Virtual I/O Device (VIRTIO) Version 1.3

Committee Specification Draft 01

06 October 2023

This stage:
https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/tex/ (Authoritative)
https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.pdf
https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.html

Previous stage:
N/A

Latest stage:
https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.pdf
https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html

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

Chairs:
Michael S. Tsirkin (mst@redhat.com), Red Hat
Cornelia Huck (cohuck@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/v1.3/csd01/listings/

Related work:
This specification replaces or supersedes:

- Virtual I/O Device (VIRTIO) Version 1.2. Edited by Michael S. Tsirkin and Cornelia Huck. Latest stage:
  https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html
- Virtual I/O Device (VIRTIO) Version 1.1. Edited by Michael S. Tsirkin and Cornelia Huck. Latest stage:
  https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html
- Virtual I/O Device (VIRTIO) Version 1.0. Edited by Rusty Russell, Michael S. Tsirkin, Cornelia Huck, and Pawel Moll. Latest stage:
  https://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.html
- 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 stage” 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 page in the TC’s GitHub repository (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-v1.3]
Virtual I/O Device (VIRTIO) Version 1.3. Edited by Michael S. Tsirkin and Cornelia Huck. 06 October 2023. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/virtio/virtio/v1.3/csd01/virtio-v1.3-csd01.html. Latest stage: https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.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.

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 AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS DOCUMENT OR ANY PART THEREOF.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

[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
[RFC4122]	Leach, P., Mealling, M., and R. Salz, “A Universally Unique IDentifier (UUID) URN Namespace”, RFC 4122, DOI 10.17487/RFC4122, July 2005. http://www.ietf.org/rfc/rfc4122.txt
[S390 PoP]	z/Architecture Principles of Operation, IBM Publication SA22-7832, https://www.ibm.com/docs/en/SSQ2R2_15.0.0/com.ibm.tpf.toolkit.hlasm.doc/dz9zr006.pdf, and any future revisions
[S390 Common I/O]	ESA/390 Common I/O-Device and Self-Description, IBM Publication SA22-7204, https://www.ibm.com/resources/publications/OutputPubsDetails?PubID=SA22720401, 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

1This 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.

2The Linux implementation further separates the virtio transport code from the specific virtio drivers: these drivers are shared between different transports.

[IEEE 802] IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture, http://www.ieee802.org/, IEEE

[IEEE 802.3-2022] IEEE Standard for Ethernet, https://doi.org/10.1109/IEEESTD.2022.9844436, IEEE 802.3-2022

[IEEE 802 Ethertypes] IEEE 802 Ethertypes, https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml, IANA

[IANA Protocol Numbers] IANA Protocol Numbers, https://www.iana.org/assignments/protocol-numbers, IANA

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

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

[FUSE] Linux FUSE interface, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/fuse.h

[errno] Linux error names and numbers, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/asm-generic/errno-base.h

[eMMC] eMMC Electrical Standard (5.1), JESD84-B51, http://www.jedec.org/sites/default/files/docs/JESD84-B51.pdf

[HDA] High Definition Audio Specification, https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/high-definition-audio-specification.pdf

[I2C] I2C-bus specification and user manual, https://www.nxp.com/docs/en/user-guide/UM10204.pdf

[SCMI] Arm System Control and Management Interface, DEN0056, https://developer.arm.com/docs/den0056/c, version C and any future revisions

[RFC3447] J. Jonsson.,“Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography”, February 2003. https://www.ietf.org/rfc/rfc3447.txt

[FIPS186-3] National Institute of Standards and Technology (NIST), FIPS Publication 180-3: Secure Hash Standard, October 2008. https://csrc.nist.gov/csrc/media/publications/fips/186/3/archive/2009-06-25/documents/fips_186-3.pdf

[RFC5915] “Elliptic Curve Private Key Structure”, June 2010. https://www.rfc-editor.org/rfc/rfc5915

[RFC6025] C.Wallace., “ASN.1 Translation”, October 2010. https://www.ietf.org/rfc/rfc6025.txt

[RFC3279] W.Polk., “Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile”, April 2002. https://www.ietf.org/rfc/rfc3279.txt

[SEC1] Standards for Efficient Cryptography Group(SECG), “SEC1: Elliptic Cureve Cryptography”, Version 1.0, September 2000. https://www.secg.org/sec1-v2.pdf

[RFC2784] Generic Routing Encapsulation. This protocol is only specified for IPv4 and used as either the payload or delivery protocol. https://datatracker.ietf.org/doc/rfc2784/

[RFC2890] Key and Sequence Number Extensions to GRE. This protocol describes extensions by which two fields, Key and Sequence Number, can be optionally carried in the GRE Header.
https://www.rfc-editor.org/rfc/rfc2890

[RFC7676] IPv6 Support for Generic Routing Encapsulation (GRE). This protocol is specified for IPv6 and used as either the payload or delivery protocol. Note that this does not change the GRE header format or any behaviors specified by RFC 2784 or RFC 2890.
https://datatracker.ietf.org/doc/rfc7676/

[GRE-in-UDP] GRE-in-UDP Encapsulation. This specifies a method of encapsulating network protocol packets within GRE and UDP headers. This protocol is specified for IPv4 and IPv6, and used as either the payload or delivery protocol.
https://www.rfc-editor.org/rfc/rfc8086

[VXLAN] Virtual eXtensible Local Area Network.
https://datatracker.ietf.org/doc/rfc7348/

[VXLAN-GPE] Generic Protocol Extension for VXLAN. This protocol describes extending Virtual eXtensible Local Area Network (VXLAN) via changes to the VXLAN header.
https://www.ietf.org/archive/id/draft-ietf-nvo3-vxlan-gpe-12.txt

[GENEVE] Generic Network Virtualization Encapsulation.
https://datatracker.ietf.org/doc/rfc8926/

[IPIP] IP Encapsulation within IP.
https://www.rfc-editor.org/rfc/rfc2003

[NVGRE] NVGRE: Network Virtualization Using Generic Routing Encapsulation
https://www.rfc-editor.org/rfc/rfc7637.html

[IP] INTERNET PROTOCOL
https://www.rfc-editor.org/rfc/rfc791

[Internet Header Format] Internet Header Format
https://datatracker.ietf.org/doc/html/rfc791#section-3.1

[IPv6 Header Format] IPv6 Header Format
https://www.rfc-editor.org/rfc/rfc8200#section-3

[UDP] User Datagram Protocol
https://www.rfc-editor.org/rfc/rfc768

[TCP] TRANSMISSION CONTROL PROTOCOL
https://www.rfc-editor.org/rfc/rfc793

[TCP Header Format] TCP Header Format
https://www.rfc-editor.org/rfc/rfc9293#name-header-format

[IPSEC] IPsec Protocol
https://www.rfc-editor.org/rfc/rfc4301

[ESP] IPsec ESP
https://www.rfc-editor.org/rfc/rfc4303

[ESN] IPsec ESN
https://www.rfc-editor.org/rfc/rfc4304

[UDP Encapsulation] IPsec UDP Encapsulation
https://www.rfc-editor.org/rfc/rfc3948

[CAN] ISO 11898-1:2015 Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling

[RFC8174] Leiba, B., ”Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words”, BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017
http://www.ietf.org/rfc/rfc8174.txt

[EPOCH] POSIX.1-2024, Base Definitions, Seconds Since the Epoch
https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/V1_chap04.html#tag_04_19

[UTC-SLS] UTC with Smoothed Leap Seconds (UTC-SLS)
https://www.cl.cam.ac.uk/~mgk25/time/utc-sls/

1.2 Non-Normative References

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

1.3 Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119] and [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3.1 Legacy Interface: Terminology

Specification drafts preceding version 1.0 of this specification (e.g. see [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)
```

1.5 Constant Specifications

In many cases, numeric values used in the interface between the device and the driver are documented using the C #define and /* */ comment syntax. Multiple related values are grouped together with a common name as a prefix, using _ as a separator. Using _XXX as a suffix refers to all values in a group. For example:

```
/* Field Fld value A description */
#define VIRTIO_FLD_A        (1 << 0)
/* Field Fld value B description */
#define VIRTIO_FLD_B        (1 << 1)
```

documents two numeric values for a field *Fld*, with *Fld* having value 1 referring to *A* and *Fld* having value 2 referring to *B*. Note that << refers to the shift-left operation.

Further, in this case VIRTIO_FLD_A and VIRTIO_FLD_B refer to values 1 and 2 of Fld respectively. Further, VIRTIO_FLD_XXX refers to either VIRTIO_FLD_A or VIRTIO_FLD_B.

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.

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

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

SUSPEND (16) When VIRTIO_F_SUSPEND is negotiated, indicates that the device has been suspended by the driver.

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

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

The *device status* field starts out as 0, and is reinitialized to 0 by the device during reset.

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 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, 41, 42 and 50 to 127 Feature bits for the specific device type

24 to 40, and 43 Feature bits reserved for extensions to the queue and feature negotiation mechanisms, see 6

44 to 49, and 128 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.

To keep the feature negotiation mechanism extensible, it is important that devices *do not offer any feature bits that they would not be able to handle if the driver accepted them (even though drivers are not supposed to accept any unspecified, reserved, or unsupported features even if offered, according to the specification.) Likewise, it is important that drivers* do not accept feature bits they do not know how to handle (even though devices are not supposed to offer any unspecified, reserved, or unsupported features in the first place, according to the specification.) The preferred way for handling reserved and unexpected features is that the driver ignores them.

In particular, this is especially important for features limited to specific transports, as enabling these for more transports in future versions of the specification is highly likely to require changing the behaviour from drivers and devices. Drivers and devices supporting multiple transports need to carefully maintain per-transport lists of allowed features.

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 MUST validate the feature bits offered by the device. The driver MUST ignore and MUST NOT accept any feature bit that is

- not described in this specification,
- marked as reserved,
- not supported for the specific transport,
- not defined for the device type.

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.

By contrast, the driver MUST NOT fail solely because a feature it does not understand has been offered by the device.

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.

The device MUST NOT offer feature bits corresponding to features it would not support if accepted by the driver (even if the driver is prohibited from accepting the feature bits by the specification); for the sake of clarity, this refers to feature bits not described in this specification, reserved feature bits and feature bits reserved or not supported for the specific transport or the specific device type, but this does not preclude devices written to a future version of this specification from offering such feature bits should such a specification have a provision for devices to support the corresponding features.

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 Reset

The driver may want to initiate a device reset at various times; notably, it is required to do so during device initialization and device cleanup.

The mechanism used by the driver to initiate the reset is transport specific.

2.4.1 Device Requirements: Device Reset

A device MUST reinitialize *device status* to 0 after receiving a reset.

A device MUST NOT send notifications or interact with the queues after indicating completion of the reset by reinitializing *device status* to 0, until the driver re-initializes the device.

2.4.2 Driver Requirements: Device Reset

The driver SHOULD consider a driver-initiated reset complete when it reads *device status* as 0.

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.6 Virtqueues

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

A virtio device can have maximum of 65536 virtqueues. Each virtqueue is identified by a virtqueue index. A virtqueue index has a value in the range of 0 to 65535.

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.

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 2.7):

- 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 2.7 Split Virtqueues) and Packed Virtqueues (see 2.8 Packed Virtqueues).

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

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

2.6.1 Virtqueue Reset

When VIRTIO_F_RING_RESET is negotiated, the driver can reset a virtqueue individually. The way to reset the virtqueue is transport specific.

Virtqueue reset is divided into two parts. The driver first resets a queue and can afterwards optionally re-enable it.

2.6.1.1 Virtqueue Reset

2.6.1.1.1 Device Requirements: Virtqueue Reset

After a queue has been reset by the driver, the device MUST NOT execute any requests from that virtqueue, or notify the driver for it.

The device MUST reset any state of a virtqueue to the default state, including the available state and the used state.

2.6.1.1.2 Driver Requirements: Virtqueue Reset

After the driver tells the device to reset a queue, the driver MUST verify that the queue has actually been reset.

After the queue has been successfully reset, the driver MAY release any resource associated with that virtqueue.

2.6.1.2 Virtqueue Re-enable

This process is the same as the initialization process of a single queue during the initialization of the entire device.

2.6.1.2.1 Device Requirements: Virtqueue Re-enable

The device MUST observe any queue configuration that may have been changed by the driver, like the maximum queue size.

2.6.1.2.2 Driver Requirements: Virtqueue Re-enable

When re-enabling a queue, the driver MUST configure the queue resources as during initial virtqueue discovery, but optionally with different parameters.

2.7 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.

2.7.1 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.

2.7.2 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;
};
```

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

2.7.3 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.

2.7.4 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!

2.7.4.1 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.

2.7.4.2 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.

2.7.4.3 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.

2.7.5 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.

2.7.5.1 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). A device MUST NOT write to any descriptor table entry.

2.7.5.2 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.

2.7.5.3 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.

2.7.5.3.1 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.

2.7.5.3.2 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.

2.7.6 The Virtqueue Available Ring

The available ring has the following layout structure:

```
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.

2.7.6.1 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).

2.7.7 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.

2.7.7.1 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.

2.7.7.2 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).

2.7.8 The Virtqueue Used Ring

The used ring has the following layout structure:

```
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;
        /*
         * The number of bytes written into the device writable portion of
         * the buffer described by the descriptor chain.
         */
        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.

2.7.8.1 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.

2.7.8.2 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.

2.7.8.3 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.

2.7.9 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.

2.7.10 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 2.7.7. 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.

2.7.10.1 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.

2.7.10.2 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.

2.7.11 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.

2.7.12 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.

2.7.13 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 2.7.5 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.

2.7.13.1 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.

2.7.13.2 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;
```

2.7.13.3 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.

2.7.13.3.1 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.

2.7.13.4 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 2.7.10.

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

2.7.13.4.1 Driver Requirements: Notifying The Device

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

2.7.14 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 2.7.7.

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++;
}
```

2.8 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 the device and the driver.

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.

2.8.1 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.

2.8.2 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.

2.8.3 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.

2.8.4 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.

2.8.5 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 virtqueue size.

2.8.6 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.

2.8.7 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.

2.8.8 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.

2.8.9 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.

2.8.10 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_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.

2.8.10.1 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.

2.8.11 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.

2.8.12 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.

2.8.13 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.

2.8.14 Event Suppression Structure Format

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

```
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;
};
```

2.8.15 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*.

2.8.16 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.

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

2.8.17 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.

2.8.18 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.

2.8.19 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 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 VIRTQ_DESC_F_INDIRECT set in a scatter-gather list linked by VIRTQ_DESC_F_NEXT. *flags*.

2.8.20 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.

2.8.21 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.

2.8.21.1 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.

2.8.21.1.1 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.

2.8.21.2 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 2.8.14.

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

2.8.21.3 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_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);
}
```

2.8.21.3.1 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.

2.8.22 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 2.8.14.

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);
}
```

2.9 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 contains either a virtqueue index if VIRTIO_F_NOTIF_CONFIG_DATA is not negotiated or device supplied virtqueue notification config data if VIRTIO_F_NOTIF_CONFIG_DATA is negotiated.

The notification method and supplying any such virtqueue notification config data is transport specific.

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:

vq_index or vq_notif_config_data Either virtqueue index or device supplied queue notification config data corresponding to a virtqueue.

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.

2.10 Shared Memory Regions

Shared memory regions are an additional facility available to devices that need a region of memory that's continuously shared between the device and the driver, rather than passed between them in the way virtqueue elements are.

Example uses include shared caches and version pools for versioned data structures.

The memory region is allocated by the device and presented to the driver. Where the device is implemented in software on a host, this arrangement allows the memory region to be allocated by a library on the host, which the device may not have full control over.

A device may have multiple shared memory regions associated with it. Each region has a *shmid* to identify it, the meaning of which is device-specific.

Enumeration and location of shared memory regions is performed in a transport-specific way.

Memory consistency rules vary depending on the region and the device and they will be specified as required by each device.

2.10.1 Addressing within regions

References into shared memory regions are represented as offsets from the beginning of the region instead of absolute memory addresses. Offsets are used both for references between structures stored within shared memory and for requests placed in virtqueues that refer to shared memory. The *shmid* may be explicit or may be inferred from the context of the reference.

2.10.2 Device Requirements: Shared Memory Regions

Shared memory regions MUST NOT expose shared memory regions which are used to control the operation of the device, nor to stream data.

2.11 Exporting Objects

When an object created by one virtio device needs to be shared with a separate virtio device, the first device can export the object by generating a UUID which can then be passed to the second device to identify the object.

What constitutes an object, how to export objects, and how to import objects are defined by the individual device types. It is RECOMMENDED that devices generate version 4 UUIDs as specified by [RFC4122].

2.12 Device groups

It is occasionally useful to have a device control a group of other devices (the group may occasionally include the device itself) within a group. The owner device itself is not a member of the group (except in the special case of the self group). Terminology used in such cases:

Device group or just group, includes zero or more devices.

Owner device or owner, the device controlling the group.

Member device a device within a group. The owner device itself is not a member of the group except for the *Self group type*.

Member identifier each member has this identifier, unique within the group and used to address it through the owner device.

Group type identifier specifies what kind of member devices there are in a group, how the member identifier is interpreted and what kind of control the owner has. A given owner can control multiple groups

of different types but only a single group of a given type, thus the type and the owner together identify the group. [4]

Note: Each device only has a single driver, thus for the purposes of this section, ”the driver” is usually unambiguous and refers to the driver of the owner device. When there’s ambiguity, ”owner driver” refers to the driver of the owner device, while ”member driver” refers to the driver of a member device.

The following group types, and their identifiers, are currently specified:

Self group type (0x0) This device group includes the owner device itself and no other devices. The group type identifier for this group is 0x0. The member identifier for this group has a value of 0x0.

SR-IOV group type (0x1) This device group has a PCI Single Root I/O Virtualization (SR-IOV) physical function (PF) device as the owner and includes all its SR-IOV virtual functions (VFs) as members (see [PCIe]).

The PF device itself is not a member of the group.

The group type identifier for this group is 0x1.

A member identifier for this group can have a value from 0x1 to *NumVFs* as specified in the SR-IOV Extended Capability of the owner device and equals the SR-IOV VF number of the member device; the group only exists when the *VF Enable* bit in the SR-IOV Control Register within the SR-IOV Extended Capability of the owner device is set (see [PCIe]).

Both owner and member devices for this group type use the Virtio PCI transport (see 4.1).

2.12.1 Group administration commands

The driver sends group administration commands to the owner device of a group to control member devices of the group. This mechanism can be used, for example, to configure a member device before it is initialized by its driver. [5]

All the group administration commands are of the following form:

```
struct virtio_admin_cmd {
        /* Device-readable part */
        le16 opcode;
        /*
         * 0       - Self
         * 1       - SR-IOV
         * 2-65535 - reserved
         */
        le16 group_type;
        /* unused, reserved for future extensions */
        u8 reserved1[12];
        le64 group_member_id;
        le64 command_specific_data[];

        /* Device-writable part */
        le16 status;
        le16 status_qualifier;
        /* unused, reserved for future extensions */
        u8 reserved2[4];
        u8 command_specific_result[];
};
```

For all commands, *opcode*, *group_type* and if necessary *group_member_id* and *command_specific_data* are set by the driver, and the owner device sets *status* and if needed *status_qualifier* and *command_specific_result*.

Generally, any unused device-readable fields are set to zero by the driver and ignored by the device. Any unused device-writeable fields are set to zero by the device and ignored by the driver.

[4] Even though some group types only support specific transports, group type identifiers are global rather than transport-specific - a flood of new group types is not expected.

[5] The term ”administration” is intended to be interpreted widely to include any kind of control. See specific commands for detail.

opcode specifies the command. The valid values for *opcode* can be found in the following table:

opcode	Name	Command Description
0x0000	VIRTIO_ADMIN_CMD_LIST_QUERY	Provides to driver list of commands supported for this group type
0x0001	VIRTIO_ADMIN_CMD_LIST_USE	Provides to device list of commands used for this group type
0x0002	VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE	Writes into the legacy common configuration structure
0x0003	VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ	Reads from the legacy common configuration structure
0x0004	VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE	Writes into the legacy device configuration structure
0x0005	VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ	Reads into the legacy device configuration structure
0x0006	VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO	Query the notification region information
0x0007	VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY	Query the supported device capabilities bitmap
0x0008	VIRTIO_ADMIN_CMD_DEVICE_CAP_GET	Get the device capabilities
0x0009	VIRTIO_ADMIN_CMD_DRIVER_CAP_SET	Set the driver capabilities
0x000a	VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE	Create a device resource object
0x000c	VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY	Modify a device resource object
0x000b	VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY	Query a device resource object
0x000d	VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY	Destroy a device resource object
0x000e	VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET	Get the metadata of the device parts
0x000f	VIRTIO_ADMIN_CMD_DEV_PARTS_GET	Get the device parts
0x0010	VIRTIO_ADMIN_CMD_DEV_PARTS_SET	Set the device parts
0x0011	VIRTIO_ADMIN_CMD_DEV_MODE_SET	Stop or resume the device

0x0013 - 0x7FFF	-	Commands using *struct virtio_admin_cmd*
0x8000 - 0xFFFF	-	Reserved for future commands (possibly using a different structure)

The *group_type* specifies the group type identifier. The *group_member_id* specifies the member identifier within the group. See section 2.12 for the definition of the group type identifier and group member identifier.

The *status* describes the command result and possibly failure reason at an abstract level, this is appropriate for forwarding to applications. The *status_qualifier* describes failures at a low virtio specific level, as appropriate for debugging. The following table describes possible *status* values; to simplify common implementations, they are intentionally matching common Linux error names and numbers:

Status (decimal)	Name	Description
00	VIRTIO_ADMIN_STATUS_OK	successful completion
06	VIRTIO_ADMIN_STATUS_ENXIO	no such capability or resource
11	VIRTIO_ADMIN_STATUS_EAGAIN	try again
12	VIRTIO_ADMIN_STATUS_ENOMEM	insufficient resources
16	VIRTIO_ADMIN_STATUS_EBUSY	device busy
22	VIRTIO_ADMIN_STATUS_EINVAL	invalid command
28	VIRTIO_ADMIN_STATUS_ENOSPC	resources exhausted on device
other	-	group administration command error

When *status* is VIRTIO_ADMIN_STATUS_OK, *status_qualifier* is reserved and set to zero by the device.

The following table describes possible *status_qualifier* values:

Status	Name	Description
0x00	VIRTIO_ADMIN_STATUS_Q_OK	used with VIRTIO_ADMIN_STATUS_OK
0x01	VIRTIO_ADMIN_STATUS_Q_INVALID_COMMAND	command error: no additional information
0x02	VIRTIO_ADMIN_STATUS_Q_INVALID_OPCODE	unsupported or invalid *opcode*
0x03	VIRTIO_ADMIN_STATUS_Q_INVALID_FIELD	unsupported or invalid field within *command_specific_data*
0x04	VIRTIO_ADMIN_STATUS_Q_INVALID_GROUP	unsupported or invalid *group_type*
0x05	VIRTIO_ADMIN_STATUS_Q_INVALID_MEMBER	unsupported or invalid *group_member_id*
0x06	VIRTIO_ADMIN_STATUS_Q_NORESOURCE	out of internal resources: ok to retry
0x07	VIRTIO_ADMIN_STATUS_Q_TRYAGAIN	command blocks for too long: should retry
0x08-0xFFFF	-	reserved for future use

Each command uses a different *command_specific_data* and *command_specific_result* structures and the length of *command_specific_data* and *command_specific_result* depends on these structures and is described separately or is implicit in the structure description.

Before sending any group administration commands to the device, the driver needs to communicate to the device which commands it is going to use. Initially (after reset), only two commands are assumed to be used: VIRTIO_ADMIN_CMD_LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE.

Before sending any other commands for any member of a specific group to the device, the driver queries the supported commands via VIRTIO_ADMIN_CMD_LIST_QUERY and sends the commands it is capable of using via VIRTIO_ADMIN_CMD_LIST_USE.

Commands VIRTIO_ADMIN_CMD_LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE both use the following structure describing the command opcodes:

```
struct virtio_admin_cmd_list {
       /* Indicates which of the below fields were returned
       le64 device_admin_cmd_opcodes[];
};
```

This structure is an array of 64 bit values in little-endian byte order, in which a bit is set if the specific command opcode is supported. Thus, *device_admin_cmd_opcodes[0]* refers to the first 64-bit value in this array corresponding to opcodes 0 to 63, *device_admin_cmd_opcodes[1]* is the second 64-bit value corresponding to opcodes 64 to 127, etc. For example, the array of size 2 including the values 0x3 in *device_admin_cmd_opcodes[0]* and 0x1 in *device_admin_cmd_opcodes[1]* indicates that only opcodes 0, 1 and 64 are supported. The length of the array depends on the supported opcodes - it is large enough to include bits set for all supported opcodes, that is the length can be calculated by starting with the largest supported opcode adding one, dividing by 64 and rounding up. In other words, for VIRTIO_ADMIN_CMD_LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE the length of *command_specific_result* and *command_specific_data* respectively will be $DIV_ROUND_UP(max_cmd, 64) * 8$ where DIV_ROUND_UP is integer division with round up and *max_cmd* is the largest available command opcode.

The array is also allowed to be larger and to additionally include an arbitrary number of all-zero entries.

Accordingly, bits 0 and 1 corresponding to opcode 0 (VIRTIO_ADMIN_CMD_LIST_QUERY) and 1 (VIRTIO_ADMIN_CMD_LIST_USE) are always set in *device_admin_cmd_opcodes[0]* returned by VIRTIO_ADMIN_CMD_LIST_QUERY.

For the command VIRTIO_ADMIN_CMD_LIST_QUERY, *opcode* is set to 0x0. The *group_member_id* is unused. It is set to zero by driver. This command has no command specific data. The device, upon success, returns a result in *command_specific_result* in the format *struct virtio_admin_cmd_list* describing the list of group administration commands supported for the group type specified by *group_type*.

For the command VIRTIO_ADMIN_CMD_LIST_USE, *opcode* is set to 0x1. The *group_member_id* is unused. It is set to zero by driver. The *command_specific_data* is in the format *struct virtio_admin_cmd_list* describing the list of group administration commands used by the driver with the group type specified by *group_type*.

This command has no command specific result.

The driver issues the command VIRTIO_ADMIN_CMD_LIST_QUERY to query the list of commands valid for this group and before sending any commands for any member of a group.

The driver then enables use of some of the opcodes by sending to the device the command VIRTIO_ADMIN_CMD_LIST_USE with a subset of the list returned by VIRTIO_ADMIN_CMD_LIST_QUERY that is both understood and used by the driver.

If the device supports the command list used by the driver, the device completes the command with status VIRTIO_ADMIN_STATUS_OK. If the device does not support the command list (for example, if the driver is not capable to use some required commands), the device completes the command with status VIRTIO_ADMIN_STATUS_INVALID_FIELD.

Note: the driver is assumed not to set bits in device_admin_cmd_opcodes if it is not familiar with how the command opcode is used, since the device could have dependencies between command opcodes.

It is assumed that all members in a group support and are used with the same list of commands. However, for owner devices supporting multiple group types, the list of supported commands might differ between different group types.

2.12.1.1 Legacy Interfaces

In some systems, there is a need to support utilizing a legacy driver with a device that does not directly support the legacy interface. In such scenarios, a group owner device can provide the legacy interface functionality for the group member devices. The driver of the owner device can then access the legacy interface of a member device on behalf of the legacy member device driver.

For example, with the SR-IOV group type, group members (VFs) can not present the legacy interface in an I/O BAR in BAR0 as expected by the legacy pci driver. If the legacy driver is running inside a virtual machine, the hypervisor executing the virtual machine can present a virtual device with an I/O BAR in BAR0. The hypervisor intercepts the legacy driver accesses to this I/O BAR and forwards them to the group owner device (PF) using group administration commands.

The following commands support such a legacy interface functionality:

1. VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE
2. VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ
3. VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE
4. VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ

These commands are currently only defined for the SR-IOV group type and have, generally, the same effect as member device accesses through a legacy interface listed in section 4.1.4.10 except that little-endian format is assumed unconditionally.

2.12.1.1.1 VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE

This command has the same effect as writing into the virtio common configuration structure through the legacy interface. The *command_specific_data* is in the format *struct virtio_admin_cmd_legacy_common_-cfg_wr_data* describing the access to be performed.

```
struct virtio_admin_cmd_legacy_common_cfg_wr_data {
        u8 offset; /* Starting byte offset within the common configuration structure to write */
        u8 reserved[7];
        u8 data[];
};
```

For the command VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, *opcode* is set to 0x2. The *group_member_id* refers to the member device to be accessed. The *offset* refers to the offset for the write within the virtio common configuration structure, and excluding the device-specific configuration. The length of the data to write is simply the length of *data*.

No length or alignment restrictions are placed on the value of the *offset* and the length of the *data*, except that the resulting access refers to a single field and is completely within the virtio common configuration structure, excluding the device-specific configuration.

This command has no command specific result.

2.12.1.1.2 VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ

This command has the same effect as reading from the virtio common configuration structure through the legacy interface. The *command_specific_data* is in the format *struct virtio_admin_cmd_legacy_common_-cfg_rd_data* describing the access to be performed.

```
struct virtio_admin_cmd_legacy_common_cfg_rd_data {
        u8 offset; /* Starting byte offset within the common configuration structure to read */
};
```

For the command VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ, *opcode* is set to 0x3. The *group_member_id* refers to the member device to be accessed. The *offset* refers to the offset for the read from the virtio common configuration structure, and excluding the device-specific configuration.

```
struct virtio_admin_cmd_legacy_common_cfg_rd_result {
        u8 data[];
};
```

No length or alignment restrictions are placed on the value of the *offset* and the length of the *data*, except that the resulting access refers to a single field and is completely within the virtio common configuration structure, excluding the device-specific configuration.

When the command completes successfully, *command_specific_result* is in the format *struct virtio_admin_cmd_legacy_common_cfg_rd_result* returned by the device. The length of the data read is simply the length of *data*.

2.12.1.1.3 VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE

This command has the same effect as writing into the virtio device-specific configuration through the legacy interface. The *command_specific_data* is in the format *struct virtio_admin_cmd_legacy_dev_reg_wr_data* describing the access to be performed.

```
struct virtio_admin_cmd_legacy_dev_reg_wr_data {
        u8 offset; /* Starting byte offset within the device-specific configuration to write */
        u8 reserved[7];
        u8 data[];
};
```

For the command VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE, *opcode* is set to 0x4. The *group_member_id* refers to the member device to be accessed. The *offset* refers to the offset for the write within the virtio device-specific configuration. The length of the data to write is simply the length of *data*.

No length or alignment restrictions are placed on the value of the *offset* and the length of the *data*, except that the resulting access refers to a single field and is completely within the device-specific configuration.

This command has no command specific result.

2.12.1.1.4 VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ

This command has the same effect as reading from the virtio device-specific configuration through the legacy interface. The *command_specific_data* is in the format *struct virtio_admin_cmd_legacy_common_cfg_rd_data* describing the access to be performed.

```
struct virtio_admin_cmd_legacy_dev_cfg_rd_data {
        u8 offset; /* Starting byte offset within the device-specific configuration to read */
};
```

For the command VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ, *opcode* is set to 0x5. The *group_member_id* refers to the member device to be accessed. The *offset* refers to the offset for the read from the virtio device-specific configuration.

```
struct virtio_admin_cmd_legacy_dev_reg_rd_result {
        u8 data[];
};
```

No length or alignment restrictions are placed on the value of the *offset* and the length of the *data*, except that the resulting access refers to a single field and is completely within the device-specific configuration.

When the command completes successfully, *command_specific_result* is in the format *struct virtio_admin_cmd_legacy_dev_reg_rd_result* returned by the device.

The length of the data read is simply the length of *data*.

2.12.1.1.5 VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO

The driver of the owner device can send a driver notification to the member device operated using the legacy interface by executing VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE with the *offset* matching *Queue Notify* and the *data* containing a 16-bit virtqueue index to be notified.

However, as VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE is also used for slow path configuration a separate dedicated mechanism for sending such driver notifications to the member device can be made available by the owner device. For the SR-IOV group type, the optional command VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO addresses this need by returning to the driver one or more addresses which can be used to send such driver notifications. The notification address returned can be in the device memory (PCI BAR or VF BAR) of the device.

In this alternative approach, driver notifications are sent by writing a 16-bit virtqueue index to be notified, in the little-endian format, to the notification address returned by the VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO command.

Any driver notification sent through the notification address has the same effect as if it was sent using the VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE command with the *offset* matching *Queue Notify*.

This command is only defined for the SR-IOV group type.

For the command VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO, *opcode* is set to 0x6. The *group_member_id* refers to the member device to be accessed. This command does not use *command_specific_data*.

When the device supports the VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO command, the group owner device hardwires VF BAR0 to zero in the SR-IOV Extended capability.

```
struct virtio_pci_legacy_notify_info {
        u8 flags;  /* 0 = end of list, 1 = owner device, 2 = member device */
        u8 bar;    /* BAR of the member or the owner device */
        u8 padding[6];
        le64 offset; /* Offset within bar. */
};

struct virtio_admin_cmd_legacy_notify_info_result {
        struct virtio_pci_legacy_notify_info entries[4];
};
```

A *flags* value of 0x1 indicates that the notification address is of the owner device, the value of 0x2 indicates that the notification address is of the member device and the value of 0x0 indicates that all the entries starting from that entry are invalid entries in *entries*. All other values in *flags* are reserved.

The *bar* values 0x1 to 0x5 specify BAR1 to BAR5 respectively: when the *flags* is 0x1 this is specified by the Base Address Registers in the PCI header of the device, when the *flags* is 0x2 this is specified by the VF BARn registers in the SR-IOV Extended Capability of the device.

The *offset* indicates the notification address relative to BAR indicated in *bar*. This value is 2-byte aligned.

When the command completes successfully, *command_specific_result* is in the format *struct virtio_admin_cmd_legacy_notify_info_result*. The device can supply up to 4 entries each with a different notification address. In this case, any of the entries can be used by the driver. The order of the entries serves as a preference hint to the driver. The driver is expected to utilize the entries placed earlier in the array in preference to the later ones. The driver is also expected to ignore any invalid entries, as well as the end of list entry if present and any entries following the end of list.

2.12.1.1.6 Device Requirements: Legacy Interface

A device MUST either support all of, or none of VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ commands.

For VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, VIRTIO_ADMIN_CMD_LEGACY_COMMON_-CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_-DEV_CFG_READ commands, the device MUST decode and encode (respectively) the value of the *data* using the little-endian format.

For the VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_-COMMON_CFG_READ commands, the device MUST fail the command when the value of the *offset* and the length of the *data* do not refer to a single field or are not completely within the virtio common configuration excluding the device-specific configuration.

For the VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_DEV_-CFG_READ commands, the device MUST fail the command when the value of the *offset* and the length of the *data* do not refer to a single field or are not completely within the virtio device-specific configuration.

The command VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE MUST have the same effect as writing into the virtio common configuration structure through the legacy interface.

The command VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ MUST have the same effect as reading from the virtio common configuration structure through the legacy interface.

The command VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE MUST have the same effect as writing into the virtio device-specific configuration through the legacy interface.

The command VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ MUST have the same effect as reading from the virtio device-specific configuration through the legacy interface.

