_NEXT off) signals the end of the descriptor. A single indirect descriptor table can include both device-readable and device-writable descriptors.

2.4.5.3.1 Driver Requirements: Indirect Descriptors

The driver MUST NOT set the VIRTQ_DESC_F_INDIRECT flag unless the VIRTIO_F_INDIRECT_DESC feature was negotiated. The driver MUST NOT set the VIRTQ_DESC_F_INDIRECT flag within an indirect descriptor (ie. only one table per descriptor).

A driver MUST NOT create a descriptor chain longer than the Queue Size of the device.

2.4.5.3.2 Device Requirements: Indirect Descriptors

The device MUST ignore the write-only flag (*flags*&VIRTQ_DESC_F_WRITE) in the descriptor that refers to an indirect table.

2.4.6 The Virtqueue Available Ring

```
struct virtq_avail {
#define VIRTQ_AVAIL_F_NO_INTERRUPT      1
        le16 flags;
        le16 idx;
        le16 ring[ /* Queue Size */ ];
        le16 used_event; /* Only if VIRTIO_F_EVENT_IDX */
};
```

The driver uses the available ring to offer buffers to the device: each ring entry refers to the head of a descriptor chain. It is only written by the driver and read by the device.

idx field indicates where the driver would put the next descriptor entry in the ring (modulo the queue size). This starts at 0, and increases.

Note: The legacy [Virtio PCI Draft] referred to this structure as vring_avail, and the constant as VRING_AVAIL_F_NO_INTERRUPT, but the layout and value were identical.

2.4.7 Virtqueue Interrupt Suppression

If the VIRTIO_F_EVENT_IDX feature bit is not negotiated, the *flags* field in the available ring offers a crude mechanism for the driver to inform the device that it doesn't want interrupts when buffers are used. Otherwise *used_event* is a more performant alterative where the driver specifies how far the device can progress before interrupting.

Neither of these interrupt suppression methods are reliable, as they are not synchronized with the device, but they serve as useful optimizations.

2.4.7.1 Driver Requirements: Virtqueue Interrupt Suppression

If the VIRTIO_F_EVENT_IDX feature bit is not negotiated:

- The driver MUST set *flags* to 0 or 1.
- The driver MAY set *flags* to 1 to advise the device that interrupts are not needed.

Otherwise, if the VIRTIO_F_EVENT_IDX feature bit is negotiated:

- The driver MUST set *flags* to 0.
- The driver MAY use *used_event* to advise the device that interrupts are unnecessary until the device writes entry with an index specified by *used_event* into the used ring (equivalently, until *idx* in the used ring will reach the value *used_event* + 1).

The driver MUST handle spurious interrupts from the device.

2.4.7.2 Device Requirements: Virtqueue Interrupt Suppression

If the VIRTIO_F_EVENT_IDX feature bit is not negotiated:

- The device MUST ignore the *used_event* value.
- After the device writes a descriptor index into the used ring:
  - If *flags* is 1, the device SHOULD NOT send an interrupt.
  - If *flags* is 0, the device MUST send an interrupt.

Otherwise, if the VIRTIO_F_EVENT_IDX feature bit is negotiated:

- The device MUST ignore the lower bit of *flags*.
- After the device writes a descriptor index into the used ring:
  - If the *idx* field in the used ring (which determined where that descriptor index was placed) was equal to *used_event*, the device MUST send an interrupt.
  - Otherwise the device SHOULD NOT send an interrupt.

Note: For example, if *used_event* is 0, then a device using VIRTIO_F_EVENT_IDX would interrupt after the first buffer is used (and again after the 65536th buffer, etc).

2.4.8 The Virtqueue Used Ring

```
struct virtq_used {
#define VIRTQ_USED_F_NO_NOTIFY  1
        le16 flags;
        le16 idx;
        struct virtq_used_elem ring[ /* Queue Size */];
        le16 avail_event; /* Only if VIRTIO_F_EVENT_IDX */
};

/* le32 is used here for ids for padding reasons. */
struct virtq_used_elem {
        /* Index of start of used descriptor chain. */
        le32 id;
        /* Total length of the descriptor chain which was used (written to) */
        le32 len;
};
```

The used ring is where the device returns buffers once it is done with them: it is only written to by the device, and read by the driver.

Each entry in the ring is a pair: *id* indicates the head entry of the descriptor chain describing the buffer (this matches an entry placed in the available ring by the guest earlier), and *len* the total of bytes written into the buffer. The latter is extremely useful for drivers 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.

Note: The legacy [Virtio PCI Draft] referred to these structures as vring_used and vring_used_elem, and the constant as VRING_USED_F_NO_NOTIFY, but the layout and value were identical.

2.4.9 Virtqueue Notification Suppression

The device can suppress notifications in a manner analogous to the way drivers can suppress interrupts as detailed in section 2.4.7. The device manipulates *flags* or *avail_event* in the used ring the same way the driver manipulates *flags* or *used_event* in the available ring.

2.4.9.1 Driver Requirements: Virtqueue Notification Suppression

The driver MUST initialize *flags* in the used ring to 0 when allocating the used ring.

If the VIRTIO_F_EVENT_IDX feature bit is not negotiated:

- The driver MUST ignore the *avail_event* value.
- After the driver writes a descriptor index into the available ring:
    - If *flags* is 1, the driver SHOULD NOT send a notification.
    - If *flags* is 0, the driver MUST send a notification.

Otherwise, if the VIRTIO_F_EVENT_IDX feature bit is negotiated:

- The driver MUST ignore the lower bit of *flags*.
- After the driver writes a descriptor index into the available ring:
    - If the *idx* field in the available ring (which determined where that descriptor index was placed) was equal to *avail_event*, the driver MUST send a notification.
    - Otherwise the driver SHOULD NOT send a notification.

2.4.9.2 Device Requirements: Virtqueue Notification Suppression

If the VIRTIO_F_EVENT_IDX feature bit is not negotiated:

- The device MUST set *flags* to 0 or 1.
- The device MAY set *flags* to 1 to advise the driver that notifications are not needed.

Otherwise, if the VIRTIO_F_EVENT_IDX feature bit is negotiated:

- The device MUST set *flags* to 0.
- The device MAY use *avail_event* to advise the driver that notifications are unnecessary until the driver writes entry with an index specified by *avail_event* into the available ring (equivalently, until *idx* in the available ring will reach the value *avail_event* + 1).

The device MUST handle spurious notifications from the driver.

2.4.10 Helpers for Operating Virtqueues

The Linux Kernel Source code contains the definitions above and helper routines in a more usable form, in include/uapi/linux/virtio_ring.h. This was explicitly licensed by IBM and Red Hat 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 A virtio_ring.h.

3 General Initialization And Device Operation

We start with an overview of device initialization, then expand on the details of the device and how each step is preformed. This section is best read along with the bus-specific section which describes how to communicate with the specific device.

3.1 Device Initialization

3.1.1 Driver Requirements: Device Initialization

The driver MUST follow this sequence to initialize a device:

1. Reset the device.
2. Set the ACKNOWLEDGE status bit: the guest OS has notice the device.
3. Set the DRIVER status bit: the guest OS knows how to drive the device.
4. Read device feature bits, and write the subset of feature bits understood by the OS and driver to the device. During this step the driver MAY read (but MUST NOT write) the device-specific configuration fields to check that it can support the device before accepting it.
5. Set the FEATURES_OK status bit. The driver MUST NOT accept new feature bits after this step.
6. Re-read *device status* to ensure the FEATURES_OK bit is still set: otherwise, the device does not support our subset of features and the device is unusable.
7. Perform device-specific setup, including discovery of virtqueues for the device, optional per-bus setup, reading and possibly writing the device's virtio configuration space, and population of virtqueues.
8. Set the DRIVER_OK status bit. At this point the device is "live".

If any of these steps go irrecoverably wrong, the driver 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). The driver MUST NOT continue initialization in that case.

The driver MUST NOT notify the device before setting DRIVER_OK.

3.1.2 Legacy Interface: Device Initialization

Legacy devices did not support the FEATURES_OK status bit, and thus did not have a graceful way for the device to indicate unsupported feature combinations. They also did not provide a clear mechanism to end feature negotiation, which meant that devices finalized features on first-use, and no features could be introduced which radically changed the initial operation of the device.

Legacy driver implementations often used the device before setting the DRIVER_OK bit, and sometimes even before writing the feature bits to the device.

The result was the steps 5 and 6 were omitted, and steps 4, 7 and 8 were conflated.

Therefore, when using the legacy interface:

- The transitional driver MUST execute the initialization sequence as described in 3.1 but omitting the steps 5 and 6.

- The transitional device MUST support the driver writing device configuration fields before the step 4.
- The transitional device MUST support the driver using the device before the step 8.

3.2 Device Operation

There are two parts to device operation: supplying new buffers to the device, and processing used buffers from the device.

Note: As an example, the simplest virtio network device has two virtqueues: the transmit virtqueue and the receive virtqueue. The driver adds outgoing (device-readable) packets to the transmit virtqueue, and then frees them after they are used. Similarly, incoming (device-writable) buffers are added to the receive virtqueue, and processed after they are used.

3.2.1 Supplying Buffers to The Device

The driver offers buffers to one of the device's virtqueues as follows:

1. The driver places the buffer into free descriptor(s) in the descriptor table, chaining as necessary (see 2.4.5 The Virtqueue Descriptor Table).
2. The driver places the index of the head of the descriptor chain into the next ring entry of the available ring.
3. Steps 1 and 2 MAY be performed repeatedly if batching is possible.
4. The driver performs suitable a memory barrier to ensure the device sees the updated descriptor table and available ring before the next step.
5. The available *idx* is increased by the number of descriptor chain heads added to the available ring.
6. The driver performs a suitable memory barrier to ensure that it updates the *idx* field before checking for notification suppression.
7. If notifications are not suppressed, the driver notifies the device of the new available 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 (the highest power of 2 which fits in 16 bits), so the 16-bit *idx* value can always distinguish between a full and empty buffer.

What follows is the requirements of each stage in more detail.

3.2.1.1 Placing Buffers Into The Descriptor Table

A buffer consists of zero or more device-readable physically-contiguous elements followed by zero or more physically-contiguous device-writable elements (each has at least one element). This algorithm maps it into the descriptor table to form a descriptor chain:

for each buffer element, b:

1. Get the next free descriptor table entry, d
2. Set *d.addr* to the physical address of the start of b
3. Set *d.len* to the length of b.
4. If b is device-writable, set *d.flags* to VIRTQ_DESC_F_WRITE, otherwise 0.
5. If there is a buffer element after this:

(a) Set *d.next* to the index of the next free descriptor element.

(b) Set the VIRTQ_DESC_F_NEXT bit in *d.flags*.

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

3.2.1.2 Updating The Available Ring

The descriptor chain head is the first d in the algorithm above, ie. the index of the descriptor table entry referring to the first part of the buffer. A naive driver implementation MAY do the following (with the appropriate conversion to-and-from little-endian assumed):

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

However, in general the driver MAY add many descriptor chains before it updates *idx* (at which point they become visible to the device), so it is common to keep a counter of how many the driver has added:

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

3.2.1.3 Updating *idx*

idx always increments, and wraps naturally at 65536:

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

Once available *idx* is updated by the driver, this exposes the descriptor and its contents. The device MAY access the descriptor chains the driver created and the memory they refer to immediately.

3.2.1.3.1 Driver Requirements: Updating idx

The driver MUST perform a suitable memory barrier before the *idx* update, to ensure the device sees the most up-to-date copy.

3.2.1.4 Notifying The Device

The actual method of device notification is bus-specific, but generally it can be expensive. So the device MAY suppress such notifications if it doesn't need them, as detailed in section 2.4.9.

The driver has to be careful to expose the new *idx* value before checking if notifications are suppressed.

3.2.1.4.1 Driver Requirements: Notifying The Device

The driver MUST perform a suitable memory barrier before reading *flags* or *avail_event*, to avoid missing a notification.

3.2.2 Receiving Used Buffers From The Device

Once the device has used buffers referred to by a descriptor (read from or written to them, or parts of both, depending on the nature of the virtqueue and the device), it interrupts the driver as detailed in section 2.4.7.

Note: For optimal performance, a driver MAY disable interrupts while processing the used ring, but beware the problem of missing interrupts between emptying the ring and reenabling interrupts. This is usually handled by re-checking for more used buffers after interrups are re-enabled:

```
virtq_disable_interrupts(vq);

for (;;) {
        if (vq->last_seen_used != le16_to_cpu(virtq->used.idx)) {
                virtq_enable_interrupts(vq);
                mb();

                if (vq->last_seen_used != le16_to_cpu(virtq->used.idx))
                        break;

                virtq_disable_interrupts(vq);
        }

        struct virtq_used_elem *e = virtq.used->ring[vq->last_seen_used%vsz];
        process_buffer(e);
        vq->last_seen_used++;
}
```

3.2.3 Notification of Device Configuration Changes

For devices where the device-specific configuration information can be changed, an interrupt is delivered when a device-specific configuration change occurs.

In addition, this interrupt is triggered by the device setting DEVICE_NEEDS_RESET (see 2.1.2).

3.3 Device Cleanup

Once the driver has set the DRIVER_OK status bit, all the configured virtqueue of the device are considered live. None of the virtqueues of a device are live once the device has been reset.

3.3.1 Driver Requirements: Device Cleanup

A driver MUST NOT alter descriptor table entries which have been exposed in the available ring (and not marked consumed by the device in the used ring) of a live virtqueue.

A driver MUST NOT decrement the available *idx* on a live virtqueue (ie. there is no way to “unexpose” buffers).

Thus a driver MUST ensure a virtqueue isn’t live (by device reset) before removing exposed buffers.

4 Virtio Transport Options

Virtio can use various different buses, thus the standard is split into virtio general and bus-specific sections.

4.1 Virtio Over PCI Bus

Virtio devices are commonly implemented as PCI devices.

A Virtio device can be implemented as any kind of PCI device: a Conventional PCI device or a PCI Express device. To assure designs meet the latest level requirements, see the PCI-SIG home page at http://www.pcisig.com for any approved changes.

4.1.1 Device Requirements: Virtio Over PCI Bus

A Virtio device using Virtio Over PCI Bus MUST expose to guest an interface that meets the specification requirements of the appropriate PCI specification: [PCI] and [PCIe] respectively.

4.1.2 PCI Device Discovery

Any PCI device with PCI Vendor ID 0x1AF4, and PCI Device ID 0x1000 through 0x107F inclusive is a virtio device. The actual value within this range indicates which virtio device is supported by the device. The PCI Device ID is calculated by adding 0x1040 to the Virtio Device ID, as indicated in section 5. Additionally, devices MAY utilize a Transitional PCI Device ID range, 0x1000 to 0x103F depending on the device type.

4.1.2.1 Device Requirements: PCI Device Discovery

Devices MUST have the PCI Vendor ID 0x1AF4. Devices MUST either have the PCI Device ID calculated by adding 0x1040 to the Virtio Device ID, as indicated in section 5 or have the Transitional PCI Device ID depending on the device type, as follows:

Transitional PCI Device ID	Virtio Device
0x1000	network card
0x1001	block device
0x1002	memory ballooning (legacy)
0x1003	console
0x1004	SCSI host
0x1005	entropy source
0x1009	9P transport

For example, the network card device with the Virtio Device ID 1 has the PCI Device ID 0x1041 or the Transitional PCI Device ID 0x1000.

The PCI Subsystem Vendor ID and the PCI Subsystem Device ID MAY reflect the PCI Vendor and Device ID of the environment (for informational purposes by the driver).

Non-transitional devices SHOULD have a PCI Device ID in the range 0x1040 to 0x107f. Non-transitional devices SHOULD have a PCI Revision ID of 1 or higher. Non-transitional devices SHOULD have a PCI Subsystem Device ID of 0x40 or higher.

This is to reduce the chance of a legacy driver attempting to drive the device.

4.1.2.2 Driver Requirements: PCI Device Discovery

Drivers MUST match devices with the PCI Vendor ID 0x1AF4 and the PCI Device ID in the range 0x1040 to 0x107f, calculated by adding 0x1040 to the Virtio Device ID, as indicated in section 5. Drivers for device types listed in section 4.1.2 MUST match devices with the PCI Vendor ID 0x1AF4 and the Transitional PCI Device ID indicated in section 4.1.2.

Drivers MUST match any PCI Revision ID value. Drivers MAY match any PCI Subsystem Vendor ID and any PCI Subsystem Device ID value.

4.1.2.3 Legacy Interfaces: A Note on PCI Device Discovery

Transitional devices MUST have a PCI Revision ID of 0. Transitional devices MUST have the PCI Subsystem Device ID matching the Virtio Device ID, as indicated in section 5. Transitional devices MUST have the Transitional PCI Device ID in the range 0x1000 to 0x103f.

This is to match legacy drivers.

4.1.3 PCI Device Layout

The device is configured via I/O and/or memory regions (though see 4.1.4.7 for access via the PCI configuration space), as specified by Virtio Structure PCI Capabilities.

Fields of different sizes are present in the device configuration regions. All 32-bit and 16-bit fields are little-endian.

4.1.3.1 Driver Requirements: PCI Device Layout

The driver MUST access each field using the “natural” access method, i.e. 32-bit accesses for 32-bit fields, 16-bit accesses for 16-bit fields and 8-bit accesses for 8-bit fields.

4.1.4 Virtio Structure PCI Capabilities

The virtio device configuration layout includes several structures:

- Common configuration
- Notifications
- ISR Status
- Device-specific configuration (optional)

Each structure can be mapped by a Base Address register (BAR) belonging to the function, or accessed via the special VIRTIO_PCI_CAP_PCI_CFG field in the PCI configuration space.

The location of each structure is specified using a vendor-specific PCI capability located on the capability list in PCI configuration space of the device. This virtio structure capability uses little-endian format; all fields are read-only for the driver unless stated otherwise:

```
struct virtio_pci_cap {
        u8 cap_vndr;    /* Generic PCI field: PCI_CAP_ID_VNDR */
        u8 cap_next;    /* Generic PCI field: next ptr. */
        u8 cap_len;     /* Generic PCI field: capability length */
        u8 cfg_type;    /* Identifies the structure. */
        u8 bar;         /* Where to find it. */
        u8 padding[3];  /* Pad to full dword. */
        le32 offset;    /* Offset within bar. */
        le32 length;    /* Length of the structure, in bytes. */
};
```

This structure can be followed by extra data, depending on *cfg_type*, as documented below.

