# Virtual I/O Device (VIRTIO) Version 1.1

## Working Draft

## 20 December 2018

### Specification URIs

**This version:**
https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/tex/ (Authoritative)
https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/virtio-v1.1-wd01.pdf
https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/virtio-v1.1-wd01.html

**Previous version:**
N/A

**Latest version:**
https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf
https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html

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

**Chair:**
Michael S. Tsirkin (mst@redhat.com), Red Hat

**Editors:**
Michael S. Tsirkin (mst@redhat.com), Red Hat
Cornelia Huck (cohuck@redhat.com), Red Hat

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

- Example Driver Listing:
  https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/listings/

**Related work:**
This specification replaces or supersedes:

- Virtual I/O Device (VIRTIO) Version 1.0. Edited by Rusty Russell, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. Latest version:
  https://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf
- 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/.
This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. 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 TC’s web page (https://github.com/oasis-tcs/virtio-admin/blob/master/IPR.md).
Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product’s prose narrative document(s), the content in the separate plain text file prevails.

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

**[VIRTIO-v1.1]**
*Virtual I/O Device (VIRTIO) Version 1.1*. Edited by Michael S. Tsirkin and Cornelia Huck. 20 December 2018. OASIS Working Draft. https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/virtio-v1.1-wd01.html. Latest version: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html.

# Notices



All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the ”OASIS IPR Policy”). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. 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 TC’s web page (https://github.com/oasis-tcs/virtio-admin/blob/master/IPR.md).

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an ”AS IS” basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

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,<br>http://www.t10.org/cgi-bin/ac.pl?t=f&f=sam4r05.pdf |
| **[SCSI MMC]** | SCSI Multimedia Commands,<br>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<br>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.

### 1.3.2 Transition from earlier specification drafts

For devices and drivers already implementing the legacy interface, some changes will have to be made to support this specification.

In this case, it might be beneficial for the reader to focus on sections tagged ”Legacy Interface” in the section title. These highlight the changes made since the earlier drafts.

## 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.

Some of the fields to be defined in this specification don't start or don't end on a byte boundary. Such fields are called bit-fields. A set of bit-fields is always a sub-division of an integer typed field.

Bit-fields within integer fields are always listed in order, from the least significant to the most significant bit. The bit-fields are considered unsigned integers of the specified width with the next in significance relationship of the bits preserved.

For example:

```
struct S {
        be16 {
                A : 15;
                B : 1;
        } x;
        be16 y;
};
```

documents the value A stored in the low 15 bit of *x* and the value B stored in the high bit of *x*, the 16-bit integer *x* in turn stored using the big-endian byte order at the beginning of the structure S, and being followed immediately by an unsigned integer *y* stored in big-endian byte order at an offset of 2 bytes (16 bits) from the beginning of the structure.

Note that this notation somewhat resembles the C bitfield syntax but should not be naively converted to a bitfield notation for portable code: it matches the way bitfields are packed by C compilers on little-endian architectures but not the way bitfields are packed by C compilers on big-endian architectures.

Assuming that CPU_TO_BE16 converts a 16-bit integer from a native CPU to the big-endian byte order, the following is the equivalent portable C code to generate a value to be stored into *x*:

```
CPU_TO_BE16(B << 15 | A)
```

# 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
- Notifications
- 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 (listed below in the order in which they would be typically set):

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

**DRIVER (2)** Indicates that the guest OS knows how to drive the device.

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

**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.

**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.

### 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 send any used buffer notifications to 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 37** Feature bits reserved for extensions to the queue and feature negotiation mechanisms

**38 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.

If a device has successfully negotiated a set of features at least once (by accepting the FEATURES_OK *device status* bit during device initialization), then it SHOULD NOT fail re-negotiation of the same set of features after a device or system reset. Failure to do so would interfere with resuming from suspend and error recovery.

### 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 Notifications

The notion of sending a notification (driver to device or device to driver) plays an important role in this specification. The modus operandi of the notifications is transport specific.

There are three types of notifications:

- configuration change notification
- available buffer notification
- used buffer notification.

Configuration change notifications and used buffer notifications are sent by the device, the recipient is the driver. A configuration change notification indicates that the device configuration space has changed; a used buffer notification indicates that a buffer may have been made used on the virtqueue designated by the notification.

Available buffer notifications are sent by the driver, the recipient is the device. This type of notification indicates that a buffer may have been made available on the virtqueue designated by the notification.

The semantics, the transport-specific implementations, and other important aspects of the different notifications are specified in detail in the following chapters.

Most transports implement notifications sent by the device to the driver using interrupts. Therefore, in previous versions of this specification, these notifications were often called interrupts. Some names defined in this specification still retain this interrupt terminology. Occasionally, the term event is used to refer to a notification or a receipt of a notification.

## 2.4 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.

### 2.4.1 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.

### 2.4.2 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.

### 2.4.3 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.

### 2.4.4 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.

## 2.5 Virtqueues

The mechanism for bulk data transport on virtio devices is pretentiously called a virtqueue. Each device can have zero or more virtqueues[1].

Driver makes requests available to device by adding an available buffer to the queue - i.e. adding a buffer describing the request to a virtqueue, and optionally triggering a driver event - i.e. sending an available buffer notification to the device.

Device executes the requests and - when complete - adds a used buffer to the queue - i.e. lets the driver know by marking the buffer as used. Device can then trigger a device event - i.e. send a used buffer notification to the driver.

Device reports the number of bytes it has written to memory for each buffer it uses. This is referred to as "used length".

Device is not generally required to use buffers in the same order in which they have been made available by the driver.

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

Some devices always use descriptors in the same order in which they have been made available. These devices can offer the VIRTIO_F_IN_ORDER feature. If negotiated, this knowledge might allow optimizations or simplify driver and/or device code.

Each virtqueue can consist of up to 3 parts:

- Descriptor Area - used for describing buffers
- Driver Area - extra data supplied by driver to the device
- Device Area - extra data supplied by device to driver

**Note:** Note that previous versions of this spec used different names for these parts (following 2.6):

- Descriptor Table - for the Descriptor Area
- Available Ring - for the Driver Area
- Used Ring - for the Device Area

Two formats are supported: Split Virtqueues (see 2.6 Split Virtqueues) and Packed Virtqueues (see 2.7 Packed Virtqueues).

Every driver and device supports either the Packed or the Split Virtqueue format, or both.

## 2.6 Split Virtqueues

The split virtqueue format was the only format supported by the version 1.0 (and earlier) of this standard.

The split virtqueue format separates the virtqueue into several parts, where each part is write-able by either the driver or the device, but not both. Multiple parts and/or locations within a part need to be updated when making a buffer available and when marking it as used.

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 - occupies the Descriptor Area
- Available Ring - occupies the Driver Area
- Used Ring - occupies the Device Area

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

The memory alignment 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 a used buffer notification.

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

### 2.6.1 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.

### 2.6.2 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; henceforth referred to as Queue Align) and consists of three parts:

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

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 virtqueue according to the following formula (Queue Align given in qalign and Queue Size given in qsz):

```
#define ALIGN(x) (((x) + qalign) & ~qalign)
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 Queue Align boundary.
        u8 pad[ Padding ];

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

### 2.6.3 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.

### 2.6.4 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!

#### 2.6.4.1 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.

#### 2.6.4.2 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.

#### 2.6.4.3 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.

### 2.6.5 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.

If VIRTIO_F_IN_ORDER has been negotiated, driver uses descriptors in ring order: starting from offset 0 in the table, and wrapping around at the end of the table.

**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.

#### 2.6.5.1 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).

#### 2.6.5.2 Driver Requirements: The Virtqueue Descriptor Table

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

If VIRTIO_F_IN_ORDER has been negotiated, and when making a descriptor with VRING_DESC_F_NEXT set in *flags* at offset $x$ in the table available to the device, driver MUST set *next* to $0$ for the last descriptor in the table (where $x = queue\_size - 1$) and to $x + 1$ for the rest of the descriptors.

#### 2.6.5.3 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_queue.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.

If VIRTIO_F_IN_ORDER has been negotiated, indirect descriptors use sequential indices, in-order: index 0 followed by index 1 followed by index 2, etc.

##### 2.6.5.3.1 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.

A driver MUST NOT set both VIRTQ_DESC_F_INDIRECT and VIRTQ_DESC_F_NEXT in *flags*.

If VIRTIO_F_IN_ORDER has been negotiated, indirect descriptors MUST appear sequentially, with *next* taking the value of 1 for the 1st descriptor, 2 for the 2nd one, etc.

##### 2.6.5.3.2 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.

The device MUST handle the case of zero or more normal chained descriptors followed by a single descriptor with *flags*&VIRTQ_DESC_F_INDIRECT.

**Note:** While unusual (most implementations either create a chain solely using non-indirect descriptors, or use a single indirect element), such a layout is valid.

### 2.6.6 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.

#### 2.6.6.1 Driver Requirements: The Virtqueue Available Ring

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

### 2.6.7 Used Buffer Notification 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 notifications when buffers are used. Otherwise *used_event* is a more performant alternative where the driver specifies how far the device can progress before a notification is required.

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

#### 2.6.7.1 Driver Requirements: Used Buffer Notification 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 notifications 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 notifications are unnecessary until the device writes an 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 notifications from the device.

#### 2.6.7.2 Device Requirements: Used Buffer Notification 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 a notification.
  - If *flags* is 0, the device MUST send a notification.

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 a notification.
  - Otherwise the device SHOULD NOT send a notification.

**Note:** For example, if *used_event* is 0, then a device using

VIRTIO_F_EVENT_IDX would send a used buffer notification to the driver after the first buffer is used (and again after the 65536th buffer, etc).

### 2.6.8 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.

**Note:** *len* is particularly useful for drivers using untrusted buffers: if a driver does not know exactly how much has been written by the device, the driver would have to zero the buffer in advance to ensure no data leakage occurs.

For example, a network driver may hand a received buffer directly to an unprivileged userspace application. If the network device has not overwritten the bytes which were in that buffer, this could leak the contents of freed memory from other processes to the application.

*idx* field indicates where the device 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 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.

#### 2.6.8.1 Legacy Interface: The Virtqueue Used Ring

Historically, many drivers ignored the *len* value, as a result, many devices set *len* incorrectly. Thus, when using the legacy interface, it is generally a good idea to ignore the *len* value in used ring entries if possible. Specific known issues are listed per device type.

#### 2.6.8.2 Device Requirements: The Virtqueue Used Ring

The device MUST set *len* prior to updating the used *idx*.

The device MUST write at least *len* bytes to descriptor, beginning at the first device-writable buffer, prior to updating the used *idx*.

The device MAY write more than *len* bytes to descriptor.

**Note:** There are potential error cases where a device might not know what parts of the buffers have been written. This is why *len* is permitted to be an underestimate: that's preferable to the driver believing that uninitialized memory has been overwritten when it has not.

#### 2.6.8.3 Driver Requirements: The Virtqueue Used Ring

The driver MUST NOT make assumptions about data in device-writable buffers beyond the first *len* bytes, and SHOULD ignore this data.

### 2.6.9 In-order use of descriptors

Some devices always use descriptors in the same order in which they have been made available. These devices can offer the VIRTIO_F_IN_ORDER feature. If negotiated, this knowledge allows devices to notify the use of a batch of buffers to the driver by only writing out a single used ring entry with the *id* corresponding to the head entry of the descriptor chain describing the last buffer in the batch.

The device then skips forward in the ring according to the size of the batch. Accordingly, it increments the used *idx* by the size of the batch.

The driver needs to look up the used *id* and calculate the batch size to be able to advance to where the next used ring entry will be written by the device.

This will result in the used ring entry at an offset matching the first available ring entry in the batch, the used ring entry for the next batch at an offset matching the first available ring entry in the next batch, etc.

The skipped buffers (for which no used ring entry was written) are assumed to have been used (read or written) by the device completely.

### 2.6.10 Available Buffer Notification Suppression

The device can suppress available buffer notifications in a manner analogous to the way drivers can suppress used buffer notifications as detailed in section 2.6.7. 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.

#### 2.6.10.1 Driver Requirements: Available Buffer 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.

#### 2.6.10.2 Device Requirements: Available Buffer 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.

### 2.6.11 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) in A virtio_queue.h.

### 2.6.12 Virtqueue Operation

There are two parts to virtqueue operation: supplying new available 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.

What follows is the requirements of each of these two parts when using the split virtqueue format in more detail.

### 2.6.13 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 2.6.5 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 a suitable 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. The driver sends an available buffer notification to the device if such notifications are not suppressed.

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.

#### 2.6.13.1 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.

#### 2.6.13.2 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;
```

#### 2.6.13.3 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.

#### 2.6.13.3.1 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.

### 2.6.13.4 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 2.6.10.

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

#### 2.6.13.4.1 Driver Requirements: Notifying The Device

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

## 2.6.14 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 sends a used buffer notification to the driver as detailed in section 2.6.7.

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

```
virtq_disable_used_buffer_notifications(vq);

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

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

                virtq_disable_used_buffer_notifications(vq);
        }

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

# 2.7 Packed Virtqueues

Packed virtqueues is an alternative compact virtqueue layout using read-write memory, that is memory that is both read and written by both host and guest.

Use of packed virtqueues is negotiated by the VIRTIO_F_RING_PACKED feature bit.

Packed virtqueues support up to $2^{15}$ entries each.

With current transports, virtqueues are located in guest memory allocated by the driver. Each packed virtqueue consists of three parts:

- Descriptor Ring - occupies the Descriptor Area

- Driver Event Suppression - occupies the Driver Area
- Device Event Suppression - occupies the Device Area

Where the Descriptor Ring in turn consists of descriptors, and where each descriptor can contain the following parts:

- Buffer ID
- Element Address
- Element Length
- Flags

A buffer consists of zero or more device-readable physically-contiguous elements followed by zero or more physically-contiguous device-writable elements (each buffer has at least one element).

When the driver wants to send such a buffer to the device, it writes at least one available descriptor describing elements of the buffer into the Descriptor Ring. The descriptor(s) are associated with a buffer by means of a Buffer ID stored within the descriptor.

The driver then notifies the device. When the device has finished processing the buffer, it writes a used device descriptor including the Buffer ID into the Descriptor Ring (overwriting a driver descriptor previously made available), and sends a used event notification.

The Descriptor Ring is used in a circular manner: the driver writes descriptors into the ring in order. After reaching the end of the ring, the next descriptor is placed at the head of the ring. Once the ring is full of driver descriptors, the driver stops sending new requests and waits for the device to start processing descriptors and to write out some used descriptors before making new driver descriptors available.

Similarly, the device reads descriptors from the ring in order and detects that a driver descriptor has been made available. As processing of descriptors is completed, used descriptors are written by the device back into the ring.

Note: after reading driver descriptors and starting their processing in order, the device might complete their processing out of order. Used device descriptors are written in the order in which their processing is complete.

The Device Event Suppression data structure is write-only by the device. It includes information for reducing the number of device events - i.e. sending fewer available buffer notifications to the device.

The Driver Event Suppression data structure is read-only by the device. It includes information for reducing the number of driver events - i.e. sending fewer used buffer notifications to the driver.

### 2.7.1 Driver and Device Ring Wrap Counters

Each of the driver and the device are expected to maintain, internally, a single-bit ring wrap counter initialized to 1.

The counter maintained by the driver is called the Driver Ring Wrap Counter. The driver changes the value of this counter each time it makes available the last descriptor in the ring (after making the last descriptor available).

The counter maintained by the device is called the Device Ring Wrap Counter. The device changes the value of this counter each time it uses the last descriptor in the ring (after marking the last descriptor used).

It is easy to see that the Driver Ring Wrap Counter in the driver matches the Device Ring Wrap Counter in the device when both are processing the same descriptor, or when all available descriptors have been used.

To mark a descriptor as available and used, both the driver and the device use the following two flags:

```
#define VIRTQ_DESC_F_AVAIL     (1 << 7)
#define VIRTQ_DESC_F_USED      (1 << 15)
```

To mark a descriptor as available, the driver sets the VIRTQ_DESC_F_AVAIL bit in Flags to match the internal Driver Ring Wrap Counter. It also sets the VIRTQ_DESC_F_USED bit to match the *inverse* value (i.e. to not match the internal Driver Ring Wrap Counter).

To mark a descriptor as used, the device sets the VIRTQ_DESC_F_USED bit in Flags to match the internal Device Ring Wrap Counter. It also sets the VIRTQ_DESC_F_AVAIL bit to match the *same* value.

Thus VIRTQ_DESC_F_AVAIL and VIRTQ_DESC_F_USED bits are different for an available descriptor and equal for a used descriptor.

Note that this observation is mostly useful for sanity-checking as these are necessary but not sufficient conditions - for example, all descriptors are zero-initialized. To detect used and available descriptors it is possible for drivers and devices to keep track of the last observed value of VIRTQ_DESC_F_USED/VIRTQ_-DESC_F_AVAIL. Other techniques to detect VIRTQ_DESC_F_AVAIL/VIRTQ_DESC_F_USED bit changes might also be possible.

### 2.7.2 Polling of available and used descriptors

Writes of device and driver descriptors can generally be reordered, but each side (driver and device) are only required to poll (or test) a single location in memory: the next device descriptor after the one they processed previously, in circular order.

Sometimes the device needs to only write out a single used descriptor after processing a batch of multiple available descriptors. As described in more detail below, this can happen when using descriptor chaining or with in-order use of descriptors. In this case, the device writes out a used descriptor with the buffer id of the last descriptor in the group. After processing the used descriptor, both device and driver then skip forward in the ring the number of the remaining descriptors in the group until processing (reading for the driver and writing for the device) the next used descriptor.

### 2.7.3 Write Flag

In an available descriptor, the VIRTQ_DESC_F_WRITE bit within Flags is used to mark a descriptor as corresponding to a write-only or read-only element of a buffer.

```
/* This marks a descriptor as device write-only (otherwise device read-only). */
#define VIRTQ_DESC_F_WRITE     2
```

In a used descriptor, this bit is used to specify whether any data has been written by the device into any parts of the buffer.

### 2.7.4 Element Address and Length

In an available descriptor, Element Address corresponds to the physical address of the buffer element. The length of the element assumed to be physically contiguous is stored in Element Length.

In a used descriptor, Element Address is unused. Element Length specifies the length of the buffer that has been initialized (written to) by the device.

Element Length is reserved for used descriptors without the VIRTQ_DESC_F_WRITE flag, and is ignored by drivers.

### 2.7.5 Scatter-Gather Support

Some drivers need an ability to supply a list of multiple buffer elements (also known as a scatter/gather list) with a request. Two features support this: descriptor chaining and indirect descriptors.

If neither feature is in use by the driver, each buffer is physically-contiguous, either read-only or write-only and is described completely by a single descriptor.

While unusual (most implementations either create all lists solely using non-indirect descriptors, or always use a single indirect element), if both features have been negotiated, mixing indirect and non-indirect descriptors in a ring is valid, as long as each list only contains descriptors of a given type.

Scatter/gather lists only apply to available descriptors. A single used descriptor corresponds to the whole list.

The device limits the number of descriptors in a list through a transport-specific and/or device-specific value. If not limited, the maximum number of descriptors in a list is the virt queue size.

### 2.7.6 Next Flag: Descriptor Chaining

The packed ring format allows the driver to supply a scatter/gather list to the device by using multiple descriptors, and setting the VIRTQ_DESC_F_NEXT bit in Flags for all but the last available descriptor.

```
/* This marks a buffer as continuing. */
#define VIRTQ_DESC_F_NEXT   1
```

Buffer ID is included in the last descriptor in the list.

The driver always makes the first descriptor in the list available after the rest of the list has been written out into the ring. This guarantees that the device will never observe a partial scatter/gather list in the ring.

Note: all flags, including VIRTQ_DESC_F_AVAIL, VIRTQ_DESC_F_USED, VIRTQ_DESC_F_WRITE must be set/cleared correctly in all descriptors in the list, not just the first one.

The device only writes out a single used descriptor for the whole list. It then skips forward according to the number of descriptors in the list. The driver needs to keep track of the size of the list corresponding to each buffer ID, to be able to skip to where the next used descriptor is written by the device.

For example, if descriptors are used in the same order in which they are made available, this will result in the used descriptor overwriting the first available descriptor in the list, the used descriptor for the next list overwriting the first available descriptor in the next list, etc.

VIRTQ_DESC_F_NEXT is reserved in used descriptors, and should be ignored by drivers.

### 2.7.7 Indirect Flag: Scatter-Gather Support

Some devices benefit by concurrently dispatching a large number of large requests. The VIRTIO_F_INDIRECT_DESC feature allows this. To increase ring capacity the driver can store a (read-only by the device) table of indirect descriptors anywhere in memory, and insert a descriptor in the main virtqueue (with *Flags* bit VIRTQ_DESC_F_INDIRECT on) that refers to a buffer element containing this indirect descriptor table; *addr* and *len* refer to the indirect table address and length in bytes, respectively.

```
/* This means the element contains a table of descriptors. */
#define VIRTQ_DESC_F_INDIRECT   4
```

The indirect table layout structure looks like this (*len* is the Buffer Length of the descriptor that refers to this table, which is a variable):

```
struct pvirtq_indirect_descriptor_table {
        /* The actual descriptor structures (struct pvirtq_desc each) */
        struct pvirtq_desc desc[len / sizeof(struct pvirtq_desc)];
};
```

The first descriptor is located at the start of the indirect descriptor table, additional indirect descriptors come immediately afterwards. The VIRTQ_DESC_F_WRITE *flags* bit is the only valid flag for descriptors in the indirect table. Others are reserved and are ignored by the device. Buffer ID is also reserved and is ignored by the device.

In descriptors with VIRTQ_DESC_F_INDIRECT set VIRTQ_DESC_F_WRITE is reserved and is ignored by the device.

### 2.7.8 In-order use of descriptors

Some devices always use descriptors in the same order in which they have been made available. These devices can offer the VIRTIO_F_IN_ORDER feature. If negotiated, this knowledge allows devices to notify the use of a batch of buffers to the driver by only writing out a single used descriptor with the Buffer ID corresponding to the last descriptor in the batch.

The device then skips forward in the ring according to the size of the batch. The driver needs to look up the used Buffer ID and calculate the batch size to be able to advance to where the next used descriptor will be written by the device.

This will result in the used descriptor overwriting the first available descriptor in the batch, the used descriptor for the next batch overwriting the first available descriptor in the next batch, etc.

The skipped buffers (for which no used descriptor was written) are assumed to have been used (read or written) by the device completely.

### 2.7.9 Multi-buffer requests

Some devices combine multiple buffers as part of processing of a single request. These devices always mark the descriptor corresponding to the first buffer in the request used after the rest of the descriptors (corresponding to rest of the buffers) in the request - which follow the first descriptor in ring order - has been marked used and written out into the ring. This guarantees that the driver will never observe a partial request in the ring.

### 2.7.10 Driver and Device Event Suppression

In many systems used and available buffer notifications involve significant overhead. To mitigate this overhead, each virtqueue includes two identical structures used for controlling notifications between the device and the driver.

The Driver Event Suppression structure is read-only by the device and controls the used buffer notifications sent by the device to the driver.

The Device Event Suppression structure is read-only by the driver and controls the available buffer notifications sent by the driver to the device.

Each of these Event Suppression structures includes the following fields:

**Descriptor Ring Change Event Flags** Takes values:

```
/* Enable events */
#define RING_EVENT_FLAGS_ENABLE 0x0
/* Disable events */
#define RING_EVENT_FLAGS_DISABLE 0x1
/*
 * Enable events for a specific descriptor
 * (as specified by Descriptor Ring Change Event Offset/Wrap Counter).
 * Only valid if VIRTIO_F_RING_EVENT_IDX has been negotiated.
 */
#define RING_EVENT_FLAGS_DESC 0x2
/* The value 0x3 is reserved */
```

**Descriptor Ring Change Event Offset** If Event Flags set to descriptor specific event: offset within the ring (in units of descriptor size). Event will only trigger when this descriptor is made available/used respectively.

**Descriptor Ring Change Event Wrap Counter** If Event Flags set to descriptor specific event: offset within the ring (in units of descriptor size). Event will only trigger when Ring Wrap Counter matches this value and a descriptor is made available/used respectively.

After writing out some descriptors, both the device and the driver are expected to consult the relevant structure to find out whether a used respectively an available buffer notification should be sent.

#### 2.7.10.1 Structure Size and Alignment

Each part of the virtqueue is physically-contiguous in guest memory, and has different alignment requirements.

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

| Virtqueue Part | Alignment | Size |
|---|---|---|
| Descriptor Ring | 16 | $16*$(Queue Size) |
| Device Event Suppression | 4 | 4 |
| Driver Event Suppression | 4 | 4 |

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 descriptors in the virtqueue[3]. The Queue Size value does not have to be a power of 2.

### 2.7.11 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.

### 2.7.12 Device Requirements: Virtqueues

The device MUST start processing driver descriptors in the order in which they appear in the ring. The device MUST start writing device descriptors into the ring in the order in which they complete. The device MAY reorder descriptor writes once they are started.

### 2.7.13 The Virtqueue Descriptor Format

The available descriptor refers to the buffers the driver is sending to the device. *addr* is a physical address, and the descriptor is identified with a buffer using the *id* field.