For the SR-IOV group type, when the owner device supports VIRTIO_ADMIN_CMD_LEGACY_COMMON_-CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, VIRTIO_ADMIN_CMD_LEGACY_-DEV_CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_-NOTIFY_INFO commands,

- the owner device and the group member device SHOULD follow the rules for the PCI Revision ID and Subsystem Device ID of the non-transitional devices documented in section 4.1.2.
- the owner device SHOULD follow the rules for the PCI Device ID of the non-transitional devices documented in section 4.1.2.
- any driver notification received by the device at any of the notification address supplied in the command result of VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO MUST function as if the device received the notification through VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE command at an offset *offset* matching *Queue Notify*.

If the device supports the VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO command,

- the device MUST also support all of VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_-WRITE and VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_READ commands.
- in the command result of VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO, the last *struct virtio_pci_-legacy_notify_info* entry MUST have *flags* of zero.
- in the command result of VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO, valid entries MUST have a *bar* which is not hardwired to zero.
- in the command result of VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO, valid entries MUST have an *offset* aligned to 2-byte.
- the device MAY support VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO with entries of the owner device or the member device or both of them.
- for the SR-IOV group type, the group owner device MUST hardwire VF BAR0 to zero in the SR-IOV Extended capability.

2.12.1.1.7 Driver Requirements: Legacy Interface

For VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE, VIRTIO_ADMIN_CMD_LEGACY_COMMON_-CFG_READ, VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_-DEV_CFG_READ commands, the driver MUST encode and decode (respectively) the value of the *data* using the little-endian format.

For the VIRTIO_ADMIN_CMD_LEGACY_COMMON_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_-COMMON_CFG_READ commands, the driver SHOULD set *offset* and the length of the *data* to refer to a single field within the virtio common configuration structure excluding the device-specific configuration.

For the VIRTIO_ADMIN_CMD_LEGACY_DEV_CFG_WRITE and VIRTIO_ADMIN_CMD_LEGACY_DEV_-CFG_READ commands, the driver SHOULD set *offset* and the length of the *data* to refer to a single field within device specific configuration.

If VIRTIO_ADMIN_CMD_LEGACY_NOTIFY_INFO command is supported, the driver SHOULD use the notification address to send all driver notifications to the device.

If within *struct virtio_admin_cmd_legacy_notify_info_result* returned by VIRTIO_ADMIN_CMD_LEGACY_-NOTIFY_INFO, the *flags* value for a specific *struct virtio_pci_legacy_notify_info* entry is 0x0, the driver MUST ignore this entry and all the following *entries*. Additionally, for all other entries, the driver MUST validate that

- the *flags* is either 0x1 or 0x2
- the *bar* corresponds to a valid BAR of either the owner or the member device, depending on the *flags*
- the *offset* is 2-byte aligned and corresponds to an address within the BAR specified by the *bar* on *flags*

, any entry which does not meet these constraints MUST be ignored by the driver.

2.12.1.2 Device and driver capabilities

Device and driver capabilities are implemented as structured groupings for specific device functionality and their related resource objects. The device exposes its supported functionality and resource object limits through an administration command, utilizing the 'self group type.' Each capability possesses a unique ID. Through an administration command, also employing the 'self group type,' the driver reports the functionality and resource object limits it intends to use. Before executing any operations related to the capabilities, the driver communicates these capabilities to the device. The driver is allowed to set the capability at any time, provided there are no pending operations at the device level associated with that capability.

The device presents the supported capability IDs to the driver as a bitmap. The driver uses the administration command to learn about the supported capabilities bitmap.

A capability consists of one or more fields, where each field can be a limit number, a bitmap, or an array of entries. In an array field, the structure depends on the specific array and the capability type. For each bitmap field, the driver sets the desired bits - but only out of those bits in a bitmap that the device has presented. The driver sets each limit number field to a desired value that is smaller than or equal to the value the device presented. Similarly, for an array field, the driver sets the desired capability entries but only out of the capability entries that the device has presented.

It is anticipated that any necessary new fields for a capability will be appended to the structure's end, ensuring both forward and backward compatibility between the device and driver. Furthermore, to avoid indefinite growth of a single capability, it is expected that new functionality will lead to the creation of new capability rather than expanding existing ones.

Capabilities are categorized into two ranges by their IDs, as listed:

Table 2.3: Capability ids

Id	Description
0x0000-0x07ff	Generic capability for all device types
0x0800-0x0fff	Device type specific capability
0x1000 - 0xFFFF	Reserved for future

Common capabilities are listed:

Table 2.5: Common capability ids

Id	Name	Description
0x0000	VIRTIO_DEV_PARTS_CAP	Device parts capability
0x0001-0x07ff	-	Generic capability for all device types

Device type specific capabilities are described separately for each device type under *Device and driver capabilities*.

The device and driver capabilities commands are currently defined for self group type.

1. VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY
2. VIRTIO_ADMIN_CMD_DEVICE_CAP_GET
3. VIRTIO_ADMIN_CMD_DRIVER_CAP_SET

2.12.1.2.1 VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY

This command queries the bitmap of capability ids listed in 2.3.

For the command VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY, *opcode* is set to 0x7. *group_member_id* is set to zero.

This command has no command specific data.

```
struct virtio_admin_cmd_query_cap_id_result {
        le64 supported_caps[];
};
```

When the command completes successfully, *command_specific_result* is in the format *struct virtio_admin_cmd_query_cap_id_result*.

supported_caps is an array of 64 bit values in little-endian byte order, in which a bit is set if the specific capability is supported. Thus, *supported_caps[0]* refers to the first 64-bit value in this array corresponding to capability ids 0 to 63, *supported_caps[1]* is the second 64-bit value corresponding to capability ids 64 to 127, etc. For example, the array of size 2 including the values 0x3 in *supported_caps[0]* and 0x1 in *supported_caps[1]* indicates that only capability id 0, 1 and 64 are supported. The length of the array depends on the supported capabilities - it is large enough to include bits set for all supported capability ids, that is the length can be calculated by starting with the largest supported capability id adding one, dividing by 64 and rounding up. In other words, for VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY the length of *command_specific_result* will be $DIV_ROUND_UP(max_cap_id, 64) * 8$ where DIV_ROUND_UP is integer division with round up and *max_cap_id* is the largest available capability id.

The array is also allowed to be larger and to additionally include an arbitrary number of all-zero entries.

2.12.1.2.2 VIRTIO_ADMIN_CMD_DEVICE_CAP_GET

This command gets the device capability for the specified capability id *id*.

For the command VIRTIO_ADMIN_CMD_DEVICE_CAP_GET, *opcode* is set to 0x8. *group_member_id* is set to zero.

command_specific_data is in format *struct virtio_admin_cmd_cap_get_data*.

```
struct virtio_admin_cmd_cap_get_data {
        le16 id;
        u8 reserved[6];
};
```

id refers to the capability id listed in 2.3. *reserved* is reserved for future use and set to zero.

```
struct virtio_admin_cmd_cap_get_result {
        u8 cap_specific_data[];
};
```

When the command completes successfully, *command_specific_result* is in the format *struct virtio_admin_cmd_cap_get_result* responded by the device. Each capability uses different capability specific *cap_specific_data* and is described separately.

2.12.1.2.3 VIRTIO_ADMIN_CMD_DRIVER_CAP_SET

This command sets the driver capability, indicating to the device which capability the driver uses. The driver can set a resource object limit capability that is smaller than or equal to the value published by the device capability. If the capability is a set of flags, the driver sets the flag bits that are set in the device capability; the driver does not set any flag bits that are not set by the device.

For the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET, *opcode* is set to 0x9. *group_member_id* is set to zero. The *command_specific_data* is in the format *struct virtio_admin_cmd_cap_set_data*.

```
struct virtio_admin_cmd_cap_set_data {
        le16 id;
        u8 reserved[6];
        u8 cap_specific_data[];
};
```

id refers to the capability id listed in 2.3. *reserved* is reserved for future use and set to zero.

There is no command specific result. When the command completes successfully, the driver capability is updated to the values supplied in *cap_specific_data*.

2.12.1.2.4 Device Requirements: Device and driver capabilities

If the device supports capabilities, it MUST support the commands VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY, VIRTIO_ADMIN_CMD_DRIVER_CAP_SET, and VIRTIO_ADMIN_CMD_DEVICE_CAP_GET.

For the VIRTIO_ADMIN_CMD_DRIVER_CAP_SET command,

- the device MUST support the setting of resource object limit driver capability to a value that is same as or smaller than the one reported in the device capability,
- the device MUST support the setting of capability flags bits to all or fewer bits than the one reported in the device capability;

this is applicable unless specific capability fields are explicitly stated as non-writable in the VIRTIO_ADMIN_CMD_DEVICE_CAP_GET command.

The device MAY complete the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET with *status* set to VIRTIO_ADMIN_STATUS_EINVAL, if the capability resource object limit is larger than the value reported by the device's capability, or the capability flag bit is set, which is not set in the device's capability.

The device MUST complete the commands VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY, VIRTIO_ADMIN_CMD_DRIVER_CAP_GET, and VIRTIO_ADMIN_CMD_DRIVER_CAP_SET with *status* set to VIRTIO_ADMIN_STATUS_EINVAL if the commands are not for the self group type.

The device SHOULD complete the commands VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY, VIRTIO_ADMIN_CMD_DRIVER_CAP_GET, VIRTIO_ADMIN_CMD_DRIVER_CAP_SET with *status* set to VIRTIO_ADMIN_STATUS_EINVAL if the commands are not for the self group type.

The device SHOULD complete the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET with *status* set to VIRTIO_ADMIN_STATUS_EBUSY if the command requests to disable a capability while the device still has valid resource objects related to the capability being disabled.

The device SHOULD complete the comands VIRTIO_ADMIN_CMD_DEVICE_CAP_GET and VIRTIO_ADMIN_CMD_DRIVER_CAP_SET with *status* set to VIRTIO_ADMIN_STATUS_ENXIO if the capability id is not reported in command VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY.

Upon a device reset, the device MUST reset all driver capabilities.

The device SHOULD treat the driver resource limits as zero if the driver has not set such capability, unless otherwise explicitly stated.

2.12.1.2.5 Driver Requirements: Device and driver capabilities

The driver MUST send the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET before using any resource objects that depend on such a capability.

In VIRTIO_ADMIN_CMD_DRIVER_CAP_SET command, the driver MUST NOT set

- the resource object limit value larger than the value reported by the device in the command VIRTIO_ADMIN_CMD_DEVICE_CAP_GET,
- flags bits which was not reported by the device in the command VIRTIO_ADMIN_CMD_DEVICE_CAP_GET,
- array entries not reported by the device in the command VIRTIO_ADMIN_CMD_DEVICE_CAP_GET.

The driver MUST NOT disable any of the driver capability using the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET when related resource objects are created but not destroyed.

2.12.1.3 Device resource objects

Providing certain functionality consumes limited device resources such as memory, processing units, buffer memory, or end-to-end credits. A device may support multiple types of resource objects, each controlling different device functionality. To manage this, virtio provides *Device resource objects* that the driver can create, modify, and destroy using administration commands with the self group type. Creating and destroying a resource object consume and release device resources, respectively. The device resource object query command returns the resource object as maintained by the device.

For each resource type, the number of resource objects that can be created is reported by the device as part of a device capability 2.12.1.2. The driver reports the desired (same or lower) number of resource objects as part of a driver capability 2.12.1.2. For each device object type, resource object limit is defined by field *limit* using *Device and driver capabilities*.

```
le32 limit; /* maximum resource id = limit - 1 */
```

Each resource object has a unique resource object ID - a driver-assigned number in the range of 0 to *limit - 1*, where the *limit* is the maximum number set by the driver for this resource object type. These resource IDs are unique within each resource object type. The driver assigns the resource ID when creating a device resource object. Once the resource object is successfully created, subsequent resource modification, query, and destroy commands use this resource object ID. No two resource objects share the same ID. Destroying a resource object allows for the reuse of its ID for another resource object of the same type.

A valid resource object id is *limit - 1*. For example, when a device reports a *limit = 10* capability for a resource object, and drivers sets *limit = 8*, the valid resource object id range for the device and the driver is 0 to 7

for all the resource object commands. In this example, the driver can only create 8 resource objects of a specified type.

A resource object of one type may depend on the resource object of another type. Such dependency between resource objects is established by referring to the unique resource ID in the administration commands. For example, a driver creates a resource object identified by ID A of one type, then creates another resource object identified by ID B of a different type, which depends on resource object A. This dependency establishes the lifecycle of these resource objects. The driver that creates the dependent resource object must destroy the resource objects in the exact reverse order of their creation. In this example, the driver would destroy resource object B before destroying resource object A.

Some resource object types are generic, common across multiple devices. Others are specific for one device type.

Resource object type	Description
0x000-0x1ff	Generic resource object type common across all devices
0x200-0x4ff	Device type specific resource object
0x500-0xffff	Reserved for future use

Following generic resource objects are defined which are described separately.

Resource object type	Name	Description
0x000	VIRTIO_RESOURCE_OBJ_-DEV_PARTS	Device parts object, see 2.12.1.4.2
0x001-0x1ff	-	Generic resource object range reserved

When the device resets, it starts with zero resources of each type; the driver can create resources up to the published *limit*. The driver can destroy and recreate the resource one or multiple times. Upon device reset, all resource objects created by the driver are destroyed within the device.

Following administration commands control device resource objects, they are supported for the self group type, occasionally some resource objects can be created for the SR-IOV group type as well. Such sr-iov group type specific resource objects are listed where such objects is defined.

1. VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE
2. VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY
3. VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY
4. VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY

Each resource object administration command uses a common header *struct virtio_admin_cmd_resource_-obj_cmd_hdr*.

```
struct virtio_admin_cmd_resource_obj_cmd_hdr {
        le16 type;
        u8 reserved[2];
        le32 id; /* Indicates unique resource object id per resource object type */
};
```

type refers to the device resource object type. *id* uniquely identifies the resource object of a specified *type*.

2.12.1.3.1 VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE

This command creates the specified resource object of *type* identified by the resource id *id*. The valid range of *id* is defined by the device in the related device capability. The driver assigns the unique *id* for the resource for the specified *type*.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, *opcode* is set to 0xa. *group_member_id* is set to zero for self-group type and set to the member device to be accessed for the SR-IOV group type. The *command_specific_data* is in the format *struct virtio_admin_cmd_resource_obj_create_data*. *resource_obj_specific_data* refers to the resource object specific data. Each resource uses a different *resource_obj_specific_data* and is described separately.

flags is reserved for future extension for optional resource object attributes and is set to 0. Each resource object uses a different value for *flags* and it is described separately.

```
struct virtio_admin_cmd_resource_obj_create_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        le64 flags;
        u8 resource_obj_specific_data[];
};
```

When the command completes successfully, the resource object is created by the device and the device can immediately begin using it. This command has no command specific result.

2.12.1.3.2 VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY

This command modifies the attributes of an existing device resource object. For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, *opcode* is set to 0xb. The *command_specific_data* is in the format *struct virtio_admin_cmd_resource_modify_data*. *group_member_id* is set to zero for self-group type and set to the member device to be accessed for the SR-IOV group type. *id* identifies the resource object of type *type* whose attributes to modify. This command modifies the attributes supplied in *resource_obj_specific_data*.

```
struct virtio_admin_cmd_resource_modify_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        le64 flags;
        u8 resource_obj_specific_data[];
};
```

This command has no command specific result. When the command completes successfully, attributes of the resource object is set to the values supplied in *resource_obj_specific_data*.

2.12.1.3.3 VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY

This command queries attributes of the existing resource object. For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, *opcode* is set to 0xc. *group_member_id* is set to zero for self-group type and set to the member device to be accessed for the SR-IOV group type. The *command_specific_data* is in the format *struct virtio_admin_cmd_resource_obj_query_data*. *id* identifies the existing resource object of type *type* whose attributes to query.

```
struct virtio_admin_cmd_resource_obj_query_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        le64 flags;
};
```

```
struct virtio_admin_cmd_resource_obj_query_result {
        u8 resource_obj_specific_result[];
};
```

command_specific_result is in the format *virtio_admin_cmd_resource_obj_query_result*.

When the command completes successfully, the attributes of the specified resource object are are set in *resource_obj_specific_data*.

2.12.1.3.4 VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY

This command destroys the previously created device resource object. For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY, *opcode* is set to 0xd. The *command_specific_data* is in the format

struct virtio_admin_cmd_resource_obj_cmd_hdr. *group_member_id* is set to zero for self-group type and set to the member device to be accessed for the SR-IOV group type. *id* identifies the existing resource object of type *type*.

This command destroys the specified resource object of *type* identified by *id*, which is previously created using VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE command.

This command has no command specific result. When the command completes successfully, the resource object is destroyed from the device.

2.12.1.3.5 Device Requirements: Device resource objects

The device SHOULD complete the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with *status* set to VIRTIO_ADMIN_STATUS_EEXIST if a resource object already exists with supplied resource *id* for the specified *type*.

The device SHOULD complete the commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_QUERY and VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY with *status* set to VIRTIO_ADMIN_STATUS_ENXIO if the specified resource object does not exist.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_ENOSPC for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE if the device fail to create the resource object.

The device SHOULD complete the commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY or VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY commands with *status* set to VIRTIO_ADMIN_STATUS_EBUSY if other resource objects depend on the resource object being modified or destroyed.

The device MUST allow recreating the resource object using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE which was previously destroyed using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY respectively without undergoing a device reset.

The device SHOULD allow creating the resource object using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with any resource id as long as the resource object is not created.

The device MAY fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE even if the resources within the device have not reached up to the *max_limit* but the device MAY have reached an internal limit.

When a capability represents a number of resource objects, the device SHOULD allow creating as many resource objects as represented by the driver capability.

The device MUST NOT have any side effects on the resource object when the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY fails.

The device MUST complete the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY with *resource_obj_specific_data* which is matching the *resource_obj_specific_data* of last VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE or VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY command.

On device reset, the device MUST destroy all the resource objects which have been created.

2.12.1.3.6 Driver Requirements: Device resource objects

The driver MUST not create a second resource object of the same type with same ID using command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE before destroying the previously created resource object.

The driver MUST NOT create more resource objects of a specified *type* using command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE than the maximum limit set by the driver capability.

The driver SHOULD NOT modify, query and destroy the resource object which is already destroyed previously by the driver.

The driver SHOULD NOT destroy the resource object on which other resource objects are depending; the driver SHOULD destroy all the resource objects which do not depend on other resource objects.

The driver MUST NOT set the capability related to the resource objects if the resource objects have been created using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE and not yet destroyed.

The driver MUST send the command VIRTIO_ADMIN_CMD_DRIVER_CAP_SET before using any resources related to such capability.

2.12.1.4 Device parts

In some systems, there is a need to capture the state of all or part of a device and subsequently restore either the same device or a different one to this captured state. A group owner device can support administration commands to facilitate these get and set operations for the group member devices.

For example, a hypervisor can use the administration commands to capture parts of the device state and save the result as part of a VM snapshot. Later, the hypervisor can retrieve the snapshot and use the administration commands to restore parts of a device to resume VM operation.

As another example, these commands can be used to facilitate VM migration by the hypervisor: one (source) hypervisor can get parts of a device and send the results to another (destination) hypervisor, which will in turn set (restore) parts of (another) device to resume the VM operation on the destination.

The device comprises many device parts which the driver can get and set. Administration commands are provided to either get and set all the device parts at once, or to get the device parts metadata that indicates which device parts are present, and later to get and set specific device parts. To get and set the device parts or their metadata, the driver first creates a device parts resource object, indicating whether the object should handle get or set operations but not both simultaneously. The device and the driver indicate the device parts resource objects' limit using the capability VIRTIO_DEV_PARTS_CAP.

The device can be stopped to prevent device parts from changing. When the device is stopped, it does not initiate any transport requests. For instance, the device refrains from sending any configuration or virtqueue notifications and does not access any virtqueues or the driver's buffer memory. While the driver may remain active and continue to send notifications to the device, potentially updating some device parts, the device itself will not initiate any transport requests.

2.12.1.4.1 VIRTIO_DEV_PARTS_CAP

The capability VIRTIO_DEV_PARTS_CAP indicates the device parts resource objects limit. *cap_specific_data* is in the format *struct virtio_dev_parts_cap*.

```
struct virtio_dev_parts_cap {
        u8 get_parts_resource_objects_limit;
        u8 set_parts_resource_objects_limit;
};
```

get_parts_resource_objects_limit indicates the supported device parts resource objects for retrieving the device parts. *set_parts_resource_objects_limit* indicates the supported device parts resource objects for restoring the device parts.

2.12.1.4.2 VIRTIO_RESOURCE_OBJ_DEV_PARTS

A device parts resource object is used to either get or set the device parts. Before performing any get or set operation for the device parts, the driver creates the device parts resource object VIRTIO_RESOURCE_OBJ_DEV_PARTS using the administration command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE. The driver indicates the intended purpose (get or set) at the time of creating the device parts resource object. For the device parts resource object, both *resource_obj_specific_data* and *resource_obj_specific_result* are in the format *struct virtio_resource_obj_dev_parts*.

```
struct virtio_resource_obj_dev_parts {
        u8 type;
#define VIRTIO_RESOURCE_OBJ_DEV_PARTS_TYPE_GET 0
#define VIRTIO_RESOURCE_OBJ_DEV_PARTS_TYPE_SET 1
        u8 reserved[7];
};
```

When *type* is set to VIRTIO_RESOURCE_OBJ_DEV_PARTS_TYPE_GET, the driver can use the object to capture the device parts and the metadata of these device parts. When *type* is set to VIRTIO_RESOURCE_-OBJ_DEV_PARTS_TYPE_SET, the driver can use the object to restore the device parts.

2.12.1.4.3 Device parts handling commands

The owner driver uses the following resource object handling administration commands. These commands are only used for the device parts resource object after the driver creates the VIRTIO_RESOURCE_OBJ_-DEV_PARTS object. These commands are currently only defined for the SR-IOV group type:

1. VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET
2. VIRTIO_ADMIN_CMD_DEV_PARTS_GET
3. VIRTIO_ADMIN_CMD_DEV_PARTS_SET

2.12.1.4.3.1 VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET

This command obtains the metadata of the device parts. This metadata includes the maximum size of the device parts, the count of device parts, and a list of the device part headers.

For the command VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET, *opcode* is set to 0xe. The *command_specific_data* is in the format *struct virtio_admin_cmd_dev_parts_metadata_data*. *group_member_id* refers to the member device to be accessed. The resource object *type* in the *hdr* is set to VIRTIO_RESOURCE_OBJ_DEV_PARTS and *id* is set to the ID of the device parts resource object.

```
struct virtio_admin_cmd_dev_parts_metadata_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        u8 type;
        u8 reserved[7];
};

#define VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_SIZE 0
#define VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_COUNT 1
#define VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_LIST  2

struct virtio_admin_cmd_dev_parts_metadata_result {
        union {
                struct {
                        le32 size;
                        le32 reserved;
                } parts_size;
                struct {
                        le32 count;
                        le32 reserved;
                } hdr_list_count;
                struct {
                        le32 count;
                        le32 reserved;
                        struct virtio_dev_part_hdr hdrs[];
                } hdr_list;
        };
};
```

When the command completes successfully, the *command_specific_result* is in the format *struct virtio_-admin_cmd_dev_parts_metadata_result*.

When *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_SIZE, the device responds with *parts_size*. *parts_size.size* indicates the maximum size in bytes for all the device parts.

When *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_COUNT, the device responds with *hdr_list_count.count*. The *hdr_list_count.count* indicates an count of *struct virtio_dev_part_hdr* metadata entries that the device can provide when the *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_META-DATA_TYPE_LIST in a subsequent VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET command.

When *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_TYPE_LIST, the device responds with *hdr_list*. *hdr_list* indicates the device parts metadata.

reserved is reserved and set to 0.

The command responds with the *status* VIRTIO_ADMIN_STATUS_ENOMEM when the size of *command_specific_result* is not sufficient enough for the response.

2.12.1.4.3.2 VIRTIO_ADMIN_CMD_DEV_PARTS_GET

This command captures the device parts. For the command VIRTIO_ADMIN_CMD_DEV_PARTS_GET, *opcode* is set to 0xf. The *command_specific_data* is in the format *struct virtio_admin_cmd_dev_parts_get_data*. *group_member_id* refers to the member device to be accessed. The resource object *type* in the *hdr* is set to VIRTIO_RESOURCE_OBJ_DEV_PARTS and *id* is set to the ID of the device parts resource object.

```
struct virtio_admin_cmd_dev_parts_get_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        u8 type;
        u8 reserved[7];
        struct virtio_dev_part_hdr hdr_list[];
};

#define VIRTIO_ADMIN_CMD_DEV_PARTS_GET_TYPE_SELECTED 0
#define VIRTIO_ADMIN_CMD_DEV_PARTS_GET_TYPE_ALL 1

struct virtio_admin_cmd_dev_parts_get_result {
        struct virtio_dev_part parts[];
};
```

When the driver wants to capture specific device parts, *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_GET_TYPE_SELECTED and *hdr_list* is set to the device parts of interest.

When the driver wants to retrieve all the device parts, *type* is set to VIRTIO_ADMIN_CMD_DEV_PARTS_GET_TYPE_ALL, and *hdr_list* is empty.

reserved is reserved and set to 0.

When the command completes successfully, the *command_specific_result* is in the format *struct virtio_admin_cmd_dev_parts_get_result*, containing either the selected device parts or all the device parts.

If the requested device part does not exist, the device skips the device part without any error.

2.12.1.4.3.3 VIRTIO_ADMIN_CMD_DEV_PARTS_SET

This command sets one or multiple device parts. For the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET, *opcode* is set to 0x10. The *group_member_id* refers to the member device to be accessed. The resource object *type* in the *hdr* is set to VIRTIO_RESOURCE_OBJ_DEV_PARTS and *id* is set to the ID of the device parts resource object.

```
struct virtio_admin_cmd_dev_parts_set_data {
        struct virtio_admin_cmd_resource_obj_cmd_hdr hdr;
        struct virtio_dev_part parts[];
};
```

The *command_specific_data* is in the format *struct virtio_admin_cmd_dev_parts_set_data*.

This command has no command specific result.

The driver stops the device before setting any device parts.

When the command completes successfully, the device has updated device parts to the value supplied in *virtio_admin_cmd_dev_parts_set_data*.

The device parts set by this command take effect when the device is resumed using the VIRTIO_ADMIN_CMD_DEV_MODE_SET command.

When the command fails with a status other than VIRTIO_ADMIN_STATUS_OK, the device does not have any side effects.

2.12.1.4.3.4 VIRTIO_ADMIN_CMD_DEV_MODE_SET

This command either stops the device from initiating any transport requests or resumes the device operation. For the command VIRTIO_ADMIN_CMD_DEV_MODE_SET, *opcode* is set to 0x11. *group_member_id* indicates the member device to be accessed.

The *command_specific_data* is in the format *struct virtio_admin_cmd_dev_mode_set_data*.

```
struct virtio_admin_cmd_dev_mode_set_data {
        u8 flags;
};

#define VIRTIO_ADMIN_CMD_DEV_MODE_F_STOPPED 0
```

This command has no command specific result.

When the command completes successfully and if the *flags* field is set to VIRTIO_ADMIN_CMD_DEV_-MODE_F_STOPPED (bit 0), the device is stopped. When the device is stopped, the device stops initiating all transport communications, which includes:

1. stopping configuration change notifications
2. stopping all virtqueue notifications
3. stops accessing all virtqueues and the driver buffer memory

After the device is stopped, the device parts remain unchanged unless the driver initiates any transport requests.

When the device is stopped, it writes back any associated descriptors for all observed buffers to prevent out-of-order processing if the device is resumed.

When the command completes successfully and if the *flags* field is set to zero, the device resumes its operation. If the command completes with an error, it does not produce any side effects on the device.

2.12.1.4.4 Device parts order

Device parts are usually captured and restored using get and set administration commands respectively; when multiple device parts are captured or restored, they are arranged in the specific order listed:

Some of the device parts do not need to be written to the device when restored, such device parts are listed as *O*. When a such an optional device part is exchanged using *struct virtio_dev_part*, it is marked as optional by setting VIRTIO_DEV_PART_F_OPTIONAL(bit 0) in the *flags*.

Table 2.8: Device parts order

Part name	Optional	Mandatory preceding parts
VIRTIO_DEV_PART_DEV_FEATURES	O	Nil
VIRTIO_DEV_PART_DRV_FEATURES	-	Nil
VIRTIO_DEV_PART_PCI_COMMON_CFG	-	VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES
VIRTIO_DEV_PART_DEVICE_STATUS	-	VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES, VIRTIO_DEV_PART_PCI_COMMON_CFG
VIRTIO_DEV_PART_VQ_CFG	-	VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES, VIRTIO_DEV_PART_PCI_COMMON_CFG, VIRTIO_DEV_PART_DEVICE_STATUS
VIRTIO_DEV_PART_VQ_NOTIFY_CFG	-	VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES, VIRTIO_DEV_PART_PCI_COMMON_CFG, VIRTIO_DEV_PART_DEVICE_STATUS, VIRTIO_DEV_PART_VQ_CFG

2.12.1.4.4.1 Device Requirements: Device parts

A device MUST either support all of, or none of VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET, VIRTIO_ADMIN_CMD_DEV_PARTS_GET, VIRTIO_ADMIN_CMD_DEV_PARTS_SET, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, and VIRTIO_ADMIN_CMD_DEV_MODE_SET commands, where resource commands apply to the resource object VIRTIO_RESOURCE_OBJ_DEV_PARTS.

The device MUST support getting the device parts multiple times with the command VIRTIO_ADMIN_CMD_DEV_PARTS_GET.

When there are multiple device parts in the command VIRTIO_ADMIN_CMD_DEV_PARTS_GET, the device MUST respond the device parts in the same order as listed in the table Device parts order.

The device SHOULD respond with an error status for the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET if the device is not stopped.

The device MUST support the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET, allowing the same or different device parts to be set multiple times.

The device MUST respond with an error for the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET, if there is a mismatch between the device part length supplied in the VIRTIO_ADMIN_CMD_DEV_PARTS_SET and the device part length in the device.

The device MUST NOT set the device part VIRTIO_DEV_PART_DEV_FEATURES in the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET; instead, it must verify that the device features supplied in VIRTIO_DEV_PART_DEV_FEATURES match those the device has.

The device may ignore the setting of a device part that has the VIRTIO_DEV_PART_F_OPTIONAL bit set.

For the SR-IOV group type, when the device is stopped using the command VIRTIO_ADMIN_CMD_DEV_MODE_SET,

- the device MUST not initiate any PCI transaction,

- the device MUST finish all the outstanding PCI transactions before completing the command VIRTIO_ADMIN_CMD_DEV_MODE_SET,
- the device MUST write any associated descriptors to the driver memory for all the observed buffers,
- the device MUST accept driver notifications and the device MAY update any device parts,
- the device MUST respond with valid values for PCI read requests,
- the device MUST operate in the same way for the PCI architected interfaces regardless of the device mode.
- the device MUST not generate any PCI PME.

When the device is stopped,

- the device MUST not access any virtqueue memory or any memory referred by the virtqueue.
- the device MUST not generate any configuration change notification or any virtqueue notification.

For the SR-IOV group type,

- the device MUST respond to the commands VIRTIO_ADMIN_CMD_DEV_MODE_SET, VIRTIO_ADMIN_CMD_DEV_PARTS_SET after the member device completes FLR, if the FLR is in progress on the device when the device receives any of these commands.
- the member device MUST respond to the commands VIRTIO_ADMIN_CMD_DEV_MODE_SET and VIRTIO_ADMIN_CMD_DEV_PARTS_SET after the device reset completes in the device, if the device reset is in progress when the device receives any of these commands.
- the member device MUST respond to commands VIRTIO_ADMIN_CMD_DEV_MODE_SET and VIRTIO_ADMIN_CMD_DEV_PARTS_SET after the device power management state transition completes on the device, if the power management state transition is in progress when the device receives any of these commands.

When the *flags* is set to VIRTIO_ADMIN_CMD_DEV_MODE_FLAGS_STOPPED in the command VIRTIO_ADMIN_CMD_DEV_MODE_SET, and if the device is already stopped before, the device MUST complete the command successfully.

When the VIRTIO_ADMIN_CMD_DEV_MODE_FLAGS_STOPPED *flags* clear, in the command VIRTIO_ADMIN_CMD_DEV_MODE_SET, and if the device is not stopped before, the device MUST complete the command successfully.

For the SR-IOV group type, the device MUST clear all the device parts to the default value when the member device is reset or undergo an PCI FLR.

The device MAY NOT respond to the selected device part in *hdr_list* in the command VIRTIO_ADMIN_CMD_DEV_PARTS_GET if the device part is invalid in the device.

For the commands VIRTIO_ADMIN_CMD_DEV_PARTS_GET and VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET, when the device responds with:

- VIRTIO_DEV_PART_DRV_FEATURES or VIRTIO_DEV_PART_PCI_COMMON_CFG, it MUST be preceded by VIRTIO_DEV_PART_DEV_FEATURES.
- VIRTIO_DEV_PART_PCI_COMMON_CFG, it MUST be preceded by VIRTIO_DEV_PART_DEV_FEATURES.
- VIRTIO_DEV_PART_PCI_COMMON_CFG, it MUST be preceded by VIRTIO_DEV_PART_DEV_FEATURES and VIRTIO_DEV_PART_DRV_FEATURES.
- VIRTIO_DEV_PART_DEV_CFG, it MUST be preceded by VIRTIO_DEV_PART_DEV_FEATURES.
- VIRTIO_DEV_PART_DRV_CFG, it be preceded by VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES and VIRTIO_DEV_PART_DEV_CFG.
- VIRTIO_DEV_PART_DEVICE_STAtUS, it is preceded by VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_DRV_FEATURES, and VIRTIO_DEV_PART_DEV_CFG.

When the device receives a VIRTIO_ADMIN_CMD_DEV_PARTS_SET command containing the parts VIRTIO_DEV_PART_DEV_FEATURES, VIRTIO_DEV_PART_PCI_COMMON_CFG and VIRTIO_DEV_PART_DEV_CFG, the device SHOULD only verify that the provided configuration is correct but SHOULD NOT apply it, especially for the fields that are designated as read-only and invariant. This ensures that the device respects the immutability of certain configuration aspects while still performing necessary validation checks.

2.12.1.4.4.2 Driver Requirements: Device parts

The driver MUST set the mode to VIRTIO_ADMIN_CMD_DEV_MODE_F_STOPPED in the command VIRTIO_ADMIN_CMD_DEV_MODE_SET before setting parts using the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET.

When there are multiple device parts in the command VIRTIO_ADMIN_CMD_DEV_PARTS_SET, the driver MUST set the device parts in the same order as listed in the table Device parts order.

For the SR-IOV group type, the driver SHOULD NOT access the device configuration space described in section 2.5 when the device is stopped.

The driver SHOULD allocate sufficient response buffer to receive all the device parts metadata in the command VIRTIO_ADMIN_CMD_DEV_PARTS_METADATA_GET.

The driver SHOULD allocate sufficient response buffer to receive all the device parts in the command VIRTIO_ADMIN_CMD_DEV_PARTS_GET.