The fields are interpreted as follows:

cap_vndr 0x09; Identifies a vendor-specific capability.

cap_next Link to next capability in the capability list in the PCI configuration space.

cap_len Length of this capability structure, including the whole of struct virtio_pci_cap, and extra data if any. This length MAY include padding, or fields unused by the driver.

cfg_type identifies the structure, according to the following table:

```
/* Common configuration */
#define VIRTIO_PCI_CAP_COMMON_CFG        1
/* Notifications */
#define VIRTIO_PCI_CAP_NOTIFY_CFG        2
/* ISR Status */
#define VIRTIO_PCI_CAP_ISR_CFG           3
/* Device specific configuration */
#define VIRTIO_PCI_CAP_DEVICE_CFG        4
/* PCI configuration access */
#define VIRTIO_PCI_CAP_PCI_CFG           5
```

Any other value is reserved for future use.

Each structure is detailed individually below.

The device MAY offer more than one structure of any type - this makes it possible for the device to expose multiple interfaces to drivers. The order of the capabilities in the capability list specifies the order of preference suggested by the device.

Note: For example, on some hypervisors, notifications using IO accesses are faster than memory accesses. In this case, the device would expose two capabilities with *cfg_type* set to VIRTIO_PCI_CAP_NOTIFY_CFG: the first one addressing an I/O BAR, the second one addressing a memory BAR. In this example, the driver would use the I/O BAR if I/O resources are available, and fall back on memory BAR when I/O resources are unavailable.

bar values 0x0 to 0x5 specify a Base Address register (BAR) belonging to the function located beginning at 10h in PCI Configuration Space and used to map the structure into Memory or I/O Space. The BAR is permitted to be either 32-bit or 64-bit, it can map Memory Space or I/O Space.

Any other value is reserved for future use.

offset indicates where the structure begins relative to the base address associated with the BAR. The alignment requirements of *offset* are indicated in each structure-specific section below.

length indicates the length of the structure.

length MAY include padding, or fields unused by the driver, or future extensions.

Note: For example, a future device might present a large structure size of several MBytes. As current devices never utilize structures larger than 4KBytes in size, driver MAY limit the mapped structure size to e.g. 4KBytes (thus ignoring parts of structure after the first 4KBytes) to allow forward compatibility with such devices without loss of functionality and without wasting resources.

4.1.4.1 Driver Requirements: Virtio Structure PCI Capabilities

The driver MUST ignore any vendor-specific capability structure which has a reserved *cfg_type* value.

The driver SHOULD use the first instance of each virtio structure type they can support.

The driver MUST accept a *cap_len* value which is larger than specified here.

The driver MUST ignore any vendor-specific capability structure which has a reserved *bar* value.

The drivers SHOULD only map part of configuration structure large enough for device operation. The drivers MUST handle an unexpectedly large *length*, but MAY check that *length* is large enough for device operation.

The driver MUST NOT write into any field of the capability structure, with the exception of those with *cap_type* VIRTIO_PCI_CAP_PCI_CFG as detailed in 4.1.4.7.2.

4.1.4.2 Device Requirements: Virtio Structure PCI Capabilities

The device MUST include any extra data (from the beginning of the *cap_vndr* field through end of the extra data fields if any) in *cap_len*. The device MAY append extra data or padding to any structure beyond that.

If the device presents multiple structures of the same type, it SHOULD order them from optimal (first) to least-optimal (last).

4.1.4.3 Common configuration structure layout

The common configuration structure is found at the *bar* and *offset* within the VIRTIO_PCI_CAP_COMMON_CFG capability; its layout is below.

```
struct virtio_pci_common_cfg {
        /* About the whole device. */
        le32 device_feature_select;     /* read-write */
        le32 device_feature;            /* read-only for driver */
        le32 driver_feature_select;     /* read-write */
        le32 driver_feature;            /* read-write */
        le16 msix_config;               /* read-write */
        le16 num_queues;                /* read-only for driver */
        u8 device_status;               /* read-write */
        u8 config_generation;           /* read-only for driver */

        /* About a specific virtqueue. */
        le16 queue_select;              /* read-write */
        le16 queue_size;                /* read-write, power of 2, or 0. */
        le16 queue_msix_vector;         /* read-write */
        le16 queue_enable;              /* read-write */
        le16 queue_notify_off;          /* read-only for driver */
        le64 queue_desc;                /* read-write */
        le64 queue_avail;               /* read-write */
        le64 queue_used;                /* read-write */
};
```

device_feature_select The driver uses this to select which feature bits *device_feature* shows. Value 0x0 selects Feature Bits 0 to 31, 0x1 selects Feature Bits 32 to 63, etc.

device_feature The device uses this to report which feature bits it is offering to the driver: the driver writes to *device_feature_select* to select which feature bits are presented.

driver_feature_select The driver uses this to select which feature bits *driver_feature* shows. Value 0x0 selects Feature Bits 0 to 31, 0x1 selects Feature Bits 32 to 63, etc.

driver_feature The driver writes this to accept feature bits offered by the device. Driver Feature Bits selected by *driver_feature_select*.

config_msix_vector The driver sets the Configuration Vector for MSI-X.

num_queues The device specifies the maximum number of virtqueues supported here.

device_status The driver writes the device status here (see 2.1). Writing 0 into this field resets the device.

config_generation Configuration atomicity value. The device changes this every time the configuration noticeably changes.

queue_select Queue Select. The driver selects which virtqueue the following fields refer to.

queue_size Queue Size. On reset, specifies the maximum queue size supported by the hypervisor. This can be modified by driver to reduce memory requirements. A 0 means the queue is unavailable.

queue_msix_vector The driver uses this to specify the queue vector for MSI-X.

queue_enable The driver uses this to selectively prevent the device from executing requests from this virtqueue. 1 - enabled; 0 - disabled.

queue_notify_off The driver reads this to calculate the offset from start of Notification structure at which this virtqueue is located.

Note: this is *not an offset in bytes. See 4.1.4.4 below.*

queue_desc The driver writes the physical address of Descriptor Table here. See section 2.4.

queue_avail The driver writes the physical address of Available Ring here. See section 2.4.

queue_used The driver writes the physical address of Used Ring here. See section 2.4.

4.1.4.3.1 Device Requirements: Common configuration structure layout

offset MUST be 4-byte aligned.

The device MUST present at least one common configuration capability.

The device MUST present the feature bits it is offering in *device_feature*, starting at bit *device_feature_select* ∗ 32 for any *device_feature_select* written by the driver.

Note: This means that it will present 0 for any *device_feature_select* other than 0 or 1, since no feature defined here exceeds 63.

The device MUST present any valid feature bits the driver has written in *driver_feature*, starting at bit *driver_feature_select* ∗ 32 for any *driver_feature_select* written by the driver. Valid feature bits are those which are subset of the corresponding *device_feature* bits. The device MAY present invalid bits written by the driver.

Note: This means that a device can ignore writes for feature bits it never offers, and simply present 0 on reads. Or it can just mirror what the driver wrote (but it will still have to check them when the driver sets FEATURES_OK).

Note: A driver shouldn't write invalid bits anyway, as per 3.1.1, but this attempts to handle it.

The device MUST present a changed *config_generation* after the driver has read a device-specific configuration value which has changed since any part of the device-specific configuration was last read.

Note: As *config_generation* is an 8-bit value, simply incrementing it on every configuration change could violate this requirement due to wrap. Better would be to set an internal flag when it has changed, and if that flag is set when the driver reads from the device-specific configuration, increment *config_generation* and clear the flag.

The device MUST reset when 0 is written to *device_status*, and present a 0 in *device_status* once that is done.

The device MUST present a 0 in *queue_enable* on reset.

The device MUST present a 0 in *queue_size* if the virtqueue corresponding to the current *queue_select* is unavailable.

4.1.4.3.2 Driver Requirements: Common configuration structure layout

The driver MUST NOT write to *device_feature*, *num_queues*, *config_generation* or *queue_notify_off*.

The driver MUST NOT write a value which is not a power of 2 to *queue_size*.

The driver MUST configure the other virtqueue fields before enabling the virtqueue with *queue_enable*.

After writing 0 to *device_status*, the driver MUST wait for a read of *device_status* to return 0 before reinitializing the device.

The driver MUST NOT write a 0 to *queue_enable*.

4.1.4.4 Notification structure layout

The notification location is found using the VIRTIO_PCI_CAP_NOTIFY_CFG capability. This capability is immediately followed by an additional field, like so:

```
struct virtio_pci_notify_cap {
        struct virtio_pci_cap cap;
        le32 notify_off_multiplier; /* Multiplier for queue_notify_off. */
};
```

notify_off_multiplier is combined with the *queue_notify_off* to derive the Queue Notify address within a BAR for a virtqueue:

```
        cap.offset + queue_notify_off * notify_off_multiplier
```

The *cap.offset* and *notify_off_multiplier* are taken from the notification capability structure above, and the *queue_notify_off* is taken from the common configuration structure.

Note: For example, if *notifier_off_multiplier* is 0, the device uses the same Queue Notify address for all queues.

4.1.4.4.1 Device Requirements: Notification capability

The device MUST present at least one notification capability.

The *cap.offset* MUST be 2-byte aligned.

The device MUST either present *notify_off_multiplier* as an even power of 2, or present *notify_off_multiplier* as 0.

The value *cap.length* presented by the device MUST be at least 2 and MUST be large enough to support queue notification offsets for all supported queues in all possible configurations.

For all queues, the value *cap.length* presented by the device MUST satisfy:

```
cap.length >= queue_notify_off * notify_off_multiplier + 2
```

4.1.4.5 ISR status capability

The VIRTIO_PCI_CAP_ISR_CFG capability refers to at least a single byte, which contains the 8-bit ISR status field to be used for INT#x interrupt handling.

The *offset* for the *ISR status* has no alignment requirements.

The ISR bits allow the device to distinguish between device-specific configuration change interrupts and normal virtqueue interrupts:

Bits	0	1	2 to 31
Purpose	Device Configuration Interrupt	Queue Interrupt	Reserved

To avoid an extra access, simply reading this register resets it to 0 and causes the device to de-assert the interrupt.

In this way, driver read of ISR status causes the device to de-assert an interrupt.

See sections 4.1.5.3 and 4.1.5.4 for how this is used.

4.1.4.5.1 Device Requirements: ISR status capability

The device MUST present at least one VIRTIO_PCI_CAP_ISR_CFG capability.

The device MUST set the Device Configuration Interrupt bit in *ISR status* before sending a device configuration change notification to the driver.

If MSI-X capability is disabled, the device MUST set the Queue Interrupt bit in *ISR status* before sending a virtqueue notification to the driver.

If MSI-X capability is disabled, the device MUST set the Interrupt Status bit in the PCI Status register in the PCI Configuration Header of the device to the logical OR of all bits in *ISR status* of the device. The device then asserts/deasserts INT#x interrupts unless masked according to standard PCI rules [PCI].

The device MUST reset *ISR status* to 0 on driver read.

4.1.4.5.2 Driver Requirements: ISR status capability

If MSI-X capability is enabled, the driver SHOULD NOT access *ISR status* upon detecting a Queue Interrupt.

4.1.4.6 Device-specific configuration

The device MUST present at least one VIRTIO_PCI_CAP_DEVICE_CFG capability for any device type which has a device-specific configuration.

4.1.4.6.1 Device Requirements: Device-specific configuration

The *offset* for the device-specific configuration MUST be 4-byte aligned.

4.1.4.7 PCI configuration access capability

The VIRTIO_PCI_CAP_PCI_CFG capability creates an alternative (and likely suboptimal) access method to the common configuration, notification, ISR and device-specific configuration regions.

The capability is immediately followed by an additional field like so:

```
struct virtio_pci_cfg_cap {
        struct virtio_pci_cap cap;
        u8 pci_cfg_data[4]; /* Data for BAR access. */
};
```

The fields *cap.bar*, *cap.length*, *cap.offset* and *pci_cfg_data* are read-write (RW) for the driver.

To access a device region, the driver writes into the capability structure (ie. within the PCI configuration space) as follows:

- The driver sets the BAR to access by writing to *cap.bar*.
- The driver sets the size of the access by writing 1, 2 or 4 to *cap.length*.

- The driver sets the offset within the BAR by writing to *cap.offset*.

At that point, *pci_cfg_data* will provide a window of size *cap.length* into the given *cap.bar* at offset *cap.offset*.

4.1.4.7.1 Device Requirements: PCI configuration access capability

The device MUST present at least one VIRTIO_PCI_CAP_PCI_CFG capability.

Upon detecting driver write access to *pci_cfg_data*, the device MUST execute a write access at offset *cap.offset* at BAR selected by *cap.bar* using the first *cap.length* bytes from *pci_cfg_data*.

Upon detecting driver read access to *pci_cfg_data*, the device MUST execute a read access of length cap.length at offset *cap.offset* at BAR selected by *cap.bar* and store the first *cap.length* bytes in *pci_cfg_data*.

4.1.4.7.2 Driver Requirements: PCI configuration access capability

The driver MUST NOT write a *cap.offset* which is not a multiple of *cap.length* (ie. all accesses MUST be aligned).

4.1.4.8 Legacy Interfaces: A Note on PCI Device Layout

Transitional devices MUST present part of configuration registers in a legacy configuration structure in BAR0 in the first I/O region of the PCI device, as documented below. When using the legacy interface, transitional drivers MUST use the legacy configuration structure in BAR0 in the first I/O region of the PCI device, as documented below.

When using the legacy interface the driver MAY access the device-specific configuration region using any width accesses, and a transitional device MUST present driver with the same results as when accessed using the “natural” access method (i.e. 32-bit accesses for 32-bit fields, etc).

Note that this is possible because while the virtio common configuration structure is PCI (i.e. little) endian, when using the legacy interface the device-specific configuration region is encoded in the native endian of the guest (where such distinction is applicable).

When used through the legacy interface, the virtio common configuration structure 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 bits 0:31	Driver Features bits 0:31	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:

Bits	16	16
Read/Write	R+W	R+W
Purpose (MSI-X)	*config_msix_vector*	*queue_msix_vector*

Note: When MSI-X capability is enabled, device-specific configuration starts at byte offset 24 in virtio common configuration structure structure. When MSI-X capability is not enabled, device-specific configuration starts at byte offset 20 in virtio header. ie. once you enable MSI-X on the device, the other fields move. If you turn it off again, they move back!

Any device-specific configuration space immediately follows these general headers:

Bits	Device Specific	…
Read / Write	Device Specific	
Purpose	Device Specific	

When accessing the device-specific configuration space using the legacy interface, transitional drivers MUST access the device-specific configuration space at an offset immediately following the general headers.

When using the legacy interface, transitional devices MUST present the device-specific configuration space if any at an offset immediately following the general headers.

Note that only Feature Bits 0 to 31 are accessible through the Legacy Interface. When used through the Legacy Interface, Transitional Devices MUST assume that Feature Bits 32 to 63 are not acknowledged by Driver.

As legacy devices had no *config_generation* field, see 2.3.4 Legacy Interface: Device Configuration Space for workarounds.

4.1.4.9 Non-transitional Device With Legacy Driver: A Note on PCI Device Layout

Non-transitional devices, on a platform where a legacy driver for a legacy device with the same ID might have previously existed, SHOULD take the following steps to fail gracefully when a legacy driver attempts to drive them:

1. Present an I/O BAR in BAR0, and
2. Respond to a single-byte zero write to offset 18 (corresponding to Device Status register in the legacy layout) of BAR0 by presenting zeroes on every BAR and ignoring writes.

4.1.5 PCI-specific Initialization And Device Operation

4.1.5.1 Device Initialization

This documents PCI-specific steps executed during Device Initialization.

4.1.5.1.1 Virtio Device Configuration Layout Detection

As a prerequisite to device initialization, the driver scans the PCI capability list, detecting virtio configuration layout using Virtio Structure PCI capabilities as detailed in 4.1.4

4.1.5.1.2 Non-transitional Device With Legacy Driver

4.1.5.1.2.1 Driver Requirements: Non-transitional Device With Legacy Driver

Non-transitional devices, on a platform where a legacy driver for a legacy device with the same ID might have previously existed, MUST take the following steps to fail gracefully when a legacy driver attempts to drive them:

1. Present an I/O BAR in BAR0, and
2. Respond to a single-byte zero write to offset 18 (corresponding to Device Status register in the legacy layout) of BAR0 by presenting zeroes on every BAR and ignoring writes.

4.1.5.1.2.2 Legacy Interface: A Note on Device Layout Detection

Legacy drivers skipped the Device Layout Detection step, assuming legacy device configuration space in BAR0 in I/O space unconditionally.

Legacy devices did not have the Virtio PCI Capability in their capability list.

Therefore:

Transitional devices MUST expose the Legacy Interface in I/O space in BAR0.

Transitional drivers MUST look for the Virtio PCI Capabilities on the capability list. If these are not present, driver MUST assume a legacy device, and use it through the legacy interface.

Non-transitional drivers MUST look for the Virtio PCI Capabilities on the capability list. If these are not present, driver MUST assume a legacy device, and fail gracefully.

4.1.5.1.3 MSI-X Vector Configuration

When MSI-X capability is present and enabled in the device (through standard PCI configuration space) *config_msix_vector* and *queue_msix_vector* are used to map configuration change and queue interrupts to MSI-X vectors. In this case, the ISR Status is unused.

Writing a valid MSI-X Table entry number, 0 to 0x7FF, to *config_msix_vector*/*queue_msix_vector* maps interrupts triggered by the configuration change/selected queue events respectively to the corresponding MSI-X vector. To disable interrupts for an event type, the driver unmaps this event by writing a special NO_VECTOR value:

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

Note that mapping an event to vector might require device to allocate internal device resources, and thus could fail.

4.1.5.1.3.1 Device Requirements: MSI-X Vector Configuration

A device that has an MSI-X capability SHOULD support at least 2 and at most 0x800 MSI-X vectors. Device MUST report the number of vectors supported in *Table Size* in the MSI-X Capability as specified in [PCI]. The device SHOULD restrict the reported MSI-X Table Size field to a value that might benefit system performance.

Note: For example, a device which does not expect to send interrupts at a high rate might only specify 2 MSI-X vectors.

Device MUST support mapping any event type to any valid vector 0 to MSI-X *Table Size*. Device MUST support unmapping any event type.

The device MUST return vector mapped to a given event, (NO_VECTOR if unmapped) on read of *config_msix_vector*/*queue_msix_vector*. The device MUST have all queue and configuration change events are unmapped upon reset.

Devices SHOULD NOT cause mapping an event to vector to fail unless it is impossible for the device to satisfy the mapping request. Devices MUST report mapping failures by returning the NO_VECTOR value when the relevant *config_msix_vector*/*queue_msix_vector* field is read.

4.1.5.1.3.2 Driver Requirements: MSI-X Vector Configuration

Driver MUST support device with any MSI-X Table Size 0 to 0x7FF. Driver MAY fall back on using INT#x interrupts for a device which only supports one MSI-X vector (MSI-X Table Size = 0).