```
struct pvirtq_desc {
        /* Buffer Address. */
        le64 addr;
        /* Buffer Length. */
        le32 len;
        /* Buffer ID. */
        le16 id;
        /* The flags depending on descriptor type. */
        le16 flags;
};
```

The descriptor ring is zero-initialized.

### 2.7.14 Event Suppression Structure Format

The following structure is used to reduce the number of notifications sent between driver and device.

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

```
struct pvirtq_event_suppress {
        le16 {
             desc_event_off : 15; /* Descriptor Ring Change Event Offset */
             desc_event_wrap : 1; /* Descriptor Ring Change Event Wrap Counter */
        } desc; /* If desc_event_flags set to RING_EVENT_FLAGS_DESC */
        le16 {
             desc_event_flags : 2, /* Descriptor Ring Change Event Flags */
             reserved : 14; /* Reserved, set to 0 */
        } flags;
};
```

### 2.7.15 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. A device MUST NOT use a descriptor unless it observes the VIRTQ_DESC_F_AVAIL bit in its *flags* being changed (e.g. as compared to the initial zero value). A device MUST NOT change a descriptor after changing it's the VIRTQ_DESC_F_USED bit in its *flags*.

### 2.7.16 Driver Requirements: The Virtqueue Descriptor Table

A driver MUST NOT change a descriptor unless it observes the VIRTQ_DESC_F_USED bit in its *flags* being changed. A driver MUST NOT change a descriptor after changing the VIRTQ_DESC_F_AVAIL bit in its *flags*. When notifying the device, driver MUST set *next_off* and *next_wrap* to match the next descriptor not yet made available to the device. A driver MAY send multiple available buffer notifications without making any new descriptors available to the device.

### 2.7.17 Driver Requirements: Scatter-Gather Support

A driver MUST NOT create a descriptor list longer than allowed by the device.

A driver MUST NOT create a descriptor list longer than the Queue Size.

This implies that loops in the descriptor list are forbidden!

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

A driver MUST NOT depend on the device to use more descriptors to be able to write out all descriptors in a list. A driver MUST make sure there's enough space in the ring for the whole list before making the first descriptor in the list available to the device.

A driver MUST NOT make the first descriptor in the list available before all subsequent descriptors comprising the list are made available.

### 2.7.18 Device Requirements: Scatter-Gather Support

The device MUST use descriptors in a list chained by the VIRTQ_DESC_F_NEXT flag in the same order that they were made available by the driver.

The device MAY limit the number of buffers it will allow in a list.

### 2.7.19 Driver Requirements: Indirect Descriptors

The driver MUST NOT set the DESC_F_INDIRECT flag unless the VIRTIO_F_INDIRECT_DESC feature was negotiated. The driver MUST NOT set any flags except DESC_F_WRITE within an indirect descriptor.

A driver MUST NOT create a descriptor chain longer than allowed by the device.

A driver MUST NOT write direct descriptors with DESC_F_INDIRECT set in a scatter-gather list linked by VIRTQ_DESC_F_NEXT. *flags*.

### 2.7.20 Virtqueue Operation

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

What follows is the requirements of each of these two parts when using the packed virtqueue format in more detail.

### 2.7.21 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 Ring.
2. The driver performs a suitable memory barrier to ensure that it updates the descriptor(s) before checking for notification suppression.
3. If notifications are not suppressed, the driver notifies the device of the new available buffers.

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

#### 2.7.21.1 Placing Available Buffers Into The Descriptor Ring

For each buffer element, b:

1. Get the next descriptor table entry, d
2. Get the next free buffer id value
3. Set *d.addr* to the physical address of the start of b
4. Set *d.len* to the length of b.
5. Set *d.id* to the buffer id
6. Calculate the flags as follows:
   (a) If b is device-writable, set the VIRTQ_DESC_F_WRITE bit to 1, otherwise 0
   (b) Set the VIRTQ_DESC_F_AVAIL bit to the current value of the Driver Ring Wrap Counter
   (c) Set the VIRTQ_DESC_F_USED bit to inverse value
7. Perform a memory barrier to ensure that the descriptor has been initialized
8. Set *d.flags* to the calculated flags value
9. If d is the last descriptor in the ring, toggle the Driver Ring Wrap Counter
10. Otherwise, increment d to point at the next descriptor

This makes a single descriptor buffer available. However, in general the driver MAY make use of a batch of descriptors as part of a single request. In that case, it defers updating the descriptor flags for the first descriptor (and the previous memory barrier) until after the rest of the descriptors have been initialized.

Once the descriptor *flags* field is updated by the driver, this exposes the descriptor and its contents. The device MAY access the descriptor and any following descriptors the driver created and the memory they refer to immediately.

#### 2.7.21.1.1 Driver Requirements: Updating flags

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

### 2.7.21.2 Sending Available Buffer Notifications

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, using the Event Suppression structure comprising the Device Area as detailed in section 2.7.14.

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

### 2.7.21.3 Implementation Example

Below is a driver code example. It does not attempt to reduce the number of available buffer notifications, neither does it support the VIRTIO_F_RING_EVENT_IDX feature.

```
/* Note: vq->avail_wrap_count is initialized to 1 */
/* Note: vq->sgs is an array same size as the ring */

id = alloc_id(vq);

first = vq->next_avail;
sgs = 0;
for (each buffer element b) {
        sgs++;

        vq->ids[vq->next_avail] = -1;
        vq->desc[vq->next_avail].address = get_addr(b);
        vq->desc[vq->next_avail].len = get_len(b);

        avail = vq->avail_wrap_count ? VIRTQ_DESC_F_AVAIL : 0;
        used = !vq->avail_wrap_count ? VIRTQ_DESC_F_USED : 0;
        f = get_flags(b) | avail | used;
        if (b is not the last buffer element) {
                f |= VIRTQ_DESC_F_NEXT;
        }

        /* Don't mark the 1st descriptor available until all of them are ready. */
        if (vq->next_avail == first) {
                flags = f;
        } else {
                vq->desc[vq->next_avail].flags = f;
        }

        last = vq->next_avail;

        vq->next_avail++;

        if (vq->next_avail >= vq->size) {
                vq->next_avail = 0;
                vq->avail_wrap_count \^= 1;
        }
}
vq->sgs[id] = sgs;
/* ID included in the last descriptor in the list */
vq->desc[last].id = id;
write_memory_barrier();
vq->desc[first].flags = flags;

memory_barrier();

if (vq->device_event.flags != RING_EVENT_FLAGS_DISABLE) {
        notify_device(vq);
}
```

#### 2.7.21.3.1 Driver Requirements: Sending Available Buffer Notifications

The driver MUST perform a suitable memory barrier before reading the Event Suppression structure occupying the Device Area. Failing to do so could result in mandatory available buffer notifications not being sent.

### 2.7.22 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 sends a used buffer notification to the driver as detailed in section 2.7.14.

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

```
/* Note: vq->used_wrap_count is initialized to 1 */

vq->driver_event.flags = RING_EVENT_FLAGS_DISABLE;

for (;;) {
        struct pvirtq_desc *d = vq->desc[vq->next_used];

        /*
         * Check that
         * 1. Descriptor has been made available. This check is necessary
         *    if the driver is making new descriptors available in parallel
         *    with this processing of used descriptors (e.g. from another thread).
         *    Note: there are many other ways to check this, e.g.
         *    track the number of outstanding available descriptors or buffers
         *    and check that it's not 0.
         * 2. Descriptor has been used by the device.
         */
        flags = d->flags;
        bool avail = flags & VIRTQ_DESC_F_AVAIL;
        bool used = flags & VIRTQ_DESC_F_USED;
        if (avail != vq->used_wrap_count || used != vq->used_wrap_count) {
                vq->driver_event.flags = RING_EVENT_FLAGS_ENABLE;
                memory_barrier();

                /*
                 * Re-test in case the driver made more descriptors available in
                 * parallel with the used descriptor processing (e.g. from another
                 * thread) and/or the device used more descriptors before the driver
                 * enabled events.
                 */
                flags = d->flags;
                bool avail = flags & VIRTQ_DESC_F_AVAIL;
                bool used = flags & VIRTQ_DESC_F_USED;
                if (avail != vq->used_wrap_count || used != vq->used_wrap_count) {
                        break;
                }

                vq->driver_event.flags = RING_EVENT_FLAGS_DISABLE;
        }

        read_memory_barrier();

        /* skip descriptors until the next buffer */
        id = d->id;
        assert(id < vq->size);
        sgs = vq->sgs[id];
        vq->next_used += sgs;
        if (vq->next_used >= vq->size) {
                vq->next_used -= vq->size;
                vq->used_wrap_count \^= 1;
```

```
        }

        free_id(vq, id);

        process_buffer(d);
}
```

### 2.7.23 Driver notifications

The driver is sometimes required to send an available buffer notification to the device.

When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, this notification involves sending the virtqueue number to the device (method depending on the transport).

However, some devices benefit from the ability to find out the amount of available data in the queue without accessing the virtqueue in memory: for efficiency or as a debugging aid.

To help with these optimizations, when VIRTIO_F_NOTIFICATION_DATA has been negotiated, driver notifications to the device include the following information:

**vqn** VQ number to be notified.

**next_off** Offset within the ring where the next available ring entry will be written. When VIRTIO_F_RING_PACKED has not been negotiated this refers to the 15 least significant bits of the available index. When VIRTIO_F_RING_PACKED has been negotiated this refers to the offset (in units of descriptor entries) within the descriptor ring where the next available descriptor will be written.

**next_wrap** Wrap Counter. With VIRTIO_F_RING_PACKED this is the wrap counter referring to the next available descriptor. Without VIRTIO_F_RING_PACKED this is the most significant bit (bit 15) of the available index.

Note that the driver can send multiple notifications even without making any more buffers available. When VIRTIO_F_NOTIFICATION_DATA has been negotiated, these notifications would then have identical *next_off* and *next_wrap* values.

# 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 noticed 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 send any buffer available notifications to 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

When operating the device, each field in the device configuration space can be changed by either the driver or the device.

Whenever such a configuration change is triggered by the device, driver is notified. This makes it possible for drivers to cache device configuration, avoiding expensive configuration reads unless notified.

### 3.2.1 Notification of Device Configuration Changes

For devices where the device-specific configuration information can be changed, a configuration change notification is sent when a device-specific configuration change occurs.

In addition, this notification 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 virtqueue entries for exposed buffers - i.e. buffers which have been made available to the device (and not been used by the device) of a live virtqueue.

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 (traditional) |
| 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 64-bit, 32-bit and 16-bit fields are little-endian. 64-bit fields are to be treated as two 32-bit fields, with low 32 bit part followed by the high 32 bit part.

#### 4.1.3.1 Driver Requirements: PCI Device Layout

For device configuration access, 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-bit and 64-bit wide fields. For 64-bit fields, the driver MAY access each of the high and low 32-bit parts of the field independently.

#### 4.1.3.2 Device Requirements: PCI Device Layout

For 64-bit device configuration fields, the device MUST allow driver independent access to high and low 32-bit parts of the field.

### 4.1.4 Virtio Structure PCI Capabilities

The virtio device configuration layout includes several structures:

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

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 */
        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_driver;              /* read-write */
        le64 queue_device;              /* 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 device. This can be modified by the 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 Area here. See section 2.5.

***queue_driver*** The driver writes the physical address of Driver Area here. See section 2.5.

***queue_device*** The driver writes the physical address of Device Area here. See section 2.5.

#### 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.

If VIRTIO_F_RING_PACKED has not been negotiated, the device MUST present either a value of 0 or a power of 2 in *queue_size*.

#### 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*.

If VIRTIO_F_RING_PACKED has been negotiated, the driver MUST NOT write the value 0 to *queue_size*. If VIRTIO_F_RING_PACKED has not been negotiated, 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.

For devices not offering VIRTIO_F_NOTIFICATION_DATA:

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
```

For devices offering VIRTIO_F_NOTIFICATION_DATA:

The device MUST either present *notify_off_multiplier* as a number that is a power of 2 that is also a multiple 4, or present *notify_off_multiplier* as 0.

The *cap.offset* MUST be 4-byte aligned.

The value *cap.length* presented by the device MUST be at least 4 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 + 4
```

#### 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 | Queue Interrupt | Device Configuration 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).

The driver MUST NOT read or write *pci_cfg_data* unless *cap.bar*, *cap.length* and *cap.offset* address *cap.length* bytes within a BAR range specified by some other Virtio Structure PCI Capability of type other than *VIRTIO_PCI_CAP_PCI_CFG*.

### 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 2.4.4 Legacy Interface: Device Configuration Space for workarounds.

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

All known legacy drivers check either the PCI Revision or the Device and Vendor IDs, and thus won't attempt to drive a non-transitional device.

A buggy legacy driver might mistakenly attempt to drive a non-transitional device. If support for such drivers is required (as opposed to fixing the bug), the following would be the recommended way to detect and handle them.

**Note:** Such buggy drivers are not currently known to be used in production.

##### 4.1.4.9.0.1 Device 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 (including PCI Revision, Device and Vendor IDs) is known to have previously existed, SHOULD take the following

steps to cause the legacy driver to fail gracefully when it 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

###### 4.1.5.1.1.1 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.

##### 4.1.5.1.2 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.

#### 4.1.5.1.2.1 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.

#### 4.1.5.1.2.2 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.

### 4.1.5.1.3 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 2.5 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.

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

#### 4.1.5.1.3.1 Legacy Interface: A Note on Virtqueue Configuration

When using the legacy interface, the queue layout follows 2.6.2 Legacy Interfaces: A Note on Virtqueue Layout with an alignment of 4096. 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 Available Buffer Notifications

When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, the driver sends an available buffer notification to the device by writing the 16-bit virtqueue index of this virtqueue to the Queue Notify address.

When VIRTIO_F_NOTIFICATION_DATA has been negotiated, the driver sends an available buffer notification to the device by writing the following 32-bit value to the Queue Notify address:

```
le32 {
  vqn : 16;
  next_off : 15;
  next_wrap : 1;
};
```

See 2.7.23 Driver notifications for the definition of the components.

See 4.1.4.4 for how to calculate the Queue Notify address.

### 4.1.5.3 Used Buffer Notifications

If a used buffer notification 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.

#### 4.1.5.3.1 Device Requirements: Used Buffer Notifications

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.

[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.

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 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 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*<br>Offset from base<br>Direction | **Function**<br>Description |
|---|---|
| *MagicValue*<br>0x000<br>R | **Magic value**<br>0x74726976 (a Little Endian equivalent of the “virt” string). |
| *Version*<br>0x004<br>R | **Device version number**<br>0x2.<br>**Note:** Legacy devices (see 4.2.4 Legacy interface) used 0x1. |
| *DeviceID*<br>0x008<br>R | **Virtio Subsystem Device ID**<br>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*<br>0x00c<br>R | **Virtio Subsystem Vendor ID** |
| *DeviceFeatures*<br>0x010<br>R | **Flags representing features the device supports**<br>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 *DeviceFeaturesSel* $*$ 32 to (*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. |
| *DeviceFeaturesSel*<br>0x014<br>W | **Device (host) features word selection.**<br>Writing to this register selects a set of 32 device feature bits accessible by reading from *DeviceFeatures*. |
| *DriverFeatures*<br>0x020<br>W | **Flags representing device features understood and activated by the driver**<br>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 *DriverFeaturesSel* $*$ 32 to (*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*<br>0x024<br>W | **Activated (guest) features word selection**<br>Writing to this register selects a set of 32 activated feature bits accessible by writing to *DriverFeatures*. |
| *QueueSel*<br>0x030<br>W | **Virtual queue index**<br>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). |

| *Name*<br>Offset from the base<br>Direction | **Function**<br>Description |
|---|---|
| *QueueNumMax*<br>0x034<br>R | **Maximum virtual queue size**<br>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*<br>0x038<br>W | **Virtual queue size**<br>Queue size is the number of elements in the queue. 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*<br>0x044<br>RW | **Virtual queue ready bit**<br>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*<br>0x050<br>W | **Queue notifier**<br>Writing a value to this register notifies the device that there are new buffers to process in a queue.<br>When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, the value written is the queue index.<br>When VIRTIO_F_NOTIFICATION_DATA has been negotiated, the *Notification data* value has the following format:<br>`le32 { vqn : 16; next_off : 15; next_wrap : 1; };`<br>See 2.7.23 Driver notifications for the definition of the components. |
| *InterruptStatus*<br>0x60<br>R | **Interrupt status**<br>Reading from this register returns a bit mask of events that caused the device interrupt to be asserted. The following events are possible:<br>**Used Buffer Notification** - bit 0 - the interrupt was asserted because the device has used a buffer in at least one of the active virtual queues.<br>**Configuration Change Notification** - bit 1 - the interrupt was asserted because the configuration of the device has changed. |
| *InterruptACK*<br>0x064<br>W | **Interrupt acknowledge**<br>Writing a value with bits set as defined in *InterruptStatus* to this register notifies the device that events causing the interrupt have been handled. |
| *Status*<br>0x070<br>RW | **Device status**<br>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*<br>0x080<br>*QueueDescHigh*<br>0x084<br>W | **Virtual queue's Descriptor Area 64 bit long physical address**<br>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 Area of the queue selected by writing to *QueueSel* register. |

| *Name*<br>Offset from the base<br>Direction | **Function**<br>Description |
|---|---|
| *QueueDriverLow*<br>0x090<br>*QueueDriverHigh*<br>0x094<br>W | **Virtual queue's Driver Area 64 bit long physical address**<br>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 Driver Area of the queue selected by writing to *QueueSel*. |
| *QueueDeviceLow*<br>0x0a0<br>*QueueDeviceHigh*<br>0x0a4<br>W | **Virtual queue's Device Area 64 bit long physical address**<br>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 Device Area of the queue selected by writing to *QueueSel*. |
| *ConfigGeneration*<br>0x0fc<br>R | **Configuration atomicity value**<br>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 2.4. |
| *Config*<br>0x100+<br>RW | **Configuration space**<br>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 memory, making sure the memory is physically contiguous.
5. Notify the device about the queue size by writing the size to *QueueNum*.
6. Write physical addresses of the queue's Descriptor Area, Driver Area and Device Area to (respectively) the *QueueDescLow*/*QueueDescHigh*, *QueueDriverLow*/*QueueDriverHigh* and *QueueDeviceLow*/*QueueDeviceHigh* register pairs.
7. Write 0x1 to *QueueReady*.

#### 4.2.3.3 Available Buffer Notifications

When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, the driver sends an available buffer notification to the device by writing the 16-bit virtqueue index of the queue to be notified to *QueueNotify*.

When VIRTIO_F_NOTIFICATION_DATA has been negotiated, the driver sends an available buffer notification to the device by writing the following 32-bit value to *QueueNotify*:

```
le32 {
  vqn : 16;
  next_off : 15;
  next_wrap : 1;
};
```

See 2.7.23 Driver notifications for the definition of the components.

#### 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 sends a used buffer notification or a configuration change notification to the device.

##### 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). The used buffer notification bit being set SHOULD be interpreted as a used buffer notification for each active virtqueue. 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*<br>Offset from base<br>Direction | **Function**<br>Description |
|---|---|
| *MagicValue*<br>0x000<br>R | **Magic value** |
| *Version*<br>0x004<br>R | **Device version number**<br>Legacy device returns value 0x1. |
| *DeviceID*<br>0x008<br>R | **Virtio Subsystem Device ID** |
| *VendorID*<br>0x00c<br>R | **Virtio Subsystem Vendor ID** |
| *HostFeatures*<br>0x010<br>R | **Flags representing features the device supports** |
| *HostFeaturesSel*<br>0x014<br>W | **Device (host) features word selection.** |

| *Name*<br>Offset from the base<br>Direction | **Function**<br>Description |
|---|---|
| *GuestFeatures*<br>0x020<br>W | **Flags representing device features understood and activated by the driver** |
| *GuestFeaturesSel*<br>0x024<br>W | **Activated (guest) features word selection** |
| *GuestPageSize*<br>0x028<br>W | **Guest page size**<br>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*<br>0x030<br>W | **Virtual queue index**<br>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*<br>0x034<br>R | **Maximum virtual queue size**<br>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*<br>0x038<br>W | **Virtual queue size**<br>Queue size is the number of elements in the queue. 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*<br>0x03c<br>W | **Used Ring alignment in the virtual queue**<br>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*<br>0x040<br>RW | **Guest physical page number of the virtual queue**<br>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*<br>0x050<br>W | **Queue notifier** |
| *InterruptStatus*<br>0x60<br>R | **Interrupt status** |
| *InterruptACK*<br>0x064<br>W | **Interrupt acknowledge** |

| *Name*<br>Offset from the base<br>Direction | **Function**<br>Description |
|---|---|
| *Status*<br>0x070<br>RW | **Device status**<br>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*<br>0x100+<br>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. 2.6.2 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:

- NO-OPERATION (0x03)
- BASIC SENSE (0x04)
- TRANSFER IN CHANNEL (0x08)
- SENSE ID (0xe4)

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) |

A virtio-ccw proxy device facilitates:

- Discovery and attachment of virtio devices (as described above).
- Initialization of virtqueues and transport-specific facilities (using virtio-specific channel commands).
- Notifications (via hypercall and a combination of I/O interrupts and indicator bits).

#### 4.3.1.1 Channel Commands for Virtio

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
#define CCW_CMD_READ_STATUS 0x72
```

#### 4.3.1.2 Notifications

Available buffer notifications are realized as a hypercall. No additional setup by the driver is needed. The operation of available buffer notifications is described in section 4.3.3.2.

Used buffer notifications are realized either as so-called classic or adapter I/O interrupts depending on a transport level negotiation. The initialization is described in sections 4.3.2.6.1 and 4.3.2.6.3 respectively. The operation of each flavor is described in sections 4.3.3.1.1 and 4.3.3.1.2 respectively.

Configuration change notifications are done using so-called classic I/O interrupts. The initialization is described in section 4.3.2.6.2 and the operation in section 4.3.3.1.1.

#### 4.3.1.3 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 sub-channel 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.

#### 4.3.1.4 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 | 0 | <empty> | CCW_CMD_READ_STATUS support |
| 3-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 driver;
        be64 device;
};
```

*desc*, *driver* and *device* contain the guest addresses for the descriptor area, available area and used area 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. The queue layout follows 2.6.2 Legacy Interfaces: A Note on Virtqueue Layout.

### 4.3.2.3 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.

As described in 2.2.2, a device sometimes fails to set the *device status* field: For example, it might fail to accept the FEATURES_OK status bit during device initialization.

With revision 2, CCW_CMD_READ_STATUS is defined: It reads an 8 bit status value from the device and acts as a reverse operation to CCW_CMD_WRITE_STATUS.

#### 4.3.2.3.1 Driver Requirements: Communicating Status Information

If the device posts a unit check with command reject in response to the CCW_CMD_WRITE_STATUS command, the driver MUST assume that the device failed to set the status and the *device status* field retained its previous value.

If at least revision 2 has been negotiated, the driver SHOULD use the CCW_CMD_READ_STATUS command to retrieve the *device status* field after a configuration change has been detected.

If not at least revision 2 has been negotiated, the driver MUST NOT attempt to issue the CCW_CMD_-READ_STATUS command.

#### 4.3.2.3.2 Device Requirements: Communicating Status Information

If the device fails to set the *device status* field to the value written by the driver, the device MUST assure that the *device status* field is left unchanged and MUST post a unit check with command reject.

If at least revision 2 has been negotiated, the device MUST return the current *device status* field if the CCW_CMD_READ_STATUS command is issued.

### 4.3.2.4 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.

#### 4.3.2.5 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.

#### 4.3.2.6 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.

##### 4.3.2.6.1 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.

##### 4.3.2.6.2 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.

#### 4.3.2.6.3 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.

##### 4.3.2.6.3.1 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.

#### 4.3.2.6.4 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 | Notification data | |
| 4 | Host Cookie | |

When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, the *Notification data* contains the Virtqueue number.

When VIRTIO_F_NOTIFICATION_DATA has been negotiated, the value has the following format:

```
be32 {
  vqn : 16;
  next_off : 15;
  next_wrap : 1;
};
```

See 2.7.23 Driver notifications for the definition of the components.

#### 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 (traditional) |
| 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 |
| 19 | Socket device |
| 20 | Crypto device |
| 21 | Signal Distribution Module |
| 22 | pstore device |
| 23 | IOMMU device |
| 24 | Memory 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

**...**

**2(N-1)** receiveqN

**2(N-1)+1** transmitqN

**2N** 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_MTU(3)** Device maximum MTU reporting is supported. If offered by the device, device advises driver about the value of its maximum MTU. If negotiated, the driver uses *mtu* as the maximum MTU value.

**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.

**VIRTIO_NET_F_RSC_EXT(61)** Device can process duplicated ACKs and report number of coalesced segments and duplicated ACKs

**VIRTIO_NET_F_STANDBY(62)** Device may act as a standby for a primary device with the same MAC address.

#### 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.

**VIRTIO_NET_F_RSC_EXT** Requires VIRTIO_NET_F_HOST_TSO4 or VIRTIO_NET_F_HOST_TSO6.

#### 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.

**VIRTIO_NET_F_GUEST_RSC4 (41)** Device coalesces TCPIP v4 packets. This was implemented by hypervisor patch for certification purposes and current Windows driver depends on it. It will not function if virtio-net device reports this feature.

**VIRTIO_NET_F_GUEST_RSC6 (42)** Device coalesces TCPIP v6 packets. Similar to VIRTIO_NET_F_-GUEST_RSC4.

### 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.

The following driver-read-only field, *mtu* only exists if VIRTIO_NET_F_MTU is set. This field specifies the maximum MTU for the driver to use.

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

#### 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.

The device MUST set *mtu* to between 68 and 65535 inclusive, if it offers VIRTIO_NET_F_MTU.

The device SHOULD set *mtu* to at least 1280, if it offers VIRTIO_NET_F_MTU.

The device MUST NOT modify *mtu* once it has been set.

The device MUST NOT pass received packets that exceed *mtu* (plus low level ethernet header length) size with *gso_type* NONE or ECN after VIRTIO_NET_F_MTU has been successfully negotiated.

The device MUST forward transmitted packets of up to *mtu* (plus low level ethernet header length) size with *gso_type* NONE or ECN, and do so without fragmentation, after VIRTIO_NET_F_MTU has been successfully negotiated.

If the driver negotiates the VIRTIO_NET_F_STANDBY feature, the device MAY act as a standby device for a primary device with the same MAC address.

#### 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*.

A driver SHOULD negotiate VIRTIO_NET_F_MTU if the device offers it.

If the driver negotiates VIRTIO_NET_F_MTU, it MUST supply enough receive buffers to receive at least one receive packet of size *mtu* (plus low level ethernet header length) with *gso_type* NONE or ECN.

If the driver negotiates VIRTIO_NET_F_MTU, it MUST NOT transmit packets of size exceeding the value of *mtu* (plus low level ethernet header length) with *gso_type* NONE or ECN.

A driver SHOULD negotiate the VIRTIO_NET_F_STANDBY feature if the device offers it.

#### 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 Incoming 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
#define VIRTIO_NET_HDR_F_DATA_VALID    2
#define VIRTIO_NET_HDR_F_RSC_INFO      4
        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 negotiated; without that feature the structure was 2 bytes shorter.

When using the legacy interface, the driver SHOULD ignore the used length for the transmit queues and the controlq queue.

**Note:** Historically, some devices put the total descriptor length there, even though no data was actually written.

#### 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 can 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 can 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).

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

- 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

The driver MUST set *num_buffers* to zero.

If VIRTIO_NET_F_CSUM is not negotiated, the driver MUST set *flags* to zero and SHOULD supply a fully checksummed packet to the device.

If VIRTIO_NET_F_HOST_TSO4 is negotiated, the driver MAY set *gso_type* to VIRTIO_NET_HDR_GSO_-TCPV4 to request TCPv4 segmentation, otherwise the driver MUST NOT set *gso_type* to VIRTIO_NET_-HDR_GSO_TCPV4.

If VIRTIO_NET_F_HOST_TSO6 is negotiated, the driver MAY set *gso_type* to VIRTIO_NET_HDR_GSO_-TCPV6 to request TCPv6 segmentation, otherwise the driver MUST NOT set *gso_type* to VIRTIO_NET_-HDR_GSO_TCPV6.

If VIRTIO_NET_F_HOST_UFO is negotiated, the driver MAY set *gso_type* to VIRTIO_NET_HDR_GSO_-UDP to request UDP segmentation, otherwise the driver MUST NOT set *gso_type* to VIRTIO_NET_HDR_-GSO_UDP.

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

If the VIRTIO_NET_F_CSUM feature has been negotiated, the driver MAY set the VIRTIO_NET_HDR_F_-NEEDS_CSUM bit in *flags*, if so:

1. the driver MUST validate the packet checksum at offset *csum_offset* from *csum_start* as well as all preceding offsets;
2. the driver MUST set the packet checksum stored in the buffer to the TCP/UDP pseudo header;
3. the driver MUST set *csum_start* and *csum_offset* such that calculating a ones' complement checksum from *csum_start* up until the end of the packet and storing the result at offset *csum_offset* from *csum_-start* will result in a fully checksummed packet;

If none of the VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO options have been negotiated, the driver MUST set *gso_type* to VIRTIO_NET_HDR_GSO_NONE.

If *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE, then the driver MUST also set the VIRTIO_NET_-HDR_F_NEEDS_CSUM bit in *flags* and MUST set *gso_size* to indicate the desired MSS.

If one of the VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO options have been negotiated, the driver SHOULD set *hdr_len* to a value not less than the length of the headers, including the transport header.

The driver MUST NOT set the VIRTIO_NET_HDR_F_DATA_VALID and VIRTIO_NET_HDR_F_RSC_INFO bits in *flags*.

#### 5.1.6.2.2 Device Requirements: Packet Transmission

The device MUST ignore *flag* bits that it does not recognize.

If VIRTIO_NET_HDR_F_NEEDS_CSUM bit in *flags* is not set, the device MUST NOT use the *csum_start* and *csum_offset*.

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

If one of the VIRTIO_NET_F_HOST_TSO4, TSO6 or UFO options have been negotiated, the device MAY use *hdr_len* only as a hint about the transport header size. The device MUST NOT rely on *hdr_len* to be correct.

**Note:** This is due to various bugs in implementations.

If VIRTIO_NET_HDR_F_NEEDS_CSUM is not set, the device MUST NOT rely on the packet checksum being correct.

#### 5.1.6.2.3 Packet Transmission Interrupt

Often a driver will suppress transmission virtqueue interrupts and check for used packets in the transmit path of following packets.

The normal behavior in this interrupt handler is to retrieve used buffers from the virtqueue 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 least the size of the struct virtio_net_hdr.

**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 Incoming Packets

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

Processing incoming packets 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: a packet that does not fit in a single buffer can flow over to the next buffer, and so on. In this case, there will be at least *num_buffers* used buffers in the virtqueue, and the device chains them together to form a single packet in a way similar to how it would store it in a single buffer spread over multiple descriptors. 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_DATA_VALID bit in *flags* can be set: if so, device has validated the packet checksum. In case of multiple encapsulated protocols, one level of checksums has been validated.

Additionally, VIRTIO_NET_F_GUEST_CSUM, TSO4, TSO6, UDP and ECN features enable receive checksum, large receive offload and ECN support which are the input equivalents of the transmit checksum, transmit segmentation offloading and ECN features, as described in 5.1.6.2:

1. 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).
2. If the VIRTIO_NET_F_RSC_EXT option was negotiated (this implies one of VIRTIO_NET_F_GUEST_-TSO4, TSO6), the device processes also duplicated ACK segments, reports number of coalesced TCP segments in *csum_start* field and number of duplicated ACK segments in *csum_offset* field and sets bit VIRTIO_NET_HDR_F_RSC_INFO in *flags*.
3. If the VIRTIO_NET_F_GUEST_CSUM feature was negotiated, the VIRTIO_NET_HDR_F_NEEDS_-CSUM bit in *flags* can be set: if so, the packet checksum at offset *csum_offset* from *csum_start* and any preceding checksums have been validated. The checksum on the packet is incomplete and if bit VIRTIO_NET_HDR_F_RSC_INFO is not set in *flags*, then *csum_start* and *csum_offset* indicate how to calculate it (see Packet Transmission point 1).

#### 5.1.6.4.1 Device Requirements: Processing of Incoming Packets

If VIRTIO_NET_F_MRG_RXBUF has not been negotiated, the device MUST set *num_buffers* to 1.

If VIRTIO_NET_F_MRG_RXBUF has been negotiated, the device MUST set *num_buffers* to indicate the number of buffers the packet (including the header) is spread over.

If a receive packet is spread over multiple buffers, the device MUST use all buffers but the last (i.e. the first $num_buffers - 1$ buffers) completely up to the full length of each buffer supplied by the driver.

The device MUST use all buffers used by a single receive packet together, such that at least *num_buffers* are observed by driver as used.

If VIRTIO_NET_F_GUEST_CSUM is not negotiated, the device MUST set *flags* to zero and SHOULD supply a fully checksummed packet to the driver.

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.

The device SHOULD NOT send to the driver 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 the device MUST set the VIRTIO_NET_HDR_GSO_ECN bit in *gso_type*.

If the VIRTIO_NET_F_GUEST_CSUM feature has been negotiated, the device MAY set the VIRTIO_NET_-HDR_F_NEEDS_CSUM bit in *flags*, if so:

1. the device MUST validate the packet checksum at offset *csum_offset* from *csum_start* as well as all preceding offsets;
2. the device MUST set the packet checksum stored in the receive buffer to the TCP/UDP pseudo header;
3. the device MUST set *csum_start* and *csum_offset* such that calculating a ones' complement checksum from *csum_start* up until the end of the packet and storing the result at offset *csum_offset* from *csum_-start* will result in a fully checksummed packet;

If none of the VIRTIO_NET_F_GUEST_TSO4, TSO6 or UFO options have been negotiated, the device MUST set *gso_type* to VIRTIO_NET_HDR_GSO_NONE.

If *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE, then the device MUST also set the VIRTIO_-NET_HDR_F_NEEDS_CSUM bit in *flags* MUST set *gso_size* to indicate the desired MSS. If VIRTIO_NET_-F_RSC_EXT was negotiated, the device MUST also set VIRTIO_NET_HDR_F_RSC_INFO bit in *flags*, set *csum_start* to number of coalesced TCP segments and set *csum_offset* to number of received duplicated ACK segments.

If VIRTIO_NET_F_RSC_EXT was not negotiated, the device MUST not set VIRTIO_NET_HDR_F_RSC_-INFO bit in *flags*.

If one of the VIRTIO_NET_F_GUEST_TSO4, TSO6 or UFO options have been negotiated, the device SHOULD set *hdr_len* to a value not less than the length of the headers, including the transport header.

If the VIRTIO_NET_F_GUEST_CSUM feature has been negotiated, the device MAY set the VIRTIO_NET_-HDR_F_DATA_VALID bit in *flags*, if so, the device MUST validate the packet checksum (in case of multiple encapsulated protocols, one level of checksums is validated).

#### 5.1.6.4.2 Driver Requirements: Processing of Incoming Packets

The driver MUST ignore *flag* bits that it does not recognize.

If VIRTIO_NET_HDR_F_NEEDS_CSUM bit in *flags* is not set or if VIRTIO_NET_HDR_F_RSC_INFO bit *flags* is set, the driver MUST NOT use the *csum_start* and *csum_offset*.

If one of the VIRTIO_NET_F_GUEST_TSO4, TSO6 or UFO options have been negotiated, the driver MAY use *hdr_len* only as a hint about the transport header size. The driver MUST NOT rely on *hdr_len* to be correct.

**Note:** This is due to various bugs in implementations.

If neither VIRTIO_NET_HDR_F_NEEDS_CSUM nor VIRTIO_NET_HDR_F_DATA_VALID is set, the driver MUST NOT rely on the packet checksum being correct.

### 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 and VIRTIO_NET_F_CTRL_RX_EXTRA features are negotiated, the driver can send control commands for promiscuous mode, multicast, unicast and broadcast receiving.

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

```
#define VIRTIO_NET_CTRL_RX    0
 #define VIRTIO_NET_CTRL_RX_PROMISC      0
 #define VIRTIO_NET_CTRL_RX_ALLMULTI     1
 #define VIRTIO_NET_CTRL_RX_ALLUNI       2
 #define VIRTIO_NET_CTRL_RX_NOMULTI      3
 #define VIRTIO_NET_CTRL_RX_NOUNI        4
 #define VIRTIO_NET_CTRL_RX_NOBCAST      5
```

##### 5.1.6.5.1.1 Device Requirements: Packet Receive Filtering

If the VIRTIO_NET_F_CTRL_RX feature has been negotiated, the device MUST support the following VIRTIO_NET_CTRL_RX class commands:

- VIRTIO_NET_CTRL_RX_PROMISC turns promiscuous mode on and off. The command-specific-data is one byte containing 0 (off) or 1 (on). If promiscous mode is on, the device SHOULD receive all incoming packets. This SHOULD take effect even if one of the other modes set by a VIRTIO_NET_CTRL_RX class command is on.
- 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). When all-multicast receive is on the device SHOULD allow all incoming multicast packets.

If the VIRTIO_NET_F_CTRL_RX_EXTRA feature has been negotiated, the device MUST support the following VIRTIO_NET_CTRL_RX class commands:

- VIRTIO_NET_CTRL_RX_ALLUNI turns all-unicast receive on and off. The command-specific-data is one byte containing 0 (off) or 1 (on). When all-unicast receive is on the device SHOULD allow all incoming unicast packets.
- VIRTIO_NET_CTRL_RX_NOMULTI suppresses multicast receive. The command-specific-data is one byte containing 0 (multicast receive allowed) or 1 (multicast receive suppressed). When multicast receive is suppressed, the device SHOULD NOT send multicast packets to the driver. This SHOULD take effect even if VIRTIO_NET_CTRL_RX_ALLMULTI is on. This filter SHOULD NOT apply to broadcast packets.
- VIRTIO_NET_CTRL_RX_NOUNI suppresses unicast receive. The command-specific-data is one byte containing 0 (unicast receive allowed) or 1 (unicast receive suppressed). When unicast receive is suppressed, the device SHOULD NOT send unicast packets to the driver. This SHOULD take effect even if VIRTIO_NET_CTRL_RX_ALLUNI is on.
- VIRTIO_NET_CTRL_RX_NOBCAST suppresses broadcast receive. The command-specific-data is one byte containing 0 (broadcast receive allowed) or 1 (broadcast receive suppressed). When broadcast receive is suppressed, the device SHOULD NOT send broadcast packets to the driver. This SHOULD take effect even if VIRTIO_NET_CTRL_RX_ALLMULTI is on.

##### 5.1.6.5.1.2 Driver Requirements: Packet Receive Filtering

If the VIRTIO_NET_F_CTRL_RX feature has not been negotiated, the driver MUST NOT issue commands VIRTIO_NET_CTRL_RX_PROMISC or VIRTIO_NET_CTRL_RX_ALLMULTI.

If the VIRTIO_NET_F_CTRL_RX_EXTRA feature has not been negotiated, the driver MUST NOT issue commands VIRTIO_NET_CTRL_RX_ALLUNI, VIRTIO_NET_CTRL_RX_NOMULTI, VIRTIO_NET_CTRL_-RX_NOUNI or VIRTIO_NET_CTRL_RX_NOBCAST.

#### 5.1.6.5.2 Setting MAC Address Filtering

If the VIRTIO_NET_F_CTRL_RX feature is negotiated, the driver can send control commands for 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[3]. 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).

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

##### 5.1.6.5.2.1 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_TA-BLE_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.

##### 5.1.6.5.2.2 Driver Requirements: Setting MAC Address Filtering

If VIRTIO_NET_F_CTRL_RX has not been negotiated, the driver MUST NOT issue VIRTIO_NET_CTRL_-MAC class commands.

If VIRTIO_NET_F_CTRL_RX has been negotiated, the driver SHOULD issue VIRTIO_NET_CTRL_MAC_-ADDR_SET to set the default mac if it is different from *mac*.

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.

[3] 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.

#### 5.1.6.5.2.3 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.

### 5.1.6.5.3 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.

#### 5.1.6.5.3.1 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.

### 5.1.6.5.4 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.

#### 5.1.6.5.4.1 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.

#### 5.1.6.5.4.2 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.

### 5.1.6.5.5 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 unidirectional 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.

#### 5.1.6.5.5.1 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.

#### 5.1.6.5.5.2 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 a used buffer.

#### 5.1.6.5.5.3 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.

### 5.1.6.5.6 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.

#### 5.1.6.5.6.1 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.

#### 5.1.6.5.6.2 Driver Requirements: Setting Offloads State

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

#### 5.1.6.5.6.3 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 2.6.4.

## 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_FLUSH (9)** Cache flush command support.

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

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

**VIRTIO_BLK_F_DISCARD (13)** Device can support discard command, maximum discard sectors size in *max_discard_sectors* and maximum discard segment number in *max_discard_seg*.

**VIRTIO_BLK_F_WRITE_ZEROES (14)** Device can support write zeroes command, maximum write zeroes sectors size in *max_write_zeroes_sectors* and maximum write zeroes segment number in *max_write_-zeroes_seg*.

#### 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.

**Note:** In the legacy interface, VIRTIO_BLK_F_FLUSH was also called VIRTIO_BLK_F_WCE.

### 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.

The parameters in the configuration space of the device *max_discard_sectors discard_sector_alignment* are expressed in 512-byte units if the VIRTIO_BLK_F_DISCARD feature bit is negotiated. The *max_write_zeroes_sectors* is expressed in 512-byte units if the VIRTIO_BLK_F_WRITE_ZEROES feature bit is negotiated.

```
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 writeback;
        u8 unused0[3];
        le32 max_discard_sectors;
        le32 max_discard_seg;
        le32 discard_sector_alignment;
        le32 max_write_zeroes_sectors;
        le32 max_write_zeroes_seg;
        u8 write_zeroes_may_unmap;
        u8 unused1[3];
};
```

#### 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. If the VIRTIO_BLK_F_CONFIG_WCE feature is negotiated, the cache mode can be read or set through the *writeback* field. 0 corresponds to a writethrough cache, 1 to a writeback cache[4]. The cache mode after reset can be either writeback or writethrough. The actual mode can be determined by reading *writeback* after feature negotiation.
6. If the VIRTIO_BLK_F_DISCARD feature is negotiated, *max_discard_sectors* and *max_discard_seg* can be read to determine the maximum discard sectors and maximum number of discard segments for the block driver to use. *discard_sector_alignment* can be used by OS when splitting a request based on alignment.
7. if the VIRTIO_BLK_F_WRITE_ZEROES feature is negotiated, *max_write_zeroes_sectors* and *max_write_zeroes_seg* can be read to determine the maximum write zeroes sectors and maximum number of write zeroes segments for the block driver to use.

#### 5.2.5.1 Driver Requirements: Device Initialization

Drivers SHOULD NOT negotiate VIRTIO_BLK_F_FLUSH if they are incapable of sending VIRTIO_BLK_T_FLUSH commands.

If neither VIRTIO_BLK_F_CONFIG_WCE nor VIRTIO_BLK_F_FLUSH are negotiated, the driver MAY deduce the presence of a writethrough cache. If VIRTIO_BLK_F_CONFIG_WCE was not negotiated but VIRTIO_BLK_F_FLUSH was, the driver SHOULD assume presence of a writeback cache.

The driver MUST NOT read *writeback* before setting the FEATURES_OK *device status* bit.

#### 5.2.5.2 Device Requirements: Device Initialization

Devices SHOULD always offer VIRTIO_BLK_F_FLUSH, and MUST offer it if they offer VIRTIO_BLK_F_CONFIG_WCE.

If VIRTIO_BLK_F_CONFIG_WCE is negotiated but VIRTIO_BLK_F_FLUSH is not, the device MUST initialize *writeback* to 0.

The device MUST initialize padding bytes *unused0* and *unused1* to 0.

#### 5.2.5.3 Legacy Interface: Device Initialization

Because legacy devices do not have FEATURES_OK, transitional devices MUST implement slightly different behavior around feature negotiation when used through the legacy interface. In particular, when using the legacy interface:

- the driver MAY read or write *writeback* before setting the DRIVER or DRIVER_OK *device status* bit
- the device MUST NOT modify the cache mode (and *writeback*) as a result of a driver setting a status bit, unless the DRIVER_OK bit is being set and the driver has not set the VIRTIO_BLK_F_CONFIG_WCE driver feature bit.
- the device MUST NOT modify the cache mode (and *writeback*) as a result of a driver modifying the driver feature bits, for example if the driver sets the VIRTIO_BLK_F_CONFIG_WCE driver feature bit but does not set the VIRTIO_BLK_F_FLUSH bit.

### 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:

[4] Consistent with 5.2.6.2, a writethrough cache can be defined broadly as a cache that commits writes to persistent device backend storage before reporting their completion. For example, a battery-backed writeback cache actually counts as writethrough according to this definition.

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

struct virtio_blk_discard_write_zeroes {
       le64 sector;
       le32 num_sectors;
       struct {
               le32 unmap:1;
               le32 reserved:31;
       } flags;
};
```

The type of the request is either a read (VIRTIO_BLK_T_IN), a write (VIRTIO_BLK_T_OUT), a discard (VIRTIO_BLK_T_DISCARD), a write zeroes (VIRTIO_BLK_T_WRITE_ZEROES) 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
#define VIRTIO_BLK_T_DISCARD      11
#define VIRTIO_BLK_T_WRITE_ZEROES 13
```

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 commands other than read or write.

The *data* used for discard or write zeroes command is described by one or more virtio_blk_discard_write_zeroes structs. *sector* indicates the starting offset (in 512-byte units) of the segment, while *num_sectors* indicates the number of sectors in each discarded range. *unmap* is only used for write zeroes command.

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.

If the VIRTIO_BLK_F_CONFIG_WCE feature is negotiated, the driver MAY switch to writethrough or writeback mode by writing respectively 0 and 1 to the *writeback* field. After writing a 0 to *writeback*, the driver MUST NOT assume that any volatile writes have been committed to persistent device backend storage.

The *unmap* bit MUST be zero for discard commands. The driver MUST NOT assume anything about the data returned by read requests after a range of sectors has been discarded.

#### 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.

The device MUST set the *status* byte to VIRTIO_BLK_S_UNSUPP for discard and write zeroes commands if any unknown flag is set. Furthermore, the device MUST set the *status* byte to VIRTIO_BLK_S_UNSUPP for discard commands if the *unmap* flag is set.

For discard commands, the device MAY deallocate the specified range of sectors in the device backend storage.

For write zeroes commands, if the *unmap* is set, the device MAY deallocate the specified range of sectors in the device backend storage, as if the DISCARD command had been sent. After a write zeroes command is completed, reads of the specified ranges of sectors MUST return zeroes. This is true independent of whether *unmap* was set or clear.

The device SHOULD clear the *write_zeroes_may_unmap* field of the virtio configuration space if and only if a write zeroes request cannot result in deallocating one or more sectors. The device MAY change the content of the field during operation of the device; when this happens, the device SHOULD trigger a configuration change notification.

A write is considered volatile when it is submitted; the contents of sectors covered by a volatile write are undefined in persistent device backend storage until the write becomes stable. A write becomes stable once it is completed and one or more of the following conditions is true:

1. neither VIRTIO_BLK_F_CONFIG_WCE nor VIRTIO_BLK_F_FLUSH feature were negotiated, but VIRTIO_BLK_F_FLUSH was offered by the device;
2. the VIRTIO_BLK_F_CONFIG_WCE feature was negotiated and the *writeback* field in configuration space was 0 **all the time between the submission of the write and its completion**;
3. a VIRTIO_BLK_T_FLUSH request is sent **after the write is completed** and is completed itself.

If the device is backed by persistent storage, the device MUST ensure that stable writes are committed to it, before reporting completion of the write (cases 1 and 2) or the flush (case 3). Failure to do so can cause data loss in case of a crash.

If the driver changes *writeback* between the submission of the write and its completion, the write could be either volatile or stable when its completion is reported; in other words, the exact behavior is undefined.

If VIRTIO_BLK_F_FLUSH was not offered by the device[5], the device MAY also commit writes to persistent device backend storage before reporting their completion. Unlike case 1, however, this is not an absolute requirement of the specification.

**Note:** An implementation that does not offer VIRTIO_BLK_F_FLUSH and does not commit completed writes will not be resilient to data loss in case of crashes. Not offering VIRTIO_BLK_F_FLUSH is an absolute requirement for implementations that do not wish to be safe against such data losses.

#### 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.

When using the legacy interface, transitional drivers SHOULD ignore the used length values.

**Note:** Historically, some devices put the total descriptor length, or the total length of device-writable buffers there, even when only the status byte was actually written.

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.

[5] Note that in this case, according to 5.2.5.2, the device will not have offered VIRTIO_BLK_F_CONFIG_WCE either.

```
#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.

Some older legacy devices did not commit completed writes to persistent device backend storage when VIRTIO_BLK_F_FLUSH was offered but not negotiated. In order to work around this, the driver MAY set the *writeback* to 0 (if available) or it MAY send an explicit flush request after every completed write.

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*, *reserved* 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 2.6.4.

## 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[6].
2. When a buffer is used in the receiveq (signalled by a used buffer notification), 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 notification 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.

[6] Because this is high importance and low bandwidth, the current Linux implementation polls for the buffer to become used, rather than waiting for a used buffer notification, 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.

#### 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.

**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.

When using the legacy interface, the driver SHOULD ignore the used length values for the transmit queues and the control transmitq.

**Note:** Historically, some devices put the total descriptor length there, even though no data was actually written.

### 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 device 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 Traditional Memory Balloon Device

This is the traditional balloon device. The device number 13 is reserved for a new memory balloon interface, with different semantics, which is expected in a future version of the standard.

The traditional 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 has to be told before pages from the balloon are used.

**VIRTIO_BALLOON_F_STATS_VQ (1)** A virtqueue for reporting guest memory statistics is present.

**VIRTIO_BALLOON_F_DEFLATE_ON_OOM (2)** Deflate balloon on guest out of memory condition.

#### 5.5.3.1 Driver Requirements: Feature bits

The driver SHOULD accept the VIRTIO_BALLOON_F_MUST_TELL_HOST feature if offered by the device.

#### 5.5.3.2 Device Requirements: Feature bits

If the device offers the VIRTIO_BALLOON_F_MUST_TELL_HOST feature bit, and if the driver did not accept this feature bit, the device MAY signal failure by failing to set FEATURES_OK *device status* bit when the driver writes it.

##### 5.5.3.2.0.1 Legacy Interface: Feature bits

As the legacy interface does not have a way to gracefully report feature negotiation failure, when using the legacy interface, transitional devices MUST support guests which do not negotiate VIRTIO_BALLOON_F_MUST_TELL_HOST feature, and SHOULD allow guest to use memory before notifying host if VIRTIO_BALLOON_F_MUST_TELL_HOST is not negotiated.

### 5.5.4 Device configuration layout

Both fields of this configuration are always available.

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

##### 5.5.4.0.0.1 Legacy Interface: Device configuration layout

When using the legacy interface, transitional devices and drivers MUST format the fields in struct virtio_balloon_config according to the little-endian format.

**Note:** This is unlike the usual convention that legacy device fields are guest endian.

### 5.5.5 Device Initialization

The device initialization process is outlined below:

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.
   (c) DRIVER_OK is set: device operation begins.
   (d) Notify the device about the stats virtqueue buffer.

### 5.5.6 Device Operation

The device is driven either by the receipt of a configuration change notification, or by changing guest memory needs, such as performing memory compaction or responding to out of memory conditions.

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[7] and the descriptor describing the resulting 32-bit array is added to the inflateq.