2.12.1.5 Device Requirements: Group administration commands

The device MUST validate *opcode*, *group_type* and *group_member_id*, and if any of these has an invalid or unsupported value, set *status* to VIRTIO_ADMIN_STATUS_EINVAL and set *status_qualifier* accordingly:

- if *group_type* is invalid, *status_qualifier* MUST be set to VIRTIO_ADMIN_STATUS_Q_INVALID_GROUP;
- otherwise, if *opcode* is invalid, *status_qualifier* MUST be set to VIRTIO_ADMIN_STATUS_Q_INVALID_OPCODE;
- otherwise, if *group_member_id* is used by the specific command and is invalid, *status_qualifier* MUST be set to VIRTIO_ADMIN_STATUS_Q_INVALID_MEMBER.

If a command completes successfully, the device MUST set *status* to VIRTIO_ADMIN_STATUS_OK.

If a command fails, the device MUST set *status* to a value different from VIRTIO_ADMIN_STATUS_OK.

If *status* is set to VIRTIO_ADMIN_STATUS_EINVAL, the device state MUST NOT change, that is the command MUST NOT have any side effects on the device, in particular the device MUST NOT enter an error state as a result of this command.

If a command fails, the device state generally SHOULD NOT change, as far as possible.

The device MAY enforce additional restrictions and dependencies on opcodes used by the driver and MAY fail the command VIRTIO_ADMIN_CMD_LIST_USE with *status* set to VIRTIO_ADMIN_STATUS_EINVAL and *status_qualifier* set to VIRTIO_ADMIN_STATUS_Q_INVALID_FIELD if the list of commands used violate internal device dependencies.

If the device supports multiple group types, commands for each group type MUST operate independently of each other, in particular, the device MAY return different results for VIRTIO_ADMIN_CMD_LIST_QUERY for different group types.

After reset, if the device supports a given group type and before receiving VIRTIO_ADMIN_CMD_LIST_USE for this group type the device MUST assume that the list of legal commands used by the driver consists of the two commands VIRTIO_ADMIN_CMD_LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE.

After completing VIRTIO_ADMIN_CMD_LIST_USE successfully, the device MUST set the list of legal commands used by the driver to the one supplied in *command_specific_data*.

The device MUST validate commands against the list used by the driver and MUST fail any commands not in the list with *status* set to VIRTIO_ADMIN_STATUS_EINVAL and *status_qualifier* set to VIRTIO_ADMIN_-STATUS_Q_INVALID_OPCODE.

The list of supported commands reported by the device MUST NOT shrink (but MAY expand): after reporting a given command as supported through VIRTIO_ADMIN_CMD_LIST_QUERY the device MUST NOT later report it as unsupported. Further, after a given set of commands has been used (via a successful VIRTIO_-ADMIN_CMD_LIST_USE), then after a device or system reset the device SHOULD complete successfully any following calls to VIRTIO_ADMIN_CMD_LIST_USE with the same list of commands; if this command VIRTIO_ADMIN_CMD_LIST_USE fails after a device or system reset, the device MUST not fail it solely because of the command list used. Failure to do so would interfere with resuming from suspend and error recovery. Exceptions MAY apply if the system configuration assures, in some way, that the driver does not cache the previous value of VIRTIO_ADMIN_CMD_LIST_USE, such as in the case of a firmware upgrade or downgrade.

When processing a command with the SR-IOV group type, if the device does not have an SR-IOV Extended Capability or if *VF Enable* is clear then the device MUST fail all commands with *status* set to VIRTIO_AD-MIN_STATUS_EINVAL and *status_qualifier* set to VIRTIO_ADMIN_STATUS_Q_INVALID_GROUP; other-wise, if *group_member_id* is not between 1 and *NumVFs* inclusive, the device MUST fail all commands with *status* set to VIRTIO_ADMIN_STATUS_EINVAL and *status_qualifier* set to VIRTIO_ADMIN_STATUS_Q_-INVALID_MEMBER; *NumVFs*, *VF Migration Capable* and *VF Enable* refer to registers within the SR-IOV Extended Capability as specified by [PCIe].

2.12.1.6 Driver Requirements: Group administration commands

The driver MAY discover whether device supports a specific group type by issuing VIRTIO_ADMIN_CMD_-LIST_QUERY with the matching *group_type*.

The driver MUST issue VIRTIO_ADMIN_CMD_LIST_USE and wait for it to be completed with status VIR-TIO_ADMIN_STATUS_OK before issuing any commands (except for the initial VIRTIO_ADMIN_CMD_-LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE).

The driver MAY issue VIRTIO_ADMIN_CMD_LIST_USE any number of times but MUST NOT issue VIR-TIO_ADMIN_CMD_LIST_USE commands if any other command has been submitted to the device and has not yet completed processing by the device.

The driver SHOULD NOT set bits in device_admin_cmd_opcodes if it is not familiar with how the command opcode is used, as dependencies between command opcodes might exist.

The driver MUST NOT request (via VIRTIO_ADMIN_CMD_LIST_USE) the use of any commands not pre-viously reported as supported for the same group type by VIRTIO_ADMIN_CMD_LIST_QUERY.

The driver MUST NOT use any commands for a given group type before sending VIRTIO_ADMIN_CMD_-LIST_USE with the correct list of command opcodes and group type.

The driver MAY block use of VIRTIO_ADMIN_CMD_LIST_QUERY and VIRTIO_ADMIN_CMD_LIST_USE by issuing VIRTIO_ADMIN_CMD_LIST_USE with respective bits cleared in *command_specific_data*.

The driver MUST handle a command error with a reserved *status* value in the same way as *status* set to VIRTIO_ADMIN_STATUS_EINVAL (except possibly for different error reporting/diagnostic messages).

The driver MUST handle a command error with a reserved *status_qualifier* value in the same way as *sta-tus_qualifier* set to VIRTIO_ADMIN_STATUS_Q_INVALID_COMMAND (except possibly for different error reporting/diagnostic messages).

When sending commands with the SR-IOV group type, the driver specify a value for *group_member_id* between 1 and *NumVFs* inclusive, the driver MUST also make sure that as long as any such command is outstanding, *VF Migration Capable* is clear and *VF Enable* is set; *NumVFs*, *VF Migration Capable* and *VF Enable* refer to registers within the SR-IOV Extended Capability as specified by [PCIe].

2.13 Administration Virtqueues

An administration virtqueue of an owner device is used to submit group administration commands. An owner device can have more than one administration virtqueue.

If VIRTIO_F_ADMIN_VQ has been negotiated, an owner device exposes one or more administration virtqueues. The number and locations of the administration virtqueues are exposed by the owner device in a transport specific manner.

The driver enqueues requests to an arbitrary administration virtqueue, and they are used by the device on that same virtqueue. It is the responsibility of the driver to ensure strict request ordering for commands, because they will be consumed with no order constraints. For example, if consistency is required then the driver can wait for the processing of a first command by the device to be completed before submitting another command depending on the first one.

Administration virtqueues are used as follows:

- The driver submits the command using the *struct virtio_admin_cmd* structure using a buffer consisting of two parts: a device-readable one followed by a device-writable one.
- the device-readable part includes fields from *opcode* through *command_specific_data*.
- the device-writeable buffer includes fields from *status* through *command_specific_result* inclusive.

For each command, this specification describes a distinct format structure used for *command_specific_data* and *command_specific_result*, the length of these fields depends on the command.

However, to ensure forward compatibility

- drivers are allowed to submit buffers that are longer than the device expects (that is, longer than the length of *opcode* through *command_specific_data*). This allows the driver to maintain a single format structure even if some structure fields are unused by the device.
- drivers are allowed to submit buffers that are shorter than what the device expects (that is, shorter than the length of *status* through *command_specific_result*). This allows the device to maintain a single format structure even if some structure fields are unused by the driver.

The device compares the length of each part (device-readable and device-writeable) of the buffer as submitted by driver to what it expects and then silently truncates the structures to either the length submitted by the driver, or the length described in this specification, whichever is shorter. The device silently ignores any data falling outside the shorter of the two lengths. Any missing fields are interpreted as set to zero.

Similarly, the driver compares the used buffer length of the buffer to what it expects and then silently truncates the structure to the used buffer length. The driver silently ignores any data falling outside the used buffer length reported by the device. Any missing fields are interpreted as set to zero.

This simplifies driver and device implementations since the driver/device can simply maintain a single large structure (such as a C structure) for a command and its result. As new versions of the specification are designed, new fields can be added to the tail of a structure, with the driver/device using the full structure without concern for versioning.

2.13.1 Device Requirements: Group administration commands

The device MUST support device-readable and device-writeable buffers shorter than described in this specification, by

1. acting as if any data that would be read outside the device-readable buffers is set to zero, and
2. discarding data that would be written outside the specified device-writeable buffers.

The device MUST support device-readable and device-writeable buffers longer than described in this specification, by

1. ignoring any data in device-readable buffers outside the expected length, and

2. only writing the expected structure to the device-writeable buffers, ignoring any extra buffers, and reporting the actual length of data written, in bytes, as buffer used length.

The device SHOULD initialize the device-writeable buffer up to the length of the structure described by this specification or the length of the buffer supplied by the driver (even if the buffer is all set to zero), whichever is shorter.

The device MUST NOT fail a command solely because the buffers provided are shorter or longer than described in this specification.

The device MUST initialize the device-writeable part of *struct virtio_admin_cmd* that is a multiple of 64 bit in size.

The device MUST initialize *status* and *status_qualifier* in *struct virtio_admin_cmd*.

The device MUST process commands on a given administration virtqueue in the order in which they are queued.

If multiple administration virtqueues have been configured, device MAY process commands on distinct virtqueues with no order constraints.

If the device sets *status* to either VIRTIO_ADMIN_STATUS_EAGAIN or VIRTIO_ADMIN_STATUS_ENOMEM, then the command MUST NOT have any side effects, making it safe to retry.

2.13.2 Driver Requirements: Group administration commands

The driver MAY supply device-readable or device-writeable parts of *struct virtio_admin_cmd* that are longer than described in this specification.

The driver SHOULD supply device-readable part of *struct virtio_admin_cmd* that is at least as large as the structure described by this specification (even if the structure is all set to zero).

The driver MUST supply both device-readable or device-writeable parts of *struct virtio_admin_cmd* that are a multiple of 64 bit in length.

The device MUST supply both device-readable or device-writeable parts of *struct virtio_admin_cmd* that are larger than zero in length. However, *command_specific_data* and *command_specific_result* MAY be zero in length, unless specified otherwise for the command.

The driver MUST NOT assume that the device will initialize the whole device-writeable part of *struct virtio_admin_cmd* as described in the specification; instead, the driver MUST act as if the structure outside the part of the buffer used by the device is set to zero.

If multiple administration virtqueues have been configured, the driver MUST ensure ordering for commands placed on different administration virtqueues.

The driver SHOULD retry a command that completed with *status* VIRTIO_ADMIN_STATUS_EAGAIN.

2.14 Device parts

Device parts represent the device state, with parts for basic device facilities such as driver features, as well as transport specific and device type specific parts. In memory, each device part consists of a header *struct virtio_dev_part_hdr* followed by the device part data in *value*. The driver can get and set these device parts using administration commands.

```
struct virtio_dev_part_hdr {
        le16 part_type;
        u8 flags;
        u8 reserved;
        union {
                struct {
                        le32 offset;
                        le32 reserved;
                } pci_common_cfg;
                struct  {
                        le16 index;
```

```
                        u8 reserved[6];
                } vq_index;
                u8 device_type_raw[8];
        } selector;
        le32 length;
};

#define VIRTIO_DEV_PART_F_OPTIONAL 0

struct virtio_dev_part {
        struct virtio_dev_part_hdr hdr;
        u8 value[];
};
```

Each device part consists of a fixed size *hdr* followed by optional part data in field *value*. The device parts are divided into two categories and identified by *part_type*. The common device parts are independent of the device type and, are in the range *0x0000 - 0x01FF*. Common device parts are listed in 2.10 The device parts in the range *0x0200 - 0x05FF* are specific to a device type such as a network or console device. The device part is identified by the *part_type* field as listed:

0x0000 - 0x01FF - common part - used to describe a part of the device that is independent of the device type

0x0200 - 0x05FF - device type specific part - used to indicate parts that are device type specific

0x0600 - 0xFFFF - reserved

Some device parts are optional, the device can function without them. For example, such parts can help improve performance, with the device working slower, yet still correctly, even without the parts. In another example, optional parts can be used for validation, with the device being able to deduce the part itself, the part being helpful to detect driver or user errors. Such device parts are marked optional by setting bit 0 (VIRTIO_DEV_PART_F_OPTIONAL) in the *flags*.

reserved is reserved and set to zero.

length indicates the length of the *value* in bytes. The length of the device part depends on the device part itself and is described separately. The device part data is in *value* and is *part_type* specific.

selector further specifies the part. It is only used for some *part_type* values.

selector.pci_common_cfg.offset is the offset of the field in the Common configuration structure layout. It is valid only when the *part_type* is set to VIRTIO_DEV_PART_PCI_COMMON_CFG, otherwise it is reserved and set to 0.

selector.vq_index.index is the index of the virtqueue. It is valid only when the *part_type* is VIRTIO_DEV_-PART_VQ_CFG or VIRTIO_DEV_PART_VQ_NOTIFY_CFG.

selector.device_type_raw is applicable only when the *part_type* corresponds to a device-specific range. The format of *selector.device_type_raw* is device type specific.

2.14.1 Common device parts

Common parts are independent of the device type. *part_type* and *value* for each part are documented as follows:

2.14.1.1 VIRTIO_DEV_PART_DEV_FEATURES

For VIRTIO_DEV_PART_DEV_FEATURES, *part_type* is set to 0x100. The VIRTIO_DEV_PART_DEV_-FEATURES field indicates features offered by the device. *value* is in the format of *struct virtio_dev_part_-features*. *feature_bits* is in the format listed in 2.2. *length* is the length of the *struct virtio_dev_part_features*.

If the VIRTIO_DEV_PART_DEV_FEATURES device part is present, there is exactly one instance of it in the get or set commands.

The VIRTIO_DEV_PART_DEV_FEATURES part is optional for which the VIRTIO_DEV_PART_F_OPTIONAL (bit 0) *flags* is set.

Table 2.10: Common device parts

Type	Name	Description
0x100	VIRTIO_DEV_PART_DEV_FEATURES	Device features, see 2.14.1.1
0x101	VIRTIO_DEV_PART_DRV_FEATURES	Driver features, 2.14.1.2
0x102	VIRTIO_DEV_PART_PCI_COMMON_CFG	PCI common configuration, see 2.14.1.3
0x103	VIRTIO_DEV_PART_DEVICE_STATUS	Device status, see 2.14.1.4
0x104	VIRTIO_DEV_PART_VQ_CFG	Virtqueue configuration, see 2.14.1.5
0x105	VIRTIO_DEV_PART_VQ_NOTIFY_CFG	Virtqueue notification configuration, see 2.14.1.6
0x106 - 0x2FF	-	Common device parts range reserved for future

```
struct virtio_dev_part_features {
        le64 feature_bits[];
};
```

2.14.1.2 VIRTIO_DEV_PART_DRV_FEATURES

For VIRTIO_DEV_PART_DRV_FEATURES, *part_type* is set to 0x101. The VIRTIO_DEV_PART_DRV_-FEATURES field indicates features set by the driver. *value* is in the format of *struct virtio_dev_part_features*. *feature_bits* is in the format listed in 2.2. *length* is the length of the *struct virtio_dev_part_features*.

If the VIRTIO_DEV_PART_DEV_FEATURES device part present, there is exactly one instance of it in the get or set commands.

2.14.1.3 VIRTIO_DEV_PART_PCI_COMMON_CFG

For VIRTIO_DEV_PART_PCI_COMMON_CFG, *part_type* is set to 0x102. VIRTIO_DEV_PART_PCI_COMMON_CFG refers to the common device configuration fields. *offset* refers to the byte offset of single field in the common configuration layout described in *struct virtio_pci_common_cfg*. *value* is in the format depending on the *offset*, for example when *cfg_offset = 18*, *value* is in the format of *num_queues*. *length* is the length of *value* in bytes of a single structure field whose offset is *offset*.

One or multiple VIRTIO_DEV_PART_PCI_COMMON_CFG parts may exist in the get or set commands; each such part corresponds to a unique *offset*.

2.14.1.4 VIRTIO_DEV_PART_DEVICE_STATUS

For VIRTIO_DEV_PART_DEVICE_STATUS, *part_type* is set to 0x103. The VIRTIO_DEV_PART_DEVICE_-STATUS field indicates the device status as listed in 2.1. *value* is in the format *device_status* of *struct virtio_pci_common_cfg*.

If the VIRTIO_DEV_PART_DEV_FEATURES device part is present, there is exactly one instance of it in the get or set commands.

There is exactly one part may exist in the get or set commands.

2.14.1.5 VIRTIO_DEV_PART_VQ_CFG

For VIRTIO_DEV_PART_VQ_CFG, *part_type* is set to 0x104. *value* is in the format *struct virtio_dev_part_-vq_cfg*. *length* is the length of *struct virtio_dev_part_vq_cfg*.

```
struct virtio_dev_part_vq_cfg {
        le16 queue_size;
        le16 vector;
        le16 enabled;
        le16 reserved;
        le64 queue_desc;
        le64 queue_driver;
        le64 queue_device;
};
```

queue_size, *vector*, *queue_desc*, *queue_driver* and *queue_device* correspond to the fields of *struct virtio_pci_common_cfg* when used for PCI transport.

One or multiple instances of the device part VIRTIO_DEV_PART_VQ_CFG may exist in the get and set commands. Each such device part corresponds to a unique virtqueue identified by the *vq_index.index*.

2.14.1.6 VIRTIO_DEV_PART_VQ_NOTIFY_CFG

For VIRTIO_DEV_PART_VQ_NOTIFY_CFG, *part_type* is set to 0x105. *value* is in the format *struct virtio_dev_part_vq_notify_data*. *length* is the length of *struct virtio_dev_part_vq_notify_data*.

```
struct virtio_dev_part_vq_notify_data {
        le16 queue_notify_off;
        le16 queue_notif_config_data;
        u8 reserved[4];
};
```

queue_notify_off and *queue_notif_config_data* corresponds to the fields in *struct virtio_pci_common_cfg* described in the Common configuration structure layout.

One or multiple instance of the device part VIRTIO_DEV_PART_VQ_NOTIFY_CFG may exist in the get and set commands, each such device part corresponds to a unique virtqueue identified by the *vq_index.index*.

reserved is reserved and set to 0.

2.14.2 Assumptions

For the SR-IOV group type, some hypervisors do not allow the driver to access the PCI configuration space and the MSI-X Table space directly. Such hypervisors query and save these fields without the need for this device parts. Therefore, this version of the specification does not have it in the device parts. A future extension of the device part may further include them as new device part.

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 performed. 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.

3.4 Device Suspend

If VIRTIO_F_SUSPEND is negotiated, the driver is eligible to suspend the device by setting the SUSPEND bit in *device status* to 1, and the device sets the DRIVER_OK bit to 0 once it has been suspended.

If the device has been suspended, the driver can resume the device running by setting the DRIVER_OK bit in *device status* to 1, and the device sets the SUSPEND bit to 0 once it resumes running.

3.4.1 Driver Requirements: Device Suspend

The driver SHOULD NOT set SUSPEND bit if DRIVER_OK is not set or VIRTIO_F_SUSPEND is not negotiated.

Once the driver sets SUSPEND bit in *device status* to 1:

- The driver MUST verify whether the device has been suspended by re-reading *device status*, examining whether the SUSPEND bit is set to 1 and the DRIVER_OK bit is set to 0.
- The driver MUST NOT make any more buffers available to the device.
- The driver MUST NOT send notifications for any virtqueues.
- The driver MUST NOT make any changes to Device Configuration Space except for *device status* if it is part of the Configuration Space.

When the device has been suspended, once the driver sets DRIVER_OK bit in *device status* to 1, the driver MUST wait for the SUSPEND bit in *device status* to turn 0 and the DRIVER_OK bit in *device_status* to turn 1 before any normal operations."

3.4.2 Device Requirements: Device Suspend

The device MUST ignore any operations on the SUSPEND bit from the driver if the device has not been completely initialized by the procedures in 3.1

The device SHOULD ignore any write access to its Configuration Space while suspended, except for *device status* if it is part of the Configuration Space.

A device MUST NOT send any notifications for any virtqueues, access any virtqueues, or modify any fields in its Configuration Space while suspended.

If changes occur in the Configuration Space during suspended period, the device MUST NOT send any configuration change notifications. Instead, the device MUST send the notification when it resumes running.

If the driver sets the SUSPEND bit in *device status* to 1, the device MUST either suspend itself or set the DEVICE_NEEDS_RESET bit in *device status* to 1 when it fails to suspend.

If the device has been suspended and the driver resumes the device running by setting the DRIVER_OK bit in *device status* to 1, the device MUST either resume normal operation or set the DEVICE_NEEDS_RESET bit in *device status* to 1 when it fails to resume.

When the driver sets the SUSPEND bit to 1, the device SHOULD perform the following actions before presenting the SUSPEND bit as 1 and DRIVER_OK bit as 0 in the *device status*:

- Stop consuming more buffers of any virtqueues.
- Wait until all buffers that are being processed have been used.
- Send used buffer notifications to the driver.

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 device
0x1001	block device
0x1002	memory ballooning (traditional)
0x1003	console
0x1004	SCSI host
0x1005	entropy source
0x1009	9P transport

For example, the network 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 4.1.4.9 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 id;          /* Multiple capabilities of the same type */
        u8 padding[2];  /* 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
/* Shared memory region */
#define VIRTIO_PCI_CAP_SHARED_MEMORY_CFG 8
/* Vendor-specific data */
#define VIRTIO_PCI_CAP_VENDOR_CFG        9
```

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. A device may specify that this ordering mechanism be overridden by the use of the *id* field.

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.

id Used by some device types to uniquely identify multiple capabilities of a certain type. If the device type does not specify the meaning of this field, its contents are undefined.

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.

A variant of this type, struct virtio_pci_cap64, is defined for those capabilities that require offsets or lengths larger than 4GiB:

```
struct virtio_pci_cap64 {
        struct virtio_pci_cap cap;
        le32 offset_hi;
        le32 length_hi;
};
```

Given that the *cap.length* and *cap.offset* fields are only 32 bit, the additional *offset_hi* and *length_hi* fields provide the most significant 32 bits of a total 64 bit offset and length within the BAR specified by *cap.bar*.

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 4.1.4.9.2.

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 config_msix_vector;        /* 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 */
        le16 queue_notif_config_data;   /* read-only for driver */
        le16 queue_reset;               /* read-write */

        /* About the administration virtqueue. */
        le16 admin_queue_index;         /* read-only for driver */
        le16 admin_queue_num;         /* read-only for driver */
};
```

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 Set by the driver to the MSI-X vector for configuration change notifications.

num_queues The device specifies the maximum number of virtqueues supported here. This excludes administration virtqueues if any are supported.

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 Set by the driver to the MSI-X vector for virtqueue notifications.

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 2.6.

queue_driver The driver writes the physical address of Driver Area here. See section 2.6.

queue_device The driver writes the physical address of Device Area here. See section 2.6.

queue_notif_config_data This field exists only if VIRTIO_F_NOTIF_CONFIG_DATA has been negotiated. The driver will use this value when driver sends available buffer notification to the device. See section 4.1.5.2.

Note: This field provides the device with flexibility to determine how virtqueues will be referred to in available buffer notifications. In a trivial case the device can set *queue_notif_config_data* to the virtqueue index. Some devices may benefit from providing another value, for example an internal virtqueue identifier, or an internal offset related to the virtqueue index.

Note: This field was previously known as queue_notify_data.

queue_reset The driver uses this to selectively reset the queue. This field exists only if VIRTIO_F_RING_RESET has been negotiated. (see 2.6.1).

admin_queue_index The device uses this to report the index of the first administration virtqueue. This field is valid only if VIRTIO_F_ADMIN_VQ has been negotiated.

admin_queue_num The device uses this to report the number of the supported administration virtqueues. Virtqueues with index between *admin_queue_index* and (*admin_queue_index* + *admin_queue_num* - 1) inclusive serve as administration virtqueues. The value 0 indicates no supported administration virtqueues. This field is valid only if VIRTIO_F_ADMIN_VQ has been negotiated.

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.

If VIRTIO_F_RING_RESET has been negotiated, the device MUST present a 0 in *queue_reset* on reset.

If VIRTIO_F_RING_RESET has been negotiated, the device MUST present a 0 in *queue_reset* after the virtqueue is enabled with *queue_enable*.

The device MUST reset the queue when 1 is written to *queue_reset*. The device MUST continue to present 1 in *queue_reset* as long as the queue reset is ongoing. The device MUST present 0 in both *queue_reset* and *queue_enable* when queue reset has completed. (see 2.6.1).

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*.

If VIRTIO_F_ADMIN_VQ has been negotiated, the value *admin_queue_index* MUST be equal to, or bigger than *num_queues*; also, *admin_queue_num* MUST be smaller than, or equal to 0x10000 - *admin_queue_index*, to ensure that indices of valid admin queues fit into a 16 bit range beyond all other virtqueues.

4.1.4.3.2 Driver Requirements: Common configuration structure layout

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

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*.

If VIRTIO_F_RING_RESET has been negotiated, after the driver writes 1 to *queue_reset* to reset the queue, the driver MUST NOT consider queue reset to be complete until it reads back 0 in *queue_reset*. The driver MAY re-enable the queue by writing 1 to *queue_enable* after ensuring that other virtqueue fields have been set up correctly. The driver MAY set driver-writeable queue configuration values to different values than those that were used before the queue reset. (see 2.6.1).

If VIRTIO_F_ADMIN_VQ has been negotiated, and if the driver configures any administration virtqueues, the driver MUST configure the administration virtqueues using the index in the range *admin_queue_index* to *admin_queue_index* + *admin_queue_num* - 1 inclusive. The driver MAY configure fewer administration virtqueues than supported by the device.

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 driver 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 Shared memory capability

Shared memory regions 2.10 are enumerated on the PCI transport as a sequence of VIRTIO_PCI_CAP_-SHARED_MEMORY_CFG capabilities, one per region.

The capability is defined by a struct virtio_pci_cap64 and utilises the *cap.id* to allow multiple shared memory regions per device. The identifier in *cap.id* does not denote a certain order of preference; it is only used to uniquely identify a region.

4.1.4.7.1 Device Requirements: Shared memory capability

The region defined by the combination of the *cap.offset*, *offset_hi*, and *cap.length*, *length_hi* fields MUST be contained within the BAR specified by *cap.bar*.

The *cap.id* MUST be unique for any one device instance.

4.1.4.8 Vendor data capability

The optional Vendor data capability allows the device to present vendor-specific data to the driver, without conflicts, for debugging and/or reporting purposes, and without conflicting with standard functionality.

This capability augments but does not replace the standard subsystem ID and subsystem vendor ID fields (offsets 0x2C and 0x2E in the PCI configuration space header) as specified by [PCI].

Vendor data capability is enumerated on the PCI transport as a VIRTIO_PCI_CAP_VENDOR_CFG capability.

The capability has the following structure:

```
struct virtio_pci_vndr_data {
        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. */
        u16 vendor_id;  /* Identifies the vendor-specific format. */
  /* For Vendor Definition */
  /* Pads structure to a multiple of 4 bytes */
  /* Reads must not have side effects */
};
```

Where *vendor_id* identifies the PCI-SIG assigned Vendor ID as specified by [PCI].

Note that the capability size is required to be a multiple of 4.

To make it safe for a generic driver to access the capability, reads from this capability MUST NOT have any side effects.

4.1.4.8.1 Device Requirements: Vendor data capability

Devices CAN present *vendor_id* that does not match either the PCI Vendor ID or the PCI Subsystem Vendor ID.

Devices CAN present multiple Vendor data capabilities with either different or identical *vendor_id* values.

The value *vendor_id* MUST NOT equal 0x1AF4.

The size of the Vendor data capability MUST be a multiple of 4 bytes.

Reads of the Vendor data capability by the driver MUST NOT have any side effects.

4.1.4.8.2 Driver Requirements: Vendor data capability

The driver SHOULD NOT use the Vendor data capability except for debugging and reporting purposes.

The driver MUST qualify the *vendor_id* before interpreting or writing into the Vendor data capability.

4.1.4.9 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*.

4.1.4.9.1 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*.

4.1.4.9.2 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*.

4.1.4.10 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. 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 2.5.4 Legacy Interface: Device Configuration Space for workarounds.

4.1.4.11 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.

4.1.4.11.0.1 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 interpret 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 to *queue_select*.
2. Read the virtqueue size from *queue_size*. This controls how big the virtqueue is (see 2.6 Virtqueues). If this field is 0, the virtqueue does not exist.
3. Optionally, select a smaller virtqueue size and write it to *queue_size*.
4. Allocate and zero Descriptor Table, Available and Used rings for the virtqueue in contiguous physical memory.
5. Optionally, if MSI-X capability is present and enabled on the device, select a vector to use to request interrupts triggered by virtqueue events. Write the MSI-X Table entry number corresponding to this vector into *queue_msix_vector*. Read *queue_msix_vector*: on success, previously written value is returned; on failure, NO_VECTOR value is returned.

4.1.5.1.3.1 Legacy Interface: A Note on Virtqueue Configuration

When using the legacy interface, the queue layout follows 2.7.2 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 only the 16-bit notification value to the Queue Notify address of the virtqueue. A notification value depends on the negotiation of VIRTIO_F_NOTIF_CONFIG_DATA.

If 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 {
  union {
    vq_index: 16; /* Used if VIRTIO_F_NOTIF_CONFIG_DATA not negotiated */
    vq_notif_config_data: 16; /* Used if VIRTIO_F_NOTIF_CONFIG_DATA negotiated */
  };
  next_off : 15;
  next_wrap : 1;
};
```

- When VIRTIO_F_NOTIF_CONFIG_DATA is not negotiated *vq_index* is set to the virtqueue index.
- When VIRTIO_F_NOTIFICATION_DATA is negotiated, *vq_notif_config_data* is set to *queue_notif_config_data*.

See 2.9 Driver Notifications for the definition of the components.

See 4.1.4.4 for how to calculate the Queue Notify address.

4.1.5.2.1 Driver Requirements: Available Buffer Notifications

If VIRTIO_F_NOTIFICATION_DATA is not negotiated, the driver notification MUST be a 16-bit notification.

If VIRTIO_F_NOTIFICATION_DATA is negotiated, the driver notification MUST be a 32-bit notification.

If VIRTIO_F_NOTIF_CONFIG_DATA is not negotiated:

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

2The 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.

- If VIRTIO_F_NOTIFICATION_DATA is not negotiated, the driver MUST set the notification value to the virtqueue index.
- If VIRTIO_F_NOTIFICATION_DATA is negotiated, the driver MUST set the *vq_index* to the virtqueue index.

If VIRTIO_F_NOTIF_CONFIG_DATA is negotiated:

- If VIRTIO_F_NOTIFICATION_DATA is not negotiated, the driver MUST set the notification value to *queue_notif_config_data*.
- If VIRTIO_F_NOTIFICATION_DATA is negotiated, the driver MUST set the *vq_notify_config_data* to the *queue_notif_config_data* value.

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.

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 4.2.

All register values are organized as Little Endian.

Table 4.2: MMIO Device Register Layout

Name Offset from base Direction	**Function** Description
MagicValue 0x000 R	**Magic value** 0x74726976 (a Little Endian equivalent of the “virt” string).

Name Offset from the base Direction	**Function** Description
Version 0x004 R	**Device version number** 0x2. **Note:** Legacy devices (see 4.2.4 Legacy interface) used 0x1.
DeviceID 0x008 R	**Virtio Subsystem Device ID** See 5 Device Types for possible values. Value zero (0x0) is used to define a system memory map with placeholder devices at static, well known addresses, assigning functions to them depending on user's needs.
VendorID 0x00c R	**Virtio Subsystem Vendor ID**
DeviceFeatures 0x010 R	**Flags representing features the device supports** Reading from this register returns 32 consecutive flag bits, the least significant bit depending on the last value written to *DeviceFeaturesSel*. Access to this register returns bits *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 0x014 W	**Device (host) features word selection.** Writing to this register selects a set of 32 device feature bits accessible by reading from *DeviceFeatures*.
DriverFeatures 0x020 W	**Flags representing device features understood and activated by the driver** Writing to this register sets 32 consecutive flag bits, the least significant bit depending on the last value written to *DriverFeaturesSel*. Access to this register sets bits *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 0x024 W	**Activated (guest) features word selection** Writing to this register selects a set of 32 activated feature bits accessible by writing to *DriverFeatures*.
QueueSel 0x030 W	**Virtqueue index** Writing to this register selects the virtqueue that the following operations on *QueueSizeMax*, *QueueSize*, *QueueReady*, *QueueDescLow*, *QueueDescHigh*, *QueueDriverlLow*, *QueueDriverHigh*, *QueueDeviceLow*, *QueueDeviceHigh* and *QueueReset* apply to.
QueueSizeMax 0x034 R	**Maximum virtqueue size** Reading from the register returns the maximum size (number of elements) of the queue the device is ready to process or zero (0x0) if the queue is not available. This applies to the queue selected by writing to *QueueSel*. **Note:** *QueueSizeMax* was previously known as *QueueNumMax*.
QueueSize 0x038 W	**Virtqueue size** 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*. **Note:** *QueueSize* was previously known as *QueueNum*.

Name Offset from the base Direction	**Function** Description
QueueReady 0x044 RW	**Virtqueue ready bit** Writing one (0x1) to this register notifies the device that it can execute requests from this virtqueue. Reading from this register returns the last value written to it. Both read and write accesses apply to the queue selected by writing to *QueueSel*.
QueueNotify 0x050 W	**Queue notifier** Writing a value to this register notifies the device that there are new buffers to process in a queue. When VIRTIO_F_NOTIFICATION_DATA has not been negotiated, the value written is the queue index. When VIRTIO_F_NOTIFICATION_DATA has been negotiated, the *Notification data* value has the following format: `le32 { vq_index: 16; /* previously known as vqn */ next_off : 15; next_wrap : 1; };` See 2.9 Driver Notifications for the definition of the components.
InterruptStatus 0x60 R	**Interrupt status** Reading from this register returns a bit mask of events that caused the device interrupt to be asserted. The following events are possible: **Used Buffer Notification** - bit 0 - the interrupt was asserted because the device has used a buffer in at least one of the active virtqueues. **Configuration Change Notification** - bit 1 - the interrupt was asserted because the configuration of the device has changed.
InterruptACK 0x064 W	**Interrupt acknowledge** Writing a value with bits set as defined in *InterruptStatus* to this register notifies the device that events causing the interrupt have been handled.
Status 0x070 RW	**Device status** Reading from this register returns the current device status flags. Writing non-zero values to this register sets the status flags, indicating the driver progress. Writing zero (0x0) to this register triggers a device reset. See also p. 4.2.3.1 Device Initialization.
QueueDescLow 0x080 *QueueDescHigh* 0x084 W	**Virtqueue's Descriptor Area 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueDescLow*, higher 32 bits to *QueueDescHigh*) notifies the device about location of the Descriptor Area of the queue selected by writing to *QueueSel* register.
QueueDriverLow 0x090 *QueueDriverHigh* 0x094 W	**Virtqueue's Driver Area 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueDriverLow*, higher 32 bits to *QueueDriverHigh*) notifies the device about location of the Driver Area of the queue selected by writing to *QueueSel*.
QueueDeviceLow 0x0a0 *QueueDeviceHigh* 0x0a4 W	**Virtqueue's Device Area 64 bit long physical address** Writing to these two registers (lower 32 bits of the address to *QueueDeviceLow*, higher 32 bits to *QueueDeviceHigh*) notifies the device about location of the Device Area of the queue selected by writing to *QueueSel*.

Name Offset from the base Direction	**Function** Description
SHMSel 0x0ac W	**Shared memory id** Writing to this register selects the shared memory region 2.10 following operations on *SHMLenLow*, *SHMLenHigh*, *SHMBaseLow* and *SHMBaseHigh* apply to.
SHMLenLow 0x0b0 *SHMLenHigh* 0x0b4 R	**Shared memory region 64 bit long length** These registers return the length of the shared memory region in bytes, as defined by the device for the region selected by the *SHMSel* register. The lower 32 bits of the length are read from *SHMLenLow* and the higher 32 bits from *SHMLenHigh*. Reading from a non-existent region (i.e. where the ID written to *SHMSel* is unused) results in a length of -1.
SHMBaseLow 0x0b8 *SHMBaseHigh* 0x0bc R	**Shared memory region 64 bit long physical address** The driver reads these registers to discover the base address of the region in physical address space. This address is chosen by the device (or other part of the VMM). The lower 32 bits of the address are read from *SHMBaseLow* with the higher 32 bits from *SHMBaseHigh*. Reading from a non-existent region (i.e. where the ID written to *SHMSel* is unused) results in a base address of 0xffffffffffffffff.
QueueReset 0x0c0 RW	**Virtqueue reset bit** If VIRTIO_F_RING_RESET has been negotiated, writing one (0x1) to this register selectively resets the queue. Both read and write accesses apply to the queue selected by writing to *QueueSel*.
ConfigGeneration 0x0fc R	**Configuration atomicity value** Reading from this register returns a value describing a version of the device-specific configuration space (see *Config*). The driver can then access the configuration space and, when finished, read *ConfigGeneration* again. If no part of the configuration space has changed between these two *ConfigGeneration* reads, the returned values are identical. If the values are different, the configuration space accesses were not atomic and the driver has to perform the operations again. See also 2.5.
Config 0x100+ RW	**Configuration space** Device-specific configuration space starts at the offset 0x100 and is accessed with byte alignment. Its meaning and size depend on the device and the driver.

4.2.2.1 Device Requirements: MMIO Device Register Layout

The device MUST return 0x74726976 in *MagicValue*.

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

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

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

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

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

If VIRTIO_F_RING_RESET has been negotiated, the device MUST present a 0 in *QueueReset* on reset.

If VIRTIO_F_RING_RESET has been negotiated, The device MUST present a 0 in *QueueReset* after the virtqueue is enabled with *QueueReady*.

The device MUST reset the queue when 1 is written to *QueueReset*. The device MUST continue to present 1 in *QueueReset* as long as the queue reset is ongoing. The device MUST present 0 in both *QueueReset* and *QueueReady* when queue reset has completed. (see 2.6.1).

4.2.2.2 Driver Requirements: MMIO Device Register Layout

The driver MUST NOT access memory locations not described in the table 4.2 (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 4.2. 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 *QueueSize* which is less than or equal to the value presented by the device in *QueueSizeMax*.

When *QueueReady* is not zero, the driver MUST NOT access *QueueSize*, *QueueDescLow*, *QueueDescHigh*, *QueueDriverLow*, *QueueDriverHigh*, *QueueDeviceLow*, *QueueDeviceHigh*.

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.

If VIRTIO_F_RING_RESET has been negotiated, after the driver writes 1 to *QueueReset* to reset the queue, the driver MUST NOT consider queue reset to be complete until it reads back 0 in *QueueReset*. The driver MAY re-enable the queue by writing 1 to *QueueReady* after ensuring that other virtqueue fields have been set up correctly. The driver MAY set driver-writeable queue configuration values to different values than those that were used before the queue reset. (see 2.6.1).

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.

Drivers not expecting shared memory MUST NOT use the shared memory registers.

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

4.2.3.2 Virtqueue Configuration

The driver will typically initialize the virtqueue in the following way:

1. Select the queue by writing its index 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 *QueueSizeMax*. 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 *QueueSize*.
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 {
  vq_index: 16; /* previously known as vqn */
  next_off : 15;
  next_wrap : 1;
};
```

See 2.9 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 virtqueue configuration procedure.