Driver MAY intepret the Table Size as a hint from the device for the suggested number of MSI-X vectors to use.

Driver MUST NOT attempt to map an event to a vector outside the MSI-X Table supported by the device, as reported by *Table Size* in the MSI-X Capability.

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 MAY retry mapping with fewer vectors, disable MSI-X or report device failure.

4.1.5.1.4 Virtqueue Configuration

As a device can have zero or more virtqueues for bulk data transport[1], the driver needs to configure them as part of the device-specific configuration.

The driver typically does this as follows, for each virtqueue a device has:

1. Write the virtqueue index (first queue is 0) to *queue_select*.
2. Read the virtqueue size from *queue_size*. This controls how big the virtqueue is (see 2.4 Virtqueues). If this field is 0, the virtqueue does not exist.
3. Optionally, select a smaller virtqueue size and write it to *queue_size*.
4. Allocate and zero Descriptor Table, Available and Used rings for the virtqueue in contiguous physical memory.
5. 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 into *queue_msix_vector*. Read *queue_msix_vector*: on success, previously written value is returned; on failure, NO_VECTOR value is returned.

4.1.5.1.4.1 Legacy Interface: A Note on Virtqueue Configuration

When using the legacy interface, the page size for a virtqueue on a PCI virtio device is defined as 4096 bytes. Driver writes the physical address, divided by 4096 to the Queue Address field[2]. There was no mechanism to negotiate the queue size.

4.1.5.2 Notifying The Device

The driver notifies the device by writing the 16-bit virtqueue index of this virtqueue to the Queue Notify address. See 4.1.4.4 for how to calculate this address.

4.1.5.3 Virtqueue Interrupts From The Device

If an interrupt is necessary for a virtqueue, the device would typically act as follows:

- If MSI-X capability is disabled:
  1. Set the lower bit of the ISR Status field for the device.
  2. Send the appropriate PCI interrupt for the device.
- If MSI-X capability is enabled:
  1. If *queue_msix_vector* is not NO_VECTOR, request the appropriate MSI-X interrupt message for the device, *queue_msix_vector* sets the MSI-X Table entry number.

---

[1] For example, the simplest network device has two virtqueues.

[2] 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.

4.1.5.3.1 Device Requirements: Virtqueue Interrupts From The Device

If MSI-X capability is enabled and *queue_msix_vector* is NO_VECTOR for a virtqueue, the device MUST NOT deliver an interrupt for that virtqueue.

4.1.5.4 Notification of Device Configuration Changes

Some virtio PCI devices can change the device configuration state, as reflected in the device-specific configuration region of the device. In this case:

- If MSI-X capability is disabled:
  1. Set the second lower bit of the ISR Status field for the device.
  2. Send the appropriate PCI interrupt for the device.
- If MSI-X capability is enabled:
  1. If *config_msix_vector* is not NO_VECTOR, request the appropriate MSI-X interrupt message for the device, *config_msix_vector* sets the MSI-X Table entry number.

A single interrupt MAY indicate both that one or more virtqueue has been used and that the configuration space has changed.

4.1.5.4.1 Device Requirements: Notification of Device Configuration Changes

If MSI-X capability is enabled and *config_msix_vector* is NO_VECTOR, the device MUST NOT deliver an interrupt for device configuration space changes.

4.1.5.4.2 Driver Requirements: Notification of Device Configuration Changes

A driver MUST handle the case where the same interrupt is used to indicate both device configuration space change and one or more virtqueues being used.

4.1.5.5 Driver Handling Interrupts

The driver interrupt handler would typically:

- If MSI-X capability is disabled:
  - Read the ISR Status field, which will reset it to zero.
  - If the lower bit is set: look through the used rings of all virtqueues for the device, to see if any progress has been made by the device which requires servicing.
  - If the second lower bit is set: re-examine the configuration space to see what changed.
- If MSI-X capability is enabled:
  - Look through the used rings of all virtqueues mapped to that MSI-X vector for the device, to see if any progress has been made by the device which requires servicing.
  - If the MSI-X vector is equal to *config_msix_vector*, re-examine the configuration space to see what changed.

4.2 Virtio Over MMIO

Virtual environments without PCI support (a common situation in embedded devices models) might use simple memory mapped device (“virtio-mmio”) instead of the PCI device.

The memory mapped virtio device behaviour is based on the PCI device specification. Therefore most operations including device initialization, queues configuration and buffer transfers are nearly identical. Existing differences are described in the following sections.

4.2.1 MMIO Device Discovery

Unlike PCI, MMIO provides no generic device discovery mechanism. For each device, the guest OS will need to know the location of the registers and interrupt(s) used. The suggested binding for systems using flattened device trees is shown in this example:

```
// EXAMPLE: virtio_block device taking 512 bytes at 0x1e000, interrupt 42.
virtio_block@1e000 {
        compatible = "virtio,mmio";
        reg = <0x1e000 0x200>;
        interrupts = <42>;
}
```

4.2.2 MMIO Device Register Layout

MMIO virtio devices provide a set of memory mapped control registers followed by a device-specific configuration space, described in the table 4.1.

All register values are organized as Little Endian.

Table 4.1: MMIO Device Register Layout

Name Offset from base Direction	**Function** Description
MagicValue 0x000 R	**Magic value** 0x74726976 (a Little Endian equivalent of the “virt” string).
Version 0x004 R	**Device version number** 0x2. **Note:** Legacy devices (see 4.2.4 Legacy interface) used 0x1.
DeviceID 0x008 R	**Virtio Subsystem Device ID** See 5 Device Types for possible values. Value zero (0x0) is used to define a system memory map with placeholder devices at static, well known addresses, assigning functions to them depending on user’s needs.
VendorID 0x00c R	**Virtio Subsystem Vendor ID**
DeviceFeatures 0x010 R	**Flags representing features the device supports** Reading from this register returns 32 consecutive flag bits, the least significant bit depending on the last value written to *DeviceFeaturesSel*. Access to this register returns bits $\mathit{DeviceFeaturesSel} * 32$ to $(\mathit{DeviceFeaturesSel} * 32) + 31$, eg. feature bits 0 to 31 if *DeviceFeaturesSel* is set to 0 and features bits 32 to 63 if *DeviceFeaturesSel* is set to 1. Also see 2.2 Feature Bits.

Name Offset from the base Direction	**Function** Description
DeviceFeaturesSel 0x014 W	**Device (host) features word selection.** Writing to this register selects a set of 32 device feature bits accessible by reading from *DeviceFeatures*.
DriverFeatures 0x020 W	**Flags representing device features understood and activated by the driver** Writing to this register sets 32 consecutive flag bits, the least significant bit depending on the last value written to *DriverFeaturesSel*. Access to this register sets bits $\mathit{DriverFeaturesSel} * 32$ to $(\mathit{DriverFeaturesSel} * 32) + 31$, eg. feature bits 0 to 31 if *DriverFeaturesSel* is set to 0 and features bits 32 to 63 if *DriverFeaturesSel* is set to 1. Also see 2.2 Feature Bits.
DriverFeaturesSel 0x024 W	**Activated (guest) features word selection** Writing to this register selects a set of 32 activated feature bits accessible by writing to *DriverFeatures*.
QueueSel 0x030 W	**Virtual queue index** Writing to this register selects the virtual queue that the following operations on *QueueNumMax*, *QueueNum*, *QueueReady*, *QueueDescLow*, *QueueDescHigh*, *QueueAvailLow*, *QueueAvailHigh*, *QueueUsedLow* and *QueueUsedHigh* apply to. The index number of the first queue is zero (0x0).
QueueNumMax 0x034 R	**Maximum virtual queue size** Reading from the register returns the maximum size (number of elements) of the queue the device is ready to process or zero (0x0) if the queue is not available. This applies to the queue selected by writing to *QueueSel*.
QueueNum 0x038 W	**Virtual queue size** Queue size is the number of elements in the queue, therefore in each of the Descriptor Table, the Available Ring and the Used Ring. Writing to this register notifies the device what size of the queue the driver will use. This applies to the queue selected by writing to *QueueSel*.
QueueReady 0x044 RW	**Virtual queue ready bit** Writing one (0x1) to this register notifies the device that it can execute requests from this virtual queue. Reading from this register returns the last value written to it. Both read and write accesses apply to the queue selected by writing to *QueueSel*.
QueueNotify 0x050 W	**Queue notifier** Writing a queue index to this register notifies the device that there are new buffers to process in the queue.
InterruptStatus 0x60 R	**Interrupt status** Reading from this register returns a bit mask of events that caused the device interrupt to be asserted. The following events are possible: **Used Ring Update** - bit 0 - the interrupt was asserted because the device has updated the Used Ring in at least one of the active virtual queues. **Configuration Change** - bit 1 - the interrupt was asserted because the configuration of the device has changed.
InterruptACK 0x064 W	**Interrupt acknowledge** Writing a value with bits set as defined in *InterruptStatus* to this register notifies the device that events causing the interrupt have been handled.

Name Offset from the base Direction	**Function** Description
Status 0x070 RW	**Device status** Reading from this register returns the current device status flags. Writing non-zero values to this register sets the status flags, indicating the driver progress. Writing zero (0x0) to this register triggers a device reset. See also p. 4.2.3.1 Device Initialization.
QueueDescLow 0x080 *QueueDescHigh* 0x084 W	**Virtual queue's Descriptor Table 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueDescLow*, higher 32 bits to *QueueDescHigh*) notifies the device about location of the Descriptor Table of the queue selected by writing to *QueueSel* register.
QueueAvailLow 0x090 *QueueAvailHigh* 0x094 W	**Virtual queue's Available Ring 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueAvailLow*, higher 32 bits to *QueueAvailHigh*) notifies the device about location of the Available Ring of the queue selected by writing to *QueueSel*.
QueueUsedLow 0x0a0 *QueueUsedHigh* 0x0a4 W	**Virtual queue's Used Ring 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueUsedLow*, higher 32 bits to *QueueUsedHigh*) notifies the device about location of the Used Ring of the queue selected by writing to *QueueSel*.
ConfigGeneration 0x0fc R	**Configuration atomicity value** Reading from this register returns a value describing a version of the device-specific configuration space (see *Config*). The driver can then access the configuration space and, when finished, read *ConfigGeneration* again. If no part of the configuration space has changed between these two *ConfigGeneration* reads, the returned values are identical. If the values are different, the configuration space accesses were not atomic and the driver has to perform the operations again. See also 2.3.
Config 0x100+ RW	**Configuration space** Device-specific configuration space starts at the offset 0x100 and is accessed with byte alignment. Its meaning and size depend on the device and the driver.

4.2.2.1 Device Requirements: MMIO Device Register Layout

The device MUST return 0x74726976 in *MagicValue*.

The device MUST return value 0x2 in *Version*.

The device MUST present each event by setting the corresponding bit in *InterruptStatus* from the moment it takes place, until the driver acknowledges the interrupt by writing a corresponding bit mask to the *InterruptACK* register. Bits which do not represent events which took place MUST be zero.

Upon reset, the device MUST clear all bits in *InterruptStatus* and ready bits in the *QueueReady* register for all queues in the device.

The device MUST change value returned in *ConfigGeneration* if there is any risk of a driver seeing an inconsistent configuration state.

The device MUST NOT access virtual queue contents when *QueueReady* is zero (0x0).

4.2.2.2 Driver Requirements: MMIO Device Register Layout

The driver MUST NOT access memory locations not described in the table 4.1 (or, in case of the configuration space, described in the device specification), MUST NOT write to the read-only registers (direction R) and MUST NOT read from the write-only registers (direction W).

The driver MUST only use 32 bit wide and aligned reads and writes to access the control registers described in table 4.1. For the device-specific configuration space, the driver MUST use 8 bit wide accesses for 8 bit wide fields, 16 bit wide and aligned accesses for 16 bit wide fields and 32 bit wide and aligned accesses for 32 and 64 bit wide fields.

The driver MUST ignore a device with *MagicValue* which is not 0x74726976, although it MAY report an error.

The driver MUST ignore a device with *Version* which is not 0x2, although it MAY report an error.

The driver MUST ignore a device with *DeviceID* 0x0, but MUST NOT report any error.

Before reading from *DeviceFeatures*, the driver MUST write a value to *DeviceFeaturesSel*.

Before writing to the *DriverFeatures* register, the driver MUST write a value to the *DriverFeaturesSel* register.

The driver MUST write a value to *QueueNum* which is less than or equal to the value presented by the device in *QueueNumMax*.

When *QueueReady* is not zero, the driver MUST NOT access *QueueNum*, *QueueDescLow*, *QueueDescHigh*, *QueueAvailLow*, *QueueAvailHigh*, *QueueUsedLow*, *QueueUsedHigh*.

To stop using the queue the driver MUST write zero (0x0) to this *QueueReady* and MUST read the value back to ensure synchronization.

The driver MUST ignore undefined bits in *InterruptStatus*.

The driver MUST write a value with a bit mask describing events it handled into *InterruptACK* when it finishes handling an interrupt and MUST NOT set any of the undefined bits in the value.

4.2.3 MMIO-specific Initialization And Device Operation

4.2.3.1 Device Initialization

4.2.3.1.1 Driver Requirements: Device Initialization

The driver MUST start the device initialization by reading and checking values from *MagicValue* and *Version*. If both values are valid, it MUST read *DeviceID* and if its value is zero (0x0) MUST abort initialization and MUST NOT access any other register.

Further initialization MUST follow the procedure described in 3.1 Device Initialization.

4.2.3.2 Virtqueue Configuration

The driver will typically initialize the virtual queue in the following way:

1. Select the queue writing its index (first queue is 0) to *QueueSel*.
2. Check if the queue is not already in use: read *QueueReady*, and expect a returned value of zero (0x0).
3. Read maximum queue size (number of elements) from *QueueNumMax*. If the returned value is zero (0x0) the queue is not available.
4. Allocate and zero the queue pages, making sure the memory is physically contiguous. It is recommended to align the Used Ring to an optimal boundary (usually the page size).
5. Notify the device about the queue size by writing the size to *QueueNum*.

6. Write physical addresses of the queue's Descriptor Table, Available Ring and Used Ring to (respectively) the *QueueDescLow*/*QueueDescHigh*, *QueueAvailLow*/*QueueAvailHigh* and *QueueUsedLow*/*QueueUsedHigh* register pairs.
7. Write 0x1 to *QueueReady*.

4.2.3.3 Notifying The Device

The driver notifies the device about new buffers being available in a queue by writing the index of the updated queue to *QueueNotify*.

4.2.3.4 Notifications From The Device

The memory mapped virtio device is using a single, dedicated interrupt signal, which is asserted when at least one of the bits described in the description of *InterruptStatus* is set. This is how the device notifies the driver about a new used buffer being available in the queue or about a change in the device configuration.

4.2.3.4.1 Driver Requirements: Notifications From The Device

After receiving an interrupt, the driver MUST read *InterruptStatus* to check what caused the interrupt (see the register description). After the interrupt is handled, the driver MUST acknowledge it by writing a bit mask corresponding to the handled events to the InterruptACK register.

4.2.4 Legacy interface

The legacy MMIO transport used page-based addressing, resulting in a slightly different control register layout, the device initialization and the virtual queue configuration procedure.

Table 4.2 presents control registers layout, omitting descriptions of registers which did not change their function nor behaviour:

Table 4.2: MMIO Device Legacy Register Layout

Name Offset from base Direction	**Function** Description
MagicValue 0x000 R	**Magic value**
Version 0x004 R	**Device version number** Legacy device returns value 0x1.
DeviceID 0x008 R	**Virtio Subsystem Device ID**
VendorID 0x00c R	**Virtio Subsystem Vendor ID**
HostFeatures 0x010 R	**Flags representing features the device supports**

Name Offset from the base Direction	**Function** Description
HostFeaturesSel 0x014 W	**Device (host) features word selection.**
GuestFeatures 0x020 W	**Flags representing device features understood and activated by the driver**
GuestFeaturesSel 0x024 W	**Activated (guest) features word selection**
GuestPageSize 0x028 W	**Guest page size** The driver writes the guest page size in bytes to the register during initialization, before any queues are used. This value should be a power of 2 and is used by the device to calculate the Guest address of the first queue page (see QueuePFN).
QueueSel 0x030 W	**Virtual queue index** Writing to this register selects the virtual queue that the following operations on the *QueueNumMax*, *QueueNum*, *QueueAlign* and *QueuePFN* registers apply to. The index number of the first queue is zero (0x0). .
QueueNumMax 0x034 R	**Maximum virtual queue size** Reading from the register returns the maximum size of the queue the device is ready to process or zero (0x0) if the queue is not available. This applies to the queue selected by writing to *QueueSel* and is allowed only when *QueuePFN* is set to zero (0x0), so when the queue is not actively used.
QueueNum 0x038 W	**Virtual queue size** Queue size is the number of elements in the queue, therefore size of the descriptor table and both available and used rings. Writing to this register notifies the device what size of the queue the driver will use. This applies to the queue selected by writing to *QueueSel*.
QueueAlign 0x03c W	**Used Ring alignment in the virtual queue** Writing to this register notifies the device about alignment boundary of the Used Ring in bytes. This value should be a power of 2 and applies to the queue selected by writing to *QueueSel*.
QueuePFN 0x040 RW	**Guest physical page number of the virtual queue** Writing to this register notifies the device about location of the virtual queue in the Guest's physical address space. This value is the index number of a page starting with the queue Descriptor Table. Value zero (0x0) means physical address zero (0x00000000) and is illegal. When the driver stops using the queue it writes zero (0x0) to this register. Reading from this register returns the currently used page number of the queue, therefore a value other than zero (0x0) means that the queue is in use. Both read and write accesses apply to the queue selected by writing to *QueueSel*.
QueueNotify 0x050 W	**Queue notifier**
InterruptStatus 0x60 R	**Interrupt status**

Name Offset from the base Direction	**Function** Description
InterruptACK 0x064 W	**Interrupt acknowledge**
Status 0x070 RW	**Device status** Reading from this register returns the current device status flags. Writing non-zero values to this register sets the status flags, indicating the OS/driver progress. Writing zero (0x0) to this register triggers a device reset. The device sets *QueuePFN* to zero (0x0) for all queues in the device. Also see 3.1 Device Initialization.
Config 0x100+ RW	**Configuration space**

The virtual queue page size is defined by writing to *GuestPageSize*, as written by the guest. The driver does this before the virtual queues are configured.

The virtual queue layout follows p. 2.4.2 Legacy Interfaces: A Note on Virtqueue Layout, with the alignment defined in *QueueAlign*.