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

3. To remove memory from the balloon (aka. deflate):
   (a) The driver constructs an array of addresses of memory pages it has previously given to the balloon, as described above. This descriptor is added to the deflateq.
   (b) If the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is negotiated, the guest informs the device of pages before it uses them.
   (c) Otherwise, the guest is allowed to re-use pages previously given to the balloon before the device has acknowledged their withdrawal[8].
4. In either case, the device acknowledges inflate and deflate requests by using the descriptor.
5. Once the device has acknowledged the inflation or deflation, the driver updates *actual* to reflect the new number of pages in the balloon.

#### 5.5.6.1 Driver Requirements: Device Operation

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

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

The driver MAY supply pages to the balloon when *num_pages* is greater than or equal to the actual number of pages in the balloon.

If VIRTIO_BALLOON_F_DEFLATE_ON_OOM has not been negotiated, the driver MUST NOT use pages from the balloon when *num_pages* is less than or equal to the actual number of pages in the balloon.

If VIRTIO_BALLOON_F_DEFLATE_ON_OOM has been negotiated, the driver MAY use pages from the balloon when *num_pages* is less than or equal to the actual number of pages in the balloon if this is required for system stability (e.g. if memory is required by applications running within the guest).

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 NOT use pages from the balloon until the device has acknowledged the deflate request.

Otherwise, if the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is not negotiated, the driver MAY begin to re-use pages previously given to the balloon before the device has acknowledged the deflate request.

In any case, the driver MUST NOT use pages from the balloon after adding the pages to the balloon, but before the device has acknowledged the inflate request.

The driver MUST NOT request deflation of pages in the balloon before the device has acknowledged the inflate request.

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

The driver MAY update *actual* once after multiple inflate and deflate operations.

#### 5.5.6.2 Device Requirements: Device Operation

The device MAY modify the contents of a page in the balloon after detecting its physical number in an inflate request and before acknowledging the inflate request by using the inflateq descriptor.

If the VIRTIO_BALLOON_F_MUST_TELL_HOST feature is negotiated, the device MAY modify the contents of a page in the balloon after detecting its physical number in an inflate request and before detecting its physical number in a deflate request and acknowledging the deflate request.

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

#### 5.5.6.2.1 Legacy Interface: Device Operation

When using the legacy interface, the driver SHOULD ignore the used length values.

**Note:** Historically, some devices put the total descriptor length there, even though no data was actually written.

When using the legacy interface, the driver MUST write out all 4 bytes each time it updates the *actual* value in the configuration space, using a single atomic operation.

When using the legacy interface, the device SHOULD NOT use the *actual* value written by the driver in the configuration space, until the last, most-significant byte of the value has been written.

**Note:** Historically, devices used the *actual* value, even though when using Virtio Over PCI Bus the device-specific configuration space was not guaranteed to be atomic. Using intermediate values during update by driver is best avoided, except for debugging.

Historically, drivers using Virtio Over PCI Bus wrote the *actual* value by using multiple single-byte writes in order, from the least-significant to the most-significant value.

### 5.5.6.3 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 uses the buffer and sends a used buffer notification.
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.

Within the buffer, statistics are an array of 6-byte entries. 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, devices 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
#define VIRTIO_BALLOON_S_AVAIL    6
#define VIRTIO_BALLOON_S_CACHES   7
#define VIRTIO_BALLOON_S_HTLB_PGALLOC 8
#define VIRTIO_BALLOON_S_HTLB_PGFAIL  9
        le16 tag;
        le64 val;
} __attribute__((packed));
```

#### 5.5.6.3.1 Driver Requirements: Memory Statistics

Normative statements in this section apply if and only if the VIRTIO_BALLOON_F_STATS_VQ feature has been negotiated.

The driver MUST make at most one buffer available to the device in the statsq, at all times.

After initializing the device, the driver MUST make an output buffer available in the statsq.

Upon detecting that device has used a buffer in the statsq, the driver MUST make an output buffer available in the statsq.

Before making an output buffer available in the statsq, the driver MUST initialize it, including one struct virtio_balloon_stat entry for each statistic that it supports.

Driver MUST use an output buffer size which is a multiple of 6 bytes for all buffers submitted to the statsq.

Driver MAY supply struct virtio_balloon_stat entries in the output buffer submitted to the statsq in any order, without regard to *tag* values.

Driver MAY supply a subset of all statistics in the output buffer submitted to the statsq.

Driver MUST supply the same subset of statistics in all buffers submitted to the statsq.

#### 5.5.6.3.2 Device Requirements: Memory Statistics

Normative statements in this section apply if and only if the VIRTIO_BALLOON_F_STATS_VQ feature has been negotiated.

Within an output buffer submitted to the statsq, the device MUST ignore entries with *tag* values that it does not recognize.

Within an output buffer submitted to the statsq, the device MUST accept struct virtio_balloon_stat entries in any order without regard to *tag* values.

#### 5.5.6.3.3 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.

When using the legacy interface, the device SHOULD ignore all values in the first buffer in the statsq supplied by the driver after device initialization.

**Note:** Historically, drivers supplied an uninitialized buffer in the first buffer.

### 5.5.6.4 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).

**VIRTIO_BALLOON_S_AVAIL (6)** An estimate of how much memory is available (in bytes) for starting new applications, without pushing the system to swap.

**VIRTIO_BALLOON_S_CACHES (7)** The amount of memory, in bytes, that can be quickly reclaimed without additional I/O. Typically these pages are used for caching files from disk.

**VIRTIO_BALLOON_S_HTLB_PGALLOC (8)** The number of successful hugetlb page allocations in the guest.

**VIRTIO_BALLOON_S_HTLB_PGFAIL (9)** The number of failed hugetlb page allocations in the guest.

## 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*** 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.

[9] For example, INQUIRY or REPORT LUNS.
[10] For example, I_T RESET.

#### 5.6.6.0.1 Legacy Interface: Device Operation

When using the legacy interface, the driver SHOULD ignore the used length values.

**Note:** Historically, devices put the total descriptor length, or the total length of device-writable buffers there, even when only part of the buffers were actually written.

### 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 three 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 field is 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
#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*.

#### 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).

[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*.

**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 cache 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.

## 5.7 GPU Device

virtio-gpu is a virtio based graphics adapter. It can operate in 2D mode and in 3D (virgl) mode. 3D mode will offload rendering ops to the host gpu and therefore requires a gpu with 3D support on the host machine.

3D mode is not covered (yet) in this specification, even though it is mentioned here and there due to some details of the virtual hardware being designed with 3D mode in mind.

In 2D mode the virtio-gpu device provides support for ARGB Hardware cursors and multiple scanouts (aka heads).

### 5.7.1 Device ID

16

### 5.7.2 Virtqueues

**0** controlq - queue for sending control commands

**1** cursorq - queue for sending cursor updates

Both queues have the same format. Each request and each response have a fixed header, followed by command specific data fields. The separate cursor queue is the ”fast track” for cursor commands (VIRTIO_GPU_CMD_UPDATE_CURSOR and VIRTIO_GPU_CMD_MOVE_CURSOR), so they go though without being delayed by time-consuming commands in the control queue.

### 5.7.3 Feature bits

**VIRTIO_GPU_F_VIRGL (0)** virgl 3D mode is supported.

**VIRTIO_GPU_F_EDID (1)** EDID is supported.

### 5.7.4 Device configuration layout

```
#define VIRTIO_GPU_EVENT_DISPLAY (1 << 0)

struct virtio_gpu_config {
        le32 events_read;
        le32 events_clear;
        le32 num_scanouts;
        le32 reserved;
}
```

#### 5.7.4.1 Device configuration fields

***events_read*** signals pending events to the driver. The driver MUST NOT write to this field.

***events_clear*** clears pending events in the device. Writing a ’1’ into a bit will clear the corresponding bit in *events_read*, mimicking write-to-clear behavior.

***num_scanouts*** specifies the maximum number of scanouts supported by the device. Minimum value is 1, maximum value is 16.

#### 5.7.4.2 Events

**VIRTIO_GPU_EVENT_DISPLAY** Display configuration has changed. The driver SHOULD use the VIRTIO_GPU_CMD_GET_DISPLAY_INFO command to fetch the information from the device.

### 5.7.5 Device Requirements: Device Initialization

The driver SHOULD query the display information from the device using the VIRTIO_GPU_CMD_GET_DISPLAY_INFO command and use that information for the initial scanout setup. In case no information is available or all displays are disabled the driver MAY choose to use a fallback, such as 1024x768 at display 0.

### 5.7.6 Device Operation

The virtio-gpu is based around the concept of resources private to the host, the guest must DMA transfer into these resources. This is a design requirement in order to interface with future 3D rendering. In the unaccelerated 2D mode there is no support for DMA transfers from resources, just to them.

Resources are initially simple 2D resources, consisting of a width, height and format along with an identifier. The guest must then attach backing store to the resources in order for DMA transfers to work. This is like a GART in a real GPU.

#### 5.7.6.1 Device Operation: Create a framebuffer and configure scanout

- Create a host resource using VIRTIO_GPU_CMD_RESOURCE_CREATE_2D.
- Allocate a framebuffer from guest ram, and attach it as backing storage to the resource just created, using VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING. Scatter lists are supported, so the framebuffer doesn't need to be contignous in guest physical memory.
- Use VIRTIO_GPU_CMD_SET_SCANOUT to link the framebuffer to a display scanout.

#### 5.7.6.2 Device Operation: Update a framebuffer and scanout

- Render to your framebuffer memory.
- Use VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D to update the host resource from guest memory.
- Use VIRTIO_GPU_CMD_RESOURCE_FLUSH to flush the updated resource to the display.

#### 5.7.6.3 Device Operation: Using pageflip

It is possible to create multiple framebuffers, flip between them using VIRTIO_GPU_CMD_SET_SCANOUT and VIRTIO_GPU_CMD_RESOURCE_FLUSH, and update the invisible framebuffer using VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D.

#### 5.7.6.4 Device Operation: Multihead setup

In case two or more displays are present there are different ways to configure things:

- Create a single framebuffer, link it to all displays (mirroring).
- Create an framebuffer for each display.
- Create one big framebuffer, configure scanouts to display a different rectangle of that framebuffer each.

#### 5.7.6.5 Device Requirements: Device Operation: Command lifecycle and fencing

The device MAY process controlq commands asyncronously and return them to the driver before the processing is complete. If the driver needs to know when the processing is finished it can set the VIRTIO_GPU_FLAG_FENCE flag in the request. The device MUST finish the processing before returning the command then.

Note: current qemu implementation does asyncrounous processing only in 3d mode, when offloading the processing to the host gpu.

#### 5.7.6.6 Device Operation: Configure mouse cursor

The mouse cursor image is a normal resource, except that it must be 64x64 in size. The driver MUST create and populate the resource (using the usual VIRTIO_GPU_CMD_RESOURCE_CREATE_2D, VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING and VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D controlq commands) and make sure they are completed (using VIRTIO_GPU_FLAG_FENCE).

Then VIRTIO_GPU_CMD_UPDATE_CURSOR can be sent to the cursorq to set the pointer shape and position. To move the pointer without updating the shape use VIRTIO_GPU_CMD_MOVE_CURSOR instead.

#### 5.7.6.7 Device Operation: Request header

```
enum virtio_gpu_ctrl_type {

        /* 2d commands */
        VIRTIO_GPU_CMD_GET_DISPLAY_INFO = 0x0100,
        VIRTIO_GPU_CMD_RESOURCE_CREATE_2D,
        VIRTIO_GPU_CMD_RESOURCE_UNREF,
        VIRTIO_GPU_CMD_SET_SCANOUT,
        VIRTIO_GPU_CMD_RESOURCE_FLUSH,
        VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D,
        VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING,
        VIRTIO_GPU_CMD_RESOURCE_DETACH_BACKING,
        VIRTIO_GPU_CMD_GET_CAPSET_INFO,
        VIRTIO_GPU_CMD_GET_CAPSET,
        VIRTIO_GPU_CMD_GET_EDID,

        /* cursor commands */
        VIRTIO_GPU_CMD_UPDATE_CURSOR = 0x0300,
        VIRTIO_GPU_CMD_MOVE_CURSOR,

        /* success responses */
        VIRTIO_GPU_RESP_OK_NODATA = 0x1100,
        VIRTIO_GPU_RESP_OK_DISPLAY_INFO,
        VIRTIO_GPU_RESP_OK_CAPSET_INFO,
        VIRTIO_GPU_RESP_OK_CAPSET,
        VIRTIO_GPU_RESP_OK_EDID,

        /* error responses */
        VIRTIO_GPU_RESP_ERR_UNSPEC = 0x1200,
        VIRTIO_GPU_RESP_ERR_OUT_OF_MEMORY,
        VIRTIO_GPU_RESP_ERR_INVALID_SCANOUT_ID,
        VIRTIO_GPU_RESP_ERR_INVALID_RESOURCE_ID,
        VIRTIO_GPU_RESP_ERR_INVALID_CONTEXT_ID,
        VIRTIO_GPU_RESP_ERR_INVALID_PARAMETER,
};

#define VIRTIO_GPU_FLAG_FENCE (1 << 0)

struct virtio_gpu_ctrl_hdr {
        le32 type;
        le32 flags;
        le64 fence_id;
        le32 ctx_id;
        le32 padding;
};
```

All requests and responses on the virt queues have the fixed header *struct virtio_gpu_ctrl_hdr*.

***type*** specifies the type of the driver request (VIRTIO_GPU_CMD_*) or device response (VIRTIO_GPU_RESP_*).

***flags*** request / response flags.

***fence_id*** If the driver sets the VIRTIO_GPU_FLAG_FENCE bit in the request *flags* field the device MUST:

- set VIRTIO_GPU_FLAG_FENCE bit in the response,
- copy the content of the *fence_id* field from the request to the response, and

- send the response only after command processing is complete.

***ctx_id*** Rendering context (used in 3D mode only).

On success the device will return VIRTIO_GPU_RESP_OK_NODATA in case there is no payload. Otherwise the *type* field will indicate the kind of payload.

On error the device will return one of the VIRTIO_GPU_RESP_ERR_* error codes.

#### 5.7.6.8 Device Operation: controlq

For any coordinates given 0,0 is top left, larger x moves right, larger y moves down.

**VIRTIO_GPU_CMD_GET_DISPLAY_INFO** Retrieve the current output configuration. No request data (just bare *struct virtio_gpu_ctrl_hdr*). Response type is VIRTIO_GPU_RESP_OK_DISPLAY_INFO, response data is *struct virtio_gpu_resp_display_info*.

```
#define VIRTIO_GPU_MAX_SCANOUTS 16

struct virtio_gpu_rect {
        le32 x;
        le32 y;
        le32 width;
        le32 height;
};

struct virtio_gpu_resp_display_info {
        struct virtio_gpu_ctrl_hdr hdr;
        struct virtio_gpu_display_one {
                struct virtio_gpu_rect r;
                le32 enabled;
                le32 flags;
        } pmodes[VIRTIO_GPU_MAX_SCANOUTS];
};
```

The response contains a list of per-scanout information. The info contains whether the scanout is enabled and what its preferred position and size is.

The size (fields *width* and *height*) is similar to the native panel resolution in EDID display information, except that in the virtual machine case the size can change when the host window representing the guest display is gets resized.

The position (fields *x* and *y*) describe how the displays are arranged (i.e. which is – for example – the left display).

The *enabled* field is set when the user enabled the display. It is roughly the same as the connected state of a phyiscal display connector.

**VIRTIO_GPU_CMD_GET_EDID** Retrieve the EDID data for a given scanout. Request data is *struct virtio_gpu_get_edid*). Response type is VIRTIO_GPU_RESP_OK_EDID, response data is *struct virtio_gpu_resp_edid*. Support is optional and negotiated using the VIRTIO_GPU_F_EDID feature flag.

```
struct virtio_gpu_get_edid {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 scanout;
        le32 padding;
};

struct virtio_gpu_resp_edid {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 size;
        le32 padding;
        u8 edid[1024];
};
```

The response contains the EDID display data blob (as specified by VESA) for the scanout.

**VIRTIO_GPU_CMD_RESOURCE_CREATE_2D** Create a 2D resource on the host. Request data is *struct virtio_gpu_resource_create_2d*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
enum virtio_gpu_formats {
        VIRTIO_GPU_FORMAT_B8G8R8A8_UNORM  = 1,
        VIRTIO_GPU_FORMAT_B8G8R8X8_UNORM  = 2,
        VIRTIO_GPU_FORMAT_A8R8G8B8_UNORM  = 3,
        VIRTIO_GPU_FORMAT_X8R8G8B8_UNORM  = 4,

        VIRTIO_GPU_FORMAT_R8G8B8A8_UNORM  = 67,
        VIRTIO_GPU_FORMAT_X8B8G8R8_UNORM  = 68,

        VIRTIO_GPU_FORMAT_A8B8G8R8_UNORM  = 121,
        VIRTIO_GPU_FORMAT_R8G8B8X8_UNORM  = 134,
};

struct virtio_gpu_resource_create_2d {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 resource_id;
        le32 format;
        le32 width;
        le32 height;
};
```

This creates a 2D resource on the host with the specified width, height and format. The resource ids are generated by the guest.

**VIRTIO_GPU_CMD_RESOURCE_UNREF** Destroy a resource. Request data is *struct virtio_gpu_resource_unref*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_resource_unref {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 resource_id;
        le32 padding;
};
```

This informs the host that a resource is no longer required by the guest.

**VIRTIO_GPU_CMD_SET_SCANOUT** Set the scanout parameters for a single output. Request data is *struct virtio_gpu_set_scanout*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_set_scanout {
        struct virtio_gpu_ctrl_hdr hdr;
        struct virtio_gpu_rect r;
        le32 scanout_id;
        le32 resource_id;
};
```

This sets the scanout parameters for a single scanout. The resource_id is the resource to be scanned out from, along with a rectangle.

Scanout rectangles must be completely covered by the underlying resource. Overlapping (or identical) scanouts are allowed, typical use case is screen mirroring.

The driver can use resource_id = 0 to disable a scanout.

**VIRTIO_GPU_CMD_RESOURCE_FLUSH** Flush a scanout resource Request data is *struct virtio_gpu_resource_flush*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_resource_flush {
        struct virtio_gpu_ctrl_hdr hdr;
        struct virtio_gpu_rect r;
        le32 resource_id;
        le32 padding;
};
```

This flushes a resource to screen. It takes a rectangle and a resource id, and flushes any scanouts the resource is being used on.

  

**VIRTIO_GPU_CMD_TRANSFER_TO_HOST_2D** Transfer from guest memory to host resource. Request data is *struct virtio_gpu_transfer_to_host_2d*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_transfer_to_host_2d {
        struct virtio_gpu_ctrl_hdr hdr;
        struct virtio_gpu_rect r;
        le64 offset;
        le32 resource_id;
        le32 padding;
};
```

This takes a resource id along with an destination offset into the resource, and a box to transfer to the host backing for the resource.

**VIRTIO_GPU_CMD_RESOURCE_ATTACH_BACKING** Assign backing pages to a resource. Request data is *struct virtio_gpu_resource_attach_backing*, followed by *struct virtio_gpu_mem_entry* entries. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_resource_attach_backing {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 resource_id;
        le32 nr_entries;
};

struct virtio_gpu_mem_entry {
        le64 addr;
        le32 length;
        le32 padding;
};
```

This assign an array of guest pages as the backing store for a resource. These pages are then used for the transfer operations for that resource from that point on.

**VIRTIO_GPU_CMD_RESOURCE_DETACH_BACKING** Detach backing pages from a resource. Request data is *struct virtio_gpu_resource_detach_backing*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
struct virtio_gpu_resource_detach_backing {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 resource_id;
        le32 padding;
};
```

This detaches any backing pages from a resource, to be used in case of guest swapping or object destruction.

#### 5.7.6.9 Device Operation: cursorq

Both cursorq commands use the same command struct.

```
struct virtio_gpu_cursor_pos {
        le32 scanout_id;
        le32 x;
        le32 y;
        le32 padding;
};

struct virtio_gpu_update_cursor {
        struct virtio_gpu_ctrl_hdr hdr;
        struct virtio_gpu_cursor_pos pos;
        le32 resource_id;
        le32 hot_x;
        le32 hot_y;
        le32 padding;
};
```

**VIRTIO_GPU_CMD_UPDATE_CURSOR** Update cursor. Request data is *struct virtio_gpu_update_cursor*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

Full cursor update. Cursor will be loaded from the specified *resource_id* and will be moved to *pos*. The driver must transfer the cursor into the resource beforehand (using control queue commands) and make sure the commands to fill the resource are actually processed (using fencing).

**VIRTIO_GPU_CMD_MOVE_CURSOR** Move cursor. Request data is *struct virtio_gpu_update_cursor*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

Move cursor to the place specified in *pos*. The other fields are not used and will be ignored by the device.

### 5.7.7 VGA Compatibility

Applies to Virtio Over PCI only. The GPU device can come with and without VGA compatibility. The PCI class should be DISPLAY_VGA if VGA compatibility is present and DISPLAY_OTHER otherwise.

VGA compatibility: PCI region 0 has the linear framebuffer, standard vga registers are present. Configuring a scanout (VIRTIO_GPU_CMD_SET_SCANOUT) switches the device from vga compatibility mode into native virtio mode. A reset switches it back into vga compatibility mode.

Note: qemu implementation also provides bochs dispi interface io ports and mmio bar at pci region 1 and is therefore fully compatible with the qemu stdvga (see docs/specs/standard-vga.txt in the qemu source tree).

## 5.8 Input Device

The virtio input device can be used to create virtual human interface devices such as keyboards, mice and tablets. An instance of the virtio device represents one such input device. Device behavior mirrors that of the evdev layer in Linux, making pass-through implementations on top of evdev easy.

This specification defines how evdev events are transported over virtio and how the set of supported events is discovered by a driver. It does not, however, define the semantics of input events as this is dependent on the particular evdev implementation. For the list of events used by Linux input devices, see include/uapi/linux/input-event-codes.h in the Linux source tree.

### 5.8.1 Device ID

18

### 5.8.2 Virtqueues

**0** eventq

**1** statusq

### 5.8.3 Feature bits

None.

### 5.8.4 Device configuration layout

Device configuration holds all information the guest needs to handle the device, most importantly the events which are supported.

```
enum virtio_input_config_select {
  VIRTIO_INPUT_CFG_UNSET      = 0x00,
  VIRTIO_INPUT_CFG_ID_NAME    = 0x01,
  VIRTIO_INPUT_CFG_ID_SERIAL  = 0x02,
  VIRTIO_INPUT_CFG_ID_DEVIDS  = 0x03,
  VIRTIO_INPUT_CFG_PROP_BITS  = 0x10,
  VIRTIO_INPUT_CFG_EV_BITS    = 0x11,
  VIRTIO_INPUT_CFG_ABS_INFO   = 0x12,
};

struct virtio_input_absinfo {
  le32  min;
  le32  max;
  le32  fuzz;
  le32  flat;
  le32  res;
};

struct virtio_input_devids {
  le16  bustype;
  le16  vendor;
  le16  product;
  le16  version;
};

struct virtio_input_config {
  u8    select;
  u8    subsel;
  u8    size;
  u8    reserved[5];
  union {
    char string[128];
    u8   bitmap[128];
    struct virtio_input_absinfo abs;
    struct virtio_input_devids ids;
  } u;
};
```

To query a specific piece of information the driver sets *select* and *subsel* accordingly, then checks *size* to see how much information is available. *size* can be zero if no information is available. Strings do not include a NUL terminator. Related evdev ioctl names are provided for reference.

**VIRTIO_INPUT_CFG_ID_NAME** *subsel* is zero. Returns the name of the device, in *u.string*.

Similar to EVIOCGNAME ioctl for Linux evdev devices.

**VIRTIO_INPUT_CFG_ID_SERIAL** *subsel* is zero. Returns the serial number of the device, in *u.string*.

**VIRTIO_INPUT_CFG_ID_DEVIDS** *subsel* is zero. Returns ID information of the device, in *u.ids*.

Similar to EVIOCGID ioctl for Linux evdev devices.

**VIRTIO_INPUT_CFG_PROP_BITS** *subsel* is zero. Returns input properties of the device, in *u.bitmap*. Individual bits in the bitmap correspond to INPUT_PROP_* constants used by the underlying evdev implementation.

Similar to EVIOCGPROP ioctl for Linux evdev devices.

**VIRTIO_INPUT_CFG_EV_BITS** *subsel* specifies the event type using EV_* constants in the underlying evdev implementation. If *size* is non-zero the event type is supported and a bitmap of supported event codes is returned in *u.bitmap*. Individual bits in the bitmap correspond to implementation-defined input event codes, for example keys or pointing device axes.

Similar to EVIOCGBIT ioctl for Linux evdev devices.

**VIRTIO_INPUT_CFG_ABS_INFO** *subsel* specifies the absolute axis using ABS_* constants in the underlying evdev implementation. Information about the axis will be returned in *u.abs*.

Similar to EVIOCGABS ioctl for Linux evdev devices.

### 5.8.5 Device Initialization

1. The device is queried for supported event types and codes.
2. The eventq is populated with receive buffers.

#### 5.8.5.1 Driver Requirements: Device Initialization

A driver MUST set both *select* and *subsel* when querying device configuration, in any order.

A driver MUST NOT write to configuration fields other than *select* and *subsel*.

A driver SHOULD check the *size* field before accessing the configuration information.

#### 5.8.5.2 Device Requirements: Device Initialization

A device MUST set the *size* field to zero if it doesn't support a given *select* and *subsel* combination.