Table 4.3 presents control registers layout, omitting descriptions of registers which did not change their function nor behaviour:

Table 4.3: MMIO Device Legacy Register Layout

Name Offset from base Direction	**Function** Description
MagicValue 0x000 R	**Magic value**
Version 0x004 R	**Device version number** Legacy device returns value 0x1.
DeviceID 0x008 R	**Virtio Subsystem Device ID**
VendorID 0x00c R	**Virtio Subsystem Vendor ID**
HostFeatures 0x010 R	**Flags representing features the device supports**
HostFeaturesSel 0x014 W	**Device (host) features word selection.**
GuestFeatures 0x020 W	**Flags representing device features understood and activated by the driver**
GuestFeaturesSel 0x024 W	**Activated (guest) features word selection**
GuestPageSize 0x028 W	**Guest page size** The driver writes the guest page size in bytes to the register during initialization, before any queues are used. This value should be a power of 2 and is used by the device to calculate the Guest address of the first queue page (see QueuePFN).
QueueSel 0x030 W	**Virtqueue index** Writing to this register selects the virtqueue that the following operations on the *QueueSizeMax*, *QueueSize*, *QueueAlign* and *QueuePFN* registers apply to.
QueueSizeMax 0x034 R	**Maximum virtqueue size** Reading from the register returns the maximum size of the queue the device is ready to process or zero (0x0) if the queue is not available. This applies to the queue selected by writing to *QueueSel* and is allowed only when *QueuePFN* is set to zero (0x0), so when the queue is not actively used. **Note:** *QueueSizeMax* was previously known as *QueueNumMax*.
QueueSize 0x038 W	**Virtqueue size** 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*. **Note:** *QueueSize* was previously known as *QueueNum*.

Name Offset from the base Direction	**Function** Description
QueueAlign 0x03c W	**Used Ring alignment in the virtqueue** Writing to this register notifies the device about alignment boundary of the Used Ring in bytes. This value should be a power of 2 and applies to the queue selected by writing to *QueueSel*.
QueuePFN 0x040 RW	**Guest physical page number of the virtqueue** Writing to this register notifies the device about location of the virtqueue in the Guest's physical address space. This value is the index number of a page starting with the queue Descriptor Table. Value zero (0x0) means physical address zero (0x00000000) and is illegal. When the driver stops using the queue it writes zero (0x0) to this register. Reading from this register returns the currently used page number of the queue, therefore a value other than zero (0x0) means that the queue is in use. Both read and write accesses apply to the queue selected by writing to *QueueSel*.
QueueNotify 0x050 W	**Queue notifier**
InterruptStatus 0x60 R	**Interrupt status**
InterruptACK 0x064 W	**Interrupt acknowledge**
Status 0x070 RW	**Device status** Reading from this register returns the current device status flags. Writing non-zero values to this register sets the status flags, indicating the OS/driver progress. Writing zero (0x0) to this register triggers a device reset. The device sets *QueuePFN* to zero (0x0) for all queues in the device. Also see 3.1 Device Initialization.
Config 0x100+ RW	**Configuration space**

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

The virtqueue layout follows p. 2.7.2 Legacy Interfaces: A Note on Virtqueue Layout, with the alignment defined in *QueueAlign*.

The virtqueue is configured as follows:

1. Select the queue by writing its index 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 *QueueSizeMax*. 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 *QueueSizeMax*.
5. Notify the device about the queue size by writing the size to *QueueSize*.
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.

4.2.5 Features reserved for future use

Devices and drivers utilizing Virtio Over MMIO do not support the following features:

- VIRTIO_F_ADMIN_VQ

These features are reserved for future use.

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

4.3.1.4 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
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 SHOULD 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_queue_size; /* previously known as max_num */
};
```

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

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 size; /* previously known as 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 *size*.

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 size; /* previously known as num */
};
```

queue contains the guest address for queue *index*, *size* the number of buffers and *align* the alignment. The queue layout follows 2.7.2 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 contiguous 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 index.

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

```
be32 {
  vq_index: 16; /* previously known as vqn */
  next_off : 15;
  next_wrap : 1;
```

```
};
```

See 2.9 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 notification.

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. This command does not carry any payload.

The device signals completion of the virtio reset operation through successful conclusion of the CCW_-CMD_VDEV_RESET channel command. In particular, the command not only triggers the reset operation, but the reset operation is already completed when the operation concludes successfully.

4.3.3.3.1 Device Requirements: Resetting Devices

The device MUST finish the virtio reset operation and reinitialize *device status* to zero before it concludes the CCW_CMD_VDEV_RESET command successfully.

The device MUST NOT send notifications or interact with the queues after it signaled successful conclusion of the CCW_CMD_VDEV_RESET command.

4.3.3.3.2 Driver Requirements: Resetting Devices

The driver MAY consider the virtio reset operation to be complete already after successful conclusion of the CCW_CMD_VDEV_RESET channel command, although it MAY also choose to verify reset completion by reading *device status* via CCW_CMD_READ_STATUS and checking whether it is 0 afterwards.

4.3.4 Features reserved for future use

Devices and drivers utilizing Virtio over channel I/O do not support the following features:

- VIRTIO_F_ADMIN_VQ
- VIRTIO_F_RING_RESET
- Shared memory regions including VIRTIO_PMEM_F_SHMEM_REGION

These features are reserved for future use.

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 device
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	RTC (Real Time Clock) device
18	Input device
19	Socket device
20	Crypto device
21	Signal Distribution Module
22	pstore device
23	IOMMU device
24	Memory device
25	Sound device
26	file system device
27	PMEM device
28	RPMB device
29	mac80211 hwsim wireless simulation device
30	Video encoder device

31	Video decoder device
32	SCMI device
33	NitroSecureModule
34	I2C adapter
35	Watchdog
36	CAN device
38	Parameter Server
39	Audio policy device
40	Bluetooth device
41	GPIO device
42	RDMA device
43	Camera device
44	ISM device
45	SPI controller
46	TEE device
47	CPU balloon device
48	Media device
49	USB controller

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 network interface controller. It consists of a virtual Ethernet link which connects the device to the Ethernet network. The device has transmit and receive queues. The driver adds empty buffers to the receive virtqueue. The device receives incoming packets from the link; the device places these incoming packets in the receive virtqueue buffers. The driver adds outgoing packets to the transmit virtqueue. The device removes these packets from the transmit virtqueue and sends them to the link. The device may have a control virtqueue. The driver uses the control virtqueue to dynamically manipulate various features of the initialized device.

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 neither VIRTIO_NET_F_MQ nor VIRTIO_NET_F_RSS are negotiated, otherwise N is set by *max_virtqueue_pairs*.

controlq is optional; it only exists if VIRTIO_NET_F_CTRL_VQ is negotiated.

5.1.3 Feature bits

VIRTIO_NET_F_CSUM (0) Device handles packets with partial checksum offload.

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_CTRL_RX_EXTRA (20) Control channel RX extra mode support.

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_DEVICE_STATS(50) Device can provide device-level statistics to the driver through the control virtqueue.

VIRTIO_NET_F_HASH_TUNNEL(51) Device supports inner header hash for encapsulated packets.

VIRTIO_NET_F_VQ_NOTF_COAL(52) Device supports virtqueue notification coalescing.

VIRTIO_NET_F_NOTF_COAL(53) Device supports notifications coalescing.

VIRTIO_NET_F_GUEST_USO4 (54) Driver can receive USOv4 packets.

VIRTIO_NET_F_GUEST_USO6 (55) Driver can receive USOv6 packets.

VIRTIO_NET_F_HOST_USO (56) Device can receive USO packets. Unlike UFO (fragmenting the packet) the USO splits large UDP packet to several segments when each of these smaller packets has UDP header.

VIRTIO_NET_F_HASH_REPORT(57) Device can report per-packet hash value and a type of calculated hash.

VIRTIO_NET_F_GUEST_HDRLEN(59) Driver can provide the exact *hdr_len* value. Device benefits from knowing the exact header length.

VIRTIO_NET_F_RSS(60) Device supports RSS (receive-side scaling) with Toeplitz hash calculation and configurable hash parameters for receive steering.

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.

VIRTIO_NET_F_SPEED_DUPLEX(63) Device reports speed and duplex.

VIRTIO_NET_F_RSS_CONTEXT(64) Device supports multiple RSS contexts.

VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO (65) Driver can receive GSO packets carried by a UDP tunnel.

VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM (66) Driver handles packets carried by a UDP tunnel with partial csum for the outer header.

VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO (67) Device can receive GSO packets carried by a UDP tunnel.

VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM (68) Device handles packets carried by a UDP tunnel with partial csum for the outer header.

VIRTIO_NET_F_OUT_NET_HEADER(69) Driver can provide the start of *outer_nh_offset* value. Device gains advantage by not reading packet to calculate outer network header offset.

VIRTIO_NET_F_IPSEC(70) Device supports inline IPsec processing. *struct virtio_net_hdr* size expands upto field *sturct ipsec_resource_hdr* when VIRTIO_NET_F_IPSEC is negotiated. When a device offers IPsec feature, it SHOULD also offer the VIRTIO_NET_F_OUT_NET_HEADER feature.

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_GUEST_USO4 Requires VIRTIO_NET_F_GUEST_CSUM.

VIRTIO_NET_F_GUEST_USO6 Requires VIRTIO_NET_F_GUEST_CSUM.

VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO Requires VIRTIO_NET_F_GUEST_TSO4, VIRTIO_NET_F_GUEST_TSO6, VIRTIO_NET_F_GUEST_USO4 and VIRTIO_NET_F_GUEST_USO6.

VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM Requires VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO

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_HOST_USO Requires VIRTIO_NET_F_CSUM.

VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO Requires VIRTIO_NET_F_HOST_TSO4, VIRTIO_NET_F_HOST_TSO6 and VIRTIO_NET_F_HOST_USO.

VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM Requires VIRTIO_NET_F_HOST_UDP_TUNNEL_-GSO

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_NOTF_COAL Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_RSC_EXT Requires VIRTIO_NET_F_HOST_TSO4 or VIRTIO_NET_F_HOST_TSO6.

VIRTIO_NET_F_RSS Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_VQ_NOTF_COAL Requires VIRTIO_NET_F_CTRL_VQ.

VIRTIO_NET_F_HASH_TUNNEL Requires VIRTIO_NET_F_CTRL_VQ along with VIRTIO_NET_F_RSS or VIRTIO_NET_F_HASH_REPORT.

VIRTIO_NET_F_RSS_CONTEXT Requires VIRTIO_NET_F_CTRL_VQ and VIRTIO_NET_F_RSS.

Note: The dependency between UDP_TUNNEL_GSO_CSUM and UDP_TUNNEL_GSO is intentionally in the opposite direction with respect to the plain GSO features and the plain checksum offload because UDP tunnel checksum offload gives very little gain for non GSO packets and is quite complex to implement in H/W.

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

The network device has the following device configuration layout. All of the device configuration fields are read-only for the driver.

```
struct virtio_net_config {
        u8 mac[6];
        le16 status;
        le16 max_virtqueue_pairs;
        le16 mtu;
        le32 speed;
        u8 duplex;
        u8 rss_max_key_size;
        le16 rss_max_indirection_table_length;
        le32 supported_hash_types;
        le32 supported_tunnel_types;
};
```

The *mac* address field always exists (although it is only valid if VIRTIO_NET_F_MAC is set).

The *status* only exists if VIRTIO_NET_F_STATUS is set. Two bits 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 field, *max_virtqueue_pairs* only exists if VIRTIO_NET_F_MQ or VIRTIO_NET_F_RSS 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 at least one of these features is negotiated.

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

The following two fields, *speed* and *duplex*, only exist if VIRTIO_NET_F_SPEED_DUPLEX is set.

speed contains the device speed, in units of 1 MBit per second, 0 to 0x7fffffff, or 0xffffffff for unknown speed.

duplex has the values of 0x01 for full duplex, 0x00 for half duplex and 0xff for unknown duplex state.

Both *speed* and *duplex* can change, thus the driver is expected to re-read these values after receiving a configuration change notification.

The following field, *rss_max_key_size* only exists if VIRTIO_NET_F_RSS or VIRTIO_NET_F_HASH_REPORT is set. It specifies the maximum supported length of RSS key in bytes.

The following field, *rss_max_indirection_table_length* only exists if VIRTIO_NET_F_RSS is set. It specifies the maximum number of 16-bit entries in RSS indirection table.

The next field, *supported_hash_types* only exists if the device supports hash calculation, i.e. if VIRTIO_NET_F_RSS or VIRTIO_NET_F_HASH_REPORT is set.

Field *supported_hash_types* contains the bitmask of supported hash types. See 5.1.9.4.3.1 for details of supported hash types.

Field *supported_tunnel_types* only exists if the device supports inner header hash, i.e. if VIRTIO_NET_F_HASH_TUNNEL is set.

Field *supported_tunnel_types* contains the bitmask of encapsulation types supported by the device for inner header hash. Encapsulation types are defined in 5.1.9.4.4.2.

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.

The device MUST set *rss_max_key_size* to at least 40, if it offers VIRTIO_NET_F_RSS or VIRTIO_NET_F_HASH_REPORT.

The device MUST set *rss_max_indirection_table_length* to at least 128, if it offers VIRTIO_NET_F_RSS.

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.

If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, *speed* MUST contain the device speed, in units of 1 MBit per second, 0 to 0x7fffffff, or 0xffffffff for unknown.

If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, *duplex* MUST have the values of 0x00 for full duplex, 0x01 for half duplex, or 0xff for unknown.

If VIRTIO_NET_F_SPEED_DUPLEX and VIRTIO_NET_F_STATUS have both been negotiated, the device SHOULD NOT change the *speed* and *duplex* fields as long as VIRTIO_NET_S_LINK_UP is set in the *status*.

The device SHOULD NOT offer VIRTIO_NET_F_HASH_REPORT if it does not offer VIRTIO_NET_F_-CTRL_VQ.

The device SHOULD NOT offer VIRTIO_NET_F_CTRL_RX_EXTRA if it does not offer VIRTIO_NET_F_-CTRL_VQ.

5.1.4.2 Driver Requirements: Device configuration layout

The driver MUST NOT write to any of the device configuration fields.

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.

If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, the driver MUST treat any value of *speed* above 0x7fffffff as well as any value of *duplex* not matching 0x00 or 0x01 as an unknown value.

If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, the driver SHOULD re-read *speed* and *duplex* after a configuration change notification.

A driver SHOULD NOT negotiate VIRTIO_NET_F_HASH_REPORT if it does not negotiate VIRTIO_NET_-F_CTRL_VQ.

A driver SHOULD NOT negotiate VIRTIO_NET_F_CTRL_RX_EXTRA if it does not negotiate VIRTIO_-NET_F_CTRL_VQ.

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 5.1.9.3.
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 device, 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 negotiating the VIRTIO_NET_F_CSUM feature.
8. If that feature is negotiated, a driver can use TCP segmentation or UDP segmentation/fragmentation offload by negotiating the VIRTIO_NET_F_HOST_TSO4 (IPv4 TCP), VIRTIO_NET_F_HOST_TSO6 (IPv6 TCP), VIRTIO_NET_F_HOST_UFO (UDP fragmentation) and VIRTIO_NET_F_HOST_USO (UDP segmentation) features.
9. If the VIRTIO_NET_F_HOST_TSO6, VIRTIO_NET_F_HOST_TSO4 and VIRTIO_NET_F_HOST_USO segmentation features are negotiated, a driver can use TCP segmentation or UDP segmentation on top of UDP encapsulation offload, when the outer header does not require checksumming - e.g. the outer UDP checksum is zero - by negotiating the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO feature. GSO over UDP tunnels packets carry two sets of headers: the outer ones and the inner ones. The outer transport protocol is UDP, the inner could be either TCP or UDP. Only a single level of encapsulation offload is supported.
10. If VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO is negotiated, a driver can additionally use TCP segmentation or UDP segmentation on top of UDP encapsulation with the outer header requiring checksum offload, negotiating the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM feature.
11. 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, VIRTIO_NET_F_GUEST_ECN, VIRTIO_NET_F_GUEST_USO4, VIRTIO_NET_F_GUEST_USO6 VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO and VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM are the input equivalents of the features described above. See 5.1.9.3 Setting Up Receive Buffers and 5.1.9.4 Processing of Incoming Packets below.

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

5.1.6 Device and driver capabilities

The network device has the following capabilities.

Identifier	Name	Description
0x0800	VIRTIO_NET_FF_RESOURCE_CAP	Flow filter resource capability
0x0801	VIRTIO_NET_FF_SELECTOR_CAP	Flow filter classifier capability
0x0802	VIRTIO_NET_FF_ACTION_CAP	Flow filter action capability
0x0803	VIRTIO_NET_IPSEC_RESOURCE_CAP	IPsec resource capability
0x0804	VIRTIO_NET_IPSEC_SA_CAP	IPsec Security Association(SA) capability

5.1.7 Device resource objects

The network device has the following resource objects.

type	Name	Description

0x0200	VIRTIO_NET_RESOURCE_OBJ_FF_GROUP	Flow filter group resource object
0x0201	VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER	Flow filter mask object
0x0202	VIRTIO_NET_RESOURCE_OBJ_FF_RULE	Flow filter rule object
0x0203	VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA	IPsec outbound SA resource object
0x0204	VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA	IPsec inbound SA resource object

5.1.8 Device parts

Network device parts represent the configuration done by the driver using control virtqueue commands. Network device part is in the format of *struct virtio_dev_part*.

Type	Name	Description
0x200	VIRTIO_NET_DEV_PART_CVQ_CFG_PART	Represents device configuration done through a control virtqueue command, see 5.1.8.1
0x201 - 0x5FF	-	reserved for future

5.1.8.1 VIRTIO_NET_DEV_PART_CVQ_CFG_PART

For VIRTIO_NET_DEV_PART_CVQ_CFG_PART, *part_type* is set to 0x200. The VIRTIO_NET_DEV_PART_CVQ_CFG_PART part indicates configuration performed by the driver using a control virtqueue command.

```
struct virtio_net_dev_part_cvq_selector {
        u8 class;
        u8 command;
        u8 reserved[6];
};
```

There is one device part of type VIRTIO_NET_DEV_PART_CVQ_CFG_PART for each individual configuration. Each part is identified by a unique selector value. The selector, *device_type_raw*, is in the format *struct virtio_net_dev_part_cvq_selector*.

The selector consists of two fields: *class* and *command*. These fields correspond to the *class* and *command* defined in *struct virtio_net_ctrl*, as described in the relevant sections of 5.1.9.5.

The value corresponding to each part's selector follows the same format as the respective *command-specific-data* described in the relevant sections of 5.1.9.5.

For example, when the *class* is VIRTIO_NET_CTRL_MAC, the *command* can be either VIRTIO_NET_CTRL_MAC_TABLE_SET or VIRTIO_NET_CTRL_MAC_ADDR_SET; when *command* is set to VIRTIO_NET_CTRL_MAC_TABLE_SET, *value* is in the format of *struct virtio_net_ctrl_mac*.

Supported selectors are listed in the table:

Class selector	Command selector
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_PROMISC
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_ALLMULTI
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_ALLUNI
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_NOMULTI
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_NOUNI
VIRTIO_NET_CTRL_RX	VIRTIO_NET_CTRL_RX_NOBCAST
VIRTIO_NET_CTRL_MAC	VIRTIO_NET_CTRL_MAC_TABLE_SET

VIRTIO_NET_CTRL_MAC	VIRTIO_NET_CTRL_MAC_ADDR_SET
VIRTIO_NET_CTRL_VLAN	VIRTIO_NET_CTRL_VLAN_ADD
VIRTIO_NET_CTRL_ANNOUNCE	VIRTIO_NET_CTRL_ANNOUNCE_ACK
VIRTIO_NET_CTRL_MQ	VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET
VIRTIO_NET_CTRL_MQ	VIRTIO_NET_CTRL_MQ_RSS_CONFIG
VIRTIO_NET_CTRL_MQ	VIRTIO_NET_CTRL_MQ_HASH_CONFIG

For command selector VIRTIO_NET_CTRL_VLAN_ADD, device part consists of a whole VLAN table.

reserved is reserved and set to zero.