The virtual queue is configured as follows:

1. Select the queue writing its index (first queue is 0) to *QueueSel*.
2. Check if the queue is not already in use: read *QueuePFN*, expecting a returned value of zero (0x0).
3. Read maximum queue size (number of elements) from *QueueNumMax*. If the returned value is zero (0x0) the queue is not available.
4. Allocate and zero the queue pages in contiguous virtual memory, aligning the Used Ring to an optimal boundary (usually page size). The driver should choose a queue size smaller than or equal to *QueueNumMax*.
5. Notify the device about the queue size by writing the size to *QueueNum*.
6. Notify the device about the used alignment by writing its value in bytes to *QueueAlign*.
7. Write the physical number of the first page of the queue to the *QueuePFN* register.

Notification mechanisms did not change.

4.3 Virtio Over Channel I/O

S/390 based virtual machines support neither PCI nor MMIO, so a different transport is needed there.

virtio-ccw uses the standard channel I/O based mechanism used for the majority of devices on S/390. A virtual channel device with a special control unit type acts as proxy to the virtio device (similar to the way virtio-pci uses a PCI device) and configuration and operation of the virtio device is accomplished (mostly) via channel commands. This means virtio devices are discoverable via standard operating system algorithms, and adding virtio support is mainly a question of supporting a new control unit type.

As the S/390 is a big endian machine, the data structures transmitted via channel commands are big-endian: this is made clear by use of the types be16, be32 and be64.

4.3.1 Basic Concepts

As a proxy device, virtio-ccw uses a channel-attached I/O control unit with a special control unit type (0x3832) and a control unit model corresponding to the attached virtio device's subsystem device ID, accessed via a virtual I/O subchannel and a virtual channel path of type 0x32. This proxy device is discoverable via normal channel subsystem device discovery (usually a STORE SUBCHANNEL loop) and answers to the basic channel commands, most importantly SENSE ID.

For a virtio-ccw proxy device, SENSE ID will return the following information:

Bytes	Description	Contents
0	reserved	0xff
1-2	control unit type	0x3832
3	control unit model	<virtio device id>
4-5	device type	zeroes (unset)
6	device model	zeroes (unset)
7-255	extended SenseId data	zeroes (unset)

In addition to the basic channel commands, virtio-ccw defines a set of channel commands related to configuration and operation of virtio:

```
#define CCW_CMD_SET_VQ 0x13
#define CCW_CMD_VDEV_RESET 0x33
#define CCW_CMD_SET_IND 0x43
#define CCW_CMD_SET_CONF_IND 0x53
#define CCW_CMD_SET_IND_ADAPTER 0x73
#define CCW_CMD_READ_FEAT 0x12
#define CCW_CMD_WRITE_FEAT 0x11
#define CCW_CMD_READ_CONF 0x22
#define CCW_CMD_WRITE_CONF 0x21
#define CCW_CMD_WRITE_STATUS 0x31
#define CCW_CMD_READ_VQ_CONF 0x32
#define CCW_CMD_SET_VIRTIO_REV 0x83
```

4.3.1.1 Device Requirements: Basic Concepts

The virtio-ccw device acts like a normal channel device, as specified in [S390 PoP] and [S390 Common I/O]. In particular:

- A device MUST post a unit check with command reject for any command it does not support.
- If a driver did not suppress length checks for a channel command, the device MUST present a subchannel status as detailed in the architecture when the actual length did not match the expected length.
- If a driver did suppress length checks for a channel command, the device MUST present a check condition if the transmitted data does not contain enough data to process the command. If the driver submitted a buffer that was too long, the device SHOULD accept the command.

4.3.1.2 Driver Requirements: Basic Concepts

A driver for virtio-ccw devices MUST check for a control unit type of 0x3832 and MUST ignore the device type and model.

A driver SHOULD attempt to provide the correct length in a channel command even if it suppresses length checks for that command.

4.3.2 Device Initialization

virtio-ccw uses several channel commands to set up a device.

4.3.2.1 Setting the Virtio Revision

CCW_CMD_SET_VIRTIO_REV is issued by the driver to set the revision of the virtio-ccw transport it intends to drive the device with. It uses the following communication structure:

```
struct virtio_rev_info {
        be16 revision;
        be16 length;
        u8 data[];
};
```

revision contains the desired revision id, *length* the length of the data portion and *data* revision-dependent additional desired options.

The following values are supported:

revision	*length*	*data*	remarks
0	0	<empty>	legacy interface; transitional devices only
1	0	<empty>	Virtio 1.0
2-n			reserved for later revisions

Note that a change in the virtio standard does not necessarily correspond to a change in the virtio-ccw revision.

4.3.2.1.1 Device Requirements: Setting the Virtio Revision

A device MUST post a unit check with command reject for any *revision* it does not support. For any invalid combination of *revision*, *length* and *data*, it MUST post a unit check with command reject as well. A non-transitional device MUST reject revision id 0.

A device MUST answer with command reject to any virtio-ccw specific channel command that is not contained in the revision selected by the driver.

A device MUST answer with command reject to any attempt to select a different revision after a revision has been successfully selected by the driver.

A device MUST treat the revision as unset from the time the associated subchannel has been enabled until a revision has been successfully set by the driver. This implies that revisions are not persistent across disabling and enabling of the associated subchannel.

4.3.2.1.2 Driver Requirements: Setting the Virtio Revision

A driver SHOULD start with trying to set the highest revision it supports and continue with lower revisions if it gets a command reject.

A driver MUST NOT issue any other virtio-ccw specific channel commands prior to setting the revision.

After a revision has been successfully selected by the driver, it MUST NOT attempt to select a different revision.

4.3.2.1.3 Legacy Interfaces: A Note on Setting the Virtio Revision

A legacy device will not support the CCW_CMD_SET_VIRTIO_REV and answer with a command reject. A non-transitional driver MUST stop trying to operate this device in that case. A transitional driver MUST operate the device as if it had been able to set revision 0.

A legacy driver will not issue the CCW_CMD_SET_VIRTIO_REV prior to issuing other virtio-ccw specific channel commands. A non-transitional device therefore MUST answer any such attempts with a command reject. A transitional device MUST assume in this case that the driver is a legacy driver and continue as if the driver selected revision 0. This implies that the device MUST reject any command not valid for revision 0, including a subsequent CCW_CMD_SET_VIRTIO_REV.

4.3.2.2 Configuring a Virtqueue

CCW_CMD_READ_VQ_CONF is issued by the driver to obtain information about a queue. It uses the following structure for communicating:

```
struct vq_config_block {
        be16 index;
        be16 max_num;
};
```

The requested number of buffers for queue *index* is returned in *max_num*.

Afterwards, CCW_CMD_SET_VQ is issued by the driver to inform the device about the location used for its queue. The transmitted structure is

```
struct vq_info_block {
        be64 desc;
        be32 res0;
        be16 index;
        be16 num;
        be64 avail;
        be64 used;
};
```

desc, *avail* and *used* contain the guest addresses for the descriptor table, available ring and used ring for queue *index*, respectively. The actual virtqueue size (number of allocated buffers) is transmitted in *num*.

4.3.2.2.1 Device Requirements: Configuring a Virtqueue

res0 is reserved and MUST be ignored by the device.

4.3.2.2.2 Legacy Interface: A Note on Configuring a Virtqueue

For a legacy driver or for a driver that selected revision 0, CCW_CMD_SET_VQ uses the following communication block:

```
struct vq_info_block_legacy {
        be64 queue;
        be32 align;
        be16 index;
        be16 num;
};
```

queue contains the guest address for queue *index*, *num* the number of buffers and *align* the alignment.

4.3.2.3 Virtqueue Layout

The virtqueue is physically contiguous, with padding added to make the used ring meet the align value:

Descriptor Table	Available Ring (. . .padding. . .)	Used Ring

The calculation for total size is as follows:

```
#define ALIGN(x) (((x) + align) & ~align)
static inline unsigned virtq_size(unsigned int num)
{
     return ALIGN(sizeof(struct virtq_desc)*num
                  + sizeof(u16)*(3 + num))
          + ALIGN(sizeof(u16)*3 + sizeof(struct virtq_used_elem)*num);
}
```

4.3.2.4 Communicating Status Information

The driver changes the status of a device via the CCW_CMD_WRITE_STATUS command, which transmits an 8 bit status value.

4.3.2.5 Handling Device Features

Feature bits are arranged in an array of 32 bit values, making for a total of 8192 feature bits. Feature bits are in little-endian byte order.

The CCW commands dealing with features use the following communication block:

```
struct virtio_feature_desc {
        le32 features;
        u8 index;
};
```

features are the 32 bits of features currently accessed, while *index* describes which of the feature bit values is to be accessed. No padding is added at the end of the structure, it is exactly 5 bytes in length.

The guest obtains the device's device feature set via the CCW_CMD_READ_FEAT command. The device stores the features at *index* to *features*.

For communicating its supported features to the device, the driver uses the CCW_CMD_WRITE_FEAT command, denoting a *features*/*index* combination.

4.3.2.6 Device Configuration

The device's configuration space is located in host memory.

To obtain information from the configuration space, the driver uses CCW_CMD_READ_CONF, specifying the guest memory for the device to write to.

For changing configuration information, the driver uses CCW_CMD_WRITE_CONF, specifying the guest memory for the device to read from.

In both cases, the complete configuration space is transmitted. This allows the driver to compare the new configuration space with the old version, and keep a generation count internally whenever it changes.

4.3.2.7 Setting Up Indicators

In order to set up the indicator bits for host->guest notification, the driver uses different channel commands depending on whether it wishes to use traditional I/O interrupts tied to a subchannel or adapter I/O interrupts for virtqueue notifications. For any given device, the two mechanisms are mutually exclusive.

For the configuration change indicators, only a mechanism using traditional I/O interrupts is provided, regardless of whether traditional or adapter I/O interrupts are used for virtqueue notifications.

4.3.2.7.1 Setting Up Classic Queue Indicators

Indicators for notification via classic I/O interrupts are contained in a 64 bit value per virtio-ccw proxy device.

To communicate the location of the indicator bits for host->guest notification, the driver uses the CCW_CMD_SET_IND command, pointing to a location containing the guest address of the indicators in a 64 bit value.

If the driver has already set up two-staged queue indicators via the CCW_CMD_SET_IND_ADAPTER command, the device MUST post a unit check with command reject to any subsequent CCW_CMD_SET_IND command.

4.3.2.7.2 Setting Up Configuration Change Indicators

Indicators for configuration change host->guest notification are contained in a 64 bit value per virtio-ccw proxy device.

To communicate the location of the indicator bits used in the configuration change host->guest notification, the driver issues the CCW_CMD_SET_CONF_IND command, pointing to a location containing the guest address of the indicators in a 64 bit value.

4.3.2.7.3 Setting Up Two-Stage Queue Indicators

Indicators for notification via adapter I/O interrupts consist of two stages:

- a summary indicator byte covering the virtqueues for one or more virtio-ccw proxy devices
- a set of contigous indicator bits for the virtqueues for a virtio-ccw proxy device

To communicate the location of the summary and queue indicator bits, the driver uses the CCW_CMD_SET_IND_ADAPTER command with the following payload:

```
struct virtio_thinint_area {
        be64 summary_indicator;
        be64 indicator;
        be64 bit_nr;
        u8 isc;
} __attribute__ ((packed));
```

summary_indicator contains the guest address of the 8 bit summary indicator. *indicator* contains the guest address of an area wherein the indicators for the devices are contained, starting at *bit_nr*, one bit per virtqueue of the device. Bit numbers start at the left, i.e. the most significant bit in the first byte is assigned the bit number 0. *isc* contains the I/O interruption subclass to be used for the adapter I/O interrupt. It MAY be different from the isc used by the proxy virtio-ccw device's subchannel. No padding is added at the end of the structure, it is exactly 25 bytes in length.

4.3.2.7.3.1 Device Requirements: Setting Up Two-Stage Queue Indicators

If the driver has already set up classic queue indicators via the CCW_CMD_SET_IND command, the device MUST post a unit check with command reject to any subsequent CCW_CMD_SET_IND_ADAPTER command.

4.3.2.7.4 Legacy Interfaces: A Note on Setting Up Indicators

In some cases, legacy devices will only support classic queue indicators; in that case, they will reject CCW_-CMD_SET_IND_ADAPTER as they don't know that command. Some legacy devices will support two-stage queue indicators, though, and a driver will be able to successfully use CCW_CMD_SET_IND_ADAPTER to set them up.

4.3.3 Device Operation

4.3.3.1 Host->Guest Notification

There are two modes of operation regarding host->guest notification, classic I/O interrupts and adapter I/O interrupts. The mode to be used is determined by the driver by using CCW_CMD_SET_IND respectively CCW_CMD_SET_IND_ADAPTER to set up queue indicators.

For configuration changes, the driver always uses classic I/O interrupts.

4.3.3.1.1 Notification via Classic I/O Interrupts

If the driver used the CCW_CMD_SET_IND command to set up queue indicators, the device will use classic I/O interrupts for host->guest notification about virtqueue activity.

For notifying the driver of virtqueue buffers, the device sets the corresponding bit in the guest-provided indicators. If an interrupt is not already pending for the subchannel, the device generates an unsolicited I/O interrupt.

If the device wants to notify the driver about configuration changes, it sets bit 0 in the configuration indicators and generates an unsolicited I/O interrupt, if needed. This also applies if adapter I/O interrupts are used for queue notifications.

4.3.3.1.2 Notification via Adapter I/O Interrupts

If the driver used the CCW_CMD_SET_IND_ADAPTER command to set up queue indicators, the device will use adapter I/O interrupts for host->guest notification about virtqueue activity.

For notifying the driver of virtqueue buffers, the device sets the bit in the guest-provided indicator area at the corresponding offset. The guest-provided summary indicator is set to 0x01. An adapter I/O interrupt for the corresponding interruption subclass is generated.

The recommended way to process an adapter I/O interrupt by the driver is as follows:

- Process all queue indicator bits associated with the summary indicator.
- Clear the summary indicator, performing a synchronization (memory barrier) afterwards.
- Process all queue indicator bits associated with the summary indicator again.

4.3.3.1.2.1 Device Requirements: Notification via Adapter I/O Interrupts

The device SHOULD only generate an adapter I/O interrupt if the summary indicator had not been set prior to notification.

4.3.3.1.2.2 Driver Requirements: Notification via Adapter I/O Interrupts

The driver MUST clear the summary indicator after receiving an adapter I/O interrupt before it processes the queue indicators.

4.3.3.1.3 Legacy Interfaces: A Note on Host->Guest Notification

As legacy devices and drivers support only classic queue indicators, host->guest notification will always be done via classic I/O interrupts.

4.3.3.2 Guest->Host Notification

For notifying the device of virtqueue buffers, the driver unfortunately can't use a channel command (the asynchronous characteristics of channel I/O interact badly with the host block I/O backend). Instead, it uses a diagnose 0x500 call with subcode 3 specifying the queue, as follows:

GPR	Input Value	Output Value
1	0x3	
2	Subchannel ID	Host Cookie
3	Virtqueue number	
4	Host Cookie	

4.3.3.2.1 Device Requirements: Guest->Host Notification

The device MUST ignore bits 0-31 (counting from the left) of GPR2. This aligns passing the subchannel ID with the way it is passed for the existing I/O instructions.

The device MAY return a 64-bit host cookie in GPR2 to speed up the notification execution.

4.3.3.2.2 Driver Requirements: Guest->Host Notification

For each notification, the driver SHOULD use GPR4 to pass the host cookie received in GPR2 from the previous notication.

Note: For example:

```
info->cookie = do_notify(schid,
                         virtqueue_get_queue_index(vq),
                         info->cookie);
```

4.3.3.3 Resetting Devices

In order to reset a device, a driver sends the CCW_CMD_VDEV_RESET command.

5 Device Types

On top of the queues, config space and feature negotiation facilities built into virtio, several devices are defined.

The following device IDs are used to identify different types of virtio devices. Some device IDs are reserved for devices which are not currently defined in this standard.

Discovering what devices are available and their type is bus-dependent.

Device ID	Virtio Device
0	reserved (invalid)
1	network card
2	block device
3	console
4	entropy source
5	memory ballooning (legacy)
6	ioMemory
7	rpmsg
8	SCSI host
9	9P transport
10	mac80211 wlan
11	rproc serial
12	virtio CAIF
13	memory balloon
16	GPU device
17	Timer/Clock device
18	Input device

Some of the devices above are unspecified by this document, because they are seen as immature or especially niche. Be warned that some are only specified by the sole existing implementation; they could become part of a future specification, be abandoned entirely, or live on outside this standard. We shall speak of them no further.

5.1 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 are 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.

5.1.1 Device ID

1

5.1.2 Virtqueues

0 receiveq1

1 transmitq1

. . .

2N receiveqN

2N+1 transmitqN

2N+2 controlq

N=1 if VIRTIO_NET_F_MQ is not negotiated, otherwise N is set by *max_virtqueue_pairs*.

controlq only exists if VIRTIO_NET_F_CTRL_VQ set.

5.1.3 Feature bits

VIRTIO_NET_F_CSUM (0) Device handles packets with partial checksum. This “checksum offload” is a common feature on modern network cards.

VIRTIO_NET_F_GUEST_CSUM (1) Driver handles packets with partial checksum.

VIRTIO_NET_F_CTRL_GUEST_OFFLOADS (2) Control channel offloads reconfiguration support.

VIRTIO_NET_F_MAC (5) Device has given MAC address.

VIRTIO_NET_F_GUEST_TSO4 (7) Driver can receive TSOv4.

VIRTIO_NET_F_GUEST_TSO6 (8) Driver can receive TSOv6.

VIRTIO_NET_F_GUEST_ECN (9) Driver can receive TSO with ECN.

VIRTIO_NET_F_GUEST_UFO (10) Driver can receive UFO.

VIRTIO_NET_F_HOST_TSO4 (11) Device can receive TSOv4.

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) Driver 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.

VIRTIO_NET_F_GUEST_ANNOUNCE(21) Driver can send gratuitous packets.

VIRTIO_NET_F_MQ(22) Device supports multiqueue with automatic receive steering.

VIRTIO_NET_F_CTRL_MAC_ADDR(23) Set MAC address through control channel.

5.1.3.1 Feature bit requirements

Some networking feature bits require other networking feature bits (see 2.2.1):

VIRTIO_NET_F_GUEST_TSO4 Requires VIRTIO_NET_F_GUEST_CSUM.

VIRTIO_NET_F_GUEST_TSO6 Requires VIRTIO_NET_F_GUEST_CSUM.

VIRTIO_NET_F_GUEST_ECN Requires VIRTIO_NET_F_GUEST_TSO4 or VIRTIO_NET_F_GUEST_TSO6.