### 5.8.6 Device Operation

1. Input events such as press and release events for keys and buttons, and motion events for pointing devices are sent from the device to the driver using the eventq.
2. Status feedback such as keyboard LED updates are sent from the driver to the device using the statusq.
3. Both queues use the same virtio_input_event struct. *type*, *code* and *value* are filled according to the Linux input layer (evdev) interface, except that the fields are in little endian byte order whereas the evdev ioctl interface uses native endian-ness.

```
struct virtio_input_event {
  le16 type;
  le16 code;
  le32 value;
};
```

#### 5.8.6.1 Driver Requirements: Device Operation

A driver SHOULD keep the eventq populated with buffers. These buffers MUST be device-writable and MUST be at least the size of struct virtio_input_event.

Buffers placed into the statusq by a driver MUST be at least the size of struct virtio_input_event.

A driver SHOULD ignore eventq input events it does not recognize. Note that evdev devices generally maintain backward compatibility by sending redundant events and relying on the consuming side using only the events it understands and ignoring the rest.

#### 5.8.6.2 Device Requirements: Device Operation

A device MAY drop input events if the eventq does not have enough available buffers. It SHOULD NOT drop individual input events if they are part of a sequence forming one input device update. For example, a pointing device update typically consists of several input events, one for each axis, and a terminating EV_SYN event. A device SHOULD either buffer or drop the entire sequence.

## 5.9 Crypto Device

The virtio crypto device is a virtual cryptography device as well as a virtual cryptographic accelerator. The virtio crypto device provides the following crypto services: CIPHER, MAC, HASH, and AEAD. Virtio crypto devices have a single control queue and at least one data queue. Crypto operation requests are placed into a data queue, and serviced by the device. Some crypto operation requests are only valid in the context of a session. The role of the control queue is facilitating control operation requests. Sessions management is realized with control operation requests.

### 5.9.1 Device ID

20

### 5.9.2 Virtqueues

**0** dataq1

**...**

**N-1** dataqN

**N** controlq

N is set by *max_dataqueues*.

### 5.9.3 Feature bits

VIRTIO_CRYPTO_F_REVISION_1 (0) revision 1. Revision 1 has a specific request format and other enhancements (which result in some additional requirements).

VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE (1) stateless mode requests are supported by the CIPHER service.

VIRTIO_CRYPTO_F_HASH_STATELESS_MODE (2) stateless mode requests are supported by the HASH service.

VIRTIO_CRYPTO_F_MAC_STATELESS_MODE (3) stateless mode requests are supported by the MAC service.

VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE (4) stateless mode requests are supported by the AEAD service.

#### 5.9.3.1 Feature bit requirements

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

**VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE** Requires VIRTIO_CRYPTO_F_REVISION_1.

**VIRTIO_CRYPTO_F_HASH_STATELESS_MODE** Requires VIRTIO_CRYPTO_F_REVISION_1.

**VIRTIO_CRYPTO_F_MAC_STATELESS_MODE** Requires VIRTIO_CRYPTO_F_REVISION_1.

**VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE** Requires VIRTIO_CRYPTO_F_REVISION_1.

### 5.9.4 Supported crypto services

The following crypto services are defined:

```
/* CIPHER service */
#define VIRTIO_CRYPTO_SERVICE_CIPHER 0
/* HASH service */
#define VIRTIO_CRYPTO_SERVICE_HASH   1
/* MAC (Message Authentication Codes) service */
#define VIRTIO_CRYPTO_SERVICE_MAC    2
/* AEAD (Authenticated Encryption with Associated Data) service */
#define VIRTIO_CRYPTO_SERVICE_AEAD   3
```

The above constants designate bits used to indicate the which of crypto services are offered by the device as described in, see 5.9.5.

#### 5.9.4.1 CIPHER services

The following CIPHER algorithms are defined:

```
#define VIRTIO_CRYPTO_NO_CIPHER                 0
#define VIRTIO_CRYPTO_CIPHER_ARC4               1
#define VIRTIO_CRYPTO_CIPHER_AES_ECB            2
#define VIRTIO_CRYPTO_CIPHER_AES_CBC            3
#define VIRTIO_CRYPTO_CIPHER_AES_CTR            4
#define VIRTIO_CRYPTO_CIPHER_DES_ECB            5
#define VIRTIO_CRYPTO_CIPHER_DES_CBC            6
#define VIRTIO_CRYPTO_CIPHER_3DES_ECB           7
#define VIRTIO_CRYPTO_CIPHER_3DES_CBC           8
#define VIRTIO_CRYPTO_CIPHER_3DES_CTR           9
#define VIRTIO_CRYPTO_CIPHER_KASUMI_F8          10
#define VIRTIO_CRYPTO_CIPHER_SNOW3G_UEA2        11
#define VIRTIO_CRYPTO_CIPHER_AES_F8             12
#define VIRTIO_CRYPTO_CIPHER_AES_XTS            13
#define VIRTIO_CRYPTO_CIPHER_ZUC_EEA3           14
```

The above constants have two usages:

1. As bit numbers, used to tell the driver which CIPHER algorithms are supported by the device, see 5.9.5.
2. As values, used to designate the algorithm in (CIPHER type) crypto operation requests, see 5.9.7.2.1.

#### 5.9.4.2 HASH services

The following HASH algorithms are defined:

```
#define VIRTIO_CRYPTO_NO_HASH           0
#define VIRTIO_CRYPTO_HASH_MD5          1
#define VIRTIO_CRYPTO_HASH_SHA1         2
#define VIRTIO_CRYPTO_HASH_SHA_224      3
#define VIRTIO_CRYPTO_HASH_SHA_256      4
#define VIRTIO_CRYPTO_HASH_SHA_384      5
#define VIRTIO_CRYPTO_HASH_SHA_512      6
#define VIRTIO_CRYPTO_HASH_SHA3_224     7
#define VIRTIO_CRYPTO_HASH_SHA3_256     8
#define VIRTIO_CRYPTO_HASH_SHA3_384     9
#define VIRTIO_CRYPTO_HASH_SHA3_512     10
#define VIRTIO_CRYPTO_HASH_SHA3_SHAKE128     11
#define VIRTIO_CRYPTO_HASH_SHA3_SHAKE256     12
```

The above constants have two usages:

1. As bit numbers, used to tell the driver which HASH algorithms are supported by the device, see 5.9.5.
2. As values, used to designate the algorithm in (HASH type) crypto operation requires, see 5.9.7.2.1.

#### 5.9.4.3 MAC services

The following MAC algorithms are defined:

```
#define VIRTIO_CRYPTO_NO_MAC                        0
#define VIRTIO_CRYPTO_MAC_HMAC_MD5                  1
#define VIRTIO_CRYPTO_MAC_HMAC_SHA1                 2
#define VIRTIO_CRYPTO_MAC_HMAC_SHA_224              3
#define VIRTIO_CRYPTO_MAC_HMAC_SHA_256              4
#define VIRTIO_CRYPTO_MAC_HMAC_SHA_384              5
#define VIRTIO_CRYPTO_MAC_HMAC_SHA_512              6
#define VIRTIO_CRYPTO_MAC_CMAC_3DES                 25
#define VIRTIO_CRYPTO_MAC_CMAC_AES                  26
#define VIRTIO_CRYPTO_MAC_KASUMI_F9                 27
#define VIRTIO_CRYPTO_MAC_SNOW3G_UIA2               28
#define VIRTIO_CRYPTO_MAC_GMAC_AES                  41
#define VIRTIO_CRYPTO_MAC_GMAC_TWOFISH              42
#define VIRTIO_CRYPTO_MAC_CBCMAC_AES                49
#define VIRTIO_CRYPTO_MAC_CBCMAC_KASUMI_F9          50
#define VIRTIO_CRYPTO_MAC_XCBC_AES                  53
#define VIRTIO_CRYPTO_MAC_ZUC_EIA3                  54
```

The above constants have two usages:

1. As bit numbers, used to tell the driver which MAC algorithms are supported by the device, see 5.9.5.
2. As values, used to designate the algorithm in (MAC type) crypto operation requests, see 5.9.7.2.1.

#### 5.9.4.4 AEAD services

The following AEAD algorithms are defined:

```
#define VIRTIO_CRYPTO_NO_AEAD     0
#define VIRTIO_CRYPTO_AEAD_GCM    1
#define VIRTIO_CRYPTO_AEAD_CCM    2
#define VIRTIO_CRYPTO_AEAD_CHACHA20_POLY1305  3
```

The above constants have two usages:

1. As bit numbers, used to tell the driver which AEAD algorithms are supported by the device, see 5.9.5.
2. As values, used to designate the algorithm in (DEAD type) crypto operation requests, see 5.9.7.2.1.

### 5.9.5 Device configuration layout

```
struct virtio_crypto_config {
    le32 status;
    le32 max_dataqueues;
    le32 crypto_services;
    /* Detailed algorithms mask */
    le32 cipher_algo_l;
    le32 cipher_algo_h;
    le32 hash_algo;
    le32 mac_algo_l;
    le32 mac_algo_h;
    le32 aead_algo;
    /* Maximum length of cipher key in bytes */
    le32 max_cipher_key_len;
    /* Maximum length of authenticated key in bytes */
    le32 max_auth_key_len;
    le32 reserved;
    /* Maximum size of each crypto request's content in bytes */
    le64 max_size;
};
```

Currently, only one *status* bit is defined: VIRTIO_CRYPTO_S_HW_READY set indicates that the device is ready to process requests, this bit is read-only for the driver

```
#define VIRTIO_CRYPTO_S_HW_READY  (1 << 0)
```

***max_dataqueues*** is the maximum number of data virtqueues that can be configured by the device. The driver MAY use only one data queue, or it can use more to achieve better performance.

***crypto_services*** crypto service offered, see 5.9.4.

***cipher_algo_l*** CIPHER algorithms bits 0-31, see 5.9.4.1.

***cipher_algo_h*** CIPHER algorithms bits 32-63, see 5.9.4.1.

***hash_algo*** HASH algorithms bits, see 5.9.4.2.

***mac_algo_l*** MAC algorithms bits 0-31, see 5.9.4.3.

***mac_algo_h*** MAC algorithms bits 32-63, see 5.9.4.3.

***aead_algo*** AEAD algorithms bits, see 5.9.4.4.

***max_cipher_key_len*** is the maximum length of cipher key supported by the device.

***max_auth_key_len*** is the maximum length of authenticated key supported by the device.

***reserved*** is reserved for future use.

***max_size*** is the maximum size of the variable-length parameters of data operation of each crypto request's content supported by the device.

**Note:** Unless explicitly stated otherwise all lengths and sizes are in bytes.

#### 5.9.5.1 Device Requirements: Device configuration layout

- The device MUST set *max_dataqueues* to between 1 and 65535 inclusive.
- The device MUST set the *status* with valid flags, undefined flags MUST NOT be set.
- The device MUST accept and handle requests after *status* is set to VIRTIO_CRYPTO_S_HW_READY.
- The device MUST set *crypto_services* based on the crypto services the device offers.
- The device MUST set detailed algorithms masks for each service advertised by *crypto_services*. The device MUST NOT set the not defined algorithms bits.
- The device MUST set *max_size* to show the maximum size of crypto request the device supports.
- The device MUST set *max_cipher_key_len* to show the maximum length of cipher key if the device supports CIPHER service.
- The device MUST set *max_auth_key_len* to show the maximum length of authenticated key if the device supports MAC service.

#### 5.9.5.2 Driver Requirements: Device configuration layout

- The driver MUST read the *status* from the bottom bit of status to check whether the VIRTIO_CRYPTO_S_HW_READY is set, and the driver MUST reread it after device reset.
- The driver MUST NOT transmit any requests to the device if the VIRTIO_CRYPTO_S_HW_READY is not set.
- The driver MUST read *max_dataqueues* field to discover the number of data queues the device supports.
- The driver MUST read *crypto_services* field to discover which services the device is able to offer.
- The driver SHOULD ignore the not defined algorithms bits.
- The driver MUST read the detailed algorithms fields based on *crypto_services* field.
- The driver SHOULD read *max_size* to discover the maximum size of the variable-length parameters of data operation of the crypto request's content the device supports and MUST guarantee the size of each crypto request's content within the *max_size*, otherwise the request will fail and the driver MUST reset the device.

- The driver SHOULD read *max_cipher_key_len* to discover the maximum length of cipher key the device supports and MUST guarantee the *key_len* (CIPHER service or AEAD service) within the *max_cipher_key_len* of the device configuration, otherwise the request will fail.
- The driver SHOULD read *max_auth_key_len* to discover the maximum length of authenticated key the device supports and MUST guarantee the *auth_key_len* (MAC service) within the *max_auth_key_len* of the device configuration, otherwise the request will fail.

## 5.9.6 Device Initialization

### 5.9.6.1 Driver Requirements: Device Initialization

- The driver MUST configure and initialize all virtqueues.
- The driver MUST read the supported crypto services from bits of *crypto_services*.
- The driver MUST read the supported algorithms based on *crypto_services* field.

## 5.9.7 Device Operation

The operation of a virtio crypto device is driven by requests placed on the virtqueues. Requests consist of a queue-type specific header (specifying among others the operation) and an operation specific payload.

If VIRTIO_CRYPTO_F_REVISION_1 is negotiated the device may support both session mode (See 5.9.7.2.1) and stateless mode operation requests. In stateless mode all operation parameters are supplied as a part of each request, while in session mode, some or all operation parameters are managed within the session. Stateless mode is guarded by feature bits 0-4 on a service level. If stateless mode is negotiated for a service, the service accepts both session mode and stateless requests; otherwise stateless mode requests are rejected (via operation status).

### 5.9.7.1 Operation Status

The device MUST return a status code as part of the operation (both session operation and service operation) result. The valid operation status as follows:

```
enum VIRTIO_CRYPTO_STATUS {
    VIRTIO_CRYPTO_OK = 0,
    VIRTIO_CRYPTO_ERR = 1,
    VIRTIO_CRYPTO_BADMSG = 2,
    VIRTIO_CRYPTO_NOTSUPP = 3,
    VIRTIO_CRYPTO_INVSESS = 4,
    VIRTIO_CRYPTO_NOSPC = 5,
    VIRTIO_CRYPTO_MAX
};
```

- VIRTIO_CRYPTO_OK: success.
- VIRTIO_CRYPTO_BADMSG: authentication failed (only when AEAD decryption).
- VIRTIO_CRYPTO_NOTSUPP: operation or algorithm is unsupported.
- VIRTIO_CRYPTO_INVSESS: invalid session ID when executing crypto operations.
- VIRTIO_CRYPTO_NOSPC: no free session ID (only when the VIRTIO_CRYPTO_F_REVISION_1 feature bit is negotiated).
- VIRTIO_CRYPTO_ERR: any failure not mentioned above occurs.

### 5.9.7.2 Control Virtqueue

The driver uses the control virtqueue to send control commands to the device, such as session operations (See 5.9.7.2.1).

The header for controlq is of the following form:

```
#define VIRTIO_CRYPTO_OPCODE(service, op)   (((service) << 8) | (op))

struct virtio_crypto_ctrl_header {
#define VIRTIO_CRYPTO_CIPHER_CREATE_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_CIPHER, 0x02)
#define VIRTIO_CRYPTO_CIPHER_DESTROY_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_CIPHER, 0x03)
#define VIRTIO_CRYPTO_HASH_CREATE_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_HASH, 0x02)
#define VIRTIO_CRYPTO_HASH_DESTROY_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_HASH, 0x03)
#define VIRTIO_CRYPTO_MAC_CREATE_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_MAC, 0x02)
#define VIRTIO_CRYPTO_MAC_DESTROY_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_MAC, 0x03)
#define VIRTIO_CRYPTO_AEAD_CREATE_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AEAD, 0x02)
#define VIRTIO_CRYPTO_AEAD_DESTROY_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AEAD, 0x03)
    le32 opcode;
    /* algo should be service-specific algorithms */
    le32 algo;
    le32 flag;
    le32 reserved;
};
```

The controlq request is composed of four parts:

```
struct virtio_crypto_op_ctrl_req {
    /* Device read only portion */

    struct virtio_crypto_ctrl_header header;

#define VIRTIO_CRYPTO_CTRLQ_OP_SPEC_HDR_LEGACY 56
    /* fixed length fields, opcode specific */
    u8 op_flf[flf_len];

    /* variable length fields, opcode specific */
    u8 op_vlf[vlf_len];

    /* Device write only portion */

    /* op result or completion status */
    u8 op_outcome[outcome_len];
};
```

*header* is a general header (see above).

*op_flf* is the opcode (in *header*) specific fixed-length paramenters.

*flf_len* depends on the VIRTIO_CRYPTO_F_REVISION_1 feature bit (see below).

*op_vlf* is the opcode (in *header*) specific variable-length paramenters.

*vlf_len* is the size of the specific structure used.

**Note:** The *vlf_len* of session-destroy operation and the hash-session-create operation is ZERO.

- If the opcode (in *header*) is VIRTIO_CRYPTO_CIPHER_CREATE_SESSION then *op_flf* is struct virtio_crypto_sym_create_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_sym_create_session_flf is padded to 56 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_sym_create_session_vlf.
- If the opcode (in *header*) is VIRTIO_CRYPTO_HASH_CREATE_SESSION then *op_flf* is struct virtio_crypto_hash_create_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_hash_create_session_flf is padded to 56 bytes if NOT negotiated.
- If the opcode (in *header*) is VIRTIO_CRYPTO_MAC_CREATE_SESSION then *op_flf* is struct virtio_crypto_mac_create_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_-

  

crypto_mac_create_session_flf is padded to 56 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_mac_create_session_vlf.

- If the opcode (in *header*) is VIRTIO_CRYPTO_AEAD_CREATE_SESSION then *op_flf* is struct virtio_crypto_aead_create_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_aead_create_session_flf is padded to 56 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_aead_create_session_vlf.
- If the opcode (in *header*) is VIRTIO_CRYPTO_CIPHER_DESTROY_SESSION or VIRTIO_CRYPTO_HASH_DESTROY_SESSION or VIRTIO_CRYPTO_MAC_DESTROY_SESSION or VIRTIO_CRYPTO_AEAD_DESTROY_SESSION then *op_flf* is struct virtio_crypto_destroy_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_destroy_session_flf is padded to 56 bytes if NOT negotiated.

*op_outcome* stores the result of operation and must be struct virtio_crypto_destroy_session_input for destroy session or struct virtio_crypto_create_session_input for create session.

*outcome_len* is the size of the structure used.

#### 5.9.7.2.1 Session operation

The session is a handle which describes the cryptographic parameters to be applied to a number of buffers.

The following structure stores the result of session creation set by the device:

```
struct virtio_crypto_create_session_input {
    le64 session_id;
    le32 status;
    le32 padding;
};
```

A request to destroy a session includes the following information:

```
struct virtio_crypto_destroy_session_flf {
    /* Device read only portion */
    le64  session_id;
};

struct virtio_crypto_destroy_session_input {
    /* Device write only portion */
    u8  status;
};
```

##### 5.9.7.2.1.1 Session operation: HASH session

The fixed-length paramenters of HASH session requests is as follows:

```
struct virtio_crypto_hash_create_session_flf {
    /* Device read only portion */

    /* See VIRTIO_CRYPTO_HASH_* above */
    le32 algo;
    /* hash result length */
    le32 hash_result_len;
};
```

##### 5.9.7.2.1.2 Session operation: MAC session

The fixed-length and the variable-length parameters of MAC session requests are as follows:

```
struct virtio_crypto_mac_create_session_flf {
    /* Device read only portion */

    /* See VIRTIO_CRYPTO_MAC_* above */
    le32 algo;
    /* hash result length */
    le32 hash_result_len;
    /* length of authenticated key */
    le32 auth_key_len;
    le32 padding;
};

struct virtio_crypto_mac_create_session_vlf {
    /* Device read only portion */

    /* The authenticated key */
    u8 auth_key[auth_key_len];
};
```

The length of *auth_key* is specified in *auth_key_len* in the struct virtio_crypto_mac_create_session_flf.

#### 5.9.7.2.1.3 Session operation: Symmetric algorithms session

The request of symmetric session could be the CIPHER algorithms request or the chain algorithms (chaining CIPHER and HASH/MAC) request.

The fixed-length and the variable-length parameters of CIPHER session requests are as follows:

```
struct virtio_crypto_cipher_session_flf {
    /* Device read only portion */

    /* See VIRTIO_CRYPTO_CIPHER* above */
    le32 algo;
    /* length of key */
    le32 key_len;
#define VIRTIO_CRYPTO_OP_ENCRYPT  1
#define VIRTIO_CRYPTO_OP_DECRYPT  2
    /* encryption or decryption */
    le32 op;
    le32 padding;
};

struct virtio_crypto_cipher_session_vlf {
    /* Device read only portion */

    /* The cipher key */
    u8 cipher_key[key_len];
};
```

The length of *cipher_key* is specified in *key_len* in the struct virtio_crypto_cipher_session_flf.

The fixed-length and the variable-length parameters of Chain session requests are as follows:

```
struct virtio_crypto_alg_chain_session_flf {
    /* Device read only portion */

#define VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER  1
#define VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH  2
    le32 alg_chain_order;
/* Plain hash */
#define VIRTIO_CRYPTO_SYM_HASH_MODE_PLAIN    1
/* Authenticated hash (mac) */
#define VIRTIO_CRYPTO_SYM_HASH_MODE_AUTH     2
/* Nested hash */
#define VIRTIO_CRYPTO_SYM_HASH_MODE_NESTED   3
    le32 hash_mode;
    struct virtio_crypto_cipher_session_flf cipher_hdr;
```

```
#define VIRTIO_CRYPTO_ALG_CHAIN_SESS_OP_SPEC_HDR_SIZE  16
    /* fixed length fields, algo specific */
    u8 algo_flf[VIRTIO_CRYPTO_ALG_CHAIN_SESS_OP_SPEC_HDR_SIZE];

    /* length of the additional authenticated data (AAD) in bytes */
    le32 aad_len;
    le32 padding;
};

struct virtio_crypto_alg_chain_session_vlf {
    /* Device read only portion */

    /* The cipher key */
    u8 cipher_key[key_len];
    /* The authenticated key */
    u8 auth_key[auth_key_len];
};
```

*hash_mode* decides the type used by *algo_flf*.

*algo_flf* is fixed to 16 bytes and MUST contains or be one of the following types:

- struct virtio_crypto_hash_create_session_flf
- struct virtio_crypto_mac_create_session_flf

The data of unused part (if has) in *algo_flf* will be ignored.

The length of *cipher_key* is specified in *key_len* in *cipher_hdr*.

The length of *auth_key* is specified in *auth_key_len* in struct virtio_crypto_mac_create_session_flf.

The fixed-length parameters of Symmetric session requests are as follows:

```
struct virtio_crypto_sym_create_session_flf {
    /* Device read only portion */

#define VIRTIO_CRYPTO_SYM_SESS_OP_SPEC_HDR_SIZE  48
    /* fixed length fields, opcode specific */
    u8 op_flf[VIRTIO_CRYPTO_SYM_SESS_OP_SPEC_HDR_SIZE];

/* No operation */
#define VIRTIO_CRYPTO_SYM_OP_NONE  0
/* Cipher only operation on the data */
#define VIRTIO_CRYPTO_SYM_OP_CIPHER  1
/* Chain any cipher with any hash or mac operation. The order
   depends on the value of alg_chain_order param */
#define VIRTIO_CRYPTO_SYM_OP_ALGORITHM_CHAINING  2
    le32 op_type;
    le32 padding;
};
```

*op_flf* is fixed to 48 bytes, MUST contains or be one of the following types:

- struct virtio_crypto_cipher_session_flf
- struct virtio_crypto_alg_chain_session_flf

The data of unused part (if has) in *op_flf* will be ignored.

*op_type* decides the type used by *op_flf*.

The variable-length parameters of Symmetric session requests are as follows:

```
struct virtio_crypto_sym_create_session_vlf {
    /* Device read only portion */
    /* variable length fields, opcode specific */
    u8 op_vlf[vlf_len];
};
```

*op_vlf* MUST contains or be one of the following types:

- struct virtio_crypto_cipher_session_vlf

  

- struct virtio_crypto_alg_chain_session_vlf

*op_type* in struct virtio_crypto_sym_create_session_flf decides the type used by *op_vlf*.

*vlf_len* is the size of the specific structure used.

#### 5.9.7.2.1.4 Session operation: AEAD session

The fixed-length and the variable-length parameters of AEAD session requests are as follows:

```
struct virtio_crypto_aead_create_session_flf {
    /* Device read only portion */

    /* See VIRTIO_CRYPTO_AEAD_* above */
    le32 algo;
    /* length of key */
    le32 key_len;
    /* Authentication tag length */
    le32 tag_len;
    /* The length of the additional authenticated data (AAD) in bytes */
    le32 aad_len;
    /* encryption or decryption, See above VIRTIO_CRYPTO_OP_* */
    le32 op;
    le32 padding;
};

struct virtio_crypto_aead_create_session_vlf {
    /* Device read only portion */
    u8 key[key_len];
};
```

The length of *key* is specified in *key_len* in struct virtio_crypto_aead_create_session_flf.

#### 5.9.7.2.1.5 Driver Requirements: Session operation: create session

- The driver MUST set the *opcode* field based on service type: CIPHER, HASH, MAC, or AEAD.
- The driver MUST set the control general header, the opcode specific header, the opcode specific extra parameters and the opcode specific outcome buffer in turn. See 5.9.7.2.
- The driver MUST set the *reversed* field to zero.

#### 5.9.7.2.1.6 Device Requirements: Session operation: create session

- The device MUST use the corresponding opcode specific structure according to the *opcode* in the control general header.
- The device MUST extract extra parameters according to the structures used.
- The device MUST set the *status* field to one of the following values of enum VIRTIO_CRYPTO_STATUS after finish a session creation:
  - VIRTIO_CRYPTO_OK if a session is created successfully.
  - VIRTIO_CRYPTO_NOTSUPP if the requested algorithm or operation is unsupported.
  - VIRTIO_CRYPTO_NOSPC if no free session ID (only when the VIRTIO_CRYPTO_F_REVISION_-1 feature bit is negotiated).
  - VIRTIO_CRYPTO_ERR if failure not mentioned above occurs.
- The device MUST set the *session_id* field to a unique session identifier only if the status is set to VIRTIO_CRYPTO_OK.