5.1.9 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
#define VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM 8
#define VIRTIO_NET_HDR_F_SECURITY      16
#define VIRTIO_NET_HDR_F_SECURITY_ERR  32
#define VIRTIO_NET_HDR_F_SECURITY_SA_SOFT_EXPIRY_WARN 64
        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_UDP_L4      5
#define VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 0x20
#define VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 0x40
#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;
        le32 hash_value;        (Only if VIRTIO_NET_F_HASH_REPORT negotiated)
        le16 hash_report;       (Only if VIRTIO_NET_F_HASH_REPORT negotiated)
        le16 padding_reserved;  (Only if VIRTIO_NET_F_HASH_REPORT negotiated)
        le16 outer_th_offset    (Only if VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO or
            ↪ VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO negotiated)
        le16 inner_nh_offset;   (Only if VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO or
            ↪ VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO negotiated)
        le16 outer_nh_offset;   /* Only if VIRTIO_NET_F_OUT_NET_HEADER negotiated */
        /* Only if VIRTIO_NET_F_OUT_NET_HEADER or VIRTIO_NET_F_IPSEC negotiated */
        union {
                u8 padding_reserved_2[6];
                struct ipsec_resource_hdr {
                        le32 resource_id;
                        le16 resource_type;
                } ipsec_resource_hdr;
        };
};
```

The controlq is used to control various device features described further in section 5.1.9.5.

5.1.9.1 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.

5.1.9.2 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. When VIRTIO_NET_F_OUT_NET_HEADER is negotiated, the driver MAY optionally provide the *outer_-nh_offset* value. A nonzero value of *outer_nh_offset* indicates a valid outer network header with in the packet, and specifies the offset in bytes from the beginning of the packet. Otherwise *outer_nh_offset* MUST not be used.
3. 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. If the given packet has the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit set, the above checksum fields refer to the inner header checksum, see the example below.
4. If the driver negotiated VIRTIO_NET_F_HOST_TSO4, TSO6, USO or UFO, and the packet requires TCP segmentation, UDP segmentation or fragmentation, then *gso_type* is set to VIRTIO_NET_HDR_-GSO_TCPV4, TCPV6, UDP_L4 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:
   - If the VIRTIO_NET_F_GUEST_HDRLEN feature has been negotiated, *hdr_len* indicates the header length that needs to be replicated for each packet. It's the number of bytes from the beginning of the packet to the beginning of the transport payload. If the *gso_type* has the VIRTIO_NET_HDR_-GSO_UDP_TUNNEL_IPV4 bit or VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit set, *hdr_len* accounts for all the headers up to and including the inner transport. Otherwise, if the VIRTIO_-NET_F_GUEST_HDRLEN feature has not been negotiated, *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].

     Note: Some devices benefit from knowledge of the exact header length.
   - *gso_size* is the maximum size of each packet beyond that header (ie. MSS).
   - If the driver negotiated the VIRTIO_NET_F_HOST_ECN feature, the VIRTIO_NET_HDR_GSO_-ECN bit in *gso_type* indicates that the TCP packet has the ECN bit set[2].

1Due to various bugs in implementations, this field is not useful as a guarantee of the transport header size.
2This case is not handled by some older hardware, so is called out specifically in the protocol.

5. If the driver negotiated the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO feature and the *gso_type* has the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit set, the GSO protocol is encapsulated in a UDP tunnel. If the outer UDP header requires checksumming, the driver must have additionally negotiated the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM feature and offloaded the outer checksum accordingly, otherwise the outer UDP header must not require checksum validation, i.e. the outer UDP checksum must be positive zero (0x0) as defined in UDP RFC 768. The other tunnel-related fields indicate how to replicate the packet headers to cut it into smaller packets:

   - *outer_th_offset* field indicates the outer transport header within the packet. This field differs from *csum_start* as the latter points to the inner transport header within the packet.
   - *inner_nh_offset* field indicates the inner network header within the packet.

   Note: For example, consider a partially checksummed TCP (IPv4) packet carried over a Geneve UDP tunnel (again IPv4) with no tunnel options. The only relevant variable related to the tunnel type is the tunnel header length. The packet will have a 14 byte outer ethernet header, 20 byte outer IP header followed by the 8 byte UDP header (with a 0 checksum value), 8 byte Geneve header, 14 byte inner ethernet header, 20 byte inner IP header and the TCP header (with the TCP checksum field 16 bytes into that header). *csum_start* will be 14+20+8+8+14+20 = 84 (the TCP checksum includes the header), *csum_offset* will be 16. *inner_nh_offset* will be 14+20+8+8+14 = 62, *outer_th_offset* will be 14+20+8 = 42 and *gso_type* will be VIRTIO_NET_HDR_GSO_TCPV4 | VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 = 0x21

6. If the driver negotiated the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM feature, the transmitted packet is a GSO one encapsulated in a UDP tunnel, and the outer UDP header requires checksumming, the driver can skip checksumming the outer header:

   - *flags* has the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM set,
   - The outer UDP checksum field in the packet is set to the sum of the UDP pseudo header, so that replacing it by the ones' complement checksum of the outer UDP header and payload will give the correct result.

7. *num_buffers* is set to zero. This field is unused on transmitted packets.
8. 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).

5.1.9.2.1 Driver Requirements: Packet Transmission

For the transmit packet buffer, the driver MUST use the size of the structure *struct virtio_net_hdr* same as the receive packet buffer.

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 the VIRTIO_NET_F_OUT_NET_HEADER feature has been negotiated, the driver MAY set *outer_nh_offset* to nonzero value to indicate the start of the outer network header offset, if the packet contains a valid network header. Otherwise, *outer_nh_offset* is not used.

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 fragmentation, otherwise the driver MUST NOT set *gso_type* to VIRTIO_NET_HDR_GSO_UDP.

If VIRTIO_NET_F_HOST_USO is negotiated, the driver MAY set *gso_type* to VIRTIO_NET_HDR_GSO_-UDP_L4 to request UDP segmentation, otherwise the driver MUST NOT set *gso_type* to VIRTIO_NET_-HDR_GSO_UDP_L4.

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 VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO is negotiated, the driver MAY set VIRTIO_NET_HDR_-GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* according to the inner network header protocol type to request GSO packets over UDPv4 or UDPv6 tunnel segmentation, otherwise the driver MUST NOT set either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_-IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type*.

When requesting GSO segmentation over UDP tunnel, the driver MUST SET the VIRTIO_NET_HDR_-GSO_UDP_TUNNEL_IPV4 bit if the inner network header is IPv4, i.e. the packet is a TCPv4 GSO one, otherwise, if the inner network header is IPv6, the driver MUST SET the VIRTIO_NET_HDR_GSO_UDP_-TUNNEL_IPV6 bit.

The driver MUST NOT send to the device GSO packets over UDP tunnel requiring segmentation and outer UDP checksum offload, unless both the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO and VIRTIO_NET_-F_HOST_UDP_TUNNEL_GSO_CSUM features are negotiated, in which case the driver MUST set either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_-IPV6 bit in the *gso_type* and the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in the *flags*.

If VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM is not negotiated, the driver MUST not set the VIR-TIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in the *flags* and MUST NOT send to the device GSO packets over UDP tunnel requiring segmentation and outer UDP checksum offload.

The driver MUST NOT set the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_-HDR_GSO_UDP_TUNNEL_IPV6 bit together with VIRTIO_NET_HDR_GSO_UDP, as the latter is deprecated in favor of UDP_L4 and no new feature will support it.

The driver MUST NOT set the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_-HDR_GSO_UDP_TUNNEL_IPV6 bit together.

The driver MUST NOT set the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit *flags* without setting either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_-IPV6 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;

   Note: If *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE and the VIRTIO_NET_HDR_GSO_-UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit are not set in *gso_type*, *csum_offset* points to the only transport header present in the packet, and there are no additional preceding checksums validated by VIRTIO_NET_HDR_F_NEEDS_CSUM.

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, USO 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, USO or UFO options have been negotiated:

- If the VIRTIO_NET_F_GUEST_HDRLEN feature has been negotiated, and *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE, the driver MUST set *hdr_len* to a value equal to the length of the headers, including the transport header. If *gso_type* has the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit set, *hdr_len* includes the inner transport header.
- If the VIRTIO_NET_F_GUEST_HDRLEN feature has not been negotiated, or *gso_type* is VIRTIO_NET_HDR_GSO_NONE, the driver SHOULD set *hdr_len* to a value not less than the length of the headers, including the transport header.

If the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO option has been negotiated, the driver MAY set the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type*, if so:

- the driver MUST set *outer_th_offset* to the outer UDP header offset and *inner_nh_offset* to the inner network header offset. The *csum_start* and *csum_offset* fields point respectively to the inner transport header and inner transport checksum field.

If the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM feature has been negotiated, and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* are set, the driver MAY set the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags*, if so the driver MUST set the packet outer UDP header checksum to the outer UDP pseudo header checksum.

Note: calculating a ones' complement checksum from *outer_th_offset* up until the end of the packet and storing the result at offset 6 from *outer_th_offset* will result in a fully checksummed outer UDP packet;

If the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* are set and the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM feature has not been negotiated, the outer UDP header MUST NOT require checksum validation. That is, the outer UDP checksum value MUST be 0 or the validated complete checksum for such header.

Note: The valid complete checksum of the outer UDP header of individual segments can be computed by the driver prior to segmentation only if the GSO packet size is a multiple of *gso_size*, because then all segments have the same size and thus all data included in the outer UDP checksum is the same for every segment. These pre-computed segment length and checksum fields are different from those of the GSO packet. In this scenario the outer UDP header of the GSO packet must carry the segmented UDP packet length.

If the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO option has not been negotiated, the driver MUST NOT set either the VIRTIO_NET_HDR_F_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_F_GSO_UDP_TUNNEL_IPV6 in *gso_type*.

If the VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO_CSUM option has not been negotiated, the driver MUST NOT set the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags*.

The driver SHOULD accept the VIRTIO_NET_F_GUEST_HDRLEN feature if it has been offered, and if it's able to provide the exact header length.

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

The driver MUST NOT set the VIRTIO_NET_HDR_F_DATA_VALID bit in *flags* together with the VIRTIO_NET_HDR_F_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_F_GSO_UDP_TUNNEL_IPV6 bit in *gso_type*.

If the device supports IPsec Operation, the driver may set VIRTIO_NET_HDR_F_SECURITY bit in *flags*, if so:

1. the driver MUST create IPsec Outbound resource object VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA
2. the driver MUST set *resource_id* to a valid IPsec outbound resource object ID.

5.1.9.2.2 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*.

If the VIRTIO_NET_F_OUT_NET_HEADER feature has been negotiated, and *outer_nh_offset* is nonzero, the device MAY use *outer_nh_offset* as the outer network header offset. Otherwise, device MUST NOT use the *outer_nh_offset*.

If one of the VIRTIO_NET_F_HOST_TSO4, TSO6, USO or UFO options have been negotiated:

- If the VIRTIO_NET_F_GUEST_HDRLEN feature has been negotiated, and *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE, the device MAY use *hdr_len* as the transport header size.

  Note: Caution should be taken by the implementation so as to prevent a malicious driver from attacking the device by setting an incorrect hdr_len.

- If the VIRTIO_NET_F_GUEST_HDRLEN feature has not been negotiated, or *gso_type* is VIRTIO_NET_HDR_GSO_NONE, 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 both the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in in *gso_type* are set, the device MUST NOT accept the packet.

If the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* are not set, the device MUST NOT use the *outer_th_offset* and *inner_nh_offset*.

If either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* are set, and any of the following is true:

- the VIRTIO_NET_HDR_F_NEEDS_CSUM is not set in *flags*
- the VIRTIO_NET_HDR_F_DATA_VALID is set in *flags*
- the *gso_type* excluding the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit is VIRTIO_NET_HDR_GSO_NONE

the device MUST NOT accept the packet.

If the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags* is set, and both the bits VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 and VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 in *gso_type* are not set, the device MOST NOT accept the packet.

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

If VIRTIO_NET_HDR_F_SECURITY bit in *flags* is not set, the device MUST NOT use the *resource_id* and *resource_type*.

5.1.9.2.3 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.

5.1.9.3 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, VIRTIO_NET_F_GUEST_UFO, VIRTIO_NET_F_GUEST_USO4 or VIRTIO_NET_F_GUEST_USO6 features are used, the maximum incoming packet will be 65589 bytes long (14 bytes of Ethernet header, plus 40 bytes of the IPv6 header, plus 65535 bytes of maximum IPv6 payload including any extension header), otherwise 1514 bytes. When VIRTIO_NET_F_HASH_REPORT is not negotiated, the required receive buffer size is either 65601 or 1526 bytes accounting for 12 bytes of *struct virtio_net_hdr* followed by receive packet. When VIRTIO_NET_F_HASH_REPORT is negotiated, the required receive buffer size is either 65609 or 1534 bytes accounting for 20 bytes of *struct virtio_net_hdr* followed by receive packet.

5.1.9.3.1 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, VIRTIO_NET_F_GUEST_UFO, VIRTIO_NET_F_GUEST_USO4 or VIRTIO_NET_F_GUEST_USO6 are negotiated, the driver SHOULD populate the receive queue(s) with buffers of at least 65609 bytes if VIRTIO_NET_F_HASH_REPORT is negotiated, and of at least 65601 bytes if not.
  - Otherwise, the driver SHOULD populate the receive queue(s) with buffers of at least 1534 bytes if VIRTIO_NET_F_HASH_REPORT is negotiated, and of at least 1526 bytes if not.
- If VIRTIO_NET_F_MRG_RXBUF is negotiated, each buffer MUST be at least size of *struct virtio_net_hdr*, i.e. 20 bytes if VIRTIO_NET_F_HASH_REPORT is negotiated, and 12 bytes if not.

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

When calculating the size of *struct virtio_net_hdr*, the driver MUST consider all the fields inclusive up to *padding_reserved_2*, i.e. 32 bytes if VIRTIO_NET_F_OUT_NET_HEADER or VIRTIO_NET_F_IPSEC is negotiated or up to *inner_nh_offset* i.e. 24 bytes if VIRTIO_NET_F_HOST_UDP_TUNNEL_GSO is negotiated or up to *padding_reserved* i.e. 20 bytes if VIRTIO_NET_F_HASH_REPORT is negotiated, and 12 bytes if not.

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

5.1.9.3.2 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.

5.1.9.4 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. If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature has been negotiated, and the VIRTIO_NET_HDR_F_-

UDP_TUNNEL_CSUM bit is set in *flags*, both the outer UDP checksum and the inner transport checksum have been validated, otherwise only one level of checksums (the outer one in case of tunnels) has been validated.

4. If the VIRTIO_NET_F_OUT_NET_HEADER has been negotiated, and if the packet contains a valid network header, *outer_nh_offset* MAY be set to nonzero value to indicate the outer network header offset in packet.

Additionally, VIRTIO_NET_F_GUEST_CSUM, TSO4, TSO6, UDP, UDP_TUNNEL 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 5.1.9.2:

1. If the VIRTIO_NET_F_GUEST_TSO4, TSO6, UFO, USO4 or USO6 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).

   Note: If *gso_type* differs from VIRTIO_NET_HDR_GSO_NONE and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit are not set, *csum_offset* points to the only transport header present in the packet, and there are no additional preceding checksums validated by VIRTIO_NET_HDR_F_NEEDS_CSUM.
4. If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO option was negotiated and *gso_type* is not VIRTIO_NET_HDR_GSO_NONE, the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit MAY be set. In such case the *outer_th_offset* and *inner_nh_offset* fields indicate the corresponding headers information.
5. If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature was negotiated, and the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 are set in *gso_type*, the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in the *flags* can be set: if so, the outer UDP checksum has been validated and the UDP header checksum at offset 6 from from *outer_th_offset* is set to the outer UDP pseudo header checksum.

   Note: If the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit are set in *gso_type*, the *csum_start* field refers to the inner transport header offset (see Packet Transmission point 1). If the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags* is set both the inner and the outer header checksums have been validated by VIRTIO_NET_HDR_F_NEEDS_CSUM, otherwise only the inner transport header checksum has been validated.

If applicable, the device calculates per-packet hash for incoming packets as defined in 5.1.9.4.3.

If applicable, the device reports hash information for incoming packets as defined in 5.1.9.4.5.

5.1.9.4.1 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.

If none of VIRTIO_NET_F_GUEST_USO4 or VIRTIO_NET_F_GUEST_USO6 have been negotiated, the device MUST NOT set *gso_type* to VIRTIO_NET_HDR_GSO_UDP_L4.

If VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO is not negotiated, the device MUST NOT set either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type*.

If VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM is not negotiated the device MUST NOT set the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags*.

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 VIRTIO_NET_F_OUT_NET_HEADER has been negotiated, the device MAY set the *outer_nh_offset* to nonzero value to indicate outer network header offset, if packet contains a valid network header. Otherwise, the device MUST not use *outer_nh_offset*.

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;

The device MUST NOT send to the driver GSO packets encapsulated in UDP tunnel and requiring segmentation offload, unless the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO is negotiated, in which case the device MUST set the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_-GSO_UDP_TUNNEL_IPV6 bit in *gso_type* according to the inner network header protocol type, MUST set the *outer_th_offset* and *inner_nh_offset* fields to the corresponding header information, and the outer UDP header MUST NOT require checksum offload.

If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature has not been negotiated, the device MUST NOT send the driver GSO packets encapsulated in UDP tunnel and requiring segmentation and outer checksum offload.

If none of the VIRTIO_NET_F_GUEST_TSO4, TSO6, UFO, USO4 or USO6 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, UFO, USO4 or USO6 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 *gso_type* has the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit set, the referenced transport header is the inner one.

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. If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature has been negotiated, and the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit set in *flags*, both the outer UDP checksum and the inner transport checksum have been validated. Otherwise level of checksum is validated: in case of multiple encapsulated protocols the outermost one.

If either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit in *gso_type* are set, the device MUST NOT set the VIRTIO_NET_HDR_F_DATA_VALID bit in *flags*.

If the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature has been negotiated and either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit is set or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit is set in *gso_type*, the device MAY set the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit in *flags*, if so the device MUST set the packet outer UDP checksum stored in the receive buffer to the outer UDP pseudo header.

Otherwise, the VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM feature has been negotiated, either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit is set or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV6 bit is set in *gso_type*, and the bit VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM is not set in *flags*, the device MUST either provide a zero outer UDP header checksum or a fully checksummed outer UDP header.

The device MUST set the VIRTIO_NET_HDR_F_SECURITY bit in the *flags* if the packet goes through the IPsec processing. Otherwise, this bit MUST be cleared. The device MUST set or clear this bit regardless of setting VIRTIO_NET_HDR_F_SECURITY_ERR or VIRTIO_NET_HDR_F_SECURITY_SA_SOFT_EXPIRY_WARN bit.

The device MUST set the VIRTIO_NET_HDR_F_SECURITY_ERR bit in the *flags* if any error is encountered during IPsec processing. Otherwise, this bit MUST be cleared. The device MUST set or clear this bit regardless of setting VIRTIO_NET_HDR_F_SECURITY_SA_SOFT_EXPIRY_WARN bit.

The device MUST set the VIRTIO_NET_HDR_F_SECURITY_SA_SOFT_EXPIRY_WARN bit in the *flags* if the SA associated with *resource_id* reaches the SA lifetime soft limits configured in the *struct virtio_crypto_ipsec_lifetime*. See VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA.

5.1.9.4.2 Driver Requirements: Processing of Incoming Packets

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

If VIRTIO_NET_F_OUT_NET_HEADER has been negotiated, and if *outer_nh_offset* is nonzero, the driver MAY use *outer_nh_offset* as outer network header offset. Otherwise, the driver MUST not use the *outer_nh_offset*.

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, UFO, USO4 or USO6 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.

If both the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_HDR_GSO_UDP_-TUNNEL_IPV6 bit in in *gso_type* are set, the driver MUST NOT accept the packet.

If the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_-IPV6 bit in *gso_type* are not set, the driver MUST NOT use the *outer_th_offset* and *inner_nh_offset*.

If either the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit or the VIRTIO_NET_HDR_GSO_UDP_-TUNNEL_IPV6 bit in *gso_type* are set, and any of the following is true:

- the VIRTIO_NET_HDR_F_NEEDS_CSUM bit is not set in *flags*
- the VIRTIO_NET_HDR_F_DATA_VALID bit is set in *flags*
- the *gso_type* excluding the VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 bit and the VIRTIO_NET_-HDR_GSO_UDP_TUNNEL_IPV6 bit is VIRTIO_NET_HDR_GSO_NONE

the driver MUST NOT accept the packet.

If the VIRTIO_NET_HDR_F_UDP_TUNNEL_CSUM bit and the VIRTIO_NET_HDR_F_NEEDS_CSUM bit in *flags* are set, and both the bits VIRTIO_NET_HDR_GSO_UDP_TUNNEL_IPV4 and VIRTIO_NET_HDR_-GSO_UDP_TUNNEL_IPV6 in *gso_type* are not set, the driver MOST NOT accept the packet.

When VIRTIO_NET_HDR_F_SECURITY and VIRTIO_NET_HDR_F_SECURITY_ERR bits are set in the *flags*, it indicates that the device experienced a processing error on the IPsec packet. It need not be an 'error packet'. For example, a particular SA was not offloaded or SA has reached the SA lifetime hard limits configured in the *struct virtio_crypto_ipsec_lifetime*. So vnet_hdr will have both bits set.

5.1.9.4.3 Hash calculation for incoming packets

A device attempts to calculate a per-packet hash in the following cases:

- The feature VIRTIO_NET_F_RSS was negotiated. The device uses the hash to determine the receive virtqueue to place incoming packets.
- The feature VIRTIO_NET_F_HASH_REPORT was negotiated. The device reports the hash value and the hash type with the packet.

If the feature VIRTIO_NET_F_RSS was negotiated:

- The device uses *hash_types* of the virtio_net_rss_config structure as 'Enabled hash types' bitmask.
- If additionally the feature VIRTIO_NET_F_HASH_TUNNEL was negotiated, the device uses *enabled_-tunnel_types* of the virtnet_hash_tunnel structure as 'Encapsulation types enabled for inner header hash' bitmask.
- The device uses a key as defined in *hash_key_data* and *hash_key_length* of the virtio_net_rss_config structure (see 5.1.9.5.7.1).

If the feature VIRTIO_NET_F_RSS was not negotiated:

- The device uses *hash_types* of the virtio_net_hash_config structure as 'Enabled hash types' bitmask.
- If additionally the feature VIRTIO_NET_F_HASH_TUNNEL was negotiated, the device uses *enabled_-tunnel_types* of the virtnet_hash_tunnel structure as 'Encapsulation types enabled for inner header hash' bitmask.
- The device uses a key as defined in *hash_key_data* and *hash_key_length* of the virtio_net_hash_config structure (see 5.1.9.5.6.4).

Note that if the device offers VIRTIO_NET_F_HASH_REPORT, even if it supports only one pair of virtqueues, it MUST support at least one of commands of VIRTIO_NET_CTRL_MQ class to configure reported hash parameters:

- If the device offers VIRTIO_NET_F_RSS, it MUST support VIRTIO_NET_CTRL_MQ_RSS_CONFIG command per 5.1.9.5.7.1.
- Otherwise the device MUST support VIRTIO_NET_CTRL_MQ_HASH_CONFIG command per 5.1.9.5.6.4.

The per-packet hash calculation can depend on the IP packet type. See [IP], [UDP] and [TCP].

5.1.9.4.3.1 Supported/enabled hash types

Hash types applicable for IPv4 packets:

```
#define VIRTIO_NET_HASH_TYPE_IPv4              (1 << 0)
#define VIRTIO_NET_HASH_TYPE_TCPv4             (1 << 1)
#define VIRTIO_NET_HASH_TYPE_UDPv4             (1 << 2)
```

Hash types applicable for IPv6 packets without extension headers

```
#define VIRTIO_NET_HASH_TYPE_IPv6              (1 << 3)
#define VIRTIO_NET_HASH_TYPE_TCPv6             (1 << 4)
#define VIRTIO_NET_HASH_TYPE_UDPv6             (1 << 5)
```

Hash types applicable for IPv6 packets with extension headers

```
#define VIRTIO_NET_HASH_TYPE_IP_EX             (1 << 6)
#define VIRTIO_NET_HASH_TYPE_TCP_EX            (1 << 7)
#define VIRTIO_NET_HASH_TYPE_UDP_EX            (1 << 8)
```

5.1.9.4.3.2 IPv4 packets

The device calculates the hash on IPv4 packets according to 'Enabled hash types' bitmask as follows:

- If VIRTIO_NET_HASH_TYPE_TCPv4 is set and the packet has a TCP header, the hash is calculated over the following fields:
  - Source IP address
  - Destination IP address
  - Source TCP port
  - Destination TCP port
- Else if VIRTIO_NET_HASH_TYPE_UDPv4 is set and the packet has a UDP header, the hash is calculated over the following fields:
  - Source IP address
  - Destination IP address
  - Source UDP port
  - Destination UDP port
- Else if VIRTIO_NET_HASH_TYPE_IPv4 is set, the hash is calculated over the following fields:
  - Source IP address
  - Destination IP address
- Else the device does not calculate the hash

5.1.9.4.3.3 IPv6 packets without extension header

The device calculates the hash on IPv6 packets without extension headers according to 'Enabled hash types' bitmask as follows:

- If VIRTIO_NET_HASH_TYPE_TCPv6 is set and the packet has a TCPv6 header, the hash is calculated over the following fields:
  - Source IPv6 address
  - Destination IPv6 address

  - Source TCP port
  - Destination TCP port
- Else if VIRTIO_NET_HASH_TYPE_UDPv6 is set and the packet has a UDPv6 header, the hash is calculated over the following fields:
  - Source IPv6 address
  - Destination IPv6 address
  - Source UDP port
  - Destination UDP port
- Else if VIRTIO_NET_HASH_TYPE_IPv6 is set, the hash is calculated over the following fields:
  - Source IPv6 address
  - Destination IPv6 address
- Else the device does not calculate the hash

5.1.9.4.3.4 IPv6 packets with extension header

The device calculates the hash on IPv6 packets with extension headers according to 'Enabled hash types' bitmask as follows:

- If VIRTIO_NET_HASH_TYPE_TCP_EX is set and the packet has a TCPv6 header, the hash is calculated over the following fields:
  - Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 address.
  - IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 address.
  - Source TCP port
  - Destination TCP port
- Else if VIRTIO_NET_HASH_TYPE_UDP_EX is set and the packet has a UDPv6 header, the hash is calculated over the following fields:
  - Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 address.
  - IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 address.
  - Source UDP port
  - Destination UDP port
- Else if VIRTIO_NET_HASH_TYPE_IP_EX is set, the hash is calculated over the following fields:
  - Home address from the home address option in the IPv6 destination options header. If the extension header is not present, use the Source IPv6 address.
  - IPv6 address that is contained in the Routing-Header-Type-2 from the associated extension header. If the extension header is not present, use the Destination IPv6 address.
- Else skip IPv6 extension headers and calculate the hash as defined for an IPv6 packet without extension headers (see 5.1.9.4.3.3).

5.1.9.4.4 Inner Header Hash

If VIRTIO_NET_F_HASH_TUNNEL has been negotiated, the driver can send the command VIRTIO_NET_-CTRL_HASH_TUNNEL_SET to configure the calculation of the inner header hash.

```
struct virtnet_hash_tunnel {
    le32 enabled_tunnel_types;
};

#define VIRTIO_NET_CTRL_HASH_TUNNEL 7
 #define VIRTIO_NET_CTRL_HASH_TUNNEL_SET 0
```

Field *enabled_tunnel_types* contains the bitmask of encapsulation types enabled for inner header hash. See 5.1.9.4.4.2.

The class VIRTIO_NET_CTRL_HASH_TUNNEL has one command: VIRTIO_NET_CTRL_HASH_TUNNEL_SET sets *enabled_tunnel_types* for the device using the virtnet_hash_tunnel structure, which is read-only for the device.

Inner header hash is disabled by VIRTIO_NET_CTRL_HASH_TUNNEL_SET with *enabled_tunnel_types* set to 0.

Initially (before the driver sends any VIRTIO_NET_CTRL_HASH_TUNNEL_SET command) all encapsulation types are disabled for inner header hash.

5.1.9.4.4.1 Encapsulated packet

Multiple tunneling protocols allow encapsulating an inner, payload packet in an outer, encapsulated packet. The encapsulated packet thus contains an outer header and an inner header, and the device calculates the hash over either the inner header or the outer header.

If VIRTIO_NET_F_HASH_TUNNEL is negotiated and a received encapsulated packet's outer header matches one of the encapsulation types enabled in *enabled_tunnel_types*, then the device uses the inner header for hash calculations (only a single level of encapsulation is currently supported).

If VIRTIO_NET_F_HASH_TUNNEL is negotiated and a received packet's (outer) header does not match any encapsulation types enabled in *enabled_tunnel_types*, then the device uses the outer header for hash calculations.

5.1.9.4.4.2 Encapsulation types supported/enabled for inner header hash

Encapsulation types applicable for inner header hash:

```
#define VIRTIO_NET_HASH_TUNNEL_TYPE_GRE_2784    (1 << 0) /* [RFC2784] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_GRE_2890    (1 << 1) /* [RFC2890] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_GRE_7676    (1 << 2) /* [RFC7676] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_GRE_UDP     (1 << 3) /* [GRE-in-UDP] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_VXLAN       (1 << 4) /* [VXLAN] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_VXLAN_GPE   (1 << 5) /* [VXLAN-GPE] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_GENEVE      (1 << 6) /* [GENEVE] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_IPIP        (1 << 7) /* [IPIP] */
#define VIRTIO_NET_HASH_TUNNEL_TYPE_NVGRE       (1 << 8) /* [NVGRE] */
```

5.1.9.4.4.3 Advice

Example uses of the inner header hash:

- Legacy tunneling protocols, lacking the outer header entropy, can use RSS with the inner header hash to distribute flows with identical outer but different inner headers across various queues, improving performance.
- Identify an inner flow distributed across multiple outer tunnels.

As using the inner header hash completely discards the outer header entropy, care must be taken if the inner header is controlled by an adversary, as the adversary can then intentionally create configurations with insufficient entropy.

Besides disabling the inner header hash, mitigations would depend on how the hash is used. When the hash use is limited to the RSS queue selection, the inner header hash may have quality of service (QoS) limitations.

5.1.9.4.4.4 Device Requirements: Inner Header Hash

If the (outer) header of the received packet does not match any encapsulation types enabled in *enabled_tunnel_types*, the device MUST calculate the hash on the outer header.

If the device receives any bits in *enabled_tunnel_types* which are not set in *supported_tunnel_types*, it SHOULD respond to the VIRTIO_NET_CTRL_HASH_TUNNEL_SET command with VIRTIO_NET_ERR.

If the driver sets *enabled_tunnel_types* to 0 through VIRTIO_NET_CTRL_HASH_TUNNEL_SET or upon the device reset, the device MUST disable the inner header hash for all encapsulation types.

5.1.9.4.4.5 Driver Requirements: Inner Header Hash

The driver MUST have negotiated the VIRTIO_NET_F_HASH_TUNNEL feature when issuing the VIRTIO_NET_CTRL_HASH_TUNNEL_SET command.

The driver MUST NOT set any bits in *enabled_tunnel_types* which are not set in *supported_tunnel_types*.

The driver MUST ignore bits in *supported_tunnel_types* which are not documented in this specification.

5.1.9.4.5 Hash reporting for incoming packets

If VIRTIO_NET_F_HASH_REPORT was negotiated and the device has calculated the hash for the packet, the device fills *hash_report* with the report type of calculated hash and *hash_value* with the value of calculated hash.

If VIRTIO_NET_F_HASH_REPORT was negotiated but due to any reason the hash was not calculated, the device sets *hash_report* to VIRTIO_NET_HASH_REPORT_NONE.

Possible values that the device can report in *hash_report* are defined below. They correspond to supported hash types defined in 5.1.9.4.3.1 as follows:

VIRTIO_NET_HASH_TYPE_XXX = 1 « (VIRTIO_NET_HASH_REPORT_XXX - 1)

```
#define VIRTIO_NET_HASH_REPORT_NONE            0
#define VIRTIO_NET_HASH_REPORT_IPv4            1
#define VIRTIO_NET_HASH_REPORT_TCPv4           2
#define VIRTIO_NET_HASH_REPORT_UDPv4           3
#define VIRTIO_NET_HASH_REPORT_IPv6            4
#define VIRTIO_NET_HASH_REPORT_TCPv6           5
#define VIRTIO_NET_HASH_REPORT_UDPv6           6
#define VIRTIO_NET_HASH_REPORT_IPv6_EX         7
#define VIRTIO_NET_HASH_REPORT_TCPv6_EX        8
#define VIRTIO_NET_HASH_REPORT_UDPv6_EX        9
```

5.1.9.5 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;
```

```
        u8 command-specific-result[];
};

/* 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 and optionally *command-specific-result*. There is little the driver can do except issue a diagnostic if *ack* is not VIRTIO_NET_OK.

The command VIRTIO_NET_CTRL_STATS_QUERY and VIRTIO_NET_CTRL_STATS_GET contain *command-specific-result*.

5.1.9.5.1 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
```

5.1.9.5.1.1 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 promiscuous 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 broad-

cast 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.

5.1.9.5.1.2 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.

5.1.9.5.2 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 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.

5.1.9.5.2.1 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_TABLE_SET command.

The device MUST update *mac* in config space before it consumes the VIRTIO_NET_CTRL_MAC_ADDR_SET command, if VIRTIO_NET_F_MAC 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.

5.1.9.5.2.2 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.

5.1.9.5.2.3 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.

5.1.9.5.3 VLAN Filtering

If the driver negotiates the VIRTIO_NET_F_CTRL_VLAN feature, it can control a VLAN filter table in the device. The VLAN filter table applies only to VLAN tagged packets.

When VIRTIO_NET_F_CTRL_VLAN is negotiated, the device starts with an empty VLAN filter table.

Note: Similar to the MAC address based filtering, the VLAN filtering is also best-effort: unwanted packets could still arrive.

```
#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.

VIRTIO_NET_CTRL_VLAN_ADD command adds the specified VLAN to the VLAN filter table.

VIRTIO_NET_CTRL_VLAN_DEL command removes the specified VLAN from the VLAN filter table.

5.1.9.5.3.1 Device Requirements: VLAN Filtering

When VIRTIO_NET_F_CTRL_VLAN is not negotiated, the device MUST accept all VLAN tagged packets.

When VIRTIO_NET_F_CTRL_VLAN is negotiated, the device MUST accept all VLAN tagged packets whose VLAN tag is present in the VLAN filter table and SHOULD drop all VLAN tagged packets whose VLAN tag is absent in the VLAN filter table.

5.1.9.5.3.2 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.

5.1.9.5.4 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.

5.1.9.5.4.1 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.

5.1.9.5.4.2 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.

5.1.9.5.5 Device operation in multiqueue mode

This specification defines the following modes that a device MAY implement for operation with multiple transmit/receive virtqueues:

- Automatic receive steering as defined in 5.1.9.5.6. If a device supports this mode, it offers the VIRTIO_NET_F_MQ feature bit.
- Receive-side scaling as defined in 5.1.9.5.7.3. If a device supports this mode, it offers the VIRTIO_NET_F_RSS feature bit.

A device MAY support one of these features or both. The driver MAY negotiate any set of these features that the device supports.

Multiqueue is disabled by default.

The driver enables multiqueue by sending a command using *class* VIRTIO_NET_CTRL_MQ. The *command* selects the mode of multiqueue operation, as follows:

```
#define VIRTIO_NET_CTRL_MQ    4
 #define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET        0 (for automatic receive steering)
 #define VIRTIO_NET_CTRL_MQ_RSS_CONFIG          1 (for configurable receive steering)
 #define VIRTIO_NET_CTRL_MQ_HASH_CONFIG         2 (for configurable hash calculation)
```

If more than one multiqueue mode is negotiated, the resulting device configuration is defined by the last command sent by the driver.

5.1.9.5.6 Automatic receive steering in multiqueue mode

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

The driver enables multiqueue by sending 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.

```
struct virtio_net_ctrl_mq_pairs_set {
        le16 virtqueue_pairs;
};
#define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_MIN         1
#define VIRTIO_NET_CTRL_MQ_VQ_PAIRS_MAX         0x8000
```

When multiqueue is enabled by VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET command, the device MUST use automatic receive steering based on packet flow. Programming of the receive steering classificator is

implicit. After the driver transmitted a packet of a flow on transmitqX, the device SHOULD cause incoming packets for that flow to be steered to receiveqX. For uni-directional protocols, or where no packets have been transmitted yet, the device MAY steer a packet to a random queue out of the specified receiveq1. . .receiveqn.

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

5.1.9.5.6.1 Driver Requirements: Automatic receive steering in multiqueue mode

The driver MUST configure the virtqueues before enabling them with the VIRTIO_NET_CTRL_MQ_VQ_-PAIRS_SET command.

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

The driver MUST queue packets only on any transmitq1 before the VIRTIO_NET_CTRL_MQ_VQ_PAIRS_-SET command.

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

5.1.9.5.6.2 Device Requirements: Automatic receive steering in multiqueue mode

After initialization of reset, the device MUST queue packets only on receiveq1.

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.

If the destination receive queue is being reset (See 2.6.1), the device SHOULD re-select another random queue. If all receive queues are being reset, the device MUST drop the packet.

5.1.9.5.6.3 Legacy Interface: Automatic receive steering in multiqueue mode

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

5.1.9.5.6.4 Hash calculation

If VIRTIO_NET_F_HASH_REPORT was negotiated and the device uses automatic receive steering, the device MUST support a command to configure hash calculation parameters.

The driver provides parameters for hash calculation as follows:

class VIRTIO_NET_CTRL_MQ, *command* VIRTIO_NET_CTRL_MQ_HASH_CONFIG.

The *command-specific-data* has following format:

```
struct virtio_net_hash_config {
    le32 hash_types;
    le16 reserved[4];
    u8 hash_key_length;
    u8 hash_key_data[hash_key_length];
};
```

Field *hash_types* contains a bitmask of allowed hash types as defined in 5.1.9.4.3.1. Initially the device has all hash types disabled and reports only VIRTIO_NET_HASH_REPORT_NONE.

Field *reserved* MUST contain zeroes. It is defined to make the structure to match the layout of virtio_net_-rss_config structure, defined in 5.1.9.5.7.

Fields *hash_key_length* and *hash_key_data* define the key to be used in hash calculation.

5.1.9.5.7 Receive-side scaling (RSS)

A device offers the feature VIRTIO_NET_F_RSS if it supports RSS receive steering with Toeplitz hash calculation and configurable parameters.

A driver queries RSS capabilities of the device by reading device configuration as defined in 5.1.4

5.1.9.5.7.1 Setting RSS parameters

Driver sends a VIRTIO_NET_CTRL_MQ_RSS_CONFIG command using the following format for *command-specific-data*:

```
struct rss_rq_id {
   le16 vq_index_1_16: 15; /* Bits 1 to 16 of the virtqueue index */
   le16 reserved: 1; /* Set to zero */
};

struct virtio_net_rss_config {
    le32 hash_types;
    le16 indirection_table_mask;
    struct rss_rq_id unclassified_queue;
    struct rss_rq_id indirection_table[indirection_table_length];
    le16 max_tx_vq;
    u8 hash_key_length;
    u8 hash_key_data[hash_key_length];
};
```

Field *hash_types* contains a bitmask of allowed hash types as defined in 5.1.9.4.3.1.

Field *indirection_table_mask* is a mask to be applied to the calculated hash to produce an index in the *indirection_table* array. Number of entries in *indirection_table* is (*indirection_table_mask* + 1).

rss_rq_id is a receive virtqueue id. *vq_index_1_16* consists of bits 1 to 16 of a virtqueue index. For example, a *vq_index_1_16* value of 3 corresponds to virtqueue index 6, which maps to receiveq4.

Field *unclassified_queue* specifies the receive virtqueue id in which to place unclassified packets.

Field *indirection_table* is an array of receive virtqueues ids.

A driver sets *max_tx_vq* to inform a device how many transmit virtqueues it may use (transmitq1. . .transmitq *max_tx_vq*).

Fields *hash_key_length* and *hash_key_data* define the key to be used in hash calculation.

5.1.9.5.7.2 Driver Requirements: Setting RSS parameters

A driver MUST NOT send the VIRTIO_NET_CTRL_MQ_RSS_CONFIG command if the feature VIRTIO_-NET_F_RSS has not been negotiated.

A driver MUST fill the *indirection_table* array only with enabled receive virtqueues ids.

The number of entries in *indirection_table* (*indirection_table_mask* + 1) MUST be a power of two.

A driver MUST use *indirection_table_mask* values that are less than *rss_max_indirection_table_length* reported by a device.

A driver MUST NOT set any VIRTIO_NET_HASH_TYPE_ flags that are not supported by a device.

5.1.9.5.7.3 Device Requirements: RSS processing

The device MUST determine the destination queue for a network packet as follows:

- Calculate the hash of the packet as defined in 5.1.9.4.3.
- If the device did not calculate the hash for the specific packet, the device directs the packet to the receiveq specified by *unclassified_queue* of virtio_net_rss_config structure.

- Apply *indirection_table_mask* to the calculated hash and use the result as the index in the indirection table to get the destination receive virtqueue id.
- If the destination receive queue is being reset (See 2.6.1), the device MUST drop the packet.

5.1.9.5.8 RSS Context

An RSS context consists of configurable parameters specified by 5.1.9.5.7.

The RSS configuration supported by VIRTIO_NET_F_RSS is considered the default RSS configuration.

The device offers the feature VIRTIO_NET_F_RSS_CONTEXT if it supports one or multiple RSS contexts (excluding the default RSS configuration) and configurable parameters.

5.1.9.5.8.1 Querying RSS Context Capability

```
#define VIRTNET_RSS_CTX_CTRL 9
 #define VIRTNET_RSS_CTX_CTRL_CAP_GET  0
 #define VIRTNET_RSS_CTX_CTRL_ADD      1
 #define VIRTNET_RSS_CTX_CTRL_MOD      2
 #define VIRTNET_RSS_CTX_CTRL_DEL      3

struct virtnet_rss_ctx_cap {
    le16 max_rss_contexts;
}
```

Field *max_rss_contexts* specifies the maximum number of RSS contexts 5.1.9.5.8 supported by the device.

The driver queries the RSS context capability of the device by sending the command VIRTNET_RSS_-CTX_CTRL_CAP_GET with the structure virtnet_rss_ctx_cap.

For the command VIRTNET_RSS_CTX_CTRL_CAP_GET, the structure virtnet_rss_ctx_cap is write-only for the device.

5.1.9.5.8.2 Setting RSS Context Parameters

```
struct virtnet_rss_ctx_add_modify {
    le16 rss_ctx_id;
    u8 reserved[6];
    struct virtio_net_rss_config rss;
};

struct virtnet_rss_ctx_del {
    le16 rss_ctx_id;
};
```

RSS context parameters:

- *rss_ctx_id*: ID of the specific RSS context.
- *rss*: RSS context parameters of the specific RSS context whose id is *rss_ctx_id*.

reserved is reserved and it is ignored by the device.

If the feature VIRTIO_NET_F_RSS_CONTEXT has been negotiated, the driver can send the following VIRT-NET_RSS_CTX_CTRL class commands:

1. VIRTNET_RSS_CTX_CTRL_ADD: use the structure virtnet_rss_ctx_add_modify to add an RSS context configured as *rss* and id as *rss_ctx_id* for the device.
2. VIRTNET_RSS_CTX_CTRL_MOD: use the structure virtnet_rss_ctx_add_modify to configure parameters of the RSS context whose id is *rss_ctx_id* as *rss* for the device.
3. VIRTNET_RSS_CTX_CTRL_DEL: use the structure virtnet_rss_ctx_del to delete the RSS context whose id is *rss_ctx_id* for the device.

For commands VIRTNET_RSS_CTX_CTRL_ADD and VIRTNET_RSS_CTX_CTRL_MOD, the structure virtnet_rss_ctx_add_modify is read-only for the device. For the command VIRTNET_RSS_CTX_CTRL_-DEL, the structure virtnet_rss_ctx_del is read-only for the device.

5.1.9.5.8.3 Device Requirements: RSS Context

The device MUST set *max_rss_contexts* to at least 1 if it offers VIRTIO_NET_F_RSS_CONTEXT.

Upon reset, the device MUST clear all previously configured RSS contexts.

5.1.9.5.8.4 Driver Requirements: RSS Context

The driver MUST have negotiated the VIRTIO_NET_F_RSS_CONTEXT feature when issuing the VIRT-NET_RSS_CTX_CTRL class commands.

The driver MUST set *rss_ctx_id* to between 1 and *max_rss_contexts* inclusive.

The driver MUST NOT send the command VIRTIO_NET_CTRL_MQ_VQ_PAIRS_SET when the device has successfully configured at least one RSS context.

5.1.9.5.9 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.

5.1.9.5.9.1 Setting Offloads State

To configure the offloads, the following layout structure and definitions are used:

```
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_F_GUEST_UDP_TUNNEL_GSO  46
#define VIRTIO_NET_F_GUEST_UDP_TUNNEL_GSO_CSUM 47
#define VIRTIO_NET_F_GUEST_USO4       54
#define VIRTIO_NET_F_GUEST_USO6       55

#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 compatibility.

5.1.9.5.9.2 Driver Requirements: Setting Offloads State

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

5.1.9.5.9.3 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.

5.1.9.5.10 Notifications Coalescing

If the VIRTIO_NET_F_NOTF_COAL feature is negotiated, the driver can send commands VIRTIO_NET_CTRL_NOTF_COAL_TX_SET and VIRTIO_NET_CTRL_NOTF_COAL_RX_SET for notification coalescing.