VIRTIO_NET_F_GUEST_UFO Requires VIRTIO_NET_F_GUEST_CSUM.

VIRTIO_NET_F_HOST_TSO4 Requires VIRTIO_NET_F_CSUM.

VIRTIO_NET_F_HOST_TSO6 Requires VIRTIO_NET_F_CSUM.

VIRTIO_NET_F_HOST_ECN Requires VIRTIO_NET_F_HOST_TSO4 or VIRTIO_NET_F_HOST_TSO6.

VIRTIO_NET_F_HOST_UFO Requires VIRTIO_NET_F_CSUM.

VIRTIO_NET_F_CTRL_RX Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_CTRL_VLAN Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_GUEST_ANNOUNCE Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_MQ Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_CTRL_MAC_ADDR Requires VIRTIO_NET_F_CTRL_VQ.

5.1.3.2 Legacy Interface: Feature bits

VIRTIO_NET_F_GSO (6) Device handles packets with any GSO type.

This was supposed to indicate segmentation offload support, but upon further investigation it became clear that multiple bits were needed.

5.1.4 Device configuration layout

Three driver-read-only configuration fields are currently defined. The *mac* address field always exists (though is only valid if VIRTIO_NET_F_MAC is set), and *status* only exists if VIRTIO_NET_F_STATUS is set. Two read-only bits (for the driver) are currently defined for the status field: VIRTIO_NET_S_LINK_UP and VIRTIO_NET_S_ANNOUNCE.

```
#define VIRTIO_NET_S_LINK_UP     1
#define VIRTIO_NET_S_ANNOUNCE    2
```

The following driver-read-only field, *max_virtqueue_pairs* only exists if VIRTIO_NET_F_MQ is set. This field specifies the maximum number of each of transmit and receive virtqueues (receiveq1…receiveqN and transmitq1…transmitqN respectively) that can be configured once VIRTIO_NET_F_MQ is negotiated.

```
struct virtio_net_config {
        u8 mac[6];
        le16 status;
        le16 max_virtqueue_pairs;
};
```

5.1.4.1 Device Requirements: Device configuration layout

The device MUST set *max_virtqueue_pairs* to between 1 and 0x8000 inclusive, if it offers VIRTIO_NET_F_MQ.

5.1.4.2 Driver Requirements: Device configuration layout

A driver SHOULD negotiate VIRTIO_NET_F_MAC if the device offers it. If the driver negotiates the VIRTIO_NET_F_MAC feature, the driver MUST set the physical address of the NIC to *mac*. Otherwise, it SHOULD use a locally-administered MAC address (see IEEE 802, “9.2 48-bit universal LAN MAC addresses”).

If the driver does not negotiate the VIRTIO_NET_F_STATUS feature, it SHOULD assume the link is active, otherwise it SHOULD read the link status from the bottom bit of *status*.

5.1.4.3 Legacy Interface: Device configuration layout

When using the legacy interface, transitional devices and drivers MUST format *status* and *max_virtqueue_pairs* in struct virtio_net_config according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

When using the legacy interface, *mac* is driver-writable which provided a way for drivers to update the MAC without negotiating VIRTIO_NET_F_CTRL_MAC_ADDR.

5.1.5 Device Initialization

A driver would perform a typical initialization routine like so:

1. Identify and initialize the receive and transmission virtqueues, up to N of each kind. If VIRTIO_NET_F_MQ feature bit is negotiated, N=*max_virtqueue_pairs*, otherwise identify N=1.
2. If the VIRTIO_NET_F_CTRL_VQ feature bit is negotiated, identify the control virtqueue.
3. Fill the receive queues with buffers: see 5.1.6.3.
4. Even with VIRTIO_NET_F_MQ, only receiveq1, transmitq1 and controlq are used by default. The driver would send the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command specifying the number of the transmit and receive queues to use.
5. If the VIRTIO_NET_F_MAC feature bit is set, the configuration space *mac* entry indicates the “physical” address of the network card, otherwise the driver would typically generate a random local MAC address.
6. If the VIRTIO_NET_F_STATUS feature bit is negotiated, the link status comes from the bottom bit of *status*. Otherwise, the driver assumes it’s active.
7. A performant driver would indicate that it will generate checksumless packets by negotating the VIRTIO_NET_F_CSUM feature.
8. 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.
9. The converse features are also available: a driver can save the virtual device some work by negotiating these features.

   Note: For example, a network packet transported between two guests on the same system might not need checksumming at all, nor segmentation, if both guests are amenable. 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 5.1.6.3 Setting Up Receive Buffers and 5.1.6.4 Processing of Packets below.

A truly minimal driver would only accept VIRTIO_NET_F_MAC and ignore everything else.

5.1.6 Device Operation

Packets are transmitted by placing them in the transmitq1…transmitqN, and buffers for incoming packets are placed in the receiveq1…receiveqN. In each case, the packet itself is preceded 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;
        le16 hdr_len;
        le16 gso_size;
        le16 csum_start;
        le16 csum_offset;
        le16 num_buffers;
};
```

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

5.1.6.1 Legacy Interface: Device Operation

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_net_hdr according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

The legacy driver only presented *num_buffers* in the struct virtio_net_hdr when VIRTIO_NET_F_MRG_RXBUF was not negotiated; without that feature the structure was 2 bytes shorter.

5.1.6.2 Packet Transmission

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

1. The driver MAY send a completely checksummed packet. In this case, *flags* will be zero, and *gso_type* will be VIRTIO_NET_HDR_GSO_NONE.
2. If the driver negotiated VIRTIO_NET_F_CSUM, it MAY skip checksumming the packet:
    - *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 is placed by the device.
    - The TCP checksum field in the packet is set to 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.

    Note: 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.
3. If the driver negotiated VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO, and the packet requires TCP segmentation or UDP fragmentation, then *gso_type* 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[1].
  - *gso_size* is the maximum size of each packet beyond that header (ie. MSS).
  - If the driver negotiated the VIRTIO_NET_F_HOST_ECN feature, the VIRTIO_NET_HDR_GSO_ECN bit in *gso_type* indicates that the TCP packet has the ECN bit set[2].

4. *num_buffers* is set to zero. This field is unused on transmitted packets.
5. The header and packet are added as one output descriptor to the transmitq, and the device is notified of the new entry (see 5.1.5 Device Initialization).

5.1.6.2.1 Driver Requirements: Packet Transmission

If a driver has not negotiated VIRTIO_NET_F_CSUM, *flags* MUST be zero and the packet MUST be fully checksummed.

The driver MUST set *num_buffers* to zero.

A driver 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[3], in which case it MUST set the VIRTIO_NET_HDR_GSO_ECN bit in *gso_type*.

5.1.6.2.2 Packet Transmission Interrupt

Often a driver will suppress transmission interrupts using the VIRTQ_AVAIL_F_NO_INTERRUPT flag (see 3.2.2 Receiving Used Buffers From The Device) and check for used packets in the transmit path of following packets.

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

5.1.6.3 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 maximum incoming packet will be to 65550 bytes long (the maximum size of a TCP or UDP packet, plus the 14 byte ethernet header), otherwise 1514 bytes. The 12-byte struct virtio_net_hdr is prepended to this, making for 65562 or 1526 bytes.

5.1.6.3.1 Driver Requirements: Setting Up Receive Buffers

- If VIRTIO_NET_F_MRG_RXBUF is not negotiated:
  - If VIRTIO_NET_F_GUEST_TSO4, VIRTIO_NET_F_GUEST_TSO6 or VIRTIO_NET_F_GUEST_UFO are negotiated, the driver SHOULD populate the receive queue(s) with buffers of at least 65562 bytes.
  - Otherwise, the driver SHOULD populate the receive queue(s) with buffers of at least 1526 bytes.
- If VIRTIO_NET_F_MRG_RXBUF is negotiated, each buffer MUST be at greater than the size of the struct virtio_net_hdr.

---

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

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

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

Note: Obviously each buffer can be split across multiple descriptor elements.

If VIRTIO_NET_F_MQ is negotiated, each of receiveq1…receiveqN that will be used SHOULD be populated with receive buffers.

5.1.6.3.2 Device Requirements: Setting Up Receive Buffers

The device MUST set *num_buffers* to the number of descriptors used to hold the incoming packet.

The device MUST use only a single descriptor if VIRTIO_NET_F_MRG_RXBUF was not negotiated.

Note: This means that *num_buffers* will always be 1 if VIRTIO_NET_F_MRG_RXBUF is not negotiated.

5.1.6.4 Processing of Packets

When a packet is copied into a buffer in the receiveq, the optimal path is to disable further interrupts for the receiveq (see 3.2.2 Receiving Used Buffers From The Device) and process packets until no more are found, then re-enable them.

Processing packet involves:

1. *num_buffers* indicates how many descriptors this packet is spread over (including this one): this will always be 1 if VIRTIO_NET_F_MRG_RXBUF was not negotiated. 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 the device chains them together to form a single packet. The other buffers will not begin with a struct virtio_net_hdr.
2. If *num_buffers* 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 *flags* MAY be set: if so, the checksum on the packet is incomplete and *csum_start* and *csum_offset* indicate how to calculate it (see Packet Transmission point 1).
4. If the VIRTIO_NET_F_GUEST_TSO4, TSO6 or UFO options were negotiated, then *gso_type* MAY be something other than VIRTIO_NET_HDR_GSO_NONE, and *gso_size* field indicates the desired MSS (see Packet Transmission point 2).

5.1.6.4.1 Device Requirements: Processing of Packets

If VIRTIO_NET_F_CSUM is not negotiated, the device MUST set *flags* to zero and the packet MUST be fully checksummed.

If VIRTIO_NET_F_GUEST_TSO4 is not negotiated, the device MUST NOT set *gso_type* to VIRTIO_NET_HDR_GSO_TCPV4.

If VIRTIO_NET_F_GUEST_UDP is not negotiated, the device MUST NOT set *gso_type* to VIRTIO_NET_HDR_GSO_UDP.

If VIRTIO_NET_F_GUEST_TSO6 is not negotiated, the device MUST NOT set *gso_type* to VIRTIO_NET_HDR_GSO_TCPV6.

A device SHOULD NOT send TCP packets requiring segmentation offload which have the Explicit Congestion Notification bit set, unless the VIRTIO_NET_F_GUEST_ECN feature is negotiated, in which case it MUST set the VIRTIO_NET_HDR_GSO_ECN bit in *gso_type*.

5.1.6.5 Control Virtqueue

The driver uses the control virtqueue (if VIRTIO_NET_F_CTRL_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 *ack* is not VIRTIO_NET_OK.

5.1.6.5.1 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: In general, these commands are best-effort: unwanted packets could still arrive.

5.1.6.5.2 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).

5.1.6.5.3 Setting MAC Address Filtering

```
struct virtio_net_ctrl_mac {
        le32 entries;
        u8 macs[entries][6];
};

#define VIRTIO_NET_CTRL_MAC    1
 #define VIRTIO_NET_CTRL_MAC_TABLE_SET        0
 #define VIRTIO_NET_CTRL_MAC_ADDR_SET         1
```

The device can filter incoming packets by any number of destination MAC addresses[4]. 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 (as described in struct virtio_net_ctrl_mac). The first table contains unicast addresses, and the second contains multicast addresses.

The VIRTIO_NET_CTRL_MAC_ADDR_SET command is used to set the default MAC address which rx filtering accepts (and if VIRTIO_NET_F_MAC_ADDR has been negotiated, this will be reflected in *mac* in config space).

---

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

The command-specific-data for VIRTIO_NET_CTRL_MAC_ADDR_SET is the 6-byte MAC address.

5.1.6.5.3.1 Device Requirements: Setting MAC Address Filtering

The device MUST have an empty MAC filtering table on reset.

The device MUST update the MAC filtering table before it consumes the VIRTIO_NET_CTRL_MAC_TABLE_SET command.

The device MUST update *mac* in config space before it consumes the VIRTIO_NET_CTRL_MAC_ADDR_SET command, if VIRTIO_NET_F_MAC_ADDR has been negotiated.

The device SHOULD drop incoming packets which have a destination MAC which matches neither the *mac* (or that set with VIRTIO_NET_CTRL_MAC_ADDR_SET) nor the MAC filtering table.

5.1.6.5.3.2 Driver Requirements: Setting MAC Address Filtering

The driver MUST follow the VIRTIO_NET_CTRL_MAC_TABLE_SET command by a le32 number, followed by that number of non-multicast MAC addresses, followed by another le32 number, followed by that number of multicast addresses. Either number MAY be 0.

5.1.6.5.3.3 Legacy Interface: Setting MAC Address Filtering

When using the legacy interface, transitional devices and drivers MUST format *entries* in struct virtio_net_ctrl_mac according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

Legacy drivers that didn't negotiate VIRTIO_NET_F_CTRL_MAC_ADDR changed *mac* in config space when NIC is accepting incoming packets. These drivers always wrote the mac value from first to last byte, therefore after detecting such drivers, a transitional device MAY defer MAC update, or MAY defer processing incoming packets until driver writes the last byte of *mac* in the config space.

5.1.6.5.4 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 little-endian 16-bit VLAN id as the command-specific-data.

5.1.6.5.4.1 Legacy Interface: VLAN Filtering

When using the legacy interface, transitional devices and drivers MUST format the VLAN id according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.1.6.5.5 Gratuitous Packet Sending

If the driver negotiates the VIRTIO_NET_F_GUEST_ANNOUNCE (depends on VIRTIO_NET_F_CTRL_VQ), the device can ask the driver to send gratuitous packets; this is usually done after the guest has been physically migrated, and needs to announce its presence on the new network links. (As hypervisor does not have the knowledge of guest network configuration (eg. tagged vlan) it is simplest to prod the guest in this way).

```
#define VIRTIO_NET_CTRL_ANNOUNCE       3
 #define VIRTIO_NET_CTRL_ANNOUNCE_ACK             0
```

The driver checks VIRTIO_NET_S_ANNOUNCE bit in the device configuration *status* field when it notices the changes of device configuration. The command VIRTIO_NET_CTRL_ANNOUNCE_ACK is used to indicate that driver has received the notification and device clears the VIRTIO_NET_S_ANNOUNCE bit in *status*.

Processing this notification involves:

1. Sending the gratuitous packets (eg. ARP) or marking there are pending gratuitous packets to be sent and letting deferred routine to send them.
2. Sending VIRTIO_NET_CTRL_ANNOUNCE_ACK command through control vq.

5.1.6.5.5.1 Driver Requirements: Gratuitous Packet Sending

If the driver negotiates VIRTIO_NET_F_GUEST_ANNOUNCE, it SHOULD notify network peers of its new location after it sees the VIRTIO_NET_S_ANNOUNCE bit in *status*. The driver MUST send a command on the command queue with class VIRTIO_NET_CTRL_ANNOUNCE and command VIRTIO_NET_CTRL_ANNOUNCE_ACK.

5.1.6.5.5.2 Device Requirements: Gratuitous Packet Sending

If VIRTIO_NET_F_GUEST_ANNOUNCE is negotiated, the device MUST clear the VIRTIO_NET_S_ANNOUNCE bit in *status* upon receipt of a command buffer with class VIRTIO_NET_CTRL_ANNOUNCE and command VIRTIO_NET_CTRL_ANNOUNCE_ACK before marking the buffer as used.

5.1.6.5.6 Automatic receive steering in multiqueue mode

If the driver negotiates the VIRTIO_NET_F_MQ feature bit (depends on VIRTIO_NET_F_CTRL_VQ), it MAY transmit outgoing packets on one of the multiple transmitq1…transmitqN and ask the device to queue incoming packets into one of the multiple receiveq1…receiveqN depending on the packet flow.

```
struct virtio_net_ctrl_mq {
        le16 virtqueue_pairs;
};

#define VIRTIO_NET_CTRL_MQ    4
 #define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET        0
 #define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_MIN        1
 #define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_MAX        0x8000
```

Multiqueue is disabled by default. The driver enables multiqueue by executing the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command, specifying the number of the transmit and receive queues to be used up to *max_virtqueue_pairs*; subsequently, transmitq1…transmitqn and receiveq1…receiveqn where n=*virtqueue_pairs* MAY be used.

When multiqueue is enabled, the device MUST use automatic receive steering based on packet flow. Programming of the receive steering classificator is implicit. After the driver transmitted a packet of a flow on transmitqX, the device SHOULD cause incoming packets for that flow to be steered to receiveqX. For uni-directional protocols, or where no packets have been transmitted yet, the device MAY steer a packet to a random queue out of the specified receiveq1…receiveqn.

Multiqueue is disabled by setting *virtqueue_pairs* to 1 (this is the default) and waiting for the device to use the command buffer.

5.1.6.5.6.1 Driver Requirements: Automatic receive steering in multiqueue mode

The driver MUST configure the virtqueues before enabling them with the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command.

The driver MUST NOT request a *virtqueue_pairs* of 0 or greater than *max_virtqueue_pairs* in the device configuration space.

The driver MUST queue packets only on any transmitq1 before the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command.

The driver MUST NOT queue packets on transmit queues greater than *virtqueue_pairs* once it has placed the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command in the available ring.

5.1.6.5.6.2 Device Requirements: Automatic receive steering in multiqueue mode

The device MUST queue packets only on any receiveq1 before the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command.

The device MUST NOT queue packets on receive queues greater than *virtqueue_pairs* once it has placed the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command in the used ring.

5.1.6.5.6.3 Legacy Interface: Automatic receive steering in multiqueue mode

When using the legacy interface, transitional devices and drivers MUST format *virtqueue_pairs* according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.1.6.5.7 Offloads State Configuration

If the VIRTIO_NET_F_CTRL_GUEST_OFFLOADS feature is negotiated, the driver can send control commands for dynamic offloads state configuration.

5.1.6.5.7.1 Setting Offloads State

```
le64 offloads;

#define VIRTIO_NET_F_GUEST_CSUM       1
#define VIRTIO_NET_F_GUEST_TSO4       7
#define VIRTIO_NET_F_GUEST_TSO6       8
#define VIRTIO_NET_F_GUEST_ECN        9
#define VIRTIO_NET_F_GUEST_UFO        10