#### 5.9.7.2.1.7 Driver Requirements: Session operation: destroy session

- The driver MUST set the *opcode* field based on service type: CIPHER, HASH, MAC, or AEAD.

- The driver MUST set the *session_id* to a valid value assigned by the device when the session was created.

#### 5.9.7.2.1.8 Device Requirements: Session operation: destroy session

- The device MUST set the *status* field to one of the following values of enum VIRTIO_CRYPTO_STATUS.
  - VIRTIO_CRYPTO_OK if a session is created successfully.
  - VIRTIO_CRYPTO_ERR if any failure occurs.

### 5.9.7.3 Data Virtqueue

The driver uses the data virtqueues to transmit crypto operation requests to the device, and completes the crypto operations.

The header for dataq is as follows:

```
struct virtio_crypto_op_header {
#define VIRTIO_CRYPTO_CIPHER_ENCRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_CIPHER, 0x00)
#define VIRTIO_CRYPTO_CIPHER_DECRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_CIPHER, 0x01)
#define VIRTIO_CRYPTO_HASH \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_HASH, 0x00)
#define VIRTIO_CRYPTO_MAC \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_MAC, 0x00)
#define VIRTIO_CRYPTO_AEAD_ENCRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AEAD, 0x00)
#define VIRTIO_CRYPTO_AEAD_DECRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AEAD, 0x01)
    le32 opcode;
    /* algo should be service-specific algorithms */
    le32 algo;
    le64 session_id;
#define VIRTIO_CRYPTO_FLAG_SESSION_MODE 1
    /* control flag to control the request */
    le32 flag;
    le32 padding;
};
```

**Note:** If VIRTIO_CRYPTO_F_REVISION_1 is not negotiated the *flag* is ignored.

If VIRTIO_CRYPTO_F_REVISION_1 is negotiated but VIRTIO_CRYPTO_F_<SERVICE>_STATELESS_MODE is not negotiated, then the device SHOULD reject <SERVICE> requests if VIRTIO_CRYPTO_FLAG_SESSION_MODE is not set (in *flag*).

The dataq request is composed of four parts:

```
struct virtio_crypto_op_data_req {
    /* Device read only portion */

    struct virtio_crypto_op_header header;

#define VIRTIO_CRYPTO_DATAQ_OP_SPEC_HDR_LEGACY 48
    /* fixed length fields, opcode specific */
    u8 op_flf[flf_len];

    /* Device read && write portion */
    /* variable length fields, opcode specific */
    u8 op_vlf[vlf_len];

    /* Device write only portion */
    struct virtio_crypto_inhdr inhdr;
};
```

*header* is a general header (see above).

*op_flf* is the opcode (in *header*) specific header.

*flf_len* depends on the VIRTIO_CRYPTO_F_REVISION_1 feature bit (see below).

*op_vlf* is the opcode (in *header*) specific parameters.

*vlf_len* is the size of the specific structure used.

- If the the opcode (in *header*) is VIRTIO_CRYPTO_CIPHER_ENCRYPT or VIRTIO_CRYPTO_CIPHER_-DECRYPT then:
  - If VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE is negotiated, *op_flf* is struct virtio_crypto_-sym_data_flf_stateless, and *op_vlf* is struct virtio_crypto_sym_data_vlf_stateless.
  - If VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE is NOT negotiated, *op_flf* is struct virtio_-crypto_sym_data_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_-sym_data_flf is padded to 48 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_sym_-data_vlf.
- If the the opcode (in *header*) is VIRTIO_CRYPTO_HASH:
  - If VIRTIO_CRYPTO_F_HASH_STATELESS_MODE is negotiated, *op_flf* is struct virtio_crypto_-hash_data_flf_stateless, and *op_vlf* is struct virtio_crypto_hash_data_vlf_stateless.
  - If VIRTIO_CRYPTO_F_HASH_STATELESS_MODE is NOT negotiated, *op_flf* is struct virtio_-crypto_hash_data_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_-hash_data_flf is padded to 48 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_hash_-data_vlf.
- If the the opcode (in *header*) is VIRTIO_CRYPTO_MAC:
  - If VIRTIO_CRYPTO_F_MAC_STATELESS_MODE is negotiated, *op_flf* is struct virtio_crypto_-mac_data_flf_stateless, and *op_vlf* is struct virtio_crypto_mac_data_vlf_stateless.
  - If VIRTIO_CRYPTO_F_MAC_STATELESS_MODE is NOT negotiated, *op_flf* is struct virtio_crypto_-mac_data_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_mac_-data_flf is padded to 48 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_mac_data_vlf.
- If the the opcode (in *header*) is VIRTIO_CRYPTO_AEAD_ENCRYPT or VIRTIO_CRYPTO_AEAD_-DECRYPT then:
  - If VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE is negotiated, *op_flf* is struct virtio_crypto_-aead_data_flf_stateless, and *op_vlf* is struct virtio_crypto_aead_data_vlf_stateless.
  - If VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE is NOT negotiated, *op_flf* is struct virtio_-crypto_aead_data_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_-aead_data_flf is padded to 48 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_aead_-data_vlf.

*inhdr* is a unified input header that used to return the status of the operations, is defined as follows:

```
struct virtio_crypto_inhdr {
    u8 status;
};
```

#### 5.9.7.4 HASH Service Operation

Session mode HASH service requests are as follows:

```
struct virtio_crypto_hash_data_flf {
    /* length of source data */
    le32 src_data_len;
    /* hash result length */
    le32 hash_result_len;
};

struct virtio_crypto_hash_data_vlf {
    /* Device read only portion */
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
```

```
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

Each data request uses the virtio_crypto_hash_data_flf structure and the virtio_crypto_hash_data_vlf structure to store information used to run the HASH operations.

*src_data* is the source data that will be processed. *src_data_len* is the length of source data. *hash_result* is the result data and *hash_result_len* is the length of it.

Stateless mode HASH service requests are as follows:

```
struct virtio_crypto_hash_data_flf_stateless {
    struct {
        /* See VIRTIO_CRYPTO_HASH_* above */
        le32 algo;
    } sess_para;

    /* length of source data */
    le32 src_data_len;
    /* hash result length */
    le32 hash_result_len;
    le32 reserved;
};
struct virtio_crypto_hash_data_vlf_stateless {
    /* Device read only portion */
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

#### 5.9.7.4.1 Driver Requirements: HASH Service Operation

- If the driver uses the session mode, then the driver MUST set *session_id* in struct virtio_crypto_op_-header to a valid value assigned by the device when the session was created.
- If the VIRTIO_CRYPTO_F_HASH_STATELESS_MODE feature bit is negotiated, 1) if the driver uses the stateless mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to ZERO and MUST set the fields in struct virtio_crypto_hash_data_flf_stateless.sess_para, 2) if the driver uses the session mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to VIRTIO_-CRYPTO_FLAG_SESSION_MODE.
- The driver MUST set *opcode* in struct virtio_crypto_op_header to VIRTIO_CRYPTO_HASH.

#### 5.9.7.4.2 Device Requirements: HASH Service Operation

- The device MUST use the corresponding structure according to the *opcode* in the data general header.
- If the VIRTIO_CRYPTO_F_HASH_STATELESS_MODE feature bit is negotiated, the device MUST parse *flag* field in struct virtio_crypto_op_header in order to decide which mode the driver uses.
- The device MUST copy the results of HASH operations in the hash_result[] if HASH operations success.
- The device MUST set *status* in struct virtio_crypto_inhdr to one of the following values of enum VIR-TIO_CRYPTO_STATUS:
  - VIRTIO_CRYPTO_OK if the operation success.
  - VIRTIO_CRYPTO_NOTSUPP if the requested algorithm or operation is unsupported.
  - VIRTIO_CRYPTO_INVSESS if the session ID invalid when in session mode.
  - VIRTIO_CRYPTO_ERR if any failure not mentioned above occurs.

#### 5.9.7.5 MAC Service Operation

Session mode MAC service requests are as follows:

```
struct virtio_crypto_mac_data_flf {
    struct virtio_crypto_hash_data_flf hdr;
};

struct virtio_crypto_mac_data_vlf {
    /* Device read only portion */
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

Each request uses the virtio_crypto_mac_data_flf structure and the virtio_crypto_mac_data_vlf structure to store information used to run the MAC operations.

*src_data* is the source data that will be processed. *src_data_len* is the length of source data. *hash_result* is the result data and *hash_result_len* is the length of it.

Stateless mode MAC service requests are as follows:

```
struct virtio_crypto_mac_data_flf_stateless {
    struct {
        /* See VIRTIO_CRYPTO_MAC_* above */
        le32 algo;
        /* length of authenticated key */
        le32 auth_key_len;
    } sess_para;

    /* length of source data */
    le32 src_data_len;
    /* hash result length */
    le32 hash_result_len;
};

struct virtio_crypto_mac_data_vlf_stateless {
    /* Device read only portion */
    /* The authenticated key */
    u8 auth_key[auth_key_len];
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

*auth_key* is the authenticated key that will be used during the process. *auth_key_len* is the length of the key.

##### 5.9.7.5.1 Driver Requirements: MAC Service Operation

- If the driver uses the session mode, then the driver MUST set *session_id* in struct virtio_crypto_op_-header to a valid value assigned by the device when the session was created.
- If the VIRTIO_CRYPTO_F_MAC_STATELESS_MODE feature bit is negotiated, 1) if the driver uses the stateless mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to ZERO and MUST set the fields in struct virtio_crypto_mac_data_flf_stateless.sess_para, 2) if the driver uses the session mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to VIRTIO_-CRYPTO_FLAG_SESSION_MODE.
- The driver MUST set *opcode* in struct virtio_crypto_op_header to VIRTIO_CRYPTO_MAC.

#### 5.9.7.5.2 Device Requirements: MAC Service Operation

- If the VIRTIO_CRYPTO_F_MAC_STATELESS_MODE feature bit is negotiated, the device MUST parse *flag* field in struct virtio_crypto_op_header in order to decide which mode the driver uses.
- The device MUST copy the results of MAC operations in the hash_result[] if HASH operations success.
- The device MUST set *status* in struct virtio_crypto_inhdr to one of the following values of enum VIRTIO_CRYPTO_STATUS:
  - VIRTIO_CRYPTO_OK if the operation success.
  - VIRTIO_CRYPTO_NOTSUPP if the requested algorithm or operation is unsupported.
  - VIRTIO_CRYPTO_INVSESS if the session ID invalid when in session mode.
  - VIRTIO_CRYPTO_ERR if any failure not mentioned above occurs.

### 5.9.7.6 Symmetric algorithms Operation

Session mode CIPHER service requests are as follows:

```
struct virtio_crypto_cipher_data_flf {
    /*
     * Byte Length of valid IV/Counter data pointed to by the below iv data.
     *
     * For block ciphers in CBC or F8 mode, or for Kasumi in F8 mode, or for
     *   SNOW3G in UEA2 mode, this is the length of the IV (which
     *   must be the same as the block length of the cipher).
     * For block ciphers in CTR mode, this is the length of the counter
     *   (which must be the same as the block length of the cipher).
     */
    le32 iv_len;
    /* length of source data */
    le32 src_data_len;
    /* length of destination data */
    le32 dst_data_len;
    le32 padding;
};

struct virtio_crypto_cipher_data_vlf {
    /* Device read only portion */

    /*
     * Initialization Vector or Counter data.
     *
     * For block ciphers in CBC or F8 mode, or for Kasumi in F8 mode, or for
     *   SNOW3G in UEA2 mode, this is the Initialization Vector (IV)
     *   value.
     * For block ciphers in CTR mode, this is the counter.
     * For AES-XTS, this is the 128bit tweak, i, from IEEE Std 1619-2007.
     *
     * The IV/Counter will be updated after every partial cryptographic
     * operation.
     */
    u8 iv[iv_len];
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
    /* Destination data */
    u8 dst_data[dst_data_len];
};
```

Session mode requests of algorithm chaining are as follows:

```
struct virtio_crypto_alg_chain_data_flf {
    le32 iv_len;
    /* Length of source data */
    le32 src_data_len;
    /* Length of destination data */
    le32 dst_data_len;
    /* Starting point for cipher processing in source data */
```

```
    le32 cipher_start_src_offset;
    /* Length of the source data that the cipher will be computed on */
    le32 len_to_cipher;
    /* Starting point for hash processing in source data */
    le32 hash_start_src_offset;
    /* Length of the source data that the hash will be computed on */
    le32 len_to_hash;
    /* Length of the additional auth data */
    le32 aad_len;
    /* Length of the hash result */
    le32 hash_result_len;
    le32 reserved;
};

struct virtio_crypto_alg_chain_data_vlf {
    /* Device read only portion */

    /* Initialization Vector or Counter data */
    u8 iv[iv_len];
    /* Source data */
    u8 src_data[src_data_len];
    /* Additional authenticated data if exists */
    u8 aad[aad_len];

    /* Device write only portion */

    /* Destination data */
    u8 dst_data[dst_data_len];
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

Session mode requests of symmetric algorithm are as follows:

```
struct virtio_crypto_sym_data_flf {
    /* Device read only portion */

#define VIRTIO_CRYPTO_SYM_DATA_REQ_HDR_SIZE    40
    u8 op_type_flf[VIRTIO_CRYPTO_SYM_DATA_REQ_HDR_SIZE];

    /* See above VIRTIO_CRYPTO_SYM_OP_* */
    le32 op_type;
    le32 padding;
};

struct virtio_crypto_sym_data_vlf {
    u8 op_type_vlf[sym_para_len];
};
```

Each request uses the virtio_crypto_sym_data_flf structure and the virtio_crypto_sym_data_flf structure to store information used to run the CIPHER operations.

*op_type_flf* is the *op_type* specific header, it MUST starts with or be one of the following structures:

- struct virtio_crypto_cipher_data_flf
- struct virtio_crypto_alg_chain_data_flf

The length of *op_type_flf* is fixed to 40 bytes, the data of unused part (if has) will be ingored.

*op_type_vlf* is the *op_type* specific parameters, it MUST starts with or be one of the following structures:

- struct virtio_crypto_cipher_data_vlf
- struct virtio_crypto_alg_chain_data_vlf

*sym_para_len* is the size of the specific structure used.

Stateless mode CIPHER service requests are as follows:

```
struct virtio_crypto_cipher_data_flf_stateless {
    struct {
        /* See VIRTIO_CRYPTO_CIPHER* above */
```

  

```
        le32 algo;
        /* length of key */
        le32 key_len;

        /* See VIRTIO_CRYPTO_OP_* above */
        le32 op;
    } sess_para;

    /*
     * Byte Length of valid IV/Counter data pointed to by the below iv data.
     */
    le32 iv_len;
    /* length of source data */
    le32 src_data_len;
    /* length of destination data */
    le32 dst_data_len;
};

struct virtio_crypto_cipher_data_vlf_stateless {
    /* Device read only portion */

    /* The cipher key */
    u8 cipher_key[key_len];

    /* Initialization Vector or Counter data. */
    u8 iv[iv_len];
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */
    /* Destination data */
    u8 dst_data[dst_data_len];
};
```

Stateless mode requests of algorithm chaining are as follows:

```
struct virtio_crypto_alg_chain_data_flf_stateless {
    struct {
        /* See VIRTIO_CRYPTO_SYM_ALG_CHAIN_ORDER_* above */
        le32 alg_chain_order;
        /* length of the additional authenticated data in bytes */
        le32 aad_len;

        struct {
            /* See VIRTIO_CRYPTO_CIPHER* above */
            le32 algo;
            /* length of key */
            le32 key_len;
            /* See VIRTIO_CRYPTO_OP_* above */
            le32 op;
        } cipher;

        struct {
            /* See VIRTIO_CRYPTO_HASH_* or VIRTIO_CRYPTO_MAC_* above */
            le32 algo;
            /* length of authenticated key */
            le32 auth_key_len;
            /* See VIRTIO_CRYPTO_SYM_HASH_MODE_* above */
            le32 hash_mode;
        } hash;
    } sess_para;

    le32 iv_len;
    /* Length of source data */
    le32 src_data_len;
    /* Length of destination data */
    le32 dst_data_len;
    /* Starting point for cipher processing in source data */
    le32 cipher_start_src_offset;
    /* Length of the source data that the cipher will be computed on */
    le32 len_to_cipher;
```

```
    /* Starting point for hash processing in source data */
    le32 hash_start_src_offset;
    /* Length of the source data that the hash will be computed on */
    le32 len_to_hash;
    /* Length of the additional auth data */
    le32 aad_len;
    /* Length of the hash result */
    le32 hash_result_len;
    le32 reserved;
};

struct virtio_crypto_alg_chain_data_vlf_stateless {
    /* Device read only portion */

    /* The cipher key */
    u8 cipher_key[key_len];
    /* The auth key */
    u8 auth_key[auth_key_len];
    /* Initialization Vector or Counter data */
    u8 iv[iv_len];
    /* Additional authenticated data if exists */
    u8 aad[aad_len];
    /* Source data */
    u8 src_data[src_data_len];

    /* Device write only portion */

    /* Destination data */
    u8 dst_data[dst_data_len];
    /* Hash result data */
    u8 hash_result[hash_result_len];
};
```

Stateless mode requests of symmetric algorithm are as follows:

```
struct virtio_crypto_sym_data_flf_stateless {
    /* Device read only portion */
#define VIRTIO_CRYPTO_SYM_DATE_REQ_HDR_STATELESS_SIZE    72
    u8 op_type_flf[VIRTIO_CRYPTO_SYM_DATE_REQ_HDR_STATELESS_SIZE];

    /* Device write only portion */
    /* See above VIRTIO_CRYPTO_SYM_OP_* */
    le32 op_type;
};

struct virtio_crypto_sym_data_vlf_stateless {
    u8 op_type_vlf[sym_para_len];
};
```

*op_type_flf* is the *op_type* specific header, it MUST starts with or be one of the following structures:

- struct virtio_crypto_cipher_data_flf_stateless
- struct virtio_crypto_alg_chain_data_flf_stateless

The length of *op_type_flf* is fixed to 72 bytes, the data of unused part (if has) will be ingored.

*op_type_vlf* is the *op_type* specific parameters, it MUST starts with or be one of the following structures:

- struct virtio_crypto_cipher_data_vlf_stateless
- struct virtio_crypto_alg_chain_data_vlf_stateless

*sym_para_len* is the size of the specific structure used.

#### 5.9.7.6.1 Driver Requirements: Symmetric algorithms Operation

- If the driver uses the session mode, then the driver MUST set *session_id* in struct virtio_crypto_op_-header to a valid value assigned by the device when the session was created.

- If the VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE feature bit is negotiated, 1) if the driver uses the stateless mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to ZERO and MUST set the fields in struct virtio_crypto_cipher_data_flf_stateless.sess_para or struct virtio_crypto_alg_chain_data_flf_stateless.sess_para, 2) if the driver uses the session mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to VIRTIO_CRYPTO_FLAG_SESSION_MODE.
- The driver MUST set the *opcode* field in struct virtio_crypto_op_header to VIRTIO_CRYPTO_CIPHER_ENCRYPT or VIRTIO_CRYPTO_CIPHER_DECRYPT.
- The driver MUST specify the fields of struct virtio_crypto_cipher_data_flf in struct virtio_crypto_sym_data_flf and struct virtio_crypto_cipher_data_vlf in struct virtio_crypto_sym_data_vlf if the request is based on VIRTIO_CRYPTO_SYM_OP_CIPHER.
- The driver MUST specify the fields of struct virtio_crypto_alg_chain_data_flf in struct virtio_crypto_sym_data_flf and struct virtio_crypto_alg_chain_data_vlf in struct virtio_crypto_sym_data_vlf if the request is of the VIRTIO_CRYPTO_SYM_OP_ALGORITHM_CHAINING type.

#### 5.9.7.6.2 Device Requirements: Symmetric algorithms Operation

- If the VIRTIO_CRYPTO_F_CIPHER_STATELESS_MODE feature bit is negotiated, the device MUST parse *flag* field in struct virtio_crypto_op_header in order to decide which mode the driver uses.
- The device MUST parse the virtio_crypto_sym_data_req based on the *opcode* field in general header.
- The device MUST parse the fields of struct virtio_crypto_cipher_data_flf in struct virtio_crypto_sym_data_flf and struct virtio_crypto_cipher_data_vlf in struct virtio_crypto_sym_data_vlf if the request is based on VIRTIO_CRYPTO_SYM_OP_CIPHER.
- The device MUST parse the fields of struct virtio_crypto_alg_chain_data_flf in struct virtio_crypto_sym_data_flf and struct virtio_crypto_alg_chain_data_vlf in struct virtio_crypto_sym_data_vlf if the request is of the VIRTIO_CRYPTO_SYM_OP_ALGORITHM_CHAINING type.
- The device MUST copy the result of cryptographic operation in the dst_data[] in both plain CIPHER mode and algorithms chain mode.
- The device MUST check the *para.add_len* is bigger than 0 before parse the additional authenticated data in plain algorithms chain mode.
- The device MUST copy the result of HASH/MAC operation in the hash_result[] is of the VIRTIO_CRYPTO_SYM_OP_ALGORITHM_CHAINING type.
- The device MUST set the *status* field in struct virtio_crypto_inhdr to one of the following values of enum VIRTIO_CRYPTO_STATUS:
  - VIRTIO_CRYPTO_OK if the operation success.
  - VIRTIO_CRYPTO_NOTSUPP if the requested algorithm or operation is unsupported.
  - VIRTIO_CRYPTO_INVSESS if the session ID is invalid in session mode.
  - VIRTIO_CRYPTO_ERR if failure not mentioned above occurs.

### 5.9.7.7 AEAD Service Operation

Session mode requests of symmetric algorithm are as follows:

```
struct virtio_crypto_aead_data_flf {
    /*
     * Byte Length of valid IV data.
     *
     * For GCM mode, this is either 12 (for 96-bit IVs) or 16, in which
     *   case iv points to J0.
     * For CCM mode, this is the length of the nonce, which can be in the
     *   range 7 to 13 inclusive.
     */
    le32 iv_len;
    /* length of additional auth data */
    le32 aad_len;
    /* length of source data */
    le32 src_data_len;
    /* length of dst data, this should be at least src_data_len + tag_len */
    le32 dst_data_len;
```

```
    /* Authentication tag length */
    le32 tag_len;
    le32 reserved;
};

struct virtio_crypto_aead_data_vlf {
    /* Device read only portion */

    /*
     * Initialization Vector data.
     *
     * For GCM mode, this is either the IV (if the length is 96 bits) or J0
     *   (for other sizes), where J0 is as defined by NIST SP800-38D.
     *   Regardless of the IV length, a full 16 bytes needs to be allocated.
     * For CCM mode, the first byte is reserved, and the nonce should be
     *   written starting at &iv[1] (to allow space for the implementation
     *   to write in the flags in the first byte).  Note that a full 16 bytes
     *   should be allocated, even though the iv_len field will have
     *   a value less than this.
     *
     * The IV will be updated after every partial cryptographic operation.
     */
    u8 iv[iv_len];
    /* Source data */
    u8 src_data[src_data_len];
    /* Additional authenticated data if exists */
    u8 aad[aad_len];

    /* Device write only portion */
    /* Pointer to output data */
    u8 dst_data[dst_data_len];
};
```

Each request uses the virtio_crypto_aead_data_flf structure and the virtio_crypto_aead_data_flf structure to store information used to run the AEAD operations.

Stateless mode AEAD service requests are as follows:

```
struct virtio_crypto_aead_data_flf_stateless {
    struct {
        /* See VIRTIO_CRYPTO_AEAD_* above */
        le32 algo;
        /* length of key */
        le32 key_len;
        /* encrypt or decrypt, See above VIRTIO_CRYPTO_OP_* */
        le32 op;
    } sess_para;

    /* Byte Length of valid IV data. */
    le32 iv_len;
    /* Authentication tag length */
    le32 tag_len;
    /* length of additional auth data */
    le32 aad_len;
    /* length of source data */
    le32 src_data_len;
    /* length of dst data, this should be at least src_data_len + tag_len */
    le32 dst_data_len;
};

struct virtio_crypto_aead_data_vlf_stateless {
    /* Device read only portion */

    /* The cipher key */
    u8 key[key_len];
    /* Initialization Vector data. */
    u8 iv[iv_len];
    /* Source data */
    u8 src_data[src_data_len];
    /* Additional authenticated data if exists */
    u8 aad[aad_len];
```