If the VIRTIO_NET_F_VQ_NOTF_COAL feature is negotiated, the driver can send commands VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET and VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET for virtqueue notification coalescing.

```
struct virtio_net_ctrl_coal {
    le32 max_packets;
    le32 max_usecs;
};

struct virtio_net_ctrl_coal_vq {
    le16 vq_index;
    le16 reserved;
    struct virtio_net_ctrl_coal coal;
};

#define VIRTIO_NET_CTRL_NOTF_COAL 6
 #define VIRTIO_NET_CTRL_NOTF_COAL_TX_SET  0
 #define VIRTIO_NET_CTRL_NOTF_COAL_RX_SET 1
 #define VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET 2
 #define VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET 3
```

Coalescing parameters:

- *vq_index*: The virtqueue index of an enabled transmit or receive virtqueue.
- *max_usecs* for RX: Maximum number of microseconds to delay a RX notification.
- *max_usecs* for TX: Maximum number of microseconds to delay a TX notification.
- *max_packets* for RX: Maximum number of packets to receive before a RX notification.
- *max_packets* for TX: Maximum number of packets to send before a TX notification.

reserved is reserved and it is ignored by the device.

Read/Write attributes for coalescing parameters:

- For commands VIRTIO_NET_CTRL_NOTF_COAL_TX_SET and VIRTIO_NET_CTRL_NOTF_COAL_RX_SET, the structure virtio_net_ctrl_coal is write-only for the driver.
- For the command VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET, the structure virtio_net_ctrl_coal_vq is write-only for the driver.
- For the command VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET, *vq_index* and *reserved* are write-only for the driver, and the structure virtio_net_ctrl_coal is read-only for the driver.

The class VIRTIO_NET_CTRL_NOTF_COAL has the following commands:

1. VIRTIO_NET_CTRL_NOTF_COAL_TX_SET: use the structure virtio_net_ctrl_coal to set the *max_usecs* and *max_packets* parameters for all transmit virtqueues.
2. VIRTIO_NET_CTRL_NOTF_COAL_RX_SET: use the structure virtio_net_ctrl_coal to set the *max_usecs* and *max_packets* parameters for all receive virtqueues.
3. VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET: use the structure virtio_net_ctrl_coal_vq to set the *max_usecs* and *max_packets* parameters for an enabled transmit/receive virtqueue whose index is *vq_index*.
4. VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET: use the structure virtio_net_ctrl_coal_vq to get the *max_usecs* and *max_packets* parameters for an enabled transmit/receive virtqueue whose index is *vq_index*.

The device may generate notifications more or less frequently than specified by set commands of the VIRTIO_NET_CTRL_NOTF_COAL class.

If coalescing parameters are being set, the device applies the last coalescing parameters set for a virtqueue, regardless of the command used to set the parameters. Use the following command sequence with two pairs of virtqueues as an example: Each of the following commands sets *max_usecs* and *max_packets* parameters for virtqueues.

- Command1: VIRTIO_NET_CTRL_NOTF_COAL_RX_SET sets coalescing parameters for virtqueues having index 0 and index 2. Virtqueues having index 1 and index 3 retain their previous parameters.
- Command2: VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET with *vq_index* = 0 sets coalescing parameters for virtqueue having index 0. Virtqueue having index 2 retains the parameters from command1.
- Command3: VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET with *vq_index* = 0, the device responds with coalescing parameters of vq_index 0 set by command2.
- Command4: VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET with *vq_index* = 1 sets coalescing parameters for virtqueue having index 1. Virtqueue having index 3 retains its previous parameters.
- Command5: VIRTIO_NET_CTRL_NOTF_COAL_TX_SET sets coalescing parameters for virtqueues having index 1 and index 3, and overrides the parameters set by command4.
- Command6: VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET with *vq_index* = 1, the device responds with coalescing parameters of index 1 set by command5.

5.1.9.5.10.1 Operation

The device sends a used buffer notification once the notification conditions are met and if the notifications are not suppressed as explained in 2.7.7.

When the device has non-zero *max_usecs* and non-zero *max_packets*, it starts counting microseconds and packets upon receiving/sending a packet. The device counts packets and microseconds for each receive virtqueue and transmit virtqueue separately. In this case, the notification conditions are met when *max_usecs* microseconds elapse, or upon sending/receiving *max_packets* packets, whichever happens first. Afterwards, the device waits for the next packet and starts counting packets and microseconds again.

When the device has *max_usecs* = 0 or *max_packets* = 0, the notification conditions are met after every packet received/sent.

5.1.9.5.10.2 RX Example

If, for example:

- *max_usecs* = 10.
- *max_packets* = 15.

then each receive virtqueue of a device will operate as follows:

- The device will count packets received on each virtqueue until it accumulates 15, or until 10 microseconds elapsed since the first one was received.
- If the notifications are not suppressed by the driver, the device will send an used buffer notification, otherwise, the device will not send an used buffer notification as long as the notifications are suppressed.

5.1.9.5.10.3 TX Example

If, for example:

- *max_usecs* = 10.
- *max_packets* = 15.

then each transmit virtqueue of a device will operate as follows:

- The device will count packets sent on each virtqueue until it accumulates 15, or until 10 microseconds elapsed since the first one was sent.

- If the notifications are not suppressed by the driver, the device will send an used buffer notification, otherwise, the device will not send an used buffer notification as long as the notifications are suppressed.

5.1.9.5.10.4 Notifications When Coalescing Parameters Change

When the coalescing parameters of a device change, the device needs to check if the new notification conditions are met and send a used buffer notification if so.

For example, *max_packets* = 15 for a device with a single transmit virtqueue: if the device sends 10 packets and afterwards receives a VIRTIO_NET_CTRL_NOTF_COAL_TX_SET command with *max_packets* = 8, then the notification condition is immediately considered to be met; the device needs to immediately send a used buffer notification, if the notifications are not suppressed by the driver.

5.1.9.5.10.5 Driver Requirements: Notifications Coalescing

The driver MUST set *vq_index* to the virtqueue index of an enabled transmit or receive virtqueue.

The driver MUST have negotiated the VIRTIO_NET_F_NOTF_COAL feature when issuing commands VIRTIO_NET_CTRL_NOTF_COAL_TX_SET and VIRTIO_NET_CTRL_NOTF_COAL_RX_SET.

The driver MUST have negotiated the VIRTIO_NET_F_VQ_NOTF_COAL feature when issuing commands VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET and VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET.

The driver MUST ignore the values of coalescing parameters received from the VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET command if the device responds with VIRTIO_NET_ERR.

5.1.9.5.10.6 Device Requirements: Notifications Coalescing

The device MUST ignore *reserved*.

The device SHOULD respond to VIRTIO_NET_CTRL_NOTF_COAL_TX_SET and VIRTIO_NET_CTRL_NOTF_COAL_RX_SET commands with VIRTIO_NET_ERR if it was not able to change the parameters.

The device MUST respond to the VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET command with VIRTIO_NET_ERR if it was not able to change the parameters.

The device MUST respond to VIRTIO_NET_CTRL_NOTF_COAL_VQ_SET and VIRTIO_NET_CTRL_NOTF_COAL_VQ_GET commands with VIRTIO_NET_ERR if the designated virtqueue is not an enabled transmit or receive virtqueue.

Upon disabling and re-enabling a transmit virtqueue, the device MUST set the coalescing parameters of the virtqueue to those configured through the VIRTIO_NET_CTRL_NOTF_COAL_TX_SET command, or, if the driver did not set any TX coalescing parameters, to 0.

Upon disabling and re-enabling a receive virtqueue, the device MUST set the coalescing parameters of the virtqueue to those configured through the VIRTIO_NET_CTRL_NOTF_COAL_RX_SET command, or, if the driver did not set any RX coalescing parameters, to 0.

The behavior of the device in response to set commands of the VIRTIO_NET_CTRL_NOTF_COAL class is best-effort: the device MAY generate notifications more or less frequently than specified.

A device SHOULD NOT send used buffer notifications to the driver if the notifications are suppressed, even if the notification conditions are met.

Upon reset, a device MUST initialize all coalescing parameters to 0.

5.1.9.5.11 Device Statistics

If the VIRTIO_NET_F_DEVICE_STATS feature is negotiated, the driver can obtain device statistics from the device by using the following command.

Different types of virtqueues have different statistics. The statistics of the receiveq are different from those of the transmitq.

The statistics of a certain type of virtqueue are also divided into multiple types because different types require different features. This enables the expansion of new statistics.

In one command, the driver can obtain the statistics of one or multiple virtqueues. Additionally, the driver can obtain multiple type statistics of each virtqueue.

5.1.9.5.11.1 Query Statistic Capabilities

```
#define VIRTIO_NET_CTRL_STATS         8
#define VIRTIO_NET_CTRL_STATS_QUERY   0
#define VIRTIO_NET_CTRL_STATS_GET     1

struct virtio_net_stats_capabilities {

#define VIRTIO_NET_STATS_TYPE_CVQ       (1 << 32)

#define VIRTIO_NET_STATS_TYPE_RX_BASIC  (1 << 0)
#define VIRTIO_NET_STATS_TYPE_RX_CSUM   (1 << 1)
#define VIRTIO_NET_STATS_TYPE_RX_GSO    (1 << 2)
#define VIRTIO_NET_STATS_TYPE_RX_SPEED  (1 << 3)

#define VIRTIO_NET_STATS_TYPE_TX_BASIC  (1 << 16)
#define VIRTIO_NET_STATS_TYPE_TX_CSUM   (1 << 17)
#define VIRTIO_NET_STATS_TYPE_TX_GSO    (1 << 18)
#define VIRTIO_NET_STATS_TYPE_TX_SPEED  (1 << 19)

    le64 supported_stats_types[1];
}
```

To obtain device statistic capability, use the VIRTIO_NET_CTRL_STATS_QUERY command. When the command completes successfully, *command-specific-result* is in the format of *struct virtio_net_stats_capabilities*.

5.1.9.5.11.2 Get Statistics

```
struct virtio_net_ctrl_queue_stats {
       struct {
           le16 vq_index;
           le16 reserved[3];
           le64 types_bitmap[1];
       } stats[];
};

struct virtio_net_stats_reply_hdr {
#define VIRTIO_NET_STATS_TYPE_REPLY_CVQ       32

#define VIRTIO_NET_STATS_TYPE_REPLY_RX_BASIC  0
#define VIRTIO_NET_STATS_TYPE_REPLY_RX_CSUM   1
#define VIRTIO_NET_STATS_TYPE_REPLY_RX_GSO    2
#define VIRTIO_NET_STATS_TYPE_REPLY_RX_SPEED  3

#define VIRTIO_NET_STATS_TYPE_REPLY_TX_BASIC  16
#define VIRTIO_NET_STATS_TYPE_REPLY_TX_CSUM   17
#define VIRTIO_NET_STATS_TYPE_REPLY_TX_GSO    18
#define VIRTIO_NET_STATS_TYPE_REPLY_TX_SPEED  19
    u8 type;
    u8 reserved;
    le16 vq_index;
    le16 reserved1;
    le16 size;
}
```

To obtain device statistics, use the VIRTIO_NET_CTRL_STATS_GET command with the *command-specific-data* which is in the format of *struct virtio_net_ctrl_queue_stats*. When the command completes successfully, *command-specific-result* contains multiple statistic results, each statistic result has the *struct virtio_net_stats_reply_hdr* as the header.

The fields of the *struct virtio_net_ctrl_queue_stats*:

vq_index The index of the virtqueue to obtain the statistics.

types_bitmap This is a bitmask of the types of statistics to be obtained. Therefore, a *stats* inside *struct virtio_net_ctrl_queue_stats* may indicate multiple statistic replies for the virtqueue.

The fields of the *struct virtio_net_stats_reply_hdr*:

type The type of the reply statistic.

vq_index The virtqueue index of the reply statistic.

size The number of bytes for the statistics entry including size of *struct virtio_net_stats_reply_hdr*.

5.1.9.5.11.3 Controlq Statistics

The structure corresponding to the controlq statistics is *struct virtio_net_stats_cvq*. The corresponding type is VIRTIO_NET_STATS_TYPE_CVQ. This is for the controlq.

```
struct virtio_net_stats_cvq {
    struct virtio_net_stats_reply_hdr hdr;

    le64 command_num;
    le64 ok_num;
};
```

command_num The number of commands received by the device including the current command.

ok_num The number of commands completed successfully by the device including the current command.

5.1.9.5.11.4 Receiveq Basic Statistics

The structure corresponding to the receiveq basic statistics is *struct virtio_net_stats_rx_basic*. The corresponding type is VIRTIO_NET_STATS_TYPE_RX_BASIC. This is for the receiveq.

Receiveq basic statistics do not require any feature. As long as the device supports VIRTIO_NET_F_DEVICE_STATS, the following are the receiveq basic statistics.

```
struct virtio_net_stats_rx_basic {
    struct virtio_net_stats_reply_hdr hdr;

    le64 rx_notifications;

    le64 rx_packets;
    le64 rx_bytes;

    le64 rx_interrupts;

    le64 rx_drops;
    le64 rx_drop_overruns;
};
```

The packets described below were all presented on the specified virtqueue.

rx_notifications The number of driver notifications received by the device for this receiveq.

rx_packets This is the number of packets passed to the driver by the device.

rx_bytes This is the bytes of packets passed to the driver by the device.

rx_interrupts The number of interrupts generated by the device for this receiveq.

rx_drops This is the number of packets dropped by the device. The count includes all types of packets dropped by the device.

rx_drop_overruns This is the number of packets dropped by the device when no more descriptors were available.

5.1.9.5.11.5 Transmitq Basic Statistics

The structure corresponding to the transmitq basic statistics is *struct virtio_net_stats_tx_basic*. The corresponding type is VIRTIO_NET_STATS_TYPE_TX_BASIC. This is for the transmitq.

Transmitq basic statistics do not require any feature. As long as the device supports VIRTIO_NET_F_DEVICE_STATS, the following are the transmitq basic statistics.

```
struct virtio_net_stats_tx_basic {
    struct virtio_net_stats_reply_hdr hdr;

    le64 tx_notifications;

    le64 tx_packets;
    le64 tx_bytes;

    le64 tx_interrupts;

    le64 tx_drops;
    le64 tx_drop_malformed;
};
```

The packets described below are all for a specific virtqueue.

tx_notifications The number of driver notifications received by the device for this transmitq.

tx_packets This is the number of packets sent by the device (not the packets got from the driver).

tx_bytes This is the number of bytes sent by the device for all the sent packets (not the bytes sent got from the driver).

tx_interrupts The number of interrupts generated by the device for this transmitq.

tx_drops The number of packets dropped by the device. The count includes all types of packets dropped by the device.

tx_drop_malformed The number of packets dropped by the device, when the descriptors are malformed. For example, the buffer is too short.

5.1.9.5.11.6 Receiveq CSUM Statistics

The structure corresponding to the receiveq checksum statistics is *struct virtio_net_stats_rx_csum*. The corresponding type is VIRTIO_NET_STATS_TYPE_RX_CSUM. This is for the receiveq.

Only after the VIRTIO_NET_F_GUEST_CSUM is negotiated, the receiveq checksum statistics can be obtained.

```
struct virtio_net_stats_rx_csum {
    struct virtio_net_stats_reply_hdr hdr;

    le64 rx_csum_valid;
    le64 rx_needs_csum;
    le64 rx_csum_none;
    le64 rx_csum_bad;
};
```

The packets described below were all presented on the specified virtqueue.

rx_csum_valid The number of packets with VIRTIO_NET_HDR_F_DATA_VALID.

rx_needs_csum The number of packets with VIRTIO_NET_HDR_F_NEEDS_CSUM.

rx_csum_none The number of packets without hardware checksum. The packet here refers to the non-TCP/UDP packet that the device cannot recognize.

rx_csum_bad The number of packets with checksum mismatch.

5.1.9.5.11.7 Transmitq CSUM Statistics

The structure corresponding to the transmitq checksum statistics is *struct virtio_net_stats_tx_csum*. The corresponding type is VIRTIO_NET_STATS_TYPE_TX_CSUM. This is for the transmitq.

Only after the VIRTIO_NET_F_CSUM is negotiated, the transmitq checksum statistics can be obtained.

The following are the transmitq checksum statistics:

```
struct virtio_net_stats_tx_csum {
    struct virtio_net_stats_reply_hdr hdr;

    le64 tx_csum_none;
    le64 tx_needs_csum;
};
```

The packets described below are all for a specific virtqueue.

tx_csum_none The number of packets which do not require hardware checksum.

tx_needs_csum The number of packets which require checksum calculation by the device.

5.1.9.5.11.8 Receiveq GSO Statistics

The structure corresponding to the receivq GSO statistics is *struct virtio_net_stats_rx_gso*. The corresponding type is VIRTIO_NET_STATS_TYPE_RX_GSO. This is for the receiveq.

If one or more of the VIRTIO_NET_F_GUEST_TSO4, VIRTIO_NET_F_GUEST_TSO6 have been negotiated, the receiveq GSO statistics can be obtained.

GSO packets refer to packets passed by the device to the driver where *gso_type* is not VIRTIO_NET_HDR_GSO_NONE.

```
struct virtio_net_stats_rx_gso {
    struct virtio_net_stats_reply_hdr hdr;

    le64 rx_gso_packets;
    le64 rx_gso_bytes;
    le64 rx_gso_packets_coalesced;
    le64 rx_gso_bytes_coalesced;
};
```

The packets described below were all presented on the specified virtqueue.

rx_gso_packets The number of the GSO packets received by the device.

rx_gso_bytes The bytes of the GSO packets received by the device. This includes the header size of the GSO packet.

rx_gso_packets_coalesced The number of the GSO packets coalesced by the device.

rx_gso_bytes_coalesced The bytes of the GSO packets coalesced by the device. This includes the header size of the GSO packet.

5.1.9.5.11.9 Transmitq GSO Statistics

The structure corresponding to the transmitq GSO statistics is *struct virtio_net_stats_tx_gso*. The corresponding type is VIRTIO_NET_STATS_TYPE_TX_GSO. This is for the transmitq.

If one or more of the VIRTIO_NET_F_HOST_TSO4, VIRTIO_NET_F_HOST_TSO6, VIRTIO_NET_F_HOST_USO options have been negotiated, the transmitq GSO statistics can be obtained.

GSO packets refer to packets passed by the driver to the device where *gso_type* is not VIRTIO_NET_HDR_GSO_NONE. See more 5.1.9.2.

```
struct virtio_net_stats_tx_gso {
    struct virtio_net_stats_reply_hdr hdr;

    le64 tx_gso_packets;
    le64 tx_gso_bytes;
    le64 tx_gso_segments;
    le64 tx_gso_segments_bytes;
    le64 tx_gso_packets_noseg;
    le64 tx_gso_bytes_noseg;
};
```

The packets described below are all for a specific virtqueue.

tx_gso_packets The number of the GSO packets sent by the device.

tx_gso_bytes The bytes of the GSO packets sent by the device.

tx_gso_segments The number of segments prepared from GSO packets.

tx_gso_segments_bytes The bytes of segments prepared from GSO packets.

tx_gso_packets_noseg The number of the GSO packets without segmentation.

tx_gso_bytes_noseg The bytes of the GSO packets without segmentation.

5.1.9.5.11.10 Receiveq Speed Statistics

The structure corresponding to the receiveq speed statistics is *struct virtio_net_stats_rx_speed*. The corresponding type is VIRTIO_NET_STATS_TYPE_RX_SPEED. This is for the receiveq.

The device has the allowance for the speed. If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, the driver can get this by *speed*. When the received packets bitrate exceeds the *speed*, some packets may be dropped by the device.

```
struct virtio_net_stats_rx_speed {
    struct virtio_net_stats_reply_hdr hdr;

    le64 rx_packets_allowance_exceeded;
    le64 rx_bytes_allowance_exceeded;
};
```

The packets described below were all presented on the specified virtqueue.

rx_packets_allowance_exceeded The number of the packets dropped by the device due to the received packets bitrate exceeding the *speed*.

rx_bytes_allowance_exceeded The bytes of the packets dropped by the device due to the received packets bitrate exceeding the *speed*.

5.1.9.5.11.11 Transmitq Speed Statistics

The structure corresponding to the transmitq speed statistics is *struct virtio_net_stats_tx_speed*. The corresponding type is VIRTIO_NET_STATS_TYPE_TX_SPEED. This is for the transmitq.

The device has the allowance for the speed. If VIRTIO_NET_F_SPEED_DUPLEX has been negotiated, the driver can get this by *speed*. When the transmit packets bitrate exceeds the *speed*, some packets may be dropped by the device.

```
struct virtio_net_stats_tx_speed {
    struct virtio_net_stats_reply_hdr hdr;

    le64 tx_packets_allowance_exceeded;
    le64 tx_bytes_allowance_exceeded;
};
```

The packets described below were all presented on the specified virtqueue.

tx_packets_allowance_exceeded The number of the packets dropped by the device due to the transmit packets bitrate exceeding the *speed*.

tx_bytes_allowance_exceeded The bytes of the packets dropped by the device due to the transmit packets bitrate exceeding the *speed*.

5.1.9.5.11.12 Device Requirements: Device Statistics

When the VIRTIO_NET_F_DEVICE_STATS feature is negotiated, the device MUST reply to the command VIRTIO_NET_CTRL_STATS_QUERY with the *struct virtio_net_stats_capabilities*. *supported_stats_types* includes all the statistic types supported by the device.

If *struct virtio_net_ctrl_queue_stats* is incorrect (such as the following), the device MUST set *ack* to VIRTIO_NET_ERR. Even if there is only one error, the device MUST fail the entire command.

- *vq_index* exceeds the queue range.
- *types_bitmap* contains unknown types.
- One or more of the bits present in *types_bitmap* is not valid for the specified virtqueue.
- The feature corresponding to the specified *types_bitmap* was not negotiated.

The device MUST set the actual size of the bytes occupied by the reply to the *size* of the *hdr*. And the device MUST set the *type* and the *vq_index* of the statistic header.

The *command-specific-result* buffer allocated by the driver may be smaller or bigger than all the statistics specified by *struct virtio_net_ctrl_queue_stats*. The device MUST fill up only upto the valid bytes.

The statistics counter replied by the device MUST wrap around to zero by the device on the overflow.

5.1.9.5.11.13 Driver Requirements: Device Statistics

The types contained in the *types_bitmap* MUST be queried from the device via command VIRTIO_NET_CTRL_STATS_QUERY.

types_bitmap in *struct virtio_net_ctrl_queue_stats* MUST be valid to the vq specified by *vq_index*.

The *command-specific-result* buffer allocated by the driver MUST have enough capacity to store all the statistics reply headers defined in *struct virtio_net_ctrl_queue_stats*. If the *command-specific-result* buffer is fully utilized by the device but some replies are missed, it is possible that some statistics may exceed the capacity of the driver's records. In such cases, the driver should allocate additional space for the *command-specific-result* buffer.

5.1.9.6 Flow filter

A network device can support one or more flow filter rules. Each flow filter rule is applied by matching a packet and then taking an action, such as directing the packet to a specific receiveq or dropping the packet. An example of a match is matching on specific source and destination IP addresses.

A flow filter rule is a device resource object that consists of a key, a processing priority, and an action to either direct a packet to a receive queue or drop the packet.

Each rule uses a classifier. The key is matched against the packet using a classifier, defining which fields in the packet are matched. A classifier resource object consists of one or more field selectors, each with a type that specifies the header fields to be matched against, and a mask. The mask can match whole fields or parts of a field in a header. Each rule resource object depends on the classifier resource object.

When a packet is received, relevant fields are extracted (in the same way) from both the packet and the key according to the classifier. The resulting field contents are then compared - if they are identical the rule action is taken, if they are not, the rule is ignored.

Multiple flow filter rules are part of a group. The rule resource object depends on the group. Each rule within a group has a rule priority, and each group also has a group priority. For a packet, a group with the highest priority is selected first. Within a group, rules are applied from highest to lowest priority, until one of the rules

matches the packet and an action is taken. If all the rules within a group are ignored, the group with the next highest priority is selected, and so on.

The device and the driver indicates flow filter resource limits using the capability 5.1.9.6.2.1 specifying the limits on the number of flow filter rule, group and classifier resource objects. The capability 5.1.9.6.2.2 specifies which selectors the device supports. The driver indicates the selectors it is using by setting the flow filter selector capability, prior to adding any resource objects.

The capability 5.1.9.6.2.3 specifies which actions the device supports.

The driver controls the flow filter rule, classifier and group resource objects using administration commands described in 2.12.1.3.

5.1.9.6.1 Packet processing order

Note that flow filter rules are applied after MAC/VLAN filtering. Flow filter rules take precedence over steering: if a flow filter rule results in an action, the steering configuration does not apply. The steering configuration only applies to packets for which no flow filter rule action was performed. For example, incoming packets can be processed in the following order:

- apply steering configuration received using control virtqueue commands VIRTIO_NET_CTRL_RX, VIRTIO_NET_CTRL_MAC and VIRTIO_NET_CTRL_VLAN.
- apply flow filter rules if any.
- if no filter rule applied, apply steering configuration received using command VIRTIO_NET_CTRL_MQ_RSS_CONFIG or as per automatic receive steering.

Some incoming packet processing examples:

- If the packet is dropped by the flow filter rule, RSS steering is ignored for the packet.
- If the packet is directed to a specific receiveq using flow filter rule, the RSS steering is ignored for the packet.
- If a packet is dropped due to the VIRTIO_NET_CTRL_MAC configuration, both flow filter rules and the RSS steering are ignored for the packet.
- If a packet does not match any flow filter rules, the RSS steering is used to select the receiveq for the packet (if enabled).
- If there are two flow filter groups configured as group_A and group_B with respective group priorities as 4, and 5; flow filter rules of group_B are applied first having highest group priority, if there is a match, the flow filter rules of group_A are ignored; if there is no match for the flow filter rules in group_B, the flow filter rules of next level group_A are applied.

5.1.9.6.2 Device and driver capabilities

5.1.9.6.2.1 VIRTIO_NET_FF_RESOURCE_CAP

The capability VIRTIO_NET_FF_RESOURCE_CAP indicates the flow filter resource limits. *cap_specific_data* is in the format *struct virtio_net_ff_cap_data*.

```
struct virtio_net_ff_cap_data {
        le32 groups_limit;
        le32 classifiers_limit;
        le32 rules_limit;
        le32 rules_per_group_limit;
        u8 last_rule_priority;
        u8 selectors_per_classifier_limit;
};
```

groups_limit, and *classifiers_limit* represent the maximum number of flow filter groups and classifiers, respectively, that the driver can create. *rules_limit* is the maximum number of flow fiilter rules that the driver

can create across all the groups. *rules_per_group_limit* is the maximum number of flow filter rules that the driver can create for each flow filter group.

last_rule_priority is the highest priority that can be assigned to a flow filter rule.

selectors_per_classifier_limit is the maximum number of selectors that a classifier can have.

5.1.9.6.2.2 VIRTIO_NET_FF_SELECTOR_CAP

The capability VIRTIO_NET_FF_SELECTOR_CAP lists the supported selectors and the supported packet header fields for each selector. *cap_specific_data* is in the format *struct virtio_net_ff_cap_mask_data*.

```
struct virtio_net_ff_selector {
        u8 type;
        u8 flags;
        u8 reserved[2];
        u8 length;
        u8 reserved1[3];
        u8 mask[];
};

struct virtio_net_ff_cap_mask_data {
        u8 count;
        u8 reserved[7];
        struct virtio_net_ff_selector selectors[];
};

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

count indicates number of valid entries in the *selectors* array. *selectors[]* is an array of supported selectors. Within each array entry: *type* specifies the type of the packet header, as defined in table 5.6. *mask* specifies which fields of the packet header can be matched in a flow filter rule.

Each *type* is also listed in table 5.6. *mask* is a byte array in network byte order. For example, when *type* is VIRTIO_NET_FF_MASK_TYPE_IPV6, the *mask* is in the format IPv6 Header Format.

If partial masking is not set, then all bits in each field have to be either all 0s to ignore this field or all 1s to match on this field. If partial masking is set, then any combination of bits can bit set to match on these bits. For example, when a selector *type* is VIRTIO_NET_FF_MASK_TYPE_ETH, if *mask[0-12]* are zero and *mask[13-14]* are 0xff (all 1s), it indicates that matching is only supported for *EtherType* of *Ethernet MAC frame*, matching is not supported for *Destination Address* and *Source Address*.

The entries in the array *selectors* are ordered by *type*, with each *type* value only appearing once.

length is the length of a dynamic array *mask* in bytes. *reserved* and *reserved1* are reserved and set to zero.

Table 5.6: Flow filter selector types

Type	Name	Description
0x0	-	Reserved
0x1	VIRTIO_NET_FF_MASK_TYPE_ETH	14 bytes of frame header starting from destination address described in IEEE 802.3-2022
0x2	VIRTIO_NET_FF_MASK_TYPE_IPV4	20 bytes of IPv4: Internet Header Format
0x3	VIRTIO_NET_FF_MASK_TYPE_IPV6	40 bytes of IPv6 Header Format
0x4	VIRTIO_NET_FF_MASK_TYPE_TCP	20 bytes of TCP Header Format
0x5	VIRTIO_NET_FF_MASK_TYPE_UDP	8 bytes of UDP header described in UDP
0x6	VIRTIO_NET_FF_MASK_TYPE_ESP	8 bytes of ESP header
0x7 - 0xFF		Reserved for future

When VIRTIO_NET_FF_MASK_F_PARTIAL_MASK (bit 0) is set, it indicates that partial masking is supported for all the fields of the selector identified by *type*.

For the selector *type* VIRTIO_NET_FF_MASK_TYPE_IPV4, if a partial mask is unsupported, then matching on an individual bit of *Flags* in the *IPv4: Internet Header Format* is unsupported. *Flags* has to match as a whole if it is supported.

For the selector *type* VIRTIO_NET_FF_MASK_TYPE_IPV4, *mask* includes fields up to the *Destination Address*; that is, *Options* and *Padding* are excluded.

For the selector *type* VIRTIO_NET_FF_MASK_TYPE_IPV6, the *Next Header* field of the *mask* corresponds to the *Next Header* in the packet when *IPv6 Extension Headers* are not present. When the packet includes one or more *IPv6 Extension Headers*, the *Next Header* field of the *mask* corresponds to the *Next Header* of the last *IPv6 Extension Header* in the packet.

For the selector *type* VIRTIO_NET_FF_MASK_TYPE_TCP, *Control bits* are treated as individual fields for matching; that is, matching individual *Control bits* does not depend on the partial mask support.

5.1.9.6.2.3 VIRTIO_NET_FF_ACTION_CAP

The capability VIRTIO_NET_FF_ACTION_CAP lists the supported actions in a rule. *cap_specific_data* is in the format *struct virtio_net_ff_cap_actions*.

```
struct virtio_net_ff_actions {
        u8 count;
        u8 reserved[7];
        u8 actions[];
};
```

actions is an array listing all possible actions. The entries in the array are ordered from the smallest to the largest, with each supported value appearing exactly once. Each entry can have the following values:

Table 5.8: Flow filter rule actions

Action	Name	Description
0x0	-	reserved
0x1	VIRTIO_NET_FF_ACTION_DROP	Matching packet will be dropped by the device
0x2	VIRTIO_NET_FF_ACTION_DIRECT_RX_VQ	Matching packet will be directed to a receive queue
0x3	VIRTIO_NET_FF_ACTION_IPSEC	Matching packet will undergo IPsec processing
0x4	VIRTIO_NET_FF_ACTION_IPSEC_RECIRCULATE	Matching packet will first undergo IPsec processing, followed by the flow filter rules again
0x5 - 0xFF		Reserved for future

5.1.9.6.3 Resource objects

5.1.9.6.3.1 VIRTIO_NET_RESOURCE_OBJ_FF_GROUP

A flow filter group contains between 0 and *rules_limit* rules, as specified by the capability VIRTIO_NET_FF_RESOURCE_CAP. For the flow filter group object both *resource_obj_specific_data* and *resource_obj_specific_result* are in the format *struct virtio_net_resource_obj_ff_group*.

```
struct virtio_net_resource_obj_ff_group {
        le16 group_priority;
};
```

group_priority specifies the priority for the group. Each group has a distinct priority. For each incoming packet, the device tries to apply rules from groups from higher *group_priority* value to lower, until either a rule matches the packet or all groups have been tried.

5.1.9.6.3.2 VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER

A classifier is used to match a flow filter key against a packet. The classifier defines the desired packet fields to match, and is represented by the VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER device resource object.

For the flow filter classifier object both *resource_obj_specific_data* and *resource_obj_specific_result* are in the format *struct virtio_net_resource_obj_ff_classifier*.

```
struct virtio_net_resource_obj_ff_classifier {
        u8 count;
        u8 reserved[7];
        struct virtio_net_ff_selector selectors[];
};
```

A classifier is an array of *selectors*. The number of selectors in the array is indicated by *count*. The selector has a type that specifies the header fields to be matched against, and a mask. See 5.1.9.6.2.2 for details about selectors.

The first selector is always VIRTIO_NET_FF_MASK_TYPE_ETH. When there are multiple selectors, a second selector can be either VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6. If the third selector exists, it can be set to VIRTIO_NET_FF_MASK_TYPE_UDP, VIRTIO_NET_FF_MASK_TYPE_TCP and VIRTIO_NET_FF_MASK_TYPE_ESP. For example, to match a Ethernet IPv6 UDP packet, *selectors[0].type* is set to VIRTIO_NET_FF_MASK_TYPE_ETH, *selectors[1].type* is set to VIRTIO_NET_FF_MASK_TYPE_IPV6 and *selectors[2].type* is set to VIRTIO_NET_FF_MASK_TYPE_UDP; accordingly,

selectors[0].mask[0-13] is for Ethernet header fields, *selectors[1].mask[0-39]* is set for IPV6 header and *selectors[2].mask[0-7]* is set for UDP header.

When there are multiple selectors, the type of the (N+1)th selector affects the mask of the (N)th selector. If *count* is 2 or more, all the mask bits within *selectors[0]* corresponding to *EtherType* of an Ethernet header are set.

If *count* is more than 2:

- if *selector[1].type* is, VIRTIO_NET_FF_MASK_TYPE_IPV4, then, all the mask bits within *selector[1]* for *Protocol* is set.
- if *selector[1].type* is, VIRTIO_NET_FF_MASK_TYPE_IPV6, then, all the mask bits within *selector[1]* for *Next Header* is set.

If for a given packet header field, a subset of bits of a field is to be matched, and if the partial mask is supported, the flow filter mask object can specify a mask which has fewer bits set than the packet header field size. For example, a partial mask for the Ethernet header source mac address can be of 1-bit for multicast detection instead of 48-bits.

5.1.9.6.3.3 VIRTIO_NET_RESOURCE_OBJ_FF_RULE

Each flow filter rule resource object comprises a key, a priority, and an action. For the flow filter rule object, *resource_obj_specific_data* and *resource_obj_specific_result* are in the format *struct virtio_net_resource_obj_ff_rule*.