#define VIRTIO_NET_CTRL_GUEST_OFFLOADS       5
 #define VIRTIO_NET_CTRL_GUEST_OFFLOADS_SET   0
```

The class VIRTIO_NET_CTRL_GUEST_OFFLOADS has one command: VIRTIO_NET_CTRL_GUEST_OFFLOADS_SET applies the new offloads configuration.

le64 value passed as command data is a bitmask, bits set define offloads to be enabled, bits cleared - offloads to be disabled.

There is a corresponding device feature for each offload. Upon feature negotiation corresponding offload gets enabled to preserve backward compartibility.

5.1.6.5.7.2 Driver Requirements: Setting Offloads State

A driver MUST NOT enable an offload for which the appropriate feature has not been negotiated.

5.1.6.5.7.3 Legacy Interface: Setting Offloads State

When using the legacy interface, transitional devices and drivers MUST format *offloads* according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.1.6.6 Legacy Interface: Framing Requirements

When using legacy interfaces, transitional drivers which have not negotiated VIRTIO_F_ANY_LAYOUT MUST use a single descriptor for the struct virtio_net_hdr on both transmit and receive, with the network data in the following descriptors.

Additionally, when using the control virtqueue (see 5.1.6.5) , transitional drivers which have not negotiated VIRTIO_F_ANY_LAYOUT MUST:

- for all commands, use a single 2-byte descriptor including the first two fields: *class* and *command*
- for all commands except VIRTIO_NET_CTRL_MAC_TABLE_SET use a single descriptor including command-specific-data with no padding.
- for the VIRTIO_NET_CTRL_MAC_TABLE_SET command use exactly two descriptors including command-specific-data with no padding: the first of these descriptors MUST include the virtio_net_ctrl_mac table structure for the unicast addresses with no padding, the second of these descriptors MUST include the virtio_net_ctrl_mac table structure for the multicast addresses with no padding.
- for all commands, use a single 1-byte descriptor for the *ack* field

See 2.4.4.

5.2 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.

5.2.1 Device ID

2

5.2.2 Virtqueues

0 requestq

5.2.3 Feature bits

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_TOPOLOGY (10) Device exports information on optimal I/O alignment.

5.2.3.1 Legacy Interface: Feature bits

VIRTIO_BLK_F_BARRIER (0) Device supports request barriers.

VIRTIO_BLK_F_SCSI (7) Device supports scsi packet commands.

VIRTIO_BLK_F_FLUSH (9) Cache flush command support.

VIRTIO_BLK_F_CONFIG_WCE (11) Device can toggle its cache between writeback and writethrough modes.

VIRTIO_BLK_F_FLUSH was also called VIRTIO_BLK_F_WCE: Legacy drivers MUST only negotiate this feature if they are capable of sending VIRTIO_BLK_T_FLUSH commands.

5.2.4 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 {
        le64 capacity;
        le32 size_max;
        le32 seg_max;
        struct virtio_blk_geometry {
                le16 cylinders;
                u8 heads;
                u8 sectors;
        } geometry;
        le32 blk_size;
        struct virtio_blk_topology {
                // # of logical blocks per physical block (log2)
                u8 physical_block_exp;
                // offset of first aligned logical block
                u8 alignment_offset;
                // suggested minimum I/O size in blocks
                le16 min_io_size;
                // optimal (suggested maximum) I/O size in blocks
                le32 opt_io_size;
        } topology;
        u8 reserved;
};
```

5.2.4.1 Legacy Interface: Device configuration layout

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_blk_config according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.2.5 Device Initialization

1. The device size can be read from *capacity*.
2. If the VIRTIO_BLK_F_BLK_SIZE feature is negotiated, *blk_size* can be read to determine the optimal sector size for the driver to use. This does not affect the units used in the protocol (always 512 bytes), but awareness of the correct value can affect 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_TOPOLOGY feature is negotiated, the fields in the *topology* struct can be read to determine the physical block size and optimal I/O lengths for the driver to use. This also does not affect the units in the protocol, only performance.

5.2.5.1 Legacy Interface: Device Initialization

The *reserved* field used to be called *writeback*. If the VIRTIO_BLK_F_CONFIG_WCE feature is offered, the cache mode can be read from *writeback*; the driver can also write to the field in order to toggle the cache between writethrough (0) and writeback (1) mode. If the feature is not available, the driver can instead look at the result of negotiating VIRTIO_BLK_F_FLUSH: the cache will be in writeback mode after reset if and only if VIRTIO_BLK_F_FLUSH is negotiated.

Some older legacy devices did not operate in writethrough mode even after a driver announced lack of support for VIRTIO_BLK_F_FLUSH.

5.2.6 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 {
        le32 type;
        le32 reserved;
        le64 sector;
        u8 data[][512];
        u8 status;
};
```

The type of the request is either a read (VIRTIO_BLK_T_IN), a write (VIRTIO_BLK_T_OUT), or a flush (VIRTIO_BLK_T_FLUSH).

```
#define VIRTIO_BLK_T_IN           0
#define VIRTIO_BLK_T_OUT          1
#define VIRTIO_BLK_T_FLUSH        4
```

The *sector* number indicates the offset (multiplied by 512) where the read or write is to occur. This field is unused and set to 0 for scsi packet commands and for flush commands.

The final *status* byte is written by the device: either VIRTIO_BLK_S_OK for success, VIRTIO_BLK_S_IOERR for device or driver error or VIRTIO_BLK_S_UNSUPP for a request unsupported by device:

```
#define VIRTIO_BLK_S_OK        0
#define VIRTIO_BLK_S_IOERR     1
#define VIRTIO_BLK_S_UNSUPP    2
```

5.2.6.1 Driver Requirements: Device Operation

A driver MUST NOT submit a request which would cause a read or write beyond *capacity*.

A driver SHOULD accept the VIRTIO_BLK_F_RO feature if offered.

A driver MUST set *sector* to 0 for a VIRTIO_BLK_T_FLUSH request. A driver SHOULD NOT include any data in a VIRTIO_BLK_T_FLUSH request.

5.2.6.2 Device Requirements: Device Operation

A device MUST set the *status* byte to VIRTIO_BLK_S_IOERR for a write request if the VIRTIO_BLK_F_RO feature if offered, and MUST NOT write any data.

Upon receipt of a VIRTIO_BLK_T_FLUSH request, the driver SHOULD ensure that any writes which were completed are committed to non-volatile storage.

5.2.6.3 Legacy Interface: Device Operation

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_blk_req according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

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

```
#define VIRTIO_BLK_T_FLUSH_OUT    5
```

The command VIRTIO_BLK_T_FLUSH_OUT was a synonym for VIRTIO_BLK_T_FLUSH; a driver MUST treat it as a VIRTIO_BLK_T_FLUSH command.

```
#define VIRTIO_BLK_T_BARRIER     0x80000000
```

If the device has VIRTIO_BLK_F_BARRIER feature the high bit (VIRTIO_BLK_T_BARRIER) indicates that this request acts as a barrier and that all preceding requests SHOULD be complete before this one, and all following requests SHOULD NOT be started until this is complete.

Note: A barrier does not flush caches in the underlying backend device in host, and thus does not serve as data consistency guarantee. Only a VIRTIO_BLK_T_FLUSH request does that.

If the device has VIRTIO_BLK_F_SCSI feature, it can also support scsi packet command requests, each of these requests is of form:

```
/* All fields are in guest's native endian. */
struct virtio_scsi_pc_req {
        u32 type;
        u32 ioprio;
        u64 sector;
        u8 cmd[];
        u8 data[][512];
#define SCSI_SENSE_BUFFERSIZE   96
        u8 sense[SCSI_SENSE_BUFFERSIZE];
        u32 errors;
        u32 data_len;
        u32 sense_len;
        u32 residual;
        u8 status;
};
```

A request type can also be a scsi packet command (VIRTIO_BLK_T_SCSI_CMD or VIRTIO_BLK_T_SCSI_CMD_OUT). The two types are equivalent, the device does not distinguish between them:

```
#define VIRTIO_BLK_T_SCSI_CMD     2
#define VIRTIO_BLK_T_SCSI_CMD_OUT 3
```

The *cmd* field is only present for scsi packet command requests, and indicates the command to perform. This field MUST reside in a single, separate device-readable buffer; command length can be derived from the length of this buffer.

Note that these first three (four for scsi packet commands) fields are always device-readable: *data* is either device-readable or device-writable, depending on the request. The size of the read or write can be derived from the total size of the request buffers.

sense is only present for scsi packet command requests, and indicates the buffer for scsi sense data.

data_len is only present for scsi packet command requests, this field is deprecated, and SHOULD be ignored by the driver. Historically, devices copied data length there.

sense_len is only present for scsi packet command requests and indicates the number of bytes actually written to the *sense* buffer.

residual field is only present for scsi packet command requests and indicates the residual size, calculated as data length - number of bytes actually transferred.

5.2.6.4 Legacy Interface: Framing Requirements

When using legacy interfaces, transitional drivers which have not negotiated VIRTIO_F_ANY_LAYOUT:

- MUST use a single 8-byte descriptor containing *type*, *reseved* and *sector*, followed by descriptors for *data*, then finally a separate 1-byte descriptor for *status*.
- For SCSI commands there are additional constraints. *errors*, *data_len*, *sense_len* and *residual* MUST reside in a single, separate device-writable descriptor, *sense* MUST reside in a single separate device-writable descriptor of size 96 bytes, and *errors*, *data_len*, *sense_len* and *residual* MUST reside a single separate device-writable descriptor.

See 2.4.4.

5.3 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 device about whether a particular port is a console port, adding new ports, port hot-plug/unplug, etc., and indication from the driver about whether a port or a device was successfully added, port open/close, 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.

5.3.1 Device ID

3

5.3.2 Virtqueues

0 receiveq(port0)

1 transmitq(port0)

2 control receiveq

3 control transmitq

4 receiveq(port1)

5 transmitq(port1)

...

The port 0 receive and transmit queues always exist: other queues only exist if VIRTIO_CONSOLE_F_MULTIPORT is set.

5.3.3 Feature bits

VIRTIO_CONSOLE_F_SIZE (0) Configuration *cols* and *rows* are valid.

VIRTIO_CONSOLE_F_MULTIPORT (1) Device has support for multiple ports; *max_nr_ports* is valid and control virtqueues will be used.

VIRTIO_CONSOLE_F_EMERG_WRITE (2) Device has support for emergency write. Configuration field emerg_wr is valid.

5.3.4 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 can be fetched.

If VIRTIO_CONSOLE_F_EMERG_WRITE is set then the driver can use emergency write to output a single character without initializing virtio queues, or even acknowledging the feature.

```
struct virtio_console_config {
        le16 cols;
        le16 rows;
        le32 max_nr_ports;
        le32 emerg_wr;
};
```

5.3.4.1 Legacy Interface: Device configuration layout

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_console_config according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.3.5 Device Initialization

1. If the VIRTIO_CONSOLE_F_EMERG_WRITE feature is offered, *emerg_wr* field of the configuration can be written at any time. Thus it works for very early boot debugging output as well as catastophic OS failures (eg. virtio ring corruption).
2. If the VIRTIO_CONSOLE_F_SIZE feature is negotiated, the driver can read the console dimensions from *cols* and *rows*.
3. If the VIRTIO_CONSOLE_F_MULTIPORT feature is negotiated, the driver can spawn multiple ports, not all of which are necessarily attached to a console. Some could be generic ports. In this case, the control virtqueues are enabled and according to *max_nr_ports*, the appropriate number of virtqueues are created. A control message indicating the driver is ready is sent to the device. The device can then send control messages for adding new ports to the device. After creating and initializing each port, a VIRTIO_CONSOLE_PORT_READY control message is sent to the device for that port so the device can let the driver know of any additional configuration options set for that port.

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

5.3.5.1 Device Requirements: Device Initialization

The device MUST allow a write to *emerg_wr*, even on an unconfigured device.

The device SHOULD transmit the lower byte written to *emerg_wr* to an appropriate log or output method.

5.3.6 Device Operation

1. For output, a buffer containing the characters is placed in the port's transmitq[5].
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 indicates that the updated size can be read from the configuration fields. This size applies to port 0 only.
4. If the driver negotiated the VIRTIO_CONSOLE_F_MULTIPORT feature, active ports are announced by the device using the VIRTIO_CONSOLE_PORT_ADD control message. The same message is used for port hot-plug as well.

5.3.6.1 Driver Requirements: Device Operation

The driver MUST NOT put a device-readable in a receiveq. The driver MUST NOT put a device-writable buffer in a transmitq.

5.3.6.2 Multiport Device Operation

If the driver negotiated the VIRTIO_CONSOLE_F_MULTIPORT, the two control queues are used to manipulate the different console ports: the control receiveq for messages from the device to the driver, and the control sendq for driver-to-device messages. The layout of the control messages is:

```
struct virtio_console_control {
        le32 id;    /* Port number */
        le16 event; /* The kind of control event */
        le16 value; /* Extra information for the event */
};
```

The values for *event* are:

VIRTIO_CONSOLE_DEVICE_READY (0) Sent by the driver at initialization to indicate that it is ready to receive control messages. A value of 1 indicates success, and 0 indicates failure. The port number *id* is unused.

VIRTIO_CONSOLE_DEVICE_ADD (1) Sent by the device, to create a new port. *value* is unused.

VIRTIO_CONSOLE_DEVICE_REMOVE (2) Sent by the device, to remove an existing port. *value* is unused.

VIRTIO_CONSOLE_PORT_READY (3) Sent by the driver in response to the device's VIRTIO_CONSOLE_PORT_ADD message, to indicate that the port is ready to be used. A *value* of 1 indicates success, and 0 indicates failure.

---

[5] 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. However, for generic serial ports with the O_NONBLOCK flag set, the polling limitation is relaxed and the consumed buffers are freed upon the next write or poll call or when a port is closed or hot-unplugged.

VIRTIO_CONSOLE_CONSOLE_PORT (4) Sent by the device to nominate a port as a console port. There MAY be more than one console port.

VIRTIO_CONSOLE_RESIZE (5) Sent by the device to indicate a console size change. *value* is unused. The buffer is followed by the number of columns and rows:

```
struct virtio_console_resize {
        le16 cols;
        le16 rows;
};
```

VIRTIO_CONSOLE_PORT_OPEN (6) This message is sent by both the device and the driver. *value* indicates the state: 0 (port closed) or 1 (port open). This allows for ports to be used directly by guest and host processes to communicate in an application-defined manner.

VIRTIO_CONSOLE_PORT_NAME (7) Sent by the device to give a tag to the port. This control command is immediately followed by the UTF-8 name of the port for identification within the guest (without a NUL terminator).

5.3.6.2.1 Device Requirements: Multiport Device Operation

The device MUST NOT specify a port which exists in a VIRTIO_CONSOLE_DEVICE_ADD message, nor a port which is equal or greater than *max_nr_ports*.

The device MUST NOT specify a port in VIRTIO_CONSOLE_DEVICE_REMOVE which has not been created with a previous VIRTIO_CONSOLE_DEVICE_ADD.

5.3.6.2.2 Driver Requirements: Multiport Device Operation

The driver MUST send a VIRTIO_CONSOLE_DEVICE_READY message if VIRTIO_CONSOLE_F_MULTIPORT is negotiated.

Upon receipt of a VIRTIO_CONSOLE_CONSOLE_PORT message, the driver SHOULD treat the port in a manner suitable for text console access and MUST respond with a VIRTIO_CONSOLE_PORT_OPEN message, which MUST have *value* set to 1.

5.3.6.3 Legacy Interface: Device Operation

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_console_control according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.3.6.4 Legacy Interface: Framing Requirements

When using legacy interfaces, transitional drivers which have not negotiated VIRTIO_F_ANY_LAYOUT MUST use only a single descriptor for all buffers in the control receiveq and control transmitq.

5.4 Entropy Device

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

5.4.1 Device ID

4

5.4.2 Virtqueues

0 requestq

5.4.3 Feature bits

None currently defined

5.4.4 Device configuration layout

None currently defined.

5.4.5 Device Initialization

1. The virtqueue is initialized

5.4.6 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.

5.4.6.1 Driver Requirements: Device Operation

The driver MUST NOT place driver-readable buffers into the queue.

The driver MUST examine the length written by the driver to determine how many random bytes were received.

5.4.6.2 Device Requirements: Device Operation

The device MUST place one or more random bytes into the buffer, but it MAY use less than the entire buffer length.

5.5 Legacy Interface: Memory Balloon Device

This device is deprecated, and thus only exists as a legacy device illustrated here for reference. The device number 13 is reserved for a new memory balloon interface which is expected in a future version of the standard.

The virtio memory balloon device is a primitive device for managing guest memory: the device asks for a certain amount of memory, and the driver 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.

5.5.1 Device ID

5

5.5.2 Virtqueues

0 inflateq

1 deflateq

2 statsq.

Virtqueue 2 only exists if VIRTIO_BALLON_F_STATS_VQ set.

5.5.3 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.

5.5.4 Device configuration layout

Both fields of this configuration are always available.

```
struct virtio_balloon_config {
        le32 num_pages;
        le32 actual;
};
```

Note that these fields are always little endian, despite convention that legacy device fields are guest endian.

5.5.5 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 device.

Device operation begins immediately.

5.5.6 Device Operation

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

1. *num_pages* configuration field is examined. If this is greater than the *actual* number of pages, the balloon wants more memory from the guest. If it is less than *actual*, the balloon doesn't need it all.
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[6] 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.

---

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

   (b) If the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is negotiated, the guest informs the device of pages before it uses them.
   (c) Otherwise, the guest MAY begin to re-use pages previously given to the balloon before the device has acknowledged their withdrawal[7].

4. In either case, once the device has completed the inflation or deflation, the driver updates *actual* to reflect the new number of pages in the balloon[8].

5.5.6.1 Driver Requirements: Device Operation

The driver SHOULD supply pages to the balloon when *num_pages* is greater than *actual*.

The driver MAY use pages from the balloon when *num_pages* is less than *actual*.

The driver MUST use the deflateq to inform the device of pages that it wants to use from the balloon.

If the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is negotiated, the driver MUST wait until the device has used the deflateq descriptor before using the pages.

The driver MUST update *actual* after changing the number of pages in the balloon.

5.5.6.2 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.

Each statistic consists of a 16 bit tag and a 64 bit value. All statistics are optional and the driver chooses which ones to supply. To guarantee backwards compatibility, the driver SHOULD omit unsupported statistics.