```
    /* Device write only portion */
    /* Pointer to output data */
    u8 dst_data[dst_data_len];
};
```

#### 5.9.7.7.1 Driver Requirements: AEAD Service Operation

- If the driver uses the session mode, then the driver MUST set *session_id* in struct virtio_crypto_op_-header to a valid value assigned by the device when the session was created.
- If the VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE feature bit is negotiated, 1) if the driver uses the stateless mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to ZERO and MUST set the fields in struct virtio_crypto_aead_data_flf_stateless.sess_para, 2) if the driver uses the session mode, then the driver MUST set the *flag* field in struct virtio_crypto_op_header to VIRTIO_-CRYPTO_FLAG_SESSION_MODE.
- The driver MUST set the *opcode* field in struct virtio_crypto_op_header to VIRTIO_CRYPTO_AEAD_-ENCRYPT or VIRTIO_CRYPTO_AEAD_DECRYPT.

#### 5.9.7.7.2 Device Requirements: AEAD Service Operation

- If the VIRTIO_CRYPTO_F_AEAD_STATELESS_MODE feature bit is negotiated, the device MUST parse the virtio_crypto_aead_data_vlf_stateless based on the *opcode* field in general header.
- The device MUST copy the result of cryptographic operation in the dst_data[].
- The device MUST copy the authentication tag in the dst_data[] offset the cipher result.
- The device MUST set the *status* field in struct virtio_crypto_inhdr to one of the following values of enum VIRTIO_CRYPTO_STATUS:
- When the *opcode* field is VIRTIO_CRYPTO_AEAD_DECRYPT, the device MUST verify and return the verification result to the driver.
  - VIRTIO_CRYPTO_OK if the operation success.
  - VIRTIO_CRYPTO_NOTSUPP if the requested algorithm or operation is unsupported.
  - VIRTIO_CRYPTO_BADMSG if the verification result is incorrect.
  - VIRTIO_CRYPTO_INVSESS if the session ID invalid when in session mode.
  - VIRTIO_CRYPTO_ERR if any failure not mentioned above occurs.

## 5.10 Socket Device

The virtio socket device is a zero-configuration socket communications device. It facilitates data transfer between the guest and device without using the Ethernet or IP protocols.

### 5.10.1 Device ID

19

### 5.10.2 Virtqueues

**0** rx

**1** tx

**2** event

### 5.10.3 Feature bits

There are currently no feature bits defined for this device.

### 5.10.4 Device configuration layout

```
struct virtio_vsock_config {
  le64 guest_cid;
};
```

The *guest_cid* field contains the guest's context ID, which uniquely identifies the device for its lifetime. The upper 32 bits of the CID are reserved and zeroed.

The following CIDs are reserved and cannot be used as the guest's context ID:

| CID | Notes |
|---|---|
| 0 | Reserved |
| 1 | Reserved |
| 2 | Well-known CID for the host |
| 0xffffffff | Reserved |
| 0xffffffffffffffff | Reserved |

### 5.10.5 Device Initialization

1. The guest's cid is read from *guest_cid*.
2. Buffers are added to the event virtqueue to receive events from the device.
3. Buffers are added to the rx virtqueue to start receiving packets.

### 5.10.6 Device Operation

Packets transmitted or received contain a header before the payload:

```
struct virtio_vsock_hdr {
  le64 src_cid;
  le64 dst_cid;
  le32 src_port;
  le32 dst_port;
  le32 len;
  le16 type;
  le16 op;
  le32 flags;
  le32 buf_alloc;
  le32 fwd_cnt;
};
```

The upper 32 bits of src_cid and dst_cid are reserved and zeroed.

Most packets simply transfer data but control packets are also used for connection and buffer space management. *op* is one of the following operation constants:

```
enum {
  VIRTIO_VSOCK_OP_INVALID = 0,

  /* Connect operations */
  VIRTIO_VSOCK_OP_REQUEST = 1,
  VIRTIO_VSOCK_OP_RESPONSE = 2,
  VIRTIO_VSOCK_OP_RST = 3,
  VIRTIO_VSOCK_OP_SHUTDOWN = 4,
```

```
  /* To send payload */
  VIRTIO_VSOCK_OP_RW = 5,

  /* Tell the peer our credit info */
  VIRTIO_VSOCK_OP_CREDIT_UPDATE = 6,
  /* Request the peer to send the credit info to us */
  VIRTIO_VSOCK_OP_CREDIT_REQUEST = 7,
};
```

#### 5.10.6.1 Virtqueue Flow Control

The tx virtqueue carries packets initiated by applications and replies to received packets. The rx virtqueue carries packets initiated by the device and replies to previously transmitted packets.

If both rx and tx virtqueues are filled by the driver and device at the same time then it appears that a deadlock is reached. The driver has no free tx descriptors to send replies. The device has no free rx descriptors to send replies either. Therefore neither device nor driver can process virtqueues since that may involve sending new replies.

This is solved using additional resources outside the virtqueue to hold packets. With additional resources, it becomes possible to process incoming packets even when outgoing packets cannot be sent.

Eventually even the additional resources will be exhausted and further processing is not possible until the other side processes the virtqueue that it has neglected. This stop to processing prevents one side from causing unbounded resource consumption in the other side.

##### 5.10.6.1.1 Driver Requirements: Device Operation: Virtqueue Flow Control

The rx virtqueue MUST be processed even when the tx virtqueue is full so long as there are additional resources available to hold packets outside the tx virtqueue.

##### 5.10.6.1.2 Device Requirements: Device Operation: Virtqueue Flow Control

The tx virtqueue MUST be processed even when the rx virtqueue is full so long as there are additional resources available to hold packets outside the rx virtqueue.

#### 5.10.6.2 Addressing

Flows are identified by a (source, destination) address tuple. An address consists of a (cid, port number) tuple. The header fields used for this are *src_cid*, *src_port*, *dst_cid*, and *dst_port*.

Currently only stream sockets are supported. *type* is 1 for stream socket types.

Stream sockets provide in-order, guaranteed, connection-oriented delivery without message boundaries.

#### 5.10.6.3 Buffer Space Management

*buf_alloc* and *fwd_cnt* are used for buffer space management of stream sockets. The guest and the device publish how much buffer space is available per socket. Only payload bytes are counted and header bytes are not included. This facilitates flow control so data is never dropped.

*buf_alloc* is the total receive buffer space, in bytes, for this socket. This includes both free and in-use buffers. *fwd_cnt* is the free-running bytes received counter. The sender calculates the amount of free receive buffer space as follows:

```
/* tx_cnt is the sender's free-running bytes transmitted counter */
u32 peer_free = peer_buf_alloc - (tx_cnt - peer_fwd_cnt);
```

If there is insufficient buffer space, the sender waits until virtqueue buffers are returned and checks *buf_alloc* and *fwd_cnt* again. Sending the VIRTIO_VSOCK_OP_CREDIT_REQUEST packet queries how much buffer space is available. The reply to this query is a VIRTIO_VSOCK_OP_CREDIT_UPDATE packet. It is also valid to send a VIRTIO_VSOCK_OP_CREDIT_UPDATE packet without previously receiving a VIRTIO_VSOCK_OP_CREDIT_REQUEST packet. This allows communicating updates any time a change in buffer space occurs.

#### 5.10.6.3.1 Driver Requirements: Device Operation: Buffer Space Management

VIRTIO_VSOCK_OP_RW data packets MUST only be transmitted when the peer has sufficient free buffer space for the payload.

All packets associated with a stream flow MUST contain valid information in *buf_alloc* and *fwd_cnt* fields.

#### 5.10.6.3.2 Device Requirements: Device Operation: Buffer Space Management

VIRTIO_VSOCK_OP_RW data packets MUST only be transmitted when the peer has sufficient free buffer space for the payload.

All packets associated with a stream flow MUST contain valid information in *buf_alloc* and *fwd_cnt* fields.

### 5.10.6.4 Receive and Transmit

The driver queues outgoing packets on the tx virtqueue and incoming packet receive buffers on the rx virtqueue. Packets are of the following form:

```
struct virtio_vsock_packet {
    struct virtio_vsock_hdr hdr;
    u8 data[];
};
```

Virtqueue buffers for outgoing packets are read-only. Virtqueue buffers for incoming packets are write-only.

#### 5.10.6.4.1 Driver Requirements: Device Operation: Receive and Transmit

The *guest_cid* configuration field MUST be used as the source CID when sending outgoing packets.

A VIRTIO_VSOCK_OP_RST reply MUST be sent if a packet is received with an unknown *type* value.

#### 5.10.6.4.2 Device Requirements: Device Operation: Receive and Transmit

The *guest_cid* configuration field MUST NOT contain a reserved CID as listed in 5.10.4.

A VIRTIO_VSOCK_OP_RST reply MUST be sent if a packet is received with an unknown *type* value.

### 5.10.6.5 Stream Sockets

Connections are established by sending a VIRTIO_VSOCK_OP_REQUEST packet. If a listening socket exists on the destination a VIRTIO_VSOCK_OP_RESPONSE reply is sent and the connection is established. A VIRTIO_VSOCK_OP_RST reply is sent if a listening socket does not exist on the destination or the destination has insufficient resources to establish the connection.

When a connected socket receives VIRTIO_VSOCK_OP_SHUTDOWN the header *flags* field bit 0 indicates that the peer will not receive any more data and bit 1 indicates that the peer will not send any more data. These hints are permanent once sent and successive packets with bits clear do not reset them.

The VIRTIO_VSOCK_OP_RST packet aborts the connection process or forcibly disconnects a connected socket.

Clean disconnect is achieved by one or more VIRTIO_VSOCK_OP_SHUTDOWN packets that indicate no more data will be sent and received, followed by a VIRTIO_VSOCK_OP_RST response from the peer. If no VIRTIO_VSOCK_OP_RST response is received within an implementation-specific amount of time, a VIRTIO_VSOCK_OP_RST packet is sent to forcibly disconnect the socket.

The clean disconnect process ensures that neither peer reuses the (source, destination) address tuple for a new connection while the other peer is still processing the old connection.

#### 5.10.6.6 Device Events

Certain events are communicated by the device to the driver using the event virtqueue.

The event buffer is as follows:

```
enum virtio_vsock_event_id {
        VIRTIO_VSOCK_EVENT_TRANSPORT_RESET = 0,
};

struct virtio_vsock_event {
        le32 id;
};
```

The VIRTIO_VSOCK_EVENT_TRANSPORT_RESET event indicates that communication has been interrupted. This usually occurs if the guest has been physically migrated. The driver shuts down established connections and the *guest_cid* configuration field is fetched again. Existing listen sockets remain but their CID is updated to reflect the current *guest_cid*.

##### 5.10.6.6.1 Driver Requirements: Device Operation: Device Events

Event virtqueue buffers SHOULD be replenished quickly so that no events are missed.

The *guest_cid* configuration field MUST be fetched to determine the current CID when a VIRTIO_VSOCK_-EVENT_TRANSPORT_RESET event is received.

Existing connections MUST be shut down when a VIRTIO_VSOCK_EVENT_TRANSPORT_RESET event is received.

Listen connections MUST remain operational with the current CID when a VIRTIO_VSOCK_EVENT_TRANS-PORT_RESET event is received.

## 5.11 User Device

Note: This depends on the upcoming shared-mem type of virtio.

virtio-user is an interface for defining and operating virtual devices with high performance. It is intended that virtio-user serve a need for defining userspace drivers for virtual machines, but it can be used for kernel drivers as well, and there are several benefits to this approach that can potentially make it more flexible and performant than commonly suggested alternatives.

virtio-user is configured at virtio-user device realization time. The host enumerates a set of available devices for virtio-user and the guest is able to use the available ones according to a privately-defined protocol that uses a combination of virtqueues and shared memory.

virtio-user has three main virtqueue types: config, ping, and event. The config virtqueue is used to enumerate devices, create instances, and allocate shared memory. The ping virtqueue is optionally used as a doorbell to notify the host to process data. The event virtqueue is optinally used to wait for the host to complete operations from the guest.

On the host, callbacks specific to the enumerated device are issued on enumeration, instance creation, shared memory operations, and ping. These device implementations are stored in shared library plugins separate from the host hypervisor. The host hypervisor implements a minimal set of operations to allow the dispatch to happen and to send back event virtqueue messages.

The main benefit of virtio-user is to decouple definition of new drivers and devices from the underlying transport protocol and from the host hypervisor implementation. Virtio-user then serves as a platform for ”userspace” drivers for virtual machines; ”userspace” in the literal sense of allowing guest drivers to be userspace-defined, decoupled from the guest kernel, ”userspace” in the sense of device implementations being defined away from the ”kernel” of the host hypervisor code.

The second benefit of virtio-user is high performance via shared memory (Note: This depends on the upcoming shared-mem type of virtio). Each driver/device created from userspace or the guest kernel is allowed to create or share regions of shared memory. Sharing can be done with other virtio-user devices only, though it may be possible to share with other virtio devices if that is beneficial.

Another benefit derives from the separation between the driver definition from the transport protocol. The implementation of all such user-level drivers is captured by a set of primitive operations in the guest and shared library function pointers in the host. Because of this, virtio-user itself will have a very small implementation footprint, allowing it to be readily used with a wide variety of guest OSes and host VMMs, while sharing the same driver/device functionality defined in the guest and defined in a shared library on the host. This facilitates using any particular virtual device in many different guest OSes and host VMMs.

Finally, this has the benefit of being a general standardization path for existing non-standard devices to use virtio; if a new device type is introduced that can be used with virtio, it can be implemented on top of virtio-user first and work immediately all existing guest OSes / host hypervisors supporting virtio-user. virtio-user can be used to as a staging area for potential new virtio device types, and moved to new virtio device types as appropriate. Currently, virtio-vsock is often suggested as a generic pipe, but from the standardization point of view, doing so causes de-facto non-portability; there is no standard way to enumerate how such generic pipes are used.

Note that virtio-serial/virtio-vsock is not considered because they do not standardize the set of devices that operate on top of them, but in practice, are often used for fully general devices. Spec-wise, this is not a great situation because we would still have potentially non portable device implementations where there is no standard mechanism to determine whether or not things are portable. virtio-user provides a device enumeration mechanism to better control this.

In addition, for performance considerations in applications such as graphics and media, virtio-serial/virtio-vsock have the overhead of sending actual traffic through the virtqueue, while an approach based on shared memory can result in having fewer copies and virtqueue messages. virtio-serial is also limited in being specialized for console forwarding and having a cap on the number of clients. virtio-vsock is also not optimal in its choice of sockets API for transport; shared memory cannot be used, arbitrary strings can be passed without an designation of the device/driver being run de-facto, and the guest must have additional machinery to handle socket APIs. In addition, on the host, sockets are only dependable on Linux, with less predictable behavior from Windows/macOS regarding Unix sockets. Waiting for socket traffic on the host also requires a poll() loop, which is suboptimal for latency. With virtio-user, only the bare set of standard driver calls (open/close/ioctl/mmap/read) is needed, and RAM is a more universal transport abstraction. We also explicitly spec out callbacks on host that are triggered by virtqueue messages, which results in lower latency and makes it easy to dispatch to a particular device implementation without polling.

### 5.11.1 Device ID

21

### 5.11.2 Virtqueues

**0** config tx

**1** config rx

**2** ping

**3** event

### 5.11.3 Feature bits

No feature bits, unless we go with this alternative:

An alternative is to specify the possible drivers/devices in the feature bits themselves. This ensures that there is a standard place where such devices are defined. However, changing the feature bits would require updates to the spec, driver, and hypervisor, which may not be as well suited to fast iteration, and has the undesirable property of coupling device changes to hypervisor changes.

#### 5.11.3.1 Feature bit requirements

No feature bit requirements, unless we go with device enumeration in feature bits.

### 5.11.4 Device configuration layout

```
struct virtio_user_config {
    le32 enumeration_space_id;
};
```

These serve to identify the virtio-user instance for purposes of compatibility. Userspace drivers/devices enumerated under the same *enumeration_space_id* that match are considered to be compatible. The guest may not write to *enumeration_space_id*. The host writes once to *enumeration_space_id* on initialization.

### 5.11.5 Device Initialization

The enumeration space id is read from the host into *virtio_user_config.enumeration_space_id*.

On device startup, the config virtqueue is used to enumerate a set of virtual devices available on the host. They are then registered to the guest in a way that is specific to the guest OS, such as misc_register for Linux.

Buffers are added to the config virtqueues to enumerate available userspace drivers, to create / destroy userspace device contexts, or to alloc / free / import / export shared memory.

Buffers are added to the ping virtqueue to notify the host of device specific operations or to notify the host that there is available shared memory to consume. This is like a doorbell with user-defined semantics.

Buffers are added to the event virtqueue from the device to the driver to notify the driver that an operation has completed.

### 5.11.6 Device Operation

#### 5.11.6.1 Config Virtqueue Messages

Operation always begins on the config virtqueue. Messages transmitted or received on the config virtqueue are of the following structure:

```
struct virtio_user_config_msg {
    le32 msg_type;
    le32 device_count;
    le32 vendor_ids[MAX_DEVICES];
    le32 device_ids[MAX_DEVICES];
    le32 versions[MAX_DEVICES];
    le64 instance_handle;
```

```
    le64 shm_id;
    le64 shm_offset;
    le64 shm_size;
    le32 shm_flags;
    le32 error;
}
```

*MAX_DEVICES* is defined as 32. *msg_type* can only be one of the following:

```
enum {
    VIRTIO_USER_CONFIG_OP_ENUMERATE_DEVICES;
    VIRTIO_USER_CONFIG_OP_CREATE_INSTANCE;
    VIRTIO_USER_CONFIG_OP_DESTROY_INSTANCE;
    VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_ALLOC;
    VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_FREE;
    VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_EXPORT;
    VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_IMPORT;
}
```

*error* can only be one of the following:

```
enum {
    VIRTIO_USER_ERROR_CONFIG_DEVICE_INITIALIZATION_FAILED;
    VIRTIO_USER_ERROR_CONFIG_INSTANCE_CREATION_FAILED;
    VIRTIO_USER_ERROR_CONFIG_SHARED_MEMORY_ALLOC_FAILED;
    VIRTIO_USER_ERROR_CONFIG_SHARED_MEMORY_EXPORT_FAILED;
    VIRTIO_USER_ERROR_CONFIG_SHARED_MEMORY_IMPORT_FAILED;
}
```

When the guest starts, a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_-ENUMERATE_DEVICES* is sent from the guest to the host on the config tx virtqueue. All other fields are ignored.

The guest then receives a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_-ENUMERATE_DEVICES*, with *device_count* populated with the number of available devices, the *vendor_ids* array populated with *device_count* vendor ids, the *device_ids* array populated with *device_count* device ids, and the *versions* array populated with *device_count* device versions.

The results can be obtained more than once, and the same results will always be received by the guest as long as there is no change to existing virtio userspace devices.

The guest now knows which devices are available, in addition to *enumeration_space_id*. It is guaranteed that host/guest setups with the same *enumeration space id*, *device_count*, *device_ids*, *vendor_ids*, and *versions* arrays (up to *device_count*) operate the same way as far as virtio-user devices. There are the following relaxations:

1. If a particular combination of IDs in *device_ids* / *vendor_ids* is missing, the guest can still continue with the existing set of devices.
2. If a particular combination of IDs in *device_ids* / *vendor_ids* mismatch in *versions*, the guest can still continue provided the version is deemed “compatible” by the guest, which is determined by the particular device implementation. Some devices are never compatible between versions while other devices are backward and/or forward compatible.

Next, instances, which are particular userspace contexts surrounding devices, are created.

Creating instances: The guest sends a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_-CONFIG_OP_CREATE_INSTANCE* on the config tx virtqueue. The first IDs and versions number in *vendor_ids*/*device_ids*/*versions* On the host, a new *instance_handle* is generated, and a device-specific instance creation function is run based on the vendor, device, and version.

If unsuccessful, *error* is set and sent back to the guest on the config rx virtqueue, and the *instance_handle* is discarded. If successfull, a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_-CREATE_INSTANCE* and *instance_handle* equal to the generated handle is sent on the config rx virtqueue.

  

The instance creation function is a callback function that is tied to a plugin associated with the vendor and device id in question:

(le64 instance_handle) -> bool

returning true if instance creation succeeded, and false if failed.

Let's call this *on_create_instance*.

Destroying instance: The guest sends a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_DESTROY_INSTANCE* on the config tx virtqueue. The only field that needs to be populated is *instance_handle*. On the host, a device-specific instance destruction function is run:

(instance_handle) -> void

Let's call this *on_destroy_instance*.

Also, all *shm_id*'s have their memory freed by instance destruction only if the shared memory was not exported (detailed below).

Next, shared memory is set up to back device operation. This depends on the particular guest in question and what drivers/devices are being used. The shared memory configuration operations are as follows:

Allocating shared memory: The guest sends a *virtio_user_config_msg* with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_ALLOC* on the config tx virtqueue. *instance_handle* needs to be a valid instance handle generated by the host. *shm_size* must be set and greater than zero. A new shared memory region is created in the PCI address space (actual allocation is deferred). If any operation fails, a message on the config tx virtqueue with *msg_type* equal to *VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_ALLOC* and *error* equal to *VIRTIO_USER_ERROR_CONFIG_SHARED_MEMORY_ALLOC_FAILED* is sent. If all operations succeed, a new *shm_id* is generated along with *shm_offset* (offset into the PCI). and sent back on the config tx virtqueue.

Freeing shared memory objects works in a similar way, with setting *msg_type* equal to *VIRTIO_USER_CONFIG_OP_SHARED_MEMORY_FREE*. If the memory has been shared, it is refcounted based on how many instance have used it. When the refcount reaches 0, the host hypervisor will explicitly unmap that shared memory object from any existing host pointers.

To export a shared memory object, we need to have a valid *instance_handle* and an allocated shared memory object with a valid *shm_id*. The export operation itself for now is mostly administrative; it marks that allocated memory as available for sharing.

To import a shared memory object, we need to have a valid *instance_handle* and an allocated shared memory object with a valid *shm_id* that has been allocated and exported. A new *shm_id* is not generated; this is mostly administrative and marks that that *shm_id* can also be used from the second instance. This is for sharing memory, so *instance_handle* need not be the same as the *instance_handle* that allocated the shared memory.

This is similar to Vulkan *VK_KHR_external_memory*, except over raw PCI address space and *shm_id*'s.

For mapping and unmapping shared memory objects, we do not include explicit virtqueue methods for these, and instead rely on the guest kernel's memory mapping primitives.

Flow control: Only one config message is allowed to be in flight either to or from the host at any time. That is, the handshake tx/rx for device enumeration, instance creation, and shared memory operations are done in a globally visible single threaded manner. This is to make it easier to synchronize operations on shared memory and instance creation.

#### 5.11.6.2 Ping Virtqueue Messages

Once the instances have been created and configured with shared memory, we can already read/write memory, and for some device that may already be enough if they can operate lock-free and wait-free without needing notifications; we're done!

However, in order to prevent burning up CPU in most cases, most devices need some kind of mechanism to trigger activity on the device from the guest. This is captured via a new message struct, which is separate from the config struct because it's smaller and the common case is to send those messages. These messages are sent from the guest to host on the ping virtqueue.