```
struct virtio_net_resource_obj_ff_rule {
        le32 group_id;
        le32 classifier_id;
        u8 rule_priority;
        u8 key_length; /* length of key in bytes */
        u8 action;
        u8 reserved;
        le16 vq_index;
        u8 reserved1[2];
        u8 keys[][];
};
```

group_id is the resource object ID of the flow filter group to which this rule belongs. *classifier_id* is the resource object ID of the classifier used to match a packet against the *key*.

rule_priority denotes the priority of the rule within the group specified by the *group_id*. Rules within the group are applied from the highest to the lowest priority until a rule matches the packet and an action is taken. Rules with the same priority can be applied in any order.

reserved and *reserved1* are reserved and set to 0.

keys[][] is an array of keys to match against packets, using the classifier specified by *classifier_id*. Each entry (key) comprises a byte array, and they are located one immediately after another. The size (number of entries) of the array is exactly the same as that of *selectors* in the classifier, or in other words, *count* in the classifier.

key_length specifies the total length of *keys* in bytes. In other words, it equals the sum total of *length* of all selectors in *selectors* in the classifier specified by *classifier_id*.

For example, if a classifier object's *selectors[0].type* is VIRTIO_NET_FF_MASK_TYPE_ETH and *selectors[1].type* is VIRTIO_NET_FF_MASK_TYPE_IPV6, then selectors[0].length is 14 and selectors[1].length is 40. Accordingly, the *key_length* is set to 54. This setting indicates that the *key* array's length is 54 bytes comprising a first byte array of 14 bytes for the Ethernet MAC header in bytes 0-13, immediately followed by 40 bytes for the IPv6 header in bytes 14-53.

When there are multiple selectors in the classifier object, the key bytes for (N)th selector are set so that (N+1)th selector can be matched.

If *count* is 2 or more, key bytes of *EtherType* are set according to IEEE 802 Ethertypes for VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6 respectively.

If *count* is more than 2, when *selector[1].type* is VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6, key bytes of *Protocol* or *Next Header* is set as per *Protocol Numbers* defined IANA Protocol Numbers respectively.

action is the action to take when a packet matches the *key* using the *classifier_id*. Supported actions are described in 5.8.

vq_index specifies a receive virtqueue. When the *action* is set to VIRTIO_NET_FF_ACTION_DIRECT_RX_VQ, and the packet matches the *key*, the matching packet is directed to this virtqueue.

Note that at most one action is ever taken for a given packet. If a rule is applied and an action is taken, the action of other rules is not taken.

5.1.9.6.4 Device Requirements: Flow filter

When the device supports flow filter operations,

- the device MUST set VIRTIO_NET_FF_RESOURCE_CAP, VIRTIO_NET_FF_SELECTOR_CAP and VIRTIO_NET_FF_ACTION_CAP capability in the *supported_caps* in the command VIRTIO_ADMIN_CMD_CAP_SUPPORT_QUERY.
- the device MUST support the administration commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for the resource types VIRTIO_NET_RESOURCE_OBJ_FF_GROUP, VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER and VIRTIO_NET_RESOURCE_OBJ_FF_RULE.

When any of the VIRTIO_NET_FF_RESOURCE_CAP, VIRTIO_NET_FF_SELECTOR_CAP, or VIRTIO_NET_FF_ACTION_CAP capability is disabled, the device SHOULD set *status* to VIRTIO_ADMIN_STATUS_Q_INVALID_OPCODE for the commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, and VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY. These commands apply to the resource *type* of VIRTIO_NET_RESOURCE_OBJ_FF_GROUP, VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER, and VIRTIO_NET_RESOURCE_OBJ_FF_RULE.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_EINVAL for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE when the resource *type* is VIRTIO_NET_RESOURCE_OBJ_FF_GROUP, if a flow filter group already exists with the supplied *group_priority*.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_ENOSPC for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE when the resource *type* is VIRTIO_NET_RESOURCE_OBJ_FF_GROUP, if the number of flow filter group objects in the device exceeds the lower of the configured driver capabilities *groups_limit* and *rules_per_group_limit*.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_ENOSPC for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE when the resource *type* is VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER, if the number of flow filter selector objects in the device exceeds the configured driver capability *classifiers_limit*.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_EBUSY for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for a flow filter group when the flow filter group has one or more flow filter rules depending on it.

The device SHOULD set *status* to VIRTIO_ADMIN_STATUS_EBUSY for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for a flow filter classifier when the flow filter classifier has one or more flow filter rules depending on it.

The device SHOULD fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE for the flow filter rule resource object if,

- *vq_index* is not a valid receive virtqueue index for the VIRTIO_NET_FF_ACTION_DIRECT_RX_VQ action,
- *priority* is greater than or equal to *last_rule_priority*,

- *id* is greater than or equal to *rules_limit* or greater than or equal to *rules_per_group_limit*, whichever is lower,
- the length of *keys* and the length of all the mask bytes of *selectors[].mask* as referred by *classifier_id* differs,
- the supplied *action* is not supported in the capability VIRTIO_NET_FF_ACTION_CAP.

When the flow filter directs a packet to the virtqueue identified by *vq_index* and if the receive virtqueue is reset, the device MUST drop such packets.

Upon applying a flow filter rule to a packet, the device MUST STOP any further application of rules and cease applying any other steering configurations.

For multiple flow filter groups, the device MUST apply the rules from the group with the highest priority. If any rule from this group is applied, the device MUST ignore the remaining groups. If none of the rules from the highest priority group match, the device MUST apply the rules from the group with the next highest priority, until either a rule matches or all groups have been attempted.

The device MUST apply the rules within the group from the highest to the lowest priority until a rule matches the packet, and the device MUST take the action. If an action is taken, the device MUST not take any other action for this packet.

The device MAY apply the rules with the same *rule_priority* in any order within the group.

The device MUST process incoming packets in the following order:

- apply the steering configuration received using control virtqueue commands VIRTIO_NET_CTRL_RX, VIRTIO_NET_CTRL_MAC, and VIRTIO_NET_CTRL_VLAN.
- apply flow filter rules if any.
- if no filter rule is applied, apply the steering configuration received using the command VIRTIO_NET_CTRL_MQ_RSS_CONFIG or according to automatic receive steering.

When processing an incoming packet, if the packet is dropped at any stage, the device MUST skip further processing.

When the device drops the packet due to the configuration done using the control virtqueue commands VIRTIO_NET_CTRL_RX or VIRTIO_NET_CTRL_MAC or VIRTIO_NET_CTRL_VLAN, the device MUST skip flow filter rules for this packet.

When the device performs flow filter match operations and if the operation result did not have any match in all the groups, the receive packet processing continues to next level, i.e. to apply configuration done using VIRTIO_NET_CTRL_MQ_RSS_CONFIG command.

The device MUST support the creation of flow filter classifier objects using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with *flags* set to VIRTIO_NET_FF_MASK_F_PARTIAL_MASK; this support is required even if all the bits of the masks are set for a field in *selectors*, provided that partial masking is supported for the selectors.

5.1.9.6.5 Driver Requirements: Flow filter

The driver MUST enable VIRTIO_NET_FF_RESOURCE_CAP, VIRTIO_NET_FF_SELECTOR_CAP, and VIRTIO_NET_FF_ACTION_CAP capabilities to use flow filter.

The driver SHOULD NOT remove a flow filter group using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY when one or more flow filter rules depend on that group. The driver SHOULD only destroy the group after all the associated rules have been destroyed.

The driver SHOULD NOT remove a flow filter classifier using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY when one or more flow filter rules depend on the classifier. The driver SHOULD only destroy the classifier after all the associated rules have been destroyed.

The driver SHOULD NOT add multiple flow filter rules with the same *rule_priority* within a flow filter group, as these rules MAY match the same packet. The driver SHOULD assign different *rule_priority* values to different flow filter rules if multiple rules may match a single packet.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, when creating a resource of *type* VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER, the driver MUST set:

- *selectors[0].type* to VIRTIO_NET_FF_MASK_TYPE_ETH.
- *selectors[1].type* to VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6 when *count* is more than 1,
- *selectors[2].type* VIRTIO_NET_FF_MASK_TYPE_UDP or VIRTIO_NET_FF_MASK_TYPE_TCP when *count* is more than 2.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, when creating a resource of *type* VIRTIO_NET_RESOURCE_OBJ_FF_CLASSIFIER, the driver MUST set:

- *selectors[0].mask* bytes to all 1s for the *EtherType* when *count* is 2 or more.
- *selectors[1].mask* bytes to all 1s for *Protocol* or *Next Header* when *selector[1].type* is VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6, and when *count* is more than 2.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, the resource *type* VIRTIO_NET_RESOURCE_OBJ_FF_RULE, if the corresponding classifier object's *count* is 2 or more, the driver MUST SET the *keys* bytes of *EtherType* in accordance with IEEE 802 Ethertypes for either VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, when creating a resource of *type* VIRTIO_NET_RESOURCE_OBJ_FF_RULE, if the corresponding classifier object's *count* is more than 2, and the *selector[1].type* is either VIRTIO_NET_FF_MASK_TYPE_IPV4 or VIRTIO_NET_FF_MASK_TYPE_IPV6, the driver MUST set the *keys* bytes for the *Protocol* or *Next Header* according to IANA Protocol Numbers respectively.

The driver SHOULD set all the bits for a field in the mask of a selector in both the capability and the classifier object, unless the VIRTIO_NET_FF_MASK_F_PARTIAL_MASK is enabled.

5.1.9.7 IPsec Operation

A network device can support the processing of IPsec operations when VIRTIO_NET_F_IPSEC feature is negotiated. In addition to standard packet processing, the IPsec protocol processing is also handled by the network device. This occurs both pre-transmit and post-receive, providing inline IPsec capabilities.

IPsec Inbound processing: In receive path the device performs decryption, authentication, integrity checking and remove additional headers, including tunnel header if in tunnel mode, as well as the ESP/AH header from the packet (See [IPSEC]). The resulting packet contains only the plain data.

IPsec Outbound processing: In transmit path the device performs encryption, attach ICV, update/add IP header and add ESP/AH header/trailer to the packet and transmit.

5.1.9.7.1 Packet processing order

If an IPsec action rule, either VIRTIO_NET_FF_ACTION_IPSEC or VIRTIO_NET_FF_ACTION_IPSEC_RECIRCULATE, is matched during flow filter processing, IPsec processing is applied on the packet. In the case of VIRTIO_NET_FF_ACTION_IPSEC_RECIRCULATE, the packet goes through IPsec processing and is then recirculated only once to avoid the infinite loops in the device. When a packet is recirculated, it undergoes flow filters processing again with the updated packet content.

See 5.1.9.6 for details about flow filter.

Note that there is a small race condition where a SA object might be destroyed while a recieve packet is still inflight. The driver SHOULD handle this situation appropriately.

5.1.9.7.2 Device and driver capabilities

The device and the driver indicate IPsec SA resource limits using the capability VIRTIO_NET_IPSEC_RESOURCE_CAP The VIRTIO_NET_IPSEC_SA_CAP capability specifies which IPsec protocol capabilities the device supports. The driver indicates the IPsec parameters by setting the IPsec SA capability prior to adding any resource objects.

5.1.9.7.3 Resource objects

The driver controls the IPsec SA resource object using administration commands described in 2.12.1.3.

The IPsec SA resource object contains necessary parameters for packet encryption and decryption. These include the SPI, tunnel headers, IPsec mode, IPsec options, and data specific to cipher and authentication.

See VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA.

See VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA.

5.1.9.7.4 Device Requirements: IPsec Operation

When the device supports IPsec operations,

- the device MUST set VIRTIO_NET_IPSEC_RESOURCE_CAP, VIRTIO_NET_IPSEC_SA_CAP capability in the *supported_caps* in the command VIRTIO_ADMIN_CMD_CAP_SUPPORT_QUERY.
- the device MUST support the administration commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for the resource types VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA and VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA.

When any of the VIRTIO_NET_IPSEC_RESOURCE_CAP or VIRTIO_NET_IPSEC_SA_CAP capability is disabled, the device MUST set *status* to VIRTIO_ADMIN_STATUS_Q_INVALID_OPCODE for the commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, and VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for the resource types VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA and VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA.

The device MUST set *status* to VIRTIO_ADMIN_STATUS_EEXIT for the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE when the resource *type* is VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA or VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA, if the object already exists with the supplied *id*.

The device MUST fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE for the VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA object if,

- *id* is greater than or equal to *outb_sa_limit*.
- the supplied SA parameters, such as mode, options, cipher and authentication algorithms are not supported in the capabitlity VIRTIO_NET_IPSEC_SA_CAP.

The device MUST fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE for the VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA object if,

- *id* is greater than or equal to *inb_sa_limit*.
- the supplied SA parameters, such as mode, options, cipher and authentication algorithms are not supported in the capabitlity VIRTIO_NET_IPSEC_SA_CAP.

The device SHOULD maintain a table for subsequent lookups to inbound/outbound data with the corresponding SA based on the supplied *id*.

The device MUST allow recreating the resource objects using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE which was previously destroyed using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY respectively without undergoing a device reset.

The device MAY fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with the *status* set to VIRTIO_ADMIN_STATUS_EINVAL for the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTB_SA or VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INB_SA commands if the resource object with the same *spi* already exists.

On device reset, the device MUST destroy all the resource objects which have been created.

5.1.9.7.5 Driver Requirements: IPsec Operation

The driver MUST query the capabilities using VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY to discover the capability types the device offers.

The driver MUST get VIRTIO_NET_IPSEC_RESOURCE_CAP and VIRTIO_NET_IPSEC_SA_CAP if listed in VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY command result, using VIRTION_ADMIN_CMD_DEVICE_CAP_GET to discover the capabilities the device is able to offer. The driver MUST set VIRTIO_NET_IPSEC_RESOURCE_CAP and VIRTIO_NET_IPSEC_SA_CAP using VIRTIO_ADMIN_CMD_DEVICE_CAP_SET to indicate the device which capability the driver uses.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, when creating a resource VIRTIO_NET_RESOURCE_OBJ_IPSEC_OUTB_SA, the driver MUST set all the parameters in *struct virtio_crypto_resource_obj_ipsec_sa* with relevant values. And when create a resource object VIRTIO_NET_RESOURCE_OBJ_IPSEC_INB_SA, the driver MUST set all the parameters except *struct virtio_crypto_ipsec_tunnel_param*.

The driver SHOULD NOT add multiple inbound SA objects with the same *spi*.

In the outbound data path, the driver MUST ensure that packets do not exceed the PMTU prior to transmission to the device. (Refer to [IPSEC] for a detailed description of PMTU)

5.1.9.8 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 5.1.9.5) , 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 2.7.4.

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 one of its queues, and serviced (probably out of order) by the device except where noted.

5.2.1 Device ID

2

5.2.2 Virtqueues

0 requestq1

...

N-1 requestqN

N=1 if VIRTIO_BLK_F_MQ is not negotiated, otherwise N is set by *num_queues*.

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_MQ (12) Device supports multiqueue.

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*.

VIRTIO_BLK_F_LIFETIME (15) Device supports providing storage lifetime information.

VIRTIO_BLK_F_SECURE_ERASE (16) Device supports secure erase command, maximum erase sectors count in *max_secure_erase_sectors* and maximum erase segment number in *max_secure_erase_seg*.

VIRTIO_BLK_F_ZONED(17) Device is a Zoned Block Device, that is, a device that follows the zoned storage device behavior that is also supported by industry standards such as the T10 Zoned Block Command standard (ZBC r05) or the NVMe(TM) NVM Express Zoned Namespace Command Set Specification 1.1b (ZNS). For brevity, these standard documents are referred as ”ZBD standards” from this point on in the text.

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 block device has the following device configuration layout.

```
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;
        le16 num_queues;
        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];
        le32 max_secure_erase_sectors;
        le32 max_secure_erase_seg;
        le32 secure_erase_sector_alignment;
        struct virtio_blk_zoned_characteristics {
                le32 zone_sectors;
                le32 max_open_zones;
                le32 max_active_zones;
                le32 max_append_sectors;
                le32 write_granularity;
                u8 model;
                u8 unused2[3];
        } zoned;
};
```

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 field *num_queues* only exists if VIRTIO_BLK_F_MQ is set. This field specifies the number of queues.

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. The parameters in the configuration space of the device *max_secure_erase_sectors secure_erase_sector_alignment* are expressed in 512-byte units if the VIRTIO_BLK_F_SECURE_ERASE feature bit is negotiated.

If the VIRTIO_BLK_F_ZONED feature is negotiated, then in *virtio_blk_zoned_characteristics*,

- *zone_sectors* value is expressed in 512-byte sectors.
- *max_append_sectors* value is expressed in 512-byte sectors.
- *write_granularity* value is expressed in bytes.

The *model* field in *zoned* may have the following values:

```
#define VIRTIO_BLK_Z_NONE      0
#define VIRTIO_BLK_Z_HM        1
#define VIRTIO_BLK_Z_HA        2
```

Depending on their design, zoned block devices may follow several possible models of operation. The three models that are standardized for ZBDs are drive-managed, host-managed and host-aware.

While being zoned internally, drive-managed ZBDs behave exactly like regular, non-zoned block devices. For the purposes of virtio standardization, drive-managed ZBDs can always be treated as non-zoned devices. These devices have the VIRTIO_BLK_Z_NONE model value set in the *model* field in *zoned*.

Devices that offer the VIRTIO_BLK_F_ZONED feature while reporting the VIRTIO_BLK_Z_NONE zoned model are drive-managed zoned block devices. In this case, the driver treats the device as a regular non-zoned block device.

Host-managed zoned block devices have their LBA range divided into Sequential Write Required (SWR) zones that require some additional handling by the host for correct operation. All write requests to SWR zones are required be sequential and zones containing some written data need to be reset before that data can be rewritten. Host-managed devices support a set of ZBD-specific I/O requests that can be used by the host to manage device zones. Host-managed devices report VIRTIO_BLK_Z_HM in the *model* field in *zoned*.

Host-aware zoned block devices have their LBA range divided to Sequential Write Preferred (SWP) zones that support random write access, similar to regular non-zoned devices. However, the device I/O performance might not be optimal if SWP zones are used in a random I/O pattern. SWP zones also support the same set of ZBD-specific I/O requests as host-managed devices that allow host-aware devices to be managed by any host that supports zoned block devices to achieve its optimum performance. Host-aware devices report VIRTIO_BLK_Z_HA in the *model* field in *zoned*.

Both SWR zones and SWP zones are sometimes referred as sequential zones.

During device operation, sequential zones can be in one of the following states: empty, implicitly-open, explicitly-open, closed and full. The state machine that governs the transitions between these states is described later in this document.

SWR and SWP zones consume volatile device resources while being in certain states and the device may set limits on the number of zones that can be in these states simultaneously.

Zoned block devices use two internal counters to account for the device resources in use, the number of currently open zones and the number of currently active zones.

Any zone state transition from a state that doesn't consume a zone resource to a state that consumes the same resource increments the internal device counter for that resource. Any zone transition out of a state that consumes a zone resource to a state that doesn't consume the same resource decrements the counter. Any request that causes the device to exceed the reported zone resource limits is terminated by the device with a "zone resources exceeded" error as defined for specific commands later.

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.

[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.

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.
8. If the VIRTIO_BLK_F_MQ feature is negotiated, *num_queues* field can be read to determine the number of queues.
9. If the VIRTIO_BLK_F_SECURE_ERASE feature is negotiated, *max_secure_erase_sectors* and *max_secure_erase_seg* can be read to determine the maximum secure erase sectors and maximum number of secure erase segments for the block driver to use. *secure_erase_sector_alignment* can be used by OS when splitting a request based on alignment.
10. If the VIRTIO_BLK_F_ZONED feature is negotiated, the fields in *zoned* can be read by the driver to determine the zone characteristics of the device. All *zoned* fields are read-only.

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.

Drivers MUST NOT negotiate the VIRTIO_BLK_F_ZONED feature if they are incapable of supporting devices with the VIRTIO_BLK_Z_HM, VIRTIO_BLK_Z_HA or VIRTIO_BLK_Z_NONE zoned model.

If the VIRTIO_BLK_F_ZONED feature is offered by the device with the VIRTIO_BLK_Z_HM zone model, then the VIRTIO_BLK_F_DISCARD feature MUST NOT be offered by the driver.

If the VIRTIO_BLK_F_ZONED feature and VIRTIO_BLK_F_DISCARD feature are both offered by the device with the VIRTIO_BLK_Z_HA or VIRTIO_BLK_Z_NONE zone model, then the driver MAY negotiate these two bits independently.

If the VIRTIO_BLK_F_ZONED feature is negotiated, then

- if the driver that can not support host-managed zoned devices reads VIRTIO_BLK_Z_HM from the *model* field of *zoned*, the driver MUST NOT set FEATURES_OK flag and instead set the FAILED bit.
- if the driver that can not support zoned devices reads VIRTIO_BLK_Z_HA from the *model* field of *zoned*, the driver MAY handle the device as a non-zoned device. In this case, the driver SHOULD ignore all other fields in *zoned*.

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.

If the device that is being initialized is a not a zoned device, the device SHOULD NOT offer the VIRTIO_BLK_F_ZONED feature.

The VIRTIO_BLK_F_ZONED feature cannot be properly negotiated without FEATURES_OK bit. Legacy devices MUST NOT offer VIRTIO_BLK_F_ZONED feature bit.

If the VIRTIO_BLK_F_ZONED feature is not accepted by the driver,

- the device with the VIRTIO_BLK_Z_HA or VIRTIO_BLK_Z_NONE zone model SHOULD proceed with the initialization while setting all zoned characteristics fields to zero.
- the device with the VIRTIO_BLK_Z_HM zone model MUST fail to set the FEATURES_OK device status bit when the driver writes the Device Status field.

If the VIRTIO_BLK_F_ZONED feature is negotiated, then the *model* field in *zoned* struct in the configuration space MUST be set by the device

- to the value of VIRTIO_BLK_Z_NONE if it operates as a drive-managed zoned block device or a non-zoned block device.
- to the value of VIRTIO_BLK_Z_HM if it operates as a host-managed zoned block device.
- to the value of VIRTIO_BLK_Z_HA if it operates as a host-aware zoned block device.

If the VIRTIO_BLK_F_ZONED feature is negotiated and the device *model* field in *zoned* struct is VIRTIO_-BLK_Z_HM or VIRTIO_BLK_Z_HA,

- the *zone_sectors* field of *zoned* MUST be set by the device to the size of a single zone on the device. All zones of the device have the same size indicated by *zone_sectors* except for the last zone that MAY be smaller than all other zones. The driver can calculate the number of zones on the device as

```
nr_zones = (capacity + zone_sectors - 1) / zone_sectors;
```

  and the size of the last zone as

```
zs_last = capacity - (nr_zones - 1) * zone_sectors;
```

- The *max_open_zones* field of the *zoned* structure MUST be set by the device to the maximum number of zones that can be open on the device (zones in the implicit open or explicit open state). A value of zero indicates that the device does not have any limit on the number of open zones.
- The *max_active_zones* field of the *zoned* structure MUST be set by the device to the maximum number zones that can be active on the device (zones in the implicit open, explicit open or closed state). A value of zero indicates that the device does not have any limit on the number of active zones.
- the *max_append_sectors* field of *zoned* MUST be set by the device to the maximum data size of a VIRTIO_BLK_T_ZONE_APPEND request that can be successfully issued to the device. The value of this field MUST NOT exceed the *seg_max* * *size_max* value. A device MAY set the *max_append_-sectors* to zero if it doesn't support VIRTIO_BLK_T_ZONE_APPEND requests.
- the *write_granularity* field of *zoned* MUST be set by the device to the offset and size alignment constraint for VIRTIO_BLK_T_OUT and VIRTIO_BLK_T_ZONE_APPEND requests issued to a sequential zone of the device.
- the device MUST initialize padding bytes *unused2* 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 enqueues requests to the virtqueues, and they are used by the device (not necessarily in order). Each request except VIRTIO_BLK_T_ZONE_APPEND is of form:

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

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), a flush (VIRTIO_BLK_T_FLUSH), a get device ID string command (VIRTIO_BLK_T_GET_ID), a secure erase (VIRTIO_BLK_T_SECURE_ERASE), or a get device lifetime command (VIRTIO_BLK_T_GET_LIFETIME).

```
#define VIRTIO_BLK_T_IN           0
#define VIRTIO_BLK_T_OUT          1
#define VIRTIO_BLK_T_FLUSH        4
#define VIRTIO_BLK_T_GET_ID       8
#define VIRTIO_BLK_T_GET_LIFETIME 10
#define VIRTIO_BLK_T_DISCARD      11
#define VIRTIO_BLK_T_WRITE_ZEROES 13
#define VIRTIO_BLK_T_SECURE_ERASE   14
```

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, write and some zone operations.

VIRTIO_BLK_T_IN requests populate *data* with the contents of sectors read from the block device (in multiples of 512 bytes). VIRTIO_BLK_T_OUT requests write the contents of *data* to the block device (in multiples of 512 bytes).

The *data* used for discard, secure erase or write zeroes commands consists of one or more segments. The maximum number of segments is *max_discard_seg* for discard commands, *max_secure_erase_seg* for secure erase commands and *max_write_zeroes_seg* for write zeroes commands. Each segment is of form:

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

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 in write zeroes commands and allows the device to discard the specified range, provided that following reads return zeroes.

VIRTIO_BLK_T_GET_ID requests fetch the device ID string from the device into *data*. The device ID string is a NUL-padded ASCII string up to 20 bytes long. If the string is 20 bytes long then there is no NUL terminator.

The *data* used for VIRTIO_BLK_T_GET_LIFETIME requests is populated by the device, and is of the form

```
struct virtio_blk_lifetime {
  le16 pre_eol_info;
  le16 device_lifetime_est_typ_a;
  le16 device_lifetime_est_typ_b;
};
```

The *pre_eol_info* specifies the percentage of reserved blocks that are consumed and will have one of these values:

```
/* Value not available */
#define VIRTIO_BLK_PRE_EOL_INFO_UNDEFINED    0
/* < 80% of reserved blocks are consumed */
#define VIRTIO_BLK_PRE_EOL_INFO_NORMAL       1
/* 80% of reserved blocks are consumed */
#define VIRTIO_BLK_PRE_EOL_INFO_WARNING      2
/* 90% of reserved blocks are consumed */
#define VIRTIO_BLK_PRE_EOL_INFO_URGENT       3
/* All others values are reserved */
```

The *device_lifetime_est_typ_a* refers to wear of SLC cells and is provided in increments of 10used, and so on, thru to 11 meaning estimated lifetime exceeded. All values above 11 are reserved.

The *device_lifetime_est_typ_b* refers to wear of MLC cells and is provided with the same semantics as *device_lifetime_est_typ_a*.

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
```

The status of individual segments is indeterminate when a discard or write zero command produces VIRTIO_BLK_S_IOERR. A segment may have completed successfully, failed, or not been processed by the device.

The following requirements only apply if the VIRTIO_BLK_F_ZONED feature is negotiated.

In addition to the request types defined for non-zoned devices, the type of the request can be a zone report (VIRTIO_BLK_T_ZONE_REPORT), an explicit zone open (VIRTIO_BLK_T_ZONE_OPEN), a zone close (VIRTIO_BLK_T_ZONE_CLOSE), a zone finish (VIRTIO_BLK_T_ZONE_FINISH), a zone_append (VIRTIO_BLK_T_ZONE_APPEND), a zone reset (VIRTIO_BLK_T_ZONE_RESET) or a zone reset all (VIRTIO_BLK_T_ZONE_RESET_ALL).

```
#define VIRTIO_BLK_T_ZONE_APPEND    15
#define VIRTIO_BLK_T_ZONE_REPORT    16
#define VIRTIO_BLK_T_ZONE_OPEN      18
#define VIRTIO_BLK_T_ZONE_CLOSE     20
#define VIRTIO_BLK_T_ZONE_FINISH    22
#define VIRTIO_BLK_T_ZONE_RESET     24
#define VIRTIO_BLK_T_ZONE_RESET_ALL 26
```

Requests of type VIRTIO_BLK_T_OUT, VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_T_ZONE_FINISH, VIRTIO_BLK_T_ZONE_APPEND, VIRTIO_BLK_T_ZONE_RESET or VIRTIO_BLK_T_ZONE_RESET_ALL may be completed by the device with VIRTIO_BLK_S_OK, VIRTIO_-BLK_S_IOERR or VIRTIO_BLK_S_UNSUPP *status*, or, additionally, with VIRTIO_BLK_S_ZONE_INVALID_-CMD, VIRTIO_BLK_S_ZONE_UNALIGNED_WP, VIRTIO_BLK_S_ZONE_OPEN_RESOURCE or VIRTIO_-BLK_S_ZONE_ACTIVE_RESOURCE ZBD-specific status codes.

Besides the request status, VIRTIO_BLK_T_ZONE_APPEND requests return the starting sector of the appended data back to the driver. For this reason, the VIRTIO_BLK_T_ZONE_APPEND request has the layout that is extended to have the *append_sector* field to carry this value:

```
struct virtio_blk_req_za {
        le32 type;
        le32 reserved;
        le64 sector;
        u8 data[];
        le64 append_sector;
        u8 status;
};
```

```
#define VIRTIO_BLK_S_ZONE_INVALID_CMD     3
#define VIRTIO_BLK_S_ZONE_UNALIGNED_WP    4
```

```
#define VIRTIO_BLK_S_ZONE_OPEN_RESOURCE   5
#define VIRTIO_BLK_S_ZONE_ACTIVE_RESOURCE 6
```

Requests of the type VIRTIO_BLK_T_ZONE_REPORT are reads and requests of the type VIRTIO_BLK_-T_ZONE_APPEND are writes. VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_-BLK_T_ZONE_FINISH, VIRTIO_BLK_T_ZONE_RESET and VIRTIO_BLK_T_ZONE_RESET_ALL are non-data requests.

Zone sector address is a 64-bit address of the first 512-byte sector of the zone.

VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_T_ZONE_FINISH and VIRTIO_BLK_T_ZONE_RESET requests make the zone operation to act on a particular zone specified by the zone sector address in the *sector* of the request.

VIRTIO_BLK_T_ZONE_RESET_ALL request acts upon all applicable zones of the device. The *sector* value is not used for this request.

In ZBD standards, the VIRTIO_BLK_T_ZONE_REPORT request belongs to ”Zone Management Receive” command category and VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_-T_ZONE_FINISH and VIRTIO_BLK_T_ZONE_RESET/VIRTIO_BLK_T_ZONE_RESET_ALL requests are categorized as ”Zone Management Send” commands. VIRTIO_BLK_T_ZONE_APPEND is categorized separately from zone management commands and is the only request that uses the *append_secctor* field *virtio_blk_req_za* to return to the driver the sector at which the data has been appended to the zone.

VIRTIO_BLK_T_ZONE_REPORT is a read request that returns the information about the current state of zones on the device starting from the zone containing the *sector* of the request. The report consists of a header followed by zero or more zone descriptors.

A zone report reply has the following structure:

```
struct virtio_blk_zone_report {
        le64   nr_zones;
        u8     reserved[56];
        struct virtio_blk_zone_descriptor zones[];
};
```

The device sets the *nr_zones* field in the report header to the number of fully transferred zone descriptors in the data buffer.

A zone descriptor has the following structure:

```
struct virtio_blk_zone_descriptor {
        le64   z_cap;
        le64   z_start;
        le64   z_wp;
        u8     z_type;
        u8     z_state;
        u8     reserved[38];
};
```

The zone descriptor field *z_type virtio_blk_zone_descriptor* indicates the type of the zone.

The following zone types are available:

```
#define VIRTIO_BLK_ZT_CONV     1
#define VIRTIO_BLK_ZT_SWR      2
#define VIRTIO_BLK_ZT_SWP      3
```

Read and write operations into zones with the VIRTIO_BLK_ZT_CONV (Conventional) type have the same behavior as read and write operations on a regular block device. Any block in a conventional zone can be read or written at any time and in any order.

Zones with VIRTIO_BLK_ZT_SWR can be read randomly, but must be written sequentially at a certain point in the zone called the Write Pointer (WP). With every write, the Write Pointer is incremented by the number of sectors written.

Zones with VIRTIO_BLK_ZT_SWP can be read randomly and should be written sequentially, similarly to SWR zones. However, SWP zones can accept random write operations, that is, VIRTIO_BLK_T_OUT requests with a start sector different from the zone write pointer position.

The field *z_state* of *virtio_blk_zone_descriptor* indicates the state of the device zone.

The following zone states are available:

```
#define VIRTIO_BLK_ZS_NOT_WP   0
#define VIRTIO_BLK_ZS_EMPTY    1
#define VIRTIO_BLK_ZS_IOPEN    2
#define VIRTIO_BLK_ZS_EOPEN    3
#define VIRTIO_BLK_ZS_CLOSED   4
#define VIRTIO_BLK_ZS_RDONLY   13
#define VIRTIO_BLK_ZS_FULL     14
#define VIRTIO_BLK_ZS_OFFLINE  15
```

Zones of the type VIRTIO_BLK_ZT_CONV are always reported by the device to be in the VIRTIO_BLK_ZS_-NOT_WP state. Zones of the types VIRTIO_BLK_ZT_SWR and VIRTIO_BLK_ZT_SWP can not transition to the VIRTIO_BLK_ZS_NOT_WP state.

Zones in VIRTIO_BLK_ZS_EMPTY (Empty), VIRTIO_BLK_ZS_IOPEN (Implicitly Open), VIRTIO_BLK_-ZS_EOPEN (Explicitly Open) and VIRTIO_BLK_ZS_CLOSED (Closed) state are writable, but zones in VIRTIO_BLK_ZS_RDONLY (Read-Only), VIRTIO_BLK_ZS_FULL (Full) and VIRTIO_BLK_ZS_OFFLINE (Offline) state are not. The write pointer value (*z_wp*) is not valid for Read-Only, Full and Offline zones.

The zone descriptor field *z_cap* contains the maximum number of 512-byte sectors that are available to be written with user data when the zone is in the Empty state. This value shall be less than or equal to the *zone_sectors* value in *virtio_blk_zoned_characteristics* structure in the device configuration space.

The zone descriptor field *z_start* contains the zone sector address.

The zone descriptor field *z_wp* contains the sector address where the next write operation for this zone should be issued. This value is undefined for conventional zones and for zones in VIRTIO_BLK_ZS_-RDONLY, VIRTIO_BLK_ZS_FULL and VIRTIO_BLK_ZS_OFFLINE state.

Depending on their state, zones consume resources as follows:

- a zone in VIRTIO_BLK_ZS_IOPEN and VIRTIO_BLK_ZS_EOPEN state consumes one open zone resource and, additionally,
- a zone in VIRTIO_BLK_ZS_IOPEN, VIRTIO_BLK_ZS_EOPEN and VIRTIO_BLK_ZS_CLOSED state consumes one active resource.

Attempts for zone transitions that violate zone resource limits must fail with VIRTIO_BLK_S_ZONE_OPEN_-RESOURCE or VIRTIO_BLK_S_ZONE_ACTIVE_RESOURCE *status*.

Zones in the VIRTIO_BLK_ZS_EMPTY (Empty) state have the write pointer value equal to the sector address of the zone. In this state, the entire capacity of the zone is available for writing. A zone can transition from this state to

- VIRTIO_BLK_ZS_IOPEN when a successful VIRTIO_BLK_T_OUT request or VIRTIO_BLK_T_ZONE_-APPEND with a non-zero data size is received for the zone.
- VIRTIO_BLK_ZS_EOPEN when a successful VIRTIO_BLK_T_ZONE_OPEN request is received for the zone

When a VIRTIO_BLK_T_ZONE_RESET request is issued to an Empty zone, the request is completed successfully and the zone stays in the VIRTIO_BLK_ZS_EMPTY state.