```
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));
```

5.5.6.2.1 Legacy Interface: Memory Statistics

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_balloon_stat according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

---

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

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

5.5.6.3 Memory Statistics Tags

VIRTIO_BALLOON_S_SWAP_IN (0) The amount of memory that has been swapped in (in bytes).

VIRTIO_BALLOON_S_SWAP_OUT (1) The amount of memory that has been swapped out to disk (in bytes).

VIRTIO_BALLOON_S_MAJFLT (2) The number of major page faults that have occurred.

VIRTIO_BALLOON_S_MINFLT (3) The number of minor page faults that have occurred.

VIRTIO_BALLOON_S_MEMFREE (4) The amount of memory not being used for any purpose (in bytes).

VIRTIO_BALLOON_S_MEMTOT (5) The total amount of memory available (in bytes).

5.6 SCSI Host Device

The virtio SCSI host device groups together one or more virtual logical units (such as disks), and allows communicating to them using the SCSI protocol. An instance of the device represents a SCSI host to which many targets and LUNs are attached.

The virtio SCSI device services two kinds of requests:

- command requests for a logical unit;
- task management functions related to a logical unit, target or command.

The device is also able to send out notifications about added and removed logical units. Together, these capabilities provide a SCSI transport protocol that uses virtqueues as the transfer medium. In the transport protocol, the virtio driver acts as the initiator, while the virtio SCSI host provides one or more targets that receive and process the requests.

This section relies on definitions from SAM.

5.6.1 Device ID

8

5.6.2 Virtqueues

0 controlq

1 eventq

2…n request queues

5.6.3 Feature bits

VIRTIO_SCSI_F_INOUT (0) A single request can include both device-readable and device-writable data buffers.

VIRTIO_SCSI_F_HOTPLUG (1) The host SHOULD enable reporting of hot-plug and hot-unplug events for LUNs and targets on the SCSI bus. The guest SHOULD handle hot-plug and hot-unplug events.

VIRTIO_SCSI_F_CHANGE (2) The host will report changes to LUN parameters via a VIRTIO_SCSI_T_PARAM_CHANGE event; the guest SHOULD handle them.

VIRTIO_SCSI_F_T10_PI (3) The extended fields for T10 protection information (DIF/DIX) are included in the SCSI request header.

5.6.4 Device configuration layout

All fields of this configuration are always available.

```
struct virtio_scsi_config {
        le32 num_queues;
        le32 seg_max;
        le32 max_sectors;
        le32 cmd_per_lun;
        le32 event_info_size;
        le32 sense_size;
        le32 cdb_size;
        le16 max_channel;
        le16 max_target;
        le32 max_lun;
};
```

num_queues is the total number of request virtqueues exposed by the device. The driver MAY use only one request queue, or it can use more to achieve better performance.

seg_max is the maximum number of segments that can be in a command. A bidirectional command can include *seg_max* input segments and *seg_max* output segments.

max_sectors is a hint to the driver about the maximum transfer size to use.

cmd_per_lun is tells the driver the maximum number of linked commands it can send to one LUN.

event_info_size is the maximum size that the device will fill for buffers that the driver places in the eventq. It is written by the device depending on the set of negotiated features.

sense_size is the maximum size of the sense data that the device will write. The default value is written by the device and MUST be 96, but the driver can modify it. It is restored to the default when the device is reset.

cdb_size is the maximum size of the CDB that the driver will write. The default value is written by the device and MUST be 32, but the driver can likewise modify it. It is restored to the default when the device is reset.

max_channel*, *max_target* and *max_lun can be used by the driver as hints to constrain scanning the logical units on the host to channel/target/logical unit numbers that are less than or equal to the value of the fields. *max_channel* SHOULD be zero. *max_target* SHOULD be less than or equal to 255. *max_lun* SHOULD be less than or equal to 16383.

5.6.4.1 Driver Requirements: Device configuration layout

The driver MUST NOT write to device configuration fields other than *sense_size* and *cdb_size*.

The driver MUST NOT send more than *cmd_per_lun* linked commands to one LUN, and MUST NOT send more than the virtqueue size number of linked commands to one LUN.

5.6.4.2 Device Requirements: Device configuration layout

On reset, the device MUST set *sense_size* to 96 and *cdb_size* to 32.

5.6.4.3 Legacy Interface: Device configuration layout

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_scsi_config according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.6.5 Device Requirements: Device Initialization

On initialization the driver SHOULD first discover the device's virtqueues.

If the driver uses the eventq, the driver SHOULD place at least one buffer in the eventq.

The driver MAY immediately issue requests[9] or task management functions[10].

5.6.6 Device Operation

Device operation consists of operating request queues, the control queue and the event queue.

5.6.6.1 Device Operation: Request Queues

The driver queues requests to an arbitrary request queue, and they are used by the device on that same queue. It is the responsibility of the driver to ensure strict request ordering for commands placed on different queues, because they will be consumed with no order constraints.

Requests have the following format:

```
struct virtio_scsi_req_cmd {
        // Device-readable part
        u8 lun[8];
        le64 id;
        u8 task_attr;
        u8 prio;
        u8 crn;
        u8 cdb[cdb_size];
        // The next two fields are only present if VIRTIO_SCSI_F_T10_PI
        // is negotiated.
        le32 pi_bytesout;
        le32 pi_bytesin;
        u8 pi_out[pi_bytesout];
        u8 dataout[];

        // Device-writable part
        le32 sense_len;
        le32 residual;
        le16 status_qualifier;
        u8 status;
        u8 response;
        u8 sense[sense_size];
        // The next two fields are only present if VIRTIO_SCSI_F_T10_PI
        // is negotiated
        u8 pi_in[pi_bytesin];
        u8 datain[];
};


/* command-specific response values */
#define VIRTIO_SCSI_S_OK                0
#define VIRTIO_SCSI_S_OVERRUN           1
#define VIRTIO_SCSI_S_ABORTED           2
#define VIRTIO_SCSI_S_BAD_TARGET        3
#define VIRTIO_SCSI_S_RESET             4
```

---

[9] For example, INQUIRY or REPORT LUNS.

[10] For example, I_T RESET.

```
#define VIRTIO_SCSI_S_BUSY              5
#define VIRTIO_SCSI_S_TRANSPORT_FAILURE 6
#define VIRTIO_SCSI_S_TARGET_FAILURE    7
#define VIRTIO_SCSI_S_NEXUS_FAILURE     8
#define VIRTIO_SCSI_S_FAILURE           9

/* task_attr */
#define VIRTIO_SCSI_S_SIMPLE            0
#define VIRTIO_SCSI_S_ORDERED           1
#define VIRTIO_SCSI_S_HEAD              2
#define VIRTIO_SCSI_S_ACA               3
```

lun addresses the REPORT LUNS well-known logical unit, or a target and logical unit in the virtio-scsi device's SCSI domain. When used to address the REPORT LUNS logical unit, *lun* is 0xC1, 0x01 and six zero bytes. The virtio-scsi device SHOULD implement the REPORT LUNS well-known logical unit.

When used to address a target and logical unit, the only supported format for *lun* is: first byte set to 1, second byte set to target, third and fourth byte representing a single level LUN structure, followed by four zero bytes. With this representation, a virtio-scsi device can serve up to 256 targets and 16384 LUNs per target. The device MAY also support having a well-known logical units in the third and fourth byte.

id is the command identifier ("tag").

task_attr defines the task attribute as in the table above, but all task attributes MAY be mapped to SIMPLE by the device. Some commands are defined by SCSI standards as "implicit head of queue"; for such commands, all task attributes MAY also be mapped to HEAD OF QUEUE. Drivers and applications SHOULD NOT send a command with the ORDERED task attribute if the command has an implicit HEAD OF QUEUE attribute, because whether the ORDERED task attribute is honored is vendor-specific.

crn may also be provided by clients, but is generally expected to be 0. The maximum CRN value defined by the protocol is 255, since CRN is stored in an 8-bit integer.

The CDB is included in *cdb* and its size, *cdb_size*, is taken from the configuration space.

All of these fields are defined in SAM and are always device-readable.

pi_bytesout determines the size of the *pi_out* field in bytes. If it is nonzero, the *pi_out* field contains outgoing protection information for write operations. *pi_bytesin* determines the size of the *pi_in* field in the device-writable section, in bytes. All three fields are only present if VIRTIO_SCSI_F_T10_PI has been negotiated.

The remainder of the device-readable part is the data output buffer, *dataout*.

sense and subsequent fields are always device-writable. *sense_len* indicates the number of bytes actually written to the sense buffer.

residual indicates the residual size, calculated as "data_length - number_of_transferred_bytes", for read or write operations. For bidirectional commands, the number_of_transferred_bytes includes both read and written bytes. A *residual* that is less than the size of *datain* means that *dataout* was processed entirely. A *residual* that exceeds the size of *datain* means that *dataout* was processed partially and *datain* was not processed at all.

If the *pi_bytesin* is nonzero, the *pi_in* field contains incoming protection information for read operations. *pi_in* is only present if VIRTIO_SCSI_F_T10_PI has been negotiated[11].

The remainder of the device-writable part is the data input buffer, *datain*.

---

[11] There is no separate residual size for *pi_bytesout* and *pi_bytesin*. It can be computed from the *residual* field, the size of the data integrity information per sector, and the sizes of *pi_out*, *pi_in*, *dataout* and *datain*.

5.6.6.1.1 Device Requirements: Device Operation: Request Queues

The device MUST write the *status* byte as the status code as defined in SAM.

The device MUST write the *response* byte as one of the following:

VIRTIO_SCSI_S_OK when the request was completed and the *status* byte is filled with a SCSI status code (not necessarily "GOOD").

VIRTIO_SCSI_S_OVERRUN if the content of the CDB (such as the allocation length, parameter length or transfer size) requires more data than is available in the datain and dataout buffers.

VIRTIO_SCSI_S_ABORTED if the request was cancelled due to an ABORT TASK or ABORT TASK SET task management function.

VIRTIO_SCSI_S_BAD_TARGET if the request was never processed because the target indicated by *lun* does not exist.

VIRTIO_SCSI_S_RESET if the request was cancelled due to a bus or device reset (including a task management function).

VIRTIO_SCSI_S_TRANSPORT_FAILURE if the request failed due to a problem in the connection between the host and the target (severed link).

VIRTIO_SCSI_S_TARGET_FAILURE if the target is suffering a failure and to tell the driver not to retry on other paths.

VIRTIO_SCSI_S_NEXUS_FAILURE if the nexus is suffering a failure but retrying on other paths might yield a different result.

VIRTIO_SCSI_S_BUSY if the request failed but retrying on the same path is likely to work.

VIRTIO_SCSI_S_FAILURE for other host or driver error. In particular, if neither *dataout* nor *datain* is empty, and the VIRTIO_SCSI_F_INOUT feature has not been negotiated, the request will be immediately returned with a response equal to VIRTIO_SCSI_S_FAILURE.

All commands must be completed before the virtio-scsi device is reset or unplugged. The device MAY choose to abort them, or if it does not do so MUST pick the VIRTIO_SCSI_S_FAILURE response.

5.6.6.1.2 Driver Requirements: Device Operation: Request Queues

task_attr, *prio* and *crn* SHOULD be zero.

Upon receiving a VIRTIO_SCSI_S_TARGET_FAILURE response, the driver SHOULD NOT retry the request on other paths.

5.6.6.1.3 Legacy Interface: Device Operation: Request Queues

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_scsi_req_cmd according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.6.6.2 Device Operation: controlq

The controlq is used for other SCSI transport operations. Requests have the following format:

```
struct virtio_scsi_ctrl {
        le32 type;
. . .
        u8 response;
};
```

```
/* response values valid for all commands */
#define VIRTIO_SCSI_S_OK                        0
#define VIRTIO_SCSI_S_BAD_TARGET                3
#define VIRTIO_SCSI_S_BUSY                      5
#define VIRTIO_SCSI_S_TRANSPORT_FAILURE         6
#define VIRTIO_SCSI_S_TARGET_FAILURE            7
#define VIRTIO_SCSI_S_NEXUS_FAILURE             8
#define VIRTIO_SCSI_S_FAILURE                   9
#define VIRTIO_SCSI_S_INCORRECT_LUN             12
```

The *type* identifies the remaining fields.

The following commands are defined:

- Task management function.

  ```
  #define VIRTIO_SCSI_T_TMF                      0

  #define VIRTIO_SCSI_T_TMF_ABORT_TASK           0
  #define VIRTIO_SCSI_T_TMF_ABORT_TASK_SET       1
  #define VIRTIO_SCSI_T_TMF_CLEAR_ACA            2
  #define VIRTIO_SCSI_T_TMF_CLEAR_TASK_SET       3
  #define VIRTIO_SCSI_T_TMF_I_T_NEXUS_RESET      4
  #define VIRTIO_SCSI_T_TMF_LOGICAL_UNIT_RESET   5
  #define VIRTIO_SCSI_T_TMF_QUERY_TASK           6
  #define VIRTIO_SCSI_T_TMF_QUERY_TASK_SET       7

  struct virtio_scsi_ctrl_tmf
  {
          // Device-readable part
          le32 type;
          le32 subtype;
          u8   lun[8];
          le64 id;
          // Device-writable part
          u8   response;
  }

  /* command-specific response values */
  #define VIRTIO_SCSI_S_FUNCTION_COMPLETE        0
  #define VIRTIO_SCSI_S_FUNCTION_SUCCEEDED       10
  #define VIRTIO_SCSI_S_FUNCTION_REJECTED        11
  ```

  The *type* is VIRTIO_SCSI_T_TMF; *subtype* defines which task management function. All fields except *response* are filled by the driver.

  Other fields which are irrelevant for the requested TMF are ignored but they are still present. *lun* is in the same format specified for request queues; the single level LUN is ignored when the task management function addresses a whole I_T nexus. When relevant, the value of *id* is matched against the id values passed on the requestq.

  The outcome of the task management function is written by the device in *response*. The command-specific response values map 1-to-1 with those defined in SAM.

  Task management function can affect the response value for commands that are in the request queue and have not been completed yet. For example, the device MUST complete all active commands on a logical unit or target (possibly with a VIRTIO_SCSI_S_RESET response code) upon receiving a ”logical unit reset” or ”I_T nexus reset” TMF. Similarly, the device MUST complete the selected commands (possibly with a VIRTIO_SCSI_S_ABORTED response code) upon receiving an ”abort task” or ”abort task set” TMF. Such effects MUST take place before the TMF itself is successfully completed, and the device MUST use memory barriers appropriately in order to ensure that the driver sees these writes in the correct order.

- Asynchronous notification query.

  ```
  #define VIRTIO_SCSI_T_AN_QUERY                    1

  struct virtio_scsi_ctrl_an {
  ```

```
    // Device-readable part
    le32 type;
    u8   lun[8];
    le32 event_requested;
    // Device-writable part
    le32 event_actual;
    u8   response;
}

#define VIRTIO_SCSI_EVT_ASYNC_OPERATIONAL_CHANGE  2
#define VIRTIO_SCSI_EVT_ASYNC_POWER_MGMT          4
#define VIRTIO_SCSI_EVT_ASYNC_EXTERNAL_REQUEST    8
#define VIRTIO_SCSI_EVT_ASYNC_MEDIA_CHANGE        16
#define VIRTIO_SCSI_EVT_ASYNC_MULTI_HOST          32
#define VIRTIO_SCSI_EVT_ASYNC_DEVICE_BUSY         64
```

By sending this command, the driver asks the device which events the given LUN can report, as described in paragraphs 6.6 and A.6 of SCSI MMC. The driver writes the events it is interested in into *event_requested*; the device responds by writing the events that it supports into *event_actual*.

The *type* is VIRTIO_SCSI_T_AN_QUERY. *lun* and *event_requested* are written by the driver. *event_actual* and *response* fields are written by the device.

No command-specific values are defined for the *response* byte.

- Asynchronous notification subscription.

```
#define VIRTIO_SCSI_T_AN_SUBSCRIBE                2

struct virtio_scsi_ctrl_an {
        // Device-readable part
        le32 type;
        u8   lun[8];
        le32 event_requested;
        // Device-writable part
        le32 event_actual;
        u8   response;
}
```

By sending this command, the driver asks the specified LUN to report events for its physical interface, again as described in SCSI MMC. The driver writes the events it is interested in into *event_requested*; the device responds by writing the events that it supports into *event_actual*.

Event types are the same as for the asynchronous notification query message.

The *type* is VIRTIO_SCSI_T_AN_SUBSCRIBE. *lun* and *event_requested* are written by the driver. *event_actual* and *response* are written by the device.

No command-specific values are defined for the response byte.

5.6.6.2.1 Legacy Interface: Device Operation: controlq

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_scsi_ctrl, struct virtio_scsi_ctrl_tmf, struct virtio_scsi_ctrl_an and struct virtio_scsi_ctrl_an according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.6.6.3 Device Operation: eventq

The eventq is populated by the driver for the device to report information on logical units that are attached to it. In general, the device will not queue events to cope with an empty eventq, and will end up dropping events if it finds no buffer ready. However, when reporting events for many LUNs (e.g. when a whole target disappears), the device can throttle events to avoid dropping them. For this reason, placing 10-15 buffers on the event queue is sufficient.

Buffers returned by the device on the eventq will be referred to as “events” in the rest of this section. Events have the following format:

```
#define VIRTIO_SCSI_T_EVENTS_MISSED   0x80000000

struct virtio_scsi_event {
        // Device-writable part
        le32 event;
        u8  lun[8];
        le32 reason;
}
```

The devices sets bit 31 in *event* to report lost events due to missing buffers.

The meaning of *reason* depends on the contents of *event*. The following events are defined:

- No event.