```
struct virtio_user_ping {
    le64 instance_handle;
    le64 metadata;
    le64 shm_id;
    le64 shm_offset;
    le64 shm_size;
    le32 events;
}
```

*instance_handle* must be a valid instance handle. *shm_id* need not be a valid shm_id. If *shm_id* is a valid shm_id, it need not be allocated on the host yet.

On the device side, each ping results in calling a callback function of type:

(instance_handle, metadata, phys_addr, host_ptr, events) -> revents

Let us call this function *on_instance_ping*. It returns revents, which is optionally used in event virtqueue replies.

If *shm_id* is a valid shm_id, *phys_addr* is resolved given *shm_offset* by either the virtio-user driver or the host hypervisor.

If *shm_id* is a valid shm_id and there is a mapping set up for *phys_addr*, *host_ptr* refers to the corresponding memory view in the host address space. This allows coherent access to device memory from both the host and guest, given a few extra considerations. For example, for architectures that do not have store/load coherency (i.e., not x86) an explicit set of fence or synchronization instructions will also be run by virtio-user both before and after the call to *on_instance_ping*. An alternative is to leave this up to the implementor of the virtual device, but it is going to be such a common case to synchronize views of the same memory that it is probably a good idea to include synchronization out of the box.

Although, it may be common to block a guest thread until *on_instance_ping* completes on the device side. That is the purpose of the *events* field; the guest can populate it if it is desired to sync on the host completion. If *events* is not zero, then a reply is sent back to the guest via the event virtqueue after *on_instance_ping* completes, with the *revents* return value.

Flow control: Arbitrary levels of traffic can be sent on the ping virtqueue from multiple instances at the same time, but ordering within an instance is strictly preserved. Additional resources outside the virtqueue are used to hold incoming messages if the virtqueue itself fills up. This is similar to how virtio-vsock handles high traffic.

The semantics of ping messages themselves also are not restricted to guest to host only; the shared memory region named in the message can also be filled by the host and used as receive traffic by the guest. The ping message is then suitable for DMA operations in both directions, such as glTexImage2D and glReadPixels, and audio/video (de)compression (guest populates shared memory with (de)compressed buffers, sends ping message, host (de)compresses into the same memory region).

#### 5.11.6.3 Event Virtqueue Messages

Ping virtqueue messages are enough to cover all async device operations; that is, operations that do not require a round trip from the host. This is useful for most kinds of graphics API forwarding along with media codecs.

However, it can still be important to synchronize the guest on the completion of a device operation.

In the userspace driver, the interface can be similar to Linux uio interrupts for example; a blocking read() of an device is done and after unblocking, the operation has completed. The exact way of waiting is dependent on the guest OS.

However, it is all implemented on the event virtqueue. The message type:

```
struct virtio_user_event {
    le64 instance_handle;
    le32 revents;
}
```

Event messages are sent back to the guest if *events* field is nonzero, as detailed in the section on ping virtqueue messages.

The guest driver can distinguish which instance receives which ping using *instance_handle*. The field *revents* is written by the return value of *on_instance_ping* from the device side.

### 5.11.7 Kernel Drivers via virtio-user

It is not a hard restriction for instances to be created from guest userspace; there are many kernel mechanisms such as sync fd's and USB devices that can benefit from running on top of virtio-user.

Provided the functionality exists in the guest kernel, virtio-user shall expose all of its operations to other kernel drivers as well.

### 5.11.8 Kernel and Hypervisor Portability Requirements

The main goal of virto-user is to allow high performance userspace drivers/devices to be defined and implemented in a way that is decoupled from guest kernels and host hypervisors; even socket interfaces are not assumed to exist, with only virtqueues and shared memory as the basic transport.

The device implementations themselves live in shared libraries that plug in to the host hypervisor. The userspace driver implementation use existing guest userspace facilities for communicating with drivers, such as open()/ioctl()/read()/mmap() on Linux.

This set of configuration struct and virtqueue message structs is meant to be implemented across a wide variety of guest kernels and host hypervisors. What follows are the requirements to implement virtio-user for a given guest kernel and a host hypervisor.

#### 5.11.8.1 Kernel Portability Requirements

First, the guest kernel is required to be able to expose the enumerated devices in the existing way in which devices are exposed. For example, in Linux, misc_register must be available to add new entries to /dev/ for each device. Each such device is associated with the vendor id, device id, and version. For example, /dev/virtio-user/abcd:ef10:03 refers to vendor id 0xabcd, device id 0xef10, version 3.

The guest kernel also needs some way to expose config operations to userspace and to the guest kernel space (as there are a few use cases that would involve implementing some kernel drivers in terms of virtio-userspace, such as sync fd's, usb, etc) In Linux, this is done by mapping open() to instance creation, the last close() to instance destruction, ioctl() for alloc/free/export/import, and mmap() to map memory.

The guest kernel also needs some way to forward ping messages. In Linux, this can also be done via ioctl().

The guest kernel also needs some way to expose event waiting. In Linux, this can be done via read(), and the return value will be revents in the event virtqueue message.

#### 5.11.8.2 Hypervisor Portability Requirements

The first capability the host hypervisor will need to support is runtime mapping of host pointers to guest physical addresses. As of this writing, this is available in KVM, Intel HAXM, and macOS Hypervisor.framework.

Next, the host hypervisor will need to support shared library plugin loading. This is so the device implementation can be separate from the host hypervisor. The device implementations live in single shared libraries.

There is one plugin shared library for each vendor/device id. The functions exposed by each shared library shall have the following form:

```
void register_memory_mapping_funcs(
    bool (*map_guest_ram)(le64 phys_addr, void* host_ptr, le64 size),
    bool (*unmap_guest_ram)(le64 phys_addr, le64 size));
void get_device_config_info(le32* vendorId, le32* deviceId, le32* version);
bool on_create_instance(le64 instance_handle);
void on_destroy_instance(le64 instance_handle);
le32 on_instance_ping(le64 instance_handle, le64 metadata, le64 phys_addr, void* host_ptr, le32
    ↪ events);
```

The host hypervisor's plugin loading system will load set of such shared libraries and resolve their vendor id, device id, and versions, which populates the information necessary for device enumeration to work.

Each instance is able to use the results of *register_memory_mapping_funcs* to communicate with the host hypervisor to map/unmap the shared memory to host buffers.

When an instance with a given vendor and device id is created via *on_create_instance*, the host hypervisor runs the plugin's *on_create_instance* function.

When an instance is destroyed, the host hypervisor runs the plugin's *on_destroy_instance* call.

When a ping happens, the host hypervisor calls the *on_instance_ping* of the plugin that is associated with the *instance_handle*.

If *shm_id* and *shm_offset* are valid, *phys_addr* is populated with the corresponding guest physical address.

If the guest physical address is mapped to a host pointer somewhere, then *host_ptr* is populated.

The return value from the plugin is then used as revents, and if the events was nonzero, the event virtqueue will be used to send revents back to the guest.

Given the portable guest OS / host hypervisor, an existing set of shared libraries implementing a device can be used for many different guest OSes and hypervisors that support virtio-user.

In the guest side, there needs to be a similar set of libraries to send commands; these depend more on the specifics of the guest OS and how virtio-user was exposed, but it will tend to be a parallel set of shared libraries in guest userspace where only guest OS-specific customizations need to be made while the basic protocol remains the same.

## 5.12 Host Memory Device

Note: This depends on the upcoming shared-mem type of virtio that allows sharing of host memory to the guest.

virtio-hostmem is a device for sharing host memory to the guest. It runs on top of virtio-pci for virtqueue messages and uses the PCI address space for direct access like virtio-fs does.

virtio-hostmem's purpose is to allow high performance general memory accesses between guest and host, and to allow the guest to access host memory constructed at runtime, such as mapped memory from graphics APIs.

Note that vhost-pci/vhost-vsock, virtio-vsock, and virtio-fs are also general ways to share data between the guest and host, but they are specialized to socket APIs in the guest, or depend on a FUSE implementation. virtio-hostmem provides such communication mechanism over raw memory, which has benefits of being more portable across hypervisors and guest OSes, and potentially higher performance due to always being physically contiguous to the guest.

The guest can create "instances" which capture a particular use case of the device. virtio-hostmem is like virtio-input in that the guest can query for sub-devices with IDs; the guest provides vendor and device id in configuration. The host then accepts or rejects the instance creation request.

Once instance creation succeeds, shared-mem objects can be allocated from each instance. Also, different instances can share the same shared-mem objects through export/import operations. On the host, it is assumed that the hypervisor will handle all backing of the shared memory objects with actual memory of some kind.

In operating the device, a ping virtqueue is used for the guest to notify the host when something interesting has happened in the shared memory. Conversely, the event virtqueue is used for the host to notify the guest. Note that this is asymmetric; it is expected that the guest will initiate most operations via ping virtqueue, while occasionally using the event virtqueue to wait on host completions.

Both guest kernel and userspace drivers can be written using operations on virtio-hostmem in a way that mirrors UIO for Linux; open()/close()/ioctl()/read()/write()/mmap(), but concrete implementations are outside the scope of this spec.

### 5.12.1 Device ID

21

### 5.12.2 Virtqueues

**0** config tx

**1** config rx

**2** ping

**3** event

### 5.12.3 Feature bits

No feature bits.

#### 5.12.3.1 Feature bit requirements

No feature bit requirements.

### 5.12.4 Device configuration layout

```
struct virtio_hostmem_device_info {
    le32 vendor_id;
    le32 device_id;
    le32 revision;
}

struct virtio_hostmem_config {
    le64 reserved_size;
    le32 num_devices;
    virtio_hostmem_device_info available_devices[MAX_DEVICES];
};
```

*virtio_hostmem_device_info* describes a particular usage of the device in terms of the vendor / device ID and revision.

*reserved_size* is the amount of address space taken away from the guest to support virtio-hostmem. A sufficient setting for most purposes is 16 GB.

*num_devices* represents the number of valid entries in *available_devices*.

*available_devices* represents the set of available usages of virtio-hostmem (up to *MAX_DEVICES*).

*MAX_DEVICES* is the maximum number of sub-devices possible (here, set to 32).

### 5.12.5 Device Initialization

Initialization of virtio-hostmem works much like other virtio PCI devices. It will need a PCI device ID.

### 5.12.6 Device Operation

#### 5.12.6.1 Config Virtqueue Messages

Operation always begins on the config virtqueue. Messages transmitted or received on the config virtqueue are of the following structure:

```
struct virtio_hostmem_config_msg {
    le32 msg_type;
    le32 vendor_id;
    le32 device_id;
    le32 revision;
    le64 instance_handle;
    le64 shm_id;
    le64 shm_offset;
    le64 shm_size;
    le32 shm_flags;
    le32 error;
}
```

*msg_type* can only be one of the following:

```
enum {
    VIRTIO_HOSTMEM_CONFIG_OP_CREATE_INSTANCE;
    VIRTIO_HOSTMEM_CONFIG_OP_DESTROY_INSTANCE;
    VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_ALLOC;
    VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_FREE;
    VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_EXPORT;
    VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_IMPORT;
}
```

*error* can only be one of the following:

```
enum {
    VIRTIO_HOSTMEM_ERROR_CONFIG_DEVICE_INITIALIZATION_FAILED;
    VIRTIO_HOSTMEM_ERROR_CONFIG_INSTANCE_CREATION_FAILED;
    VIRTIO_HOSTMEM_ERROR_CONFIG_SHARED_MEMORY_ALLOC_FAILED;
    VIRTIO_HOSTMEM_ERROR_CONFIG_SHARED_MEMORY_EXPORT_FAILED;
    VIRTIO_HOSTMEM_ERROR_CONFIG_SHARED_MEMORY_IMPORT_FAILED;
}
```

Instances are particular contexts surrounding usages of virtio-hostmem. They control whether and how shared memory is allocated and how messages are dispatched to the host.

*vendor_id*, *device_id*, and *revision* distinguish how the hostmem device is used. If supported on the host via checking the device configuration; that is, if there exists a backend corresponding to those fields, instance creation will succeed. The vendor and device id must match, while *revision* can be more flexible depending on the use case.

Creating instances:

The guest sends a *virtio_hostmem_config_msg* with *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_-OP_CREATE_INSTANCE* and with the *vendor_id*, *device_id*, *revision* fields set. The guest sends this message on the config tx virtqueue. On the host, a new *instance_handle* is generated.

If unsuccessful, *error* is set and sent back to the guest on the config rx virtqueue, and the *instance_handle* is discarded.

If successful, a *virtio_hostmem_config_msg* with *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_OP_CREATE_INSTANCE* and *instance_handle* equal to the generated handle is sent on the config rx virtqueue.

Destroying instances:

The guest sends a *virtio_hostmem_config_msg* with *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_OP_DESTROY_INSTANCE* on the config tx virtqueue. The only field that needs to be populated is *instance_handle*.

Destroying the instance unmaps from the guest PCI space and also unmaps on the host side for all non-exported/imported allocations of the instance. For exported or imported allocations, unmapping only occurs when a shared-mem-specific refcount reaches zero.

The other kinds of config message concern creation of shared host memory regions. The shared memory configuration operations are as follows:

Shared memory operations:

The guest sends a *virtio_hostmem_config_msg* with *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_ALLOC* on the config tx virtqueue. *instance_handle* needs to be a valid instance handle generated by the host. *shm_size* must be set and greater than zero. A new shared memory region is created in the PCI address space (actual allocation is deferred). If any operation fails, a message on the config tx virtqueue with *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_ALLOC* and *error* equal to *VIRTIO_HOSTMEM_ERROR_CONFIG_SHARED_MEMORY_ALLOC_FAILED* is sent. If all operations succeed, a new *shm_id* is generated along with *shm_offset* (offset into the PCI). and sent back on the config tx virtqueue.

Freeing shared memory objects works in a similar way, with setting *msg_type* equal to *VIRTIO_HOSTMEM_CONFIG_OP_SHARED_MEMORY_FREE*. If the memory has been shared, it is refcounted based on how many instance have used it. When the refcount reaches 0, the host hypervisor will explicitly unmap that shared memory object from any existing host pointers.

To export a shared memory object, we need to have a valid *instance_handle* and an allocated shared memory object with a valid *shm_id*. The export operation itself for now is mostly administrative; it marks that allocated memory as available for sharing.

To import a shared memory object, we need to have a valid *instance_handle* and an allocated shared memory object with a valid *shm_id* that has been allocated and exported. A new *shm_id* is not generated; this is mostly administrative and marks that that *shm_id* can also be used from the second instance. This is for sharing memory, so *instance_handle* need not be the same as the *instance_handle* that allocated the shared memory.

This is similar to Vulkan *VK_KHR_external_memory*, except over raw PCI address space and *shm_id*'s.

For mapping and unmapping shared memory objects, we do not include explicit virtqueue methods for these, and instead rely on the guest kernel's memory mapping primitives.

Flow control: Only one config message is allowed to be in flight either to or from the host at any time. That is, the handshake tx/rx for device enumeration, instance creation, and shared memory operations are done in a globally visible single threaded manner. This is to make it easier to synchronize operations on shared memory and instance creation.

#### 5.12.6.2 Ping Virtqueue Messages

Once the instances have been created and configured with shared memory, we can already read/write memory, and for some devices, that may already be enough if they can operate lock-free and wait-free without needing notifications; we're done!

However, in order to prevent burning up CPU in most cases, most devices need some kind of mechanism to trigger activity on the device from the guest. This is captured via a new message struct, which is separate from the config struct because it's smaller and the common case is to send those messages. These messages are sent from the guest to host on the ping virtqueue.

```
struct virtio_hostmem_ping {
    le64 instance_handle;
    le64 metadata;
    le64 shm_id;
    le64 shm_offset;
    le64 shm_size;
    le64 phys_addr;
    le64 host_addr;
    le32 events;
}
```

*instance_handle* must be a valid instance handle. *shm_id* need not be a valid shm_id. If *shm_id* is a valid shm_id, it need not be allocated on the host yet.

If *shm_id* is a valid shm_id, For security reasons, *phys_addr* is resolved given *shm_offset* by the virtio-hostmem driver after the message arrives to the driver.

If *shm_id* is a valid shm_id and there is a mapping set up for *phys_addr*, *host_addr* refers to the corresponding memory view in the host address space. For security reasons, *host_addr* is only resolved on the host after the message arrives on the host.

This allows notifications to coherently access to device memory from both the host and guest, given a few extra considerations. For example, for architectures that do not have store/load coherency (i.e., not x86) an explicit set of fence or synchronization instructions will also be run by virtio-hostmem both before and after the call to *on_instance_ping*. An alternative is to leave this up to the implementor of the virtual device, but it is going to be such a common case to synchronize views of the same memory that it is probably a good idea to include synchronization out of the box.

Although, it may be common to block a guest thread until *on_instance_ping* completes on the device side. That is the purpose of the *events* field; the guest can populate it if it is desired to sync on the host completion. If *events* is not zero, then a reply shall be sent back to the guest via the event virtqueue, with the *revents* set to the appropriate value.

Flow control: Arbitrary levels of traffic can be sent on the ping virtqueue from multiple instances at the same time, but ordering within an instance is strictly preserved. Additional resources outside the virtqueue are used to hold incoming messages if the virtqueue itself fills up. This is similar to how virtio-vsock handles high traffic. However, there will be a limit on the maximum number of messages in flight to prevent the guest from over-notifying the host. Once the limit is reached, the guest blocks until the number of messages in flight is decreased.

The semantics of ping messages themselves also are not restricted to guest to host only; the shared memory region named in the message can also be filled by the host and used as receive traffic by the guest. The ping message is then suitable for DMA operations in both directions, such as glTexImage2D and glReadPixels, and audio/video (de)compression (guest populates shared memory with (de)compressed buffers, sends ping message, host (de)compresses into the same memory region).

#### 5.12.6.3 Event Virtqueue Messages

Ping virtqueue messages are enough to cover all async device operations; that is, operations that do not require a round trip from the host. This is useful for most kinds of graphics API forwarding along with media codecs.

However, it can still be important to synchronize the guest on the completion of a device operation.

In the driver, the interface can be similar to Linux uio interrupts for example; a blocking read() of an device is done and after unblocking, the operation has completed. The exact way of waiting is dependent on the guest OS.

However, it is all implemented on the event virtqueue. The message type:

```
struct virtio_hostmem_event {
    le64 instance_handle;
    le32 revents;
```

```
}
```

Event messages are sent back to the guest if *events* field is nonzero, as detailed in the section on ping virtqueue messages.

The guest driver can distinguish which instance receives which ping using *instance_handle*. The field *revents* is written by the return value of *on_instance_ping* from the device side.

  

# 6 Reserved Feature Bits

Currently these 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 2.6.5.3 Indirect Descriptors and 2.7.7 Indirect Flag: Scatter-Gather Support.

**VIRTIO_F_RING_EVENT_IDX(29)** This feature enables the *used_event* and the *avail_event* fields as described in 2.6.7, 2.6.8 and 2.7.10.

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

**VIRTIO_F_ACCESS_PLATFORM(33)** This feature indicates that the device can be used on a platform where device access to data in memory is limited and/or translated. E.g. this is the case if the device can be located behind an IOMMU that translates bus addresses from the device into physical addresses in memory, if the device can be limited to only access certain memory addresses or if special commands such as a cache flush can be needed to synchronise data in memory with the device. Whether accesses are actually limited or translated is described by platform-specific means. If this feature bit is set to 0, then the device has same access to memory addresses supplied to it as the driver has. In particular, the device will always use physical addresses matching addresses used by the driver (typically meaning physical addresses used by the CPU) and not translated further, and can access any address supplied to it by the driver. When clear, this overrides any platform-specific description of whether device access is limited or translated in any way, e.g. whether an IOMMU may be present.

**VIRTIO_F_RING_PACKED(34)** This feature indicates support for the packed virtqueue layout as described in 2.7 Packed Virtqueues.

**VIRTIO_F_IN_ORDER(35)** This feature indicates that all buffers are used by the device in the same order in which they have been made available.

**VIRTIO_F_ORDER_PLATFORM(36)** This feature indicates that memory accesses by the driver and the device are ordered in a way described by the platform.

If this feature bit is negotiated, the ordering in effect for any memory accesses by the driver that need to be ordered in a specific way with respect to accesses by the device is the one suitable for devices described by the platform. This implies that the driver needs to use memory barriers suitable for devices described by the platform; e.g. for the PCI transport in the case of hardware PCI devices.

If this feature bit is not negotiated, then the device and driver are assumed to be implemented in software, that is they can be assumed to run on identical CPUs in an SMP configuration. Thus a weaker form of memory barriers is sufficient to yield better performance.

**VIRTIO_F_SR_IOV(37)** This feature indicates that the device supports Single Root I/O Virtualization. Currently only PCI devices support this feature.

**VIRTIO_F_NOTIFICATION_DATA(38)** This feature indicates that the driver passes extra data (besides identifying the virtqueue) in its device notifications. See 2.7.23 Driver notifications.

## 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.

A driver SHOULD accept VIRTIO_F_ACCESS_PLATFORM if it is offered, and it MUST then either disable the IOMMU or configure the IOMMU to translate bus addresses passed to the device into physical addresses in memory. If VIRTIO_F_ACCESS_PLATFORM is not offered, then a driver MUST pass only physical addresses to the device.

A driver SHOULD accept VIRTIO_F_RING_PACKED if it is offered.

A driver SHOULD accept VIRTIO_F_ORDER_PLATFORM if it is offered. If VIRTIO_F_ORDER_PLATFORM has been negotiated, a driver MUST use the barriers suitable for hardware devices.

If VIRTIO_F_SR_IOV has been negotiated, a driver MAY enable virtual functions through the device's PCI SR-IOV capability structure. A driver MUST NOT negotiate VIRTIO_F_SR_IOV if the device does not have a PCI SR-IOV capability structure or is not a PCI device. A driver MUST negotiate VIRTIO_F_SR_IOV and complete the feature negotiation (including checking the FEATURES_OK *device status* bit) before enabling virtual functions through the device's PCI SR-IOV capability structure. After once successfully negotiating VIRTIO_F_SR_IOV, the driver MAY enable virtual functions through the device's PCI SR-IOV capability structure even if the device or the system has been fully or partially reset, and even without re-negotiating VIRTIO_F_SR_IOV after the reset.

## 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.

A device SHOULD offer VIRTIO_F_ACCESS_PLATFORM if its access to memory is through bus addresses distinct from and translated by the platform to physical addresses used by the driver, and/or if it can only access certain memory addresses with said access specified and/or granted by the platform. A device MAY fail to operate further if VIRTIO_F_ACCESS_PLATFORM is not accepted.

If VIRTIO_F_IN_ORDER has been negotiated, a device MUST use buffers in the same order in which they have been available.

A device MAY fail to operate further if VIRTIO_F_ORDER_PLATFORM is offered but not accepted. A device MAY operate in a slower emulation mode if VIRTIO_F_ORDER_PLATFORM is offered but not accepted.

It is RECOMMENDED that an add-in card based PCI device offers both VIRTIO_F_ACCESS_PLATFORM and VIRTIO_F_ORDER_PLATFORM for maximum portability.

A device SHOULD offer VIRTIO_F_SR_IOV if it is a PCI device and presents a PCI SR-IOV capability structure, otherwise it MUST NOT offer VIRTIO_F_SR_IOV.

## 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 a used buffer notification if the device runs out of available descriptors on a virtqueue, even though notifications 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 notifies 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 2.6.4.3 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, 7.2.8, 7.2.9, or 7.2.12.

**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, 7.3.8, 7.3.9, or 7.3.12.

## 7.2 Driver Conformance

A driver MUST conform to the following normative statements:

- 2.1.1
- 2.2.1
- 2.4.1
- 2.6.1
- 2.6.4.2
- 2.6.5.2
- 2.6.5.3.1
- 2.6.7.1
- 2.6.6.1
- 2.6.8.3
- 2.6.10.1
- 2.6.13.3.1
- 2.6.13.4.1
- 3.1.1
- 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
- 4.1.5.1.2.2
- 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:

- 4.3.1.4
- 4.3.2.1.2
- 4.3.2.3.1
- 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
- 5.1.6.4.2
- 5.1.6.5.1.2
- 5.1.6.5.2.2
- 5.1.6.5.4.1
- 5.1.6.5.5.1

- 5.1.6.5.6.2

### 7.2.5 Block Driver Conformance

A block driver MUST conform to the following normative statements:

- 5.2.5.1
- 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 Traditional Memory Balloon Driver Conformance

A traditional memory balloon driver MUST conform to the following normative statements:

- 5.5.3.1
- 5.5.6.1
- 5.5.6.3.1

### 7.2.9 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

### 7.2.10 Input Driver Conformance

An input driver MUST conform to the following normative statements:

- 5.8.5.1
- 5.8.6.1

### 7.2.11 Crypto Driver Conformance

A Crypto driver MUST conform to the following normative statements:

- 5.9.5.2
- 5.9.6.1
- 5.9.7.2.1.5
- 5.9.7.2.1.7
- 5.9.7.4.1
- 5.9.7.5.1
- 5.9.7.6.1
- 5.9.7.7.1

### 7.2.12 Socket Driver Conformance

A socket driver MUST conform to the following normative statements:

- 5.10.6.3.1
- 5.10.6.4.1
- 5.10.6.6.1

## 7.3 Device Conformance

A device MUST conform to the following normative statements:

- 2.1.2
- 2.2.2
- 2.4.2
- 2.6.4.1
- 2.6.5.1
- 2.6.5.3.2
- 2.6.7.2
- 2.6.8.2
- 2.6.10.2
- 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
- 4.1.3.2
- 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
- 4.1.4.9.0.1
- 4.1.5.1.2.1
- 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:

- 4.3.1.3
- 4.3.2.1.1
- 4.3.2.2.1
- 4.3.2.3.2
- 4.3.2.6.3.1
- 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
- 5.1.6.2.2
- 5.1.6.3.2
- 5.1.6.4.1
- 5.1.6.5.1.1
- 5.1.6.5.2.1
- 5.1.6.5.4.2
- 5.1.6.5.5.2

### 7.3.5 Block Device Conformance

A block device MUST conform to the following normative statements:

- 5.2.5.2
- 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 Traditional Memory Balloon Device Conformance

A traditional memory balloon device MUST conform to the following normative statements:

- 5.5.3.2
- 5.5.6.2
- 5.5.6.3.2

### 7.3.9 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

### 7.3.10 Input Device Conformance

An input device MUST conform to the following normative statements:

- 5.8.5.2
- 5.8.6.2

### 7.3.11 Crypto Device Conformance

A Crypto device MUST conform to the following normative statements:

- 5.9.5.1
- 5.9.7.2.1.6

- 5.9.7.2.1.8
- 5.9.7.4.2
- 5.9.7.5.2
- 5.9.7.6.2
- 5.9.7.7.2

### 7.3.12 Socket Device Conformance

A socket device MUST conform to the following normative statements:

- 5.10.6.3.2
- 5.10.6.4.2

## 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 2.4.3
- Section 2.4.4
- Section 2.6.2
- Section 2.6.3
- Section 2.6.4.3
- Section 3.1.2
- Section 4.1.2.3
- Section 4.1.4.8
- Section 4.1.5.1.1.1
- Section 4.1.5.1.3.1
- Section 4.2.4
- Section 4.3.2.1.3
- Section 4.3.2.2.2
- Section 4.3.3.1.3
- Section 4.3.2.6.4
- Section 5.1.3.2

- Section 5.1.4.3
- Section 5.1.6.1
- Section 5.1.6.5.2.3
- Section 5.1.6.5.3.1
- Section 5.1.6.5.5.3
- Section 5.1.6.5.6.3
- Section 5.2.3.1
- Section 5.2.4.1
- Section 5.2.5.3
- Section 5.2.6.3
- Section 5.3.4.1
- Section 5.3.6.3
- Section 5.5.3.2.0.1
- Section 5.5.6.2.1
- Section 5.5.6.3.3
- Section 5.6.4.3
- Section 5.6.6.0.1
- Section 5.6.6.1.3
- Section 5.6.6.2.1
- Section 5.6.6.3.3
- Section 6.3

# Appendix A. virtio_queue.h

This file is also available at the link https://docs.oasis-open.org/virtio/virtio/v1.1/wd01/listings/virtio_queue.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_event and used_event 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. Therefore, no writeable field which triggers an action ought to be wider than 32 bits.

## 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, Red Hat
Daniel Kiper, Oracle
Geoff Brown, Machine-to-Machine Intelligence (M2MI) Corporation
Gershon Janssen, Individual Member
Halil Pasic, IBM
James Bottomley, Parallels IP Holdings GmbH
Jian Zhou, Huawei
Lei Gong, Huawei
Luiz Capitulino, Red Hat
Michael S. Tsirkin, Red Hat
Paolo Bonzini, Red Hat
Pawel Moll, ARM
Peng Long, Huawei
Richard Sohn, Alcatel-Lucent
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
Brian Foley, ARM
David Alan Gilbert, Red Hat
Fam Zheng, Red Hat
Gerd Hoffmann, Red Hat
Halil Pasic, IBM
Jason Wang, Red Hat
Laura Novich, Red Hat
Patrick Durusau, Technical Advisory Board, OASIS
Thomas Huth, Red Hat
Yan Vugenfirer, Red Hat / Daynix
Kevin Lo, MSI

# Appendix D. Revision History

The following changes have been made since the previous version of this specification:

| Revision | Date | Editor | Changes Made |
|---|---|---|---|
| REV | DATE | Author | Description: TODO |