  ```
  #define VIRTIO_SCSI_T_NO_EVENT         0
  ```

  This event is fired in the following cases:

  - When the device detects in the eventq a buffer that is shorter than what is indicated in the configuration field, it MAY use it immediately and put this dummy value in *event*. A well-written driver will never observe this situation.
  - When events are dropped, the device MAY signal this event as soon as the drivers makes a buffer available, in order to request action from the driver. In this case, of course, this event will be reported with the VIRTIO_SCSI_T_EVENTS_MISSED flag.

- Transport reset

  ```
  #define VIRTIO_SCSI_T_TRANSPORT_RESET  1

  #define VIRTIO_SCSI_EVT_RESET_HARD         0
  #define VIRTIO_SCSI_EVT_RESET_RESCAN       1
  #define VIRTIO_SCSI_EVT_RESET_REMOVED      2
  ```

  By sending this event, the device signals that a logical unit on a target has been reset, including the case of a new device appearing or disappearing on the bus. The device fills in all fields. *event* is set to VIRTIO_SCSI_T_TRANSPORT_RESET. *lun* addresses a logical unit in the SCSI host.

  The *reason* value is one of the three #define values appearing above:

  VIRTIO_SCSI_EVT_RESET_REMOVED (“LUN/target removed”) is used if the target or logical unit is no longer able to receive commands.

  VIRTIO_SCSI_EVT_RESET_HARD (“LUN hard reset”) is used if the logical unit has been reset, but is still present.

  VIRTIO_SCSI_EVT_RESET_RESCAN (“rescan LUN/target”) is used if a target or logical unit has just appeared on the device.

  The “removed” and “rescan” events can happen when VIRTIO_SCSI_F_HOTPLUG feature was negotiated; when sent for LUN 0, they MAY apply to the entire target so the driver can ask the initiator to rescan the target to detect this.

  Events will also be reported via sense codes (this obviously does not apply to newly appeared buses or targets, since the application has never discovered them):

  - “LUN/target removed” maps to sense key ILLEGAL REQUEST, asc 0x25, ascq 0x00 (LOGICAL UNIT NOT SUPPORTED)

  - “LUN hard reset” maps to sense key UNIT ATTENTION, asc 0x29 (POWER ON, RESET OR BUS DEVICE RESET OCCURRED)
  - “rescan LUN/target” maps to sense key UNIT ATTENTION, asc 0x3f, ascq 0x0e (REPORTED LUNS DATA HAS CHANGED)

  The preferred way to detect transport reset is always to use events, because sense codes are only seen by the driver when it sends a SCSI command to the logical unit or target. However, in case events are dropped, the initiator will still be able to synchronize with the actual state of the controller if the driver asks the initiator to rescan of the SCSI bus. During the rescan, the initiator will be able to observe the above sense codes, and it will process them as if it the driver had received the equivalent event.

- Asynchronous notification

```
#define VIRTIO_SCSI_T_ASYNC_NOTIFY     2
```

  By sending this event, the device signals that an asynchronous event was fired from a physical interface.

  All fields are written by the device. *event* is set to VIRTIO_SCSI_T_ASYNC_NOTIFY. *lun* addresses a logical unit in the SCSI host. *reason* is a subset of the events that the driver has subscribed to via the “Asynchronous notification subscription” command.

- LUN parameter change

```
#define VIRTIO_SCSI_T_PARAM_CHANGE  3
```

  By sending this event, the device signals a change in the configuration parameters of a logical unit, for example the capacity or caching mode. *event* is set to VIRTIO_SCSI_T_PARAM_CHANGE. *lun* addresses a logical unit in the SCSI host.

  The same event SHOULD also be reported as a unit attention condition. *reason* contains the additional sense code and additional sense code qualifier, respectively in bits 0…7 and 8…15.

  Note: For example, a change in capacity will be reported as asc 0x2a, ascq 0x09 (CAPACITY DATA HAS CHANGED).

  For MMC devices (inquiry type 5) there would be some overlap between this event and the asynchronous notification event, so for simplicity the host never reports this event for MMC devices.

5.6.6.3.1 Driver Requirements: Device Operation: eventq

The driver SHOULD keep the eventq populated with buffers. These buffers MUST be device-writable, and SHOULD be at least *event_info_size* bytes long, and MUST be at least the size of struct virtio_scsi_event.

If *event* has bit 31 set, the driver SHOULD poll the logical units for unit attention conditions, and/or do whatever form of bus scan is appropriate for the guest operating system and SHOULD poll for asynchronous events manually using SCSI commands.

When receiving a VIRTIO_SCSI_T_TRANSPORT_RESET message with *reason* set to VIRTIO_SCSI_EVT_RESET_REMOVED or VIRTIO_SCSI_EVT_RESET_RESCAN for LUN 0, the driver SHOULD ask the initiator to rescan the target, in order to detect the case when an entire target has appeared or disappeared.

5.6.6.3.2 Device Requirements: Device Operation: eventq

The device MUST set bit 31 in *event* if events were lost due to missing buffers, and it MAY use a VIRTIO_SCSI_T_NO_EVENT event to report this.

The device MUST NOT send VIRTIO_SCSI_T_TRANSPORT_RESET messages with *reason* set to VIRTIO_SCSI_EVT_RESET_REMOVED or VIRTIO_SCSI_EVT_RESET_RESCAN unless VIRTIO_SCSI_F_HOTPLUG was negotiated.

The device MUST NOT report VIRTIO_SCSI_T_PARAM_CHANGE for MMC devices.

5.6.6.3.3 Legacy Interface: Device Operation: eventq

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_scsi_event according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.6.6.4 Legacy Interface: Framing Requirements

When using legacy interfaces, transitional drivers which have not negotiated VIRTIO_F_ANY_LAYOUT MUST use a single descriptor for the *lun*, *id*, *task_attr*, *prio*, *crn* and *cdb* fields, and MUST only use a single descriptor for the *sense_len*, *residual*, *status_qualifier*, *status*, *response* and *sense* fields.

6 Reserved Feature Bits

Currently there are three device-independent feature bits defined:

VIRTIO_F_RING_INDIRECT_DESC (28) Negotiating this feature indicates that the driver can use descriptors with the VIRTQ_DESC_F_INDIRECT flag set, as described in 2.4.5.3 Indirect Descriptors.

VIRTIO_F_RING_EVENT_IDX(29) This feature enables the *used_event* and the *avail_event* fields as described in 2.4.7 and 2.4.8.

VIRTIO_F_VERSION_1(32) This indicates compliance with this specification, giving a simple way to detect legacy devices or drivers.

6.1 Driver Requirements: Reserved Feature Bits

A driver MUST accept VIRTIO_F_VERSION_1 if it is offered. A driver MAY fail to operate further if VIRTIO_F_VERSION_1 is not offered.

6.2 Device Requirements: Reserved Feature Bits

A device MUST offer VIRTIO_F_VERSION_1. A device MAY fail to operate further if VIRTIO_F_VERSION_1 is not accepted.

6.3 Legacy Interface: Reserved Feature Bits

Transitional devices MAY offer the following:

VIRTIO_F_NOTIFY_ON_EMPTY (24) If this feature has been negotiated by driver, the device MUST issue an interrupt if the device runs out of available descriptors on a virtqueue, even though interrupts are suppressed using the VIRTQ_AVAIL_F_NO_INTERRUPT flag or the *used_event* field.

Note: An example of a driver using this feature is the legacy 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.

Transitional devices MUST offer, and if offered by the device transitional drivers MUST accept the following:

VIRTIO_F_ANY_LAYOUT (27) This feature indicates that the device accepts arbitrary descriptor layouts, as described in Section 2.4.4.3 Legacy Interface: Message Framing.

UNUSED (30) Bit 30 is used by qemu's implementation to check for experimental early versions of virtio which did not perform correct feature negotiation, and SHOULD NOT be negotiated.

7 Conformance

This chapter lists the conformance targets and clauses for each; this also forms a useful checklist which authors are asked to consult for their implementations!

7.1 Conformance Targets

Conformance targets:

Driver A driver MUST conform to three conformance clauses:

- Clause 7.2,
- One of clauses 7.2.1, 7.2.2 or 7.2.3.
- One of clauses 7.2.4, 7.2.5, 7.2.6, 7.2.7 or 7.2.8.

Device A device MUST conform to three conformance clauses:

- Clause 7.3,
- One of clauses 7.3.1, 7.3.2 or 7.3.3.
- One of clauses 7.3.4, 7.3.5, 7.3.6, 7.3.7 or 7.3.8.

7.2 Driver Conformance

A driver MUST conform to the following normative statements:

- 2.1.1
- 2.2.1
- 2.3.1
- 2.4.1
- 2.4.4.2
- 2.4.5.2
- 2.4.5.3.1
- 2.4.7.1
- 2.4.9.1
- 3.1.1
- 3.2.1.3.1
- 3.2.1.4.1
- 3.3.1
- 6.1

7.2.1 PCI Driver Conformance

A PCI driver MUST conform to the following normative statements:

- 4.1.2.2
- 4.1.3.1
- 4.1.4.1
- 4.1.4.3.2
- 4.1.4.5.2
- 4.1.4.7.2
- 4.1.5.1.2.1
- 4.1.5.1.3.2
- 4.1.5.4.2

7.2.2 MMIO Driver Conformance

An MMIO driver MUST conform to the following normative statements:

- 4.2.2.2
- 4.2.3.1.1
- 4.2.3.4.1

7.2.3 Channel I/O Driver Conformance

A Channel I/O driver MUST conform to the following normative statements:

- 4.3.1.2
- 4.3.2.1.2
- 4.3.3.1.2.2
- 4.3.3.2.2

7.2.4 Network Driver Conformance

A network driver MUST conform to the following normative statements:

- 5.1.4.2
- 5.1.6.2.1
- 5.1.6.3.1
- 5.1.6.5.3.2
- 5.1.6.5.5.1
- 5.1.6.5.6.1
- 5.1.6.5.7.2

7.2.5 Block Driver Conformance

A block driver MUST conform to the following normative statements:

- 5.2.6.1

7.2.6 Console Driver Conformance

A console driver MUST conform to the following normative statements:

- 5.3.6.1
- 5.3.6.2.2

7.2.7 Entropy Driver Conformance

An entropy driver MUST conform to the following normative statements:

- 5.4.6.1

7.2.8 SCSI Host Driver Conformance

An SCSI host driver MUST conform to the following normative statements:

- 5.6.4.1
- 5.6.6.1.2
- 5.6.6.3.1

7.3 Device Conformance

A device MUST conform to the following normative statements:

- 2.1.2
- 2.2.2
- 2.3.2
- 2.4.4.1
- 2.4.5.1
- 2.4.5.3.2
- 2.4.7.2
- 2.4.9.2
- 6.2

7.3.1 PCI Device Conformance

A PCI device MUST conform to the following normative statements:

- 4.1.1
- 4.1.2.1
- 4.1.4.2

- 4.1.4.3.1
- 4.1.4.4.1
- 4.1.4.5.1
- 4.1.4.6.1
- 4.1.4.7.1
- 4.1.5.1.3.1
- 4.1.5.3.1
- 4.1.5.4.1

7.3.2 MMIO Device Conformance

An MMIO device MUST conform to the following normative statements:

- 4.2.2.1

7.3.3 Channel I/O Device Conformance

A Channel I/O device MUST conform to the following normative statements:

- 4.3.1.1
- 4.3.2.1.1
- 4.3.2.2.1
- 4.3.2.7.3.1
- 4.3.3.1.2.1
- 4.3.3.2.1

7.3.4 Network Device Conformance

A network device MUST conform to the following normative statements:

- 5.1.4.1
- 5.1.6.3.2
- 5.1.6.4.1
- 5.1.6.5.3.1
- 5.1.6.5.5.2
- 5.1.6.5.6.2

7.3.5 Block Device Conformance

A block device MUST conform to the following normative statements:

- 5.2.6.2

7.3.6 Console Device Conformance

A console device MUST conform to the following normative statements:

- 5.3.5.1
- 5.3.6.2.1

7.3.7 Entropy Device Conformance

An entropy device MUST conform to the following normative statements:

- 5.4.6.2

7.3.8 SCSI Host Device Conformance

An SCSI host device MUST conform to the following normative statements:

- 5.6.4.2
- 5.6.5
- 5.6.6.1.1
- 5.6.6.3.2

7.4 Legacy Interface: Transitional Device and Transitional Driver Conformance

A conformant implementation MUST be either transitional or non-transitional, see 1.3.1.

A non-transitional implementation conforms to this specification if it satisfies all of the MUST or REQUIRED level requirements defined above.

An implementation MAY choose to implement OPTIONAL support for the legacy interface, including support for legacy drivers or devices, by additionally conforming to all of the MUST or REQUIRED level requirements for the legacy interface for the transitional devices and drivers.

The requirements for the legacy interface for transitional implementations are located in sections named “Legacy Interface” listed below:

- Section 2.2.3
- Section 2.3.3
- Section 2.3.4
- Section 2.4.2
- Section 2.4.3
- Section 2.4.4.3
- Section 3.1.2
- Section 4.1.2.3
- Section 4.1.4.8
- Section 4.1.5.1.2.2
- Section 4.1.5.1.4.1
- Section 4.2.4

- Section 4.3.2.1.3
- Section 4.3.2.2.2
- Section 4.3.3.1.3
- Section 4.3.2.7.4
- Section 5.1.3.2
- Section 5.1.4.3
- Section 5.1.6.1
- Section 5.1.6.5.3.3
- Section 5.1.6.5.4.1
- Section 5.1.6.5.6.3
- Section 5.1.6.5.7.3
- Section 5.2.3.1
- Section 5.2.4.1
- Section 5.2.5.1
- Section 5.2.6.3
- Section 5.3.4.1
- Section 5.3.6.3
- Section 5.5.6.1
- Section 5.5.6.2.1
- Section 5.6.4.3
- Section 5.6.6.1.3
- Section 5.6.6.2.1
- Section 5.6.6.3.3
- Section 6.3

Appendix A. virtio_ring.h

This file is also available at the link http://docs.oasis-open.org/virtio/virtio/v1.0/cs02/listings/virtio_ring.h. All definitions in this section are for non-normative reference only.

```
#ifndef VIRTQUEUE_H
#define VIRTQUEUE_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
 * Copyright 2011, Red Hat, Inc
 * 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'' AND
 * 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.
 */
#include <stdint.h>

/* This marks a buffer as continuing via the next field. */
#define VIRTQ_DESC_F_NEXT       1
/* This marks a buffer as write-only (otherwise read-only). */
#define VIRTQ_DESC_F_WRITE      2
/* This means the buffer contains a list of buffer descriptors. */
#define VIRTQ_DESC_F_INDIRECT   4

/* The device uses this in used->flags to advise the driver: don't kick me
 * when you add a buffer.  It's unreliable, so it's simply an
 * optimization. */
#define VIRTQ_USED_F_NO_NOTIFY  1
/* The driver uses this in avail->flags to advise the device: don't
 * interrupt me when you consume a buffer.  It's unreliable, so it's
 * simply an optimization.  */
#define VIRTQ_AVAIL_F_NO_INTERRUPT      1

/* Support for indirect descriptors */
#define VIRTIO_F_INDIRECT_DESC    28

/* Support for avail_idx and used_idx fields */
#define VIRTIO_F_EVENT_IDX        29

```

```
/* Arbitrary descriptor layouts. */
#define VIRTIO_F_ANY_LAYOUT       27

/* Virtqueue descriptors: 16 bytes.
 * These can chain together via "next". */
struct virtq_desc {
        /* Address (guest-physical). */
        le64 addr;
        /* Length. */
        le32 len;
        /* The flags as indicated above. */
        le16 flags;
        /* We chain unused descriptors via this, too */
        le16 next;
};

struct virtq_avail {
        le16 flags;
        le16 idx;
        le16 ring[];
        /* Only if VIRTIO_F_EVENT_IDX: le16 used_event; */
};

/* le32 is used here for ids for padding reasons. */
struct virtq_used_elem {
        /* Index of start of used descriptor chain. */
        le32 id;
        /* Total length of the descriptor chain which was written to. */
        le32 len;
};

struct virtq_used {
        le16 flags;
        le16 idx;
        struct virtq_used_elem ring[];
        /* Only if VIRTIO_F_EVENT_IDX: le16 avail_event; */
};

struct virtq {
        unsigned int num;

        struct virtq_desc *desc;
        struct virtq_avail *avail;
        struct virtq_used *used;
};

static inline int virtq_need_event(uint16_t event_idx, uint16_t new_idx, uint16_t old_idx)
{
         return (uint16_t)(new_idx - event_idx - 1) < (uint16_t)(new_idx - old_idx);
}

/* Get location of event indices (only with VIRTIO_F_EVENT_IDX) */
static inline le16 *virtq_used_event(struct virtq *vq)
{
        /* For backwards compat, used event index is at *end* of avail ring. */
        return &vq->avail->ring[vq->num];
}

static inline le16 *virtq_avail_event(struct virtq *vq)
{
        /* For backwards compat, avail event index is at *end* of used ring. */
        return (le16 *)&vq->used->ring[vq->num];
}
#endif /* VIRTQUEUE_H */
```

Appendix B. Creating New Device Types

Various considerations are necessary when creating a new device type.

B.1 How Many Virtqueues?

It is possible that a very simple device will operate entirely through its device 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.

B.2 What Device Configuration Space Layout?

Device configuration space should only be used for initialization-time parameters. It is a limited resource with no synchronization between field written by the driver, so for most uses it is 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).

Remember that configuration fields over 32 bits wide might not be atomically writable by the driver.

B.3 What Device Number?

Device numbers can be reserved by the OASIS committee: email virtio-dev@lists.oasis-open.org to secure a unique one.

Meanwhile for experimental drivers, use 65535 and work backwards.

B.4 How many MSI-X vectors? (for PCI)

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 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.

B.5 Device Improvements

Any change to device configuration space, or new virtqueues, or behavioural changes, should be indicated by negotiation of a new feature bit. This establishes clarity[1] 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.

---

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

Appendix C. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:
Amit Shah, Red Hat
Amos Kong, Red Hat
Anthony Liguori, IBM
Bruce Rogers, Novell
Bryan Venteicher, NetApp
Cornelia Huck, IBM
Daniel Kiper, Oracle
Geoff Brown, Machine-to-Machine Intelligence (M2MI) Corporation
Gershon Janssen, Individual Member
James Bottomley, Parallels IP Holdings GmbH
Luiz Capitulino, Red Hat
Michael S. Tsirkin, Red Hat
Paolo Bonzini, Red Hat
Pawel Moll, ARM
Rusty Russell, IBM
Sasha Levin, Oracle
Sergey Tverdyshev, Thales e-Security
Stefan Hajnoczi, Red Hat
Tom Lyon, Samya Systems, Inc.

The following non-members have provided valuable feedback on this specification and are gratefully acknowledged:

Reviewers:
Andrew Thornton, Google
Arun Subbarao, LynuxWorks
Fam Zheng, Red Hat
Gerd Hoffmann, Red Hat
Laura Novich, Red Hat
Patrick Durusau, Technical Advisory Board, OASIS
Thomas Huth, IBM
Yan Vugenfirer, Red Hat / Daynix
Brian Foley, ARM
David Alan Gilbert, Red Hat

Appendix D. Revision History

The following changes have been made since the previous version of this specification:

Revision	Date	Editor	Changes Made
448	22 Dec 2014	Michael S. Tsirkin	VIRTIO-120: virtio: fix used element size General ring description lists size for used ring elements as 4, it must be 8. See 2.4.
449	22 Doc 2014	Cornelia Huck	VIRTIO-125: block: fixup section levels The specification for the configuration layout for block devices should be its own subsection as for all other devices and not be hidden beneath ”Feature bits”. The normative sections for device operation should appear under the device operation section. See 5.2.4.
450	22 Dec 2014	Cornelia Huck	VIRTIO-127: ccw: two-stage indicators for legacy devices Some legacy devices will support two-stage queue indicators and therefore won’t reject CCW_CMD_SET_IND_ADAPTER. Note this. See 4.3.2.7.4.
452	22 Dec 2014	Michael S. Tsirkin	VIRTIO-115: formatting: escape \ldots in lstlisting \ldots does not work within lstlisting, the result is \ldots verbatim in the PDF output. To fix, make $ an escape character, and escape the sequence: $\ldots$ See 5.6.6.2.
455,457	23 Dec 2014	Michael S. Tsirkin	acknowledgements: acknowledge dgilbert Acknowledge David Alan Gilbert for reporting VIRTIO-120. See C.