Zones in the VIRTIO_BLK_ZS_IOPEN (Implicitly Open) state transition from this state to

- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET request is received for the zone,
- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET_ALL request is received by the device,

- VIRTIO_BLK_ZS_EOPEN when a successful VIRTIO_BLK_T_ZONE_OPEN request is received for the zone,
- VIRTIO_BLK_ZS_CLOSED when a successful VIRTIO_BLK_T_ZONE_CLOSE request is received for the zone,
- VIRTIO_BLK_ZS_CLOSED implicitly by the device when another zone is entering the VIRTIO_BLK_ZS_IOPEN or VIRTIO_BLK_ZS_EOPEN state and the number of currently open zones is at *max_open_zones* limit,
- VIRTIO_BLK_ZS_FULL when a successful VIRTIO_BLK_T_ZONE_FINISH request is received for the zone.
- VIRTIO_BLK_ZS_FULL when a successful VIRTIO_BLK_T_OUT or VIRTIO_BLK_T_ZONE_APPEND request that causes the zone to reach its writable capacity is received for the zone.

Zones in the VIRTIO_BLK_ZS_EOPEN (Explicitly Open) state transition from this state to

- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET request is received for the zone,
- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET_ALL request is received by the device,
- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_CLOSE request is received for the zone and the write pointer of the zone has the value equal to the start sector of the zone,
- VIRTIO_BLK_ZS_CLOSED when a successful VIRTIO_BLK_T_ZONE_CLOSE request is received for the zone and the zone write pointer is larger then the start sector of the zone,
- VIRTIO_BLK_ZS_FULL when a successful VIRTIO_BLK_T_ZONE_FINISH request is received for the zone,
- VIRTIO_BLK_ZS_FULL when a successful VIRTIO_BLK_T_OUT or VIRTIO_BLK_T_ZONE_APPEND request that causes the zone to reach its writable capacity is received for the zone.

When a VIRTIO_BLK_T_ZONE_EOPEN request is issued to an Explicitly Open zone, the request is completed successfully and the zone stays in the VIRTIO_BLK_ZS_EOPEN state.

Zones in the VIRTIO_BLK_ZS_CLOSED (Closed) state transition from this state to

- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET request is received for the zone,
- VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET_ALL request is received by the device,
- VIRTIO_BLK_ZS_IOPEN when a successful VIRTIO_BLK_T_OUT request or VIRTIO_BLK_T_ZONE_APPEND with a non-zero data size is received for the zone.
- VIRTIO_BLK_ZS_EOPEN when a successful VIRTIO_BLK_T_ZONE_OPEN request is received for the zone,

When a VIRTIO_BLK_T_ZONE_CLOSE request is issued to a Closed zone, the request is completed successfully and the zone stays in the VIRTIO_BLK_ZS_CLOSED state.

Zones in the VIRTIO_BLK_ZS_FULL (Full) state transition from this state to VIRTIO_BLK_ZS_EMPTY when a successful VIRTIO_BLK_T_ZONE_RESET request is received for the zone or a successful VIRTIO_BLK_T_ZONE_RESET_ALL request is received by the device.

When a VIRTIO_BLK_T_ZONE_FINISH request is issued to a Full zone, the request is completed successfully and the zone stays in the VIRTIO_BLK_ZS_FULL state.

The device may automatically transition zones to VIRTIO_BLK_ZS_RDONLY (Read-Only) or VIRTIO_BLK_ZS_OFFLINE (Offline) state from any other state. The device may also automatically transition zones in the Read-Only state to the Offline state. Zones in the Offline state may not transition to any other state.

Such automatic transitions usually indicate hardware failures. The previously written data may only be read from zones in the Read-Only state. Zones in the Offline state can not be read or written.

VIRTIO_BLK_S_ZONE_UNALIGNED_WP is set by the device when the request received from the driver attempts to perform a write to an SWR zone and at least one of the following conditions is met:

- the starting sector of the request is not equal to the current value of the zone write pointer.
- the ending sector of the request data multiplied by 512 is not a multiple of the value reported by the device in the field *write_granularity* in the device configuration space.

VIRTIO_BLK_S_ZONE_OPEN_RESOURCE is set by the device when a zone operation or write request received from the driver can not be handled without exceeding the *max_open_zones* limit value reported by the device in the configuration space.

VIRTIO_BLK_S_ZONE_ACTIVE_RESOURCE is set by the device when a zone operation or write request received from the driver can not be handled without exceeding the *max_active_zones* limit value reported by the device in the configuration space.

A zone transition request that leads to both the *max_open_zones* and the *max_active_zones* limits to be exceeded is terminated by the device with VIRTIO_BLK_S_ZONE_ACTIVE_RESOURCE *status* value.

The device reports all other error conditions related to zoned block model operation by setting the VIRTIO_-BLK_S_ZONE_INVALID_CMD value in *status* of *virtio_blk_req* structure.

5.2.6.1 Driver Requirements: Device Operation

The driver SHOULD check if the content of the *capacity* field has changed upon receiving a configuration change notification.

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.

The length of *data* MUST be a multiple of 512 bytes for VIRTIO_BLK_T_IN and VIRTIO_BLK_T_OUT requests.

The length of *data* MUST be a multiple of the size of struct virtio_blk_discard_write_zeroes for VIRTIO_-BLK_T_DISCARD, VIRTIO_BLK_T_SECURE_ERASE and VIRTIO_BLK_T_WRITE_ZEROES requests.

The length of *data* MUST be 20 bytes for VIRTIO_BLK_T_GET_ID requests.

VIRTIO_BLK_T_DISCARD requests MUST NOT contain more than *max_discard_seg* struct virtio_blk_-discard_write_zeroes segments in *data*.

VIRTIO_BLK_T_SECURE_ERASE requests MUST NOT contain more than *max_secure_erase_seg* struct virtio_blk_discard_write_zeroes segments in *data*.

VIRTIO_BLK_T_WRITE_ZEROES requests MUST NOT contain more than *max_write_zeroes_seg* struct virtio_blk_discard_write_zeroes segments in *data*.

If the VIRTIO_BLK_F_CONFIG_WCE feature is negotiated, the driver MAY switch to writethrough or write-back 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.

A driver MUST NOT assume that individual segments in a multi-segment VIRTIO_BLK_T_DISCARD or VIRTIO_BLK_T_WRITE_ZEROES request completed successfully, failed, or were processed by the device at all if the request failed with VIRTIO_BLK_S_IOERR.

The following requirements only apply if the VIRTIO_BLK_F_ZONED feature is negotiated.

A zone sector address provided by the driver MUST be a multiple of 512 bytes.

When forming a VIRTIO_BLK_T_ZONE_REPORT request, the driver MUST set a sector within the sector range of the starting zone to report to *sector* field. It MAY be a sector that is different from the zone sector address.

In VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_T_ZONE_FINISH and VIRTIO_BLK_T_ZONE_RESET requests, the driver MUST set *sector* field to point at the first sector in the target zone.

In VIRTIO_BLK_T_ZONE_RESET_ALL request, the driver MUST set the field *sector* to zero value.

The *sector* field of the VIRTIO_BLK_T_ZONE_APPEND request MUST specify the zone sector address of the zone to which data is to be appended at the position of the write pointer. The size of the data that is appended MUST be a multiple of *write_granularity* bytes and MUST NOT exceed the *max_append_sectors* value provided by the device in *virtio_blk_zoned_characteristics* configuration space structure.

Upon a successful completion of a VIRTIO_BLK_T_ZONE_APPEND request, the driver MAY read the starting sector location of the written data from the request field *append_sector*.

All VIRTIO_BLK_T_OUT requests issued by the driver to sequential zones and VIRTIO_BLK_T_ZONE_APPEND requests MUST have:

1. the data size that is a multiple of the number of bytes reported by the device in the field *write_granularity* in the *virtio_blk_zoned_characteristics* configuration space structure.
2. the value of the field *sector* that is a multiple of the number of bytes reported by the device in the field *write_granularity* in the *virtio_blk_zoned_characteristics* configuration space structure.
3. the data size that will not exceed the writable zone capacity when its value is added to the current value of the write pointer of the zone.

5.2.6.2 Device Requirements: Device Operation

The device MAY change the content of the *capacity* field during operation of the device. When this happens, the device SHOULD trigger a configuration change notification.

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, secure erase 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.

If the device is backed by storage providing lifetime metrics (such as eMMC or UFS persistent storage), the device SHOULD offer the VIRTIO_BLK_F_LIFETIME flag. The flag MUST NOT be offered if the device is backed by storage for which the lifetime metrics described in this document cannot be obtained or for which such metrics have no useful meaning. If the metrics are offered, the device MUST NOT send any reserved values, as defined in this specification.

Note: The device lifetime metrics *pre_eol_info*, *device_lifetime_est_a* and *device_lifetime_est_b* are discussed in the JESD84-B50 specification.

The complete JESD84-B50 is available at the JEDEC website (https://www.jedec.org) pursuant to JEDEC's licensing terms and conditions. This information is provided to simplfy passthrough implementations from eMMC devices.

If the VIRTIO_BLK_F_ZONED feature is not negotiated, the device MUST reject VIRTIO_BLK_T_ZONE_REPORT, VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_T_ZONE_FINISH, VIRTIO_BLK_T_ZONE_APPEND, VIRTIO_BLK_T_ZONE_RESET and VIRTIO_BLK_T_ZONE_RESET_ALL requests with VIRTIO_BLK_S_UNSUPP status.

The following device requirements only apply if the VIRTIO_BLK_F_ZONED feature is negotiated.

If a request of type VIRTIO_BLK_T_ZONE_OPEN, VIRTIO_BLK_T_ZONE_CLOSE, VIRTIO_BLK_T_ZONE_FINISH or VIRTIO_BLK_T_ZONE_RESET is issued for a Conventional zone (type VIRTIO_BLK_ZT_CONV), the device MUST complete the request with VIRTIO_BLK_S_ZONE_INVALID_CMD *status*.

If the zone specified by the VIRTIO_BLK_T_ZONE_APPEND request is not a SWR zone, then the request SHALL be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD *status*.

The device handles a VIRTIO_BLK_T_ZONE_OPEN request by attempting to change the state of the zone with the *sector* address to VIRTIO_BLK_ZS_EOPEN. If the transition to this state can not be performed, the request MUST be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD *status*. If, while processing this request, the available zone resources are insufficient, then the zone state does not change and the request MUST be completed with VIRTIO_BLK_S_ZONE_OPEN_RESOURCE or VIRTIO_BLK_S_ZONE_ACTIVE_RESOURCE value in the field *status*.

The device handles a VIRTIO_BLK_T_ZONE_CLOSE request by attempting to change the state of the zone with the *sector* address to VIRTIO_BLK_ZS_CLOSED. If the transition to this state can not be performed, the request MUST be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD value in the field *status*.

The device handles a VIRTIO_BLK_T_ZONE_FINISH request by attempting to change the state of the zone with the *sector* address to VIRTIO_BLK_ZS_FULL. If the transition to this state can not be performed, the zone state does not change and the request MUST be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD value in the field *status*.

The device handles a VIRTIO_BLK_T_ZONE_RESET request by attempting to change the state of the zone with the *sector* address to VIRTIO_BLK_ZS_EMPTY state. If the transition to this state can not be performed, the zone state does not change and the request MUST be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD value in the field *status*.

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

The device handles a VIRTIO_BLK_T_ZONE_RESET_ALL request by transitioning all sequential device zones in VIRTIO_BLK_ZS_IOPEN, VIRTIO_BLK_ZS_EOPEN, VIRTIO_BLK_ZS_CLOSED and VIRTIO_-BLK_ZS_FULL state to VIRTIO_BLK_ZS_EMPTY state.

Upon receiving a VIRTIO_BLK_T_ZONE_APPEND request or a VIRTIO_BLK_T_OUT request issued to a SWR zone in VIRTIO_BLK_ZS_EMPTY or VIRTIO_BLK_ZS_CLOSED state, the device attempts to perform the transition of the zone to VIRTIO_BLK_ZS_IOPEN state before writing data. This transition may fail due to insufficient open and/or active zone resources available on the device. In this case, the request MUST be completed with VIRTIO_BLK_S_ZONE_OPEN_RESOURCE or VIRTIO_BLK_S_ZONE_ACTIVE_RE-SOURCE value in the *status*.

If the *sector* field in the VIRTIO_BLK_T_ZONE_APPEND request does not specify the lowest sector for a zone, then the request SHALL be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD value in *status*.

A VIRTIO_BLK_T_ZONE_APPEND request or a VIRTIO_BLK_T_OUT request that has the data range that exceeds the remaining writable capacity for the zone, then the request SHALL be completed with VIRTIO_-BLK_S_ZONE_INVALID_CMD value in *status*.

If a request of the type VIRTIO_BLK_T_ZONE_APPEND is completed with VIRTIO_BLK_S_OK status, the field *append_sector* in *virtio_blk_req_za* MUST be set by the device to contain the first sector of the data written to the zone.

If a request of the type VIRTIO_BLK_T_ZONE_APPEND is completed with a status other than VIRTIO_-BLK_S_OK, the value of *append_sector* field in *virtio_blk_req_za* is undefined.

A VIRTIO_BLK_T_ZONE_APPEND request that has the data size that exceeds *max_append_sectors* configuration space value, then,

- if *max_append_sectors* configuration space value is reported as zero by the device, the request SHALL be completed with VIRTIO_BLK_S_UNSUPP *status*.
- if *max_append_sectors* configuration space value is reported as a non-zero value by the device, the request SHALL be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD *status*.

If a VIRTIO_BLK_T_ZONE_APPEND request, a VIRTIO_BLK_T_IN request or a VIRTIO_BLK_T_OUT request issued to a SWR zone has the range that has sectors in more than one zone, then the request SHALL be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD value in the field *status*.

A VIRTIO_BLK_T_OUT request that has the *sector* value that is not aligned with the write pointer for the zone, then the request SHALL be completed with VIRTIO_BLK_S_ZONE_UNALIGNED_WP value in the field *status*.

In order to avoid resource-related errors while opening zones implicitly, the device MAY automatically transition zones in VIRTIO_BLK_ZS_IOPEN state to VIRTIO_BLK_ZS_CLOSED state.

All VIRTIO_BLK_T_OUT requests or VIRTIO_BLK_T_ZONE_APPEND requests issued to a zone in the VIRTIO_BLK_ZS_RDONLY state SHALL be completed with VIRTIO_BLK_S_ZONE_INVALID_CMD *status*.

All requests issued to a zone in the VIRTIO_BLK_ZS_OFFLINE state SHALL be completed with VIRTIO_-BLK_S_ZONE_INVALID_CMD value in the field *status*.

The device MUST consider the sectors that are read between the write pointer position of a zone and the end of the last sector of the zone as unwritten data. The sectors between the write pointer position and the end of the last sector within the zone capacity during VIRTIO_BLK_T_ZONE_FINISH request processing are also considered unwritten data.

When unwritten data is present in the sector range of a read request, the device MUST process this data in one of the following ways -

1. Fill the unwritten data with a device-specific byte pattern. The configuration, control and reporting of this byte pattern is beyond the scope of this standard. This is the preferred approach.
2. Fail the request. Depending on the driver implementation, this may prevent the device from becoming operational.

If both the VIRTIO_BLK_F_ZONED and VIRTIO_BLK_F_SECURE_ERASE features are negotiated, then

1. the field *secure_erase_sector_alignment* in the configuration space of the device MUST be a multiple of *zone_sectors* value reported in the device configuration space.
2. the data size in VIRTIO_BLK_T_SECURE_ERASE requests MUST be a multiple of *zone_sectors* value in the device configuration space.

The device MUST handle a VIRTIO_BLK_T_SECURE_ERASE request in the same way it handles VIRTIO_BLK_T_ZONE_RESET request for the zone range specified in the VIRTIO_BLK_T_SECURE_ERASE request.

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.

```
#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. *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 2.7.4.

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].

[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

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.

5.3.6.1 Driver Requirements: Device Operation

The driver MUST NOT put a device-readable buffer 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).

is closed or hot-unplugged.

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 device-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

3 free_page_vq

4 reporting_vq

statsq only exists if VIRTIO_BALLOON_F_STATS_VQ is set.

free_page_vq only exists if VIRTIO_BALLOON_F_FREE_PAGE_HINT is set.

reporting_vq only exists if VIRTIO_BALLOON_F_PAGE_REPORTING is 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.

VIRTIO_BALLOON_F_FREE_PAGE_HINT(3) The device has support for free page hinting. A virtqueue for providing hints as to what memory is currently free is present. Configuration field *free_page_hint_cmd_id* is valid.

VIRTIO_BALLOON_F_PAGE_POISON(4) A hint to the device, that the driver will immediately write *poison_val* to pages after deflating them. Configuration field *poison_val* is valid.

VIRTIO_BALLOON_F_PAGE_REPORTING(5) The device has support for free page reporting. A virtqueue for reporting free guest memory is present.

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.

The driver SHOULD clear the VIRTIO_BALLOON_F_PAGE_POISON flag if it will not immediately write *poison_val* to deflated pages (e.g., to initialize them, or fill them with a poison value).

If the driver is expecting the pages to retain some initialized value, it MUST NOT accept VIRTIO_BALLOON_F_PAGE_REPORTING unless it also negotiates VIRTIO_BALLOON_F_PAGE_POISON.

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

num_pages and *actual* are always available.

free_page_hint_cmd_id is available if VIRTIO_BALLOON_F_FREE_PAGE_HINT has been negotiated. The field is read-only by the driver. *poison_val* is available if VIRTIO_BALLOON_F_PAGE_POISON has been negotiated.

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

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.
3. If the VIRTIO_BALLOON_F_FREE_PAGE_HINT feature bit is negotiated, identify the free_page_vq.
4. If the VIRTIO_BALLOON_F_PAGE_POISON feature bit is negotiated, update the *poison_val* configuration field.

5. If the VIRTIO_BALLOON_F_PAGE_REPORTING feature bit is negotiated, identify the reporting_vq.
6. DRIVER_OK is set: device operation begins.
7. If the VIRTIO_BALLOON_F_STATS_VQ feature bit is negotiated, then notify the device about the stats virtqueue buffer.
8. If the VIRTIO_BALLOON_F_PAGE_REPORTING feature bit is negotiated, then begin reporting free pages to the device.

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.
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.

[7] This is historical, and independent of the guest page size.
[8] In this case, deflation advice is merely a courtesy.

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.

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 10-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.

5.5.6.5 Free Page Hinting

Free page hinting is designed to be used during migration to determine what pages within the guest are currently unused so that they can be skipped over while migrating the guest. The device will indicate that it is ready to start performing hinting by setting the *free_page_hint_cmd_id* to one of the non-reserved values that can be used as a command ID. The following values are reserved:

VIRTIO_BALLOON_CMD_ID_STOP (0) Any command ID previously supplied by the device is invalid. The driver should stop hinting free pages until a new command ID is supplied, but should not release any hinted pages for use by the guest.

VIRTIO_BALLOON_CMD_ID_DONE (1) Any command ID previously supplied by the device is invalid. The driver should stop hinting free pages, and should release all hinted pages for use by the guest.

When a hint is provided by the driver it indicates that the data contained in the given page is no longer needed and can be discarded. If the driver writes to the page this overrides the hint and the data will be retained. The contents of any stale pages that have not been written to since the page was hinted may be lost, and if read the contents of such pages will be uninitialized memory.

A request for free page hinting proceeds as follows:

1. The driver examines the *free_page_hint_cmd_id* configuration field. If it contains a non-reserved value then free page hinting will begin.
2. To supply free page hints:
   (a) The driver constructs an output buffer containing the new value from the *free_page_hint_cmd_id* configuration field and adds it to the free_page_vq.
   (b) The driver maps a series of pages and adds them to the free_page_vq as individual scatter-gather input buffer entries.
   (c) When the driver is no longer able to fetch additional pages to add to the free_page_vq, it will construct an output buffer containing the command ID VIRTIO_BALLOON_CMD_ID_STOP.
3. A round of hinting ends either when the driver is no longer able to supply more pages for hinting as described above, or when the device updates *free_page_hint_cmd_id* configuration field to contain either VIRTIO_BALLOON_CMD_ID_STOP or VIRTIO_BALLOON_CMD_ID_DONE.
4. The device may follow VIRTIO_BALLOON_CMD_ID_STOP with a new non-reserved value for the *free_page_hint_cmd_id* configuration field in which case it will resume supplying free page hints.
5. Otherwise, if the device provides VIRTIO_BALLOON_CMD_ID_DONE then hinting is complete and the driver may release all previously hinted pages for use by the guest.

5.5.6.5.1 Driver Requirements: Free Page Hinting

Normative statements in this section apply if the VIRTIO_BALLOON_F_FREE_PAGE_HINT feature has been negotiated.

The driver MUST use an output buffer size of 4 bytes for all output buffers submitted to the free_page_vq.

The driver MUST start hinting by providing an output buffer containing the current command ID for the given block of pages.

The driver MUST NOT provide more than one output buffer containing the current command ID.

The driver SHOULD supply pages to the free_page_vq as input buffers when *free_page_hint_cmd_id* specifies a value of 2 or greater.

The driver SHOULD stop supplying pages for hinting when *free_page_hint_cmd_id* specifies a value of VIRTIO_BALLOON_CMD_ID_STOP or VIRTIO_BALLOON_CMD_ID_DONE.

If the driver is unable to supply pages, it MUST complete hinting by adding an output buffer containing the command ID VIRTIO_BALLOON_CMD_ID_STOP.

The driver MAY release hinted pages for use by the guest including when the device has not yet used the descriptor containing the hinting request.

The driver MUST treat the content of all hinted pages as uninitialized memory.

The driver MUST initialize the contents of any previously hinted page released before *free_page_hint_cmd_id* specifies a value of VIRTIO_BALLOON_CMD_ID_DONE.

The driver SHOULD release all previously hinted pages once *free_page_hint_cmd_id* specifies a value of VIRTIO_BALLOON_CMD_ID_DONE.

5.5.6.5.2 Device Requirements: Free Page Hinting

Normative statements in this section apply if the VIRTIO_BALLOON_F_FREE_PAGE_HINT feature has been negotiated.

The device SHOULD set *free_page_hint_cmd_id* to VIRTIO_BALLOON_CMD_ID_STOP any time that it will not be able to make use of the hints provided by the driver.

The device MUST NOT reuse a command ID until it has received an output buffer containing VIRTIO_BALLOON_CMD_ID_STOP from the driver.

The device MUST ignore pages that are provided with a command ID that does not match the current value in *free_page_hint_cmd_id*.

If the content of a previously hinted page has not been modified by the guest since the device issued the *free_page_hint_cmd_id* associated with the hint, the device MAY modify the contents of the page.

The device MUST NOT modify the content of a previously hinted page after *free_page_hint_cmd_id* is set to VIRTIO_BALLOON_CMD_ID_DONE.

The device MUST report a value of VIRTIO_BALLOON_CMD_ID_DONE in *free_page_hint_cmd_id* when it no longer has need for the previously hinted pages.

5.5.6.5.3 Legacy Interface: Free Page Hinting

When using the legacy interface, transitional devices and drivers MUST format the command ID field in output buffers according to the native endian of the guest rather than (necessarily when not using the legacy interface) little-endian.

5.5.6.6 Page Poison

Page Poison provides a way to notify the host that the guest is initializing free pages with *poison_val*. When the feature is enabled, pages will be immediately written to by the driver after deflating, and pages reported by free page reporting will retain the value indicated by *poison_val*.

If the guest is not initializing freed pages, the driver should reject the VIRTIO_BALLOON_F_PAGE_POISON feature.

If VIRTIO_BALLOON_F_PAGE_POISON feature has been negotiated, the driver will place the initialization value into the *poison_val* configuration field data.

5.5.6.6.1 Driver Requirements: Page Poison

Normative statements in this section apply if the VIRTIO_BALLOON_F_PAGE_POISON feature has been negotiated.

The driver MUST initialize the deflated pages with *poison_val* when they are reused by the driver.

The driver MUST populate the *poison_val* configuration data before setting the DRIVER_OK bit.

The driver MUST NOT modify *poison_val* while the DRIVER_OK bit is set.

5.5.6.6.2 Device Requirements: Page Poison

Normative statements in this section apply if the VIRTIO_BALLOON_F_PAGE_POISON feature has been negotiated.

The device MAY use the content of *poison_val* as a hint to guest behavior.

5.5.6.7 Free Page Reporting

Free Page Reporting provides a mechanism similar to balloon inflation, however it does not provide a deflation queue. Reported free pages can be reused by the driver after the reporting request has been acknowledged without notifying the device.

The driver will begin reporting free pages. When exactly and which free pages are reported is up to the driver.

1. The driver determines it has enough pages available to begin reporting free pages.
2. The driver gathers free pages into a scatter-gather list and adds them to the reporting_vq.
3. The device acknowledges the reporting request by using the reporting_vq descriptor.
4. Once the device has acknowledged the report, the driver can reuse the reported free pages when needed (e.g., by putting them back to free page lists in the guest operating system).
5. The driver can then continue to gather and report free pages until it has determined it has reported a sufficient quantity of pages.

5.5.6.7.1 Driver Requirements: Free Page Reporting

Normative statements in this section apply if the VIRTIO_BALLOON_F_PAGE_REPORTING feature has been negotiated.

If the VIRTIO_BALLOON_F_PAGE_POISON feature has not been negotiated, then the driver MUST treat all reported pages as uninitialized memory.

If the VIRTIO_BALLOON_F_PAGE_POISON feature has been negotiated, the driver MUST initialize all free pages with *poison_val* before reporting them.

The driver MUST NOT use the reported pages until the device has acknowledged the reporting request.

The driver MAY report free pages any time after DRIVER_OK is set.

The driver SHOULD attempt to report large pages rather than smaller ones.

The driver SHOULD avoid reading/writing reported pages if not strictly necessary.

5.5.6.7.2 Device Requirements: Free Page Reporting

Normative statements in this section apply if the VIRTIO_BALLOON_F_PAGE_REPORTING feature has been negotiated.

If the VIRTIO_BALLOON_F_PAGE_POISON feature has not been negotiated, the device MAY modify the contents of any page supplied in a report request before acknowledging that request by using the reporting_vq descriptor.

If the VIRTIO_BALLOON_F_PAGE_POISON feature has been negotiated, the device MUST NOT modify the the content of a reported page to a value other than *poison_val*.

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.

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.

[9] For example, INQUIRY or REPORT LUNS.

[10] For example, I_T RESET.

5.6.6.1 Device Operation: Request Queues

The driver enqueues 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).

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.

11There 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*.

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 VIR-TIO_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 mode. 3D mode will offload rendering ops to the host gpu and therefore requires a gpu with 3D support on the host machine.

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 through 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.

VIRTIO_GPU_F_RESOURCE_UUID (2) assigning resources UUIDs for export to other virtio devices is supported.

VIRTIO_GPU_F_RESOURCE_BLOB (3) creating and using size-based blob resources is supported.

VIRTIO_GPU_F_CONTEXT_INIT (4) multiple context types and synchronization timelines supported. Requires VIRTIO_GPU_F_VIRGL.

5.7.4 Device configuration layout

GPU device configuration uses the following layout structure and definitions:

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

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

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.

num_capsets specifies the maximum number of capability sets supported by the device. The minimum value is zero.

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. In case EDID support is negotiated (VIRTIO_GPU_F_EDID feature flag) the device SHOULD also fetch the updated EDID blobs using the VIRTIO_GPU_CMD_GET_EDID command.

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 EDID support is negotiated (VIRTIO_GPU_F_EDID feature flag) the device SHOULD also fetch the EDID information using the VIRTIO_GPU_CMD_GET_EDID command. If no information is available or all displays are disabled the driver MAY choose to use a fallback, such as 1024x768 at display 0.

The driver SHOULD query all shared memory regions supported by the device. If the device supports shared memory, the *shmid* of a region MUST (see 2.10 Shared Memory Regions) be one of the following:

```
enum virtio_gpu_shm_id {
        VIRTIO_GPU_SHM_ID_UNDEFINED = 0,
        VIRTIO_GPU_SHM_ID_HOST_VISIBLE = 1,
};
```

The shared memory region with VIRTIO_GPU_SHM_ID_HOST_VISIBLE is referred as the "host visible memory region". The device MUST support the VIRTIO_GPU_CMD_RESOURCE_MAP_BLOB and VIRTIO_GPU_CMD_RESOURCE_UNMAP_BLOB if the host visible memory region is available.

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, unless shared memory regions are supported. 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 contiguous 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 asynchronously 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

All requests and responses on the virtqueues have a fixed header using the following layout structure and definitions:

```
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,
        VIRTIO_GPU_CMD_RESOURCE_ASSIGN_UUID,
        VIRTIO_GPU_CMD_RESOURCE_CREATE_BLOB,
        VIRTIO_GPU_CMD_SET_SCANOUT_BLOB,

        /* 3d commands */
        VIRTIO_GPU_CMD_CTX_CREATE = 0x0200,
        VIRTIO_GPU_CMD_CTX_DESTROY,
        VIRTIO_GPU_CMD_CTX_ATTACH_RESOURCE,
        VIRTIO_GPU_CMD_CTX_DETACH_RESOURCE,
        VIRTIO_GPU_CMD_RESOURCE_CREATE_3D,
        VIRTIO_GPU_CMD_TRANSFER_TO_HOST_3D,
        VIRTIO_GPU_CMD_TRANSFER_FROM_HOST_3D,
        VIRTIO_GPU_CMD_SUBMIT_3D,
        VIRTIO_GPU_CMD_RESOURCE_MAP_BLOB,
        VIRTIO_GPU_CMD_RESOURCE_UNMAP_BLOB,

        /* 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,
        VIRTIO_GPU_RESP_OK_RESOURCE_UUID,
        VIRTIO_GPU_RESP_OK_MAP_INFO,

        /* 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)
#define VIRTIO_GPU_FLAG_INFO_RING_IDX (1 << 1)

struct virtio_gpu_ctrl_hdr {
        le32 type;
        le32 flags;
        le64 fence_id;
        le32 ctx_id;
        u8 ring_idx;
        u8 padding[3];
};
```

The fixed header *struct virtio_gpu_ctrl_hdr* in each request includes the following fields:

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).

ring_idx If VIRTIO_GPU_F_CONTEXT_INIT is supported, then the driver MAY set VIRTIO_GPU_FLAG_-INFO_RING_IDX bit in the request *flags*. In that case:

- *ring_idx* indicates the value of a context-specific ring index. The minimum value is 0 and maximum value is 63 (inclusive).
- If VIRTIO_GPU_FLAG_FENCE is set, *fence_id* acts as a sequence number on the synchronization timeline defined by *ctx_idx* and the ring index.
- If VIRTIO_GPU_FLAG_FENCE is set and when the command associated with *fence_id* is complete, the device MUST send a response for all outstanding commands with a sequence number less than or equal to *fence_id* on the same synchronization timeline.

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 physical 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.

VIRTIO_GPU_CMD_GET_CAPSET_INFO Gets the information associated with a particular *capset_index*, which MUST less than *num_capsets* defined in the device configuration. Request data is *struct virtio_gpu_get_capset_info*. Response type is VIRTIO_GPU_RESP_OK_CAPSET_INFO.

On success, *struct virtio_gpu_resp_capset_info* contains the *capset_id*, *capset_max_version*, *capset_max_size* associated with capset at the specified capset_idex. fieldcapset_id MUST be one of the following (see listing for values):

- VIRTIO_GPU_CAPSET_VIRGL – the first edition of Virgl (Gallium OpenGL) protocol.
- VIRTIO_GPU_CAPSET_VIRGL2 – the second edition of Virgl (Gallium OpenGL) protocol after the capset fix.
- VIRTIO_GPU_CAPSET_GFXSTREAM – gfxtream's (mostly) autogenerated GLES and Vulkan streaming protocols.
- VIRTIO_GPU_CAPSET_VENUS – Mesa's (mostly) autogenerated Vulkan protocol.
- VIRTIO_GPU_CAPSET_CROSS_DOMAIN – protocol for display virtualization via Wayland proxying.

```
struct virtio_gpu_get_capset_info {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 capset_index;
        le32 padding;
};

#define VIRTIO_GPU_CAPSET_VIRGL 1
#define VIRTIO_GPU_CAPSET_VIRGL2 2
#define VIRTIO_GPU_CAPSET_GFXSTREAM 3
#define VIRTIO_GPU_CAPSET_VENUS 4
#define VIRTIO_GPU_CAPSET_CROSS_DOMAIN 5
struct virtio_gpu_resp_capset_info {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 capset_id;
        le32 capset_max_version;
        le32 capset_max_size;
        le32 padding;
};
```

VIRTIO_GPU_CMD_GET_CAPSET Gets the capset associated with a particular *capset_id* and *capset_version*. Request data is *struct virtio_gpu_get_capset*. Response type is VIRTIO_GPU_RESP_OK_CAPSET.

```
struct virtio_gpu_get_capset {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 capset_id;
        le32 capset_version;
};

struct virtio_gpu_resp_capset {
        struct virtio_gpu_ctrl_hdr hdr;
        u8 capset_data[];
};
```

VIRTIO_GPU_CMD_RESOURCE_ASSIGN_UUID Creates an exported object from a resource. Request data is *struct virtio_gpu_resource_assign_uuid*. Response type is VIRTIO_GPU_RESP_OK_RESOURCE_UUID, response data is *struct virtio_gpu_resp_resource_uuid*. Support is optional and negotiated using the VIRTIO_GPU_F_RESOURCE_UUID feature flag.

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

struct virtio_gpu_resp_resource_uuid {
        struct virtio_gpu_ctrl_hdr hdr;
        u8 uuid[16];
};
```

The response contains a UUID which identifies the exported object created from the host private resource. Note that if the resource has an attached backing, modifications made to the host private resource through the exported object by other devices are not visible in the attached backing until they are transferred into the backing.

VIRTIO_GPU_CMD_RESOURCE_CREATE_BLOB Creates a virtio-gpu blob resource. Request data is *struct virtio_gpu_resource_create_blob*, followed by *struct virtio_gpu_mem_entry* entries. Response type is VIRTIO_GPU_RESP_OK_NODATA. Support is optional and negotiated using the VIRTIO_-GPU_F_RESOURCE_BLOB feature flag.

```
#define VIRTIO_GPU_BLOB_MEM_GUEST             0x0001
#define VIRTIO_GPU_BLOB_MEM_HOST3D            0x0002
#define VIRTIO_GPU_BLOB_MEM_HOST3D_GUEST      0x0003

#define VIRTIO_GPU_BLOB_FLAG_USE_MAPPABLE     0x0001
#define VIRTIO_GPU_BLOB_FLAG_USE_SHAREABLE    0x0002
#define VIRTIO_GPU_BLOB_FLAG_USE_CROSS_DEVICE 0x0004

struct virtio_gpu_resource_create_blob {
       struct virtio_gpu_ctrl_hdr hdr;
       le32 resource_id;
       le32 blob_mem;
       le32 blob_flags;
       le32 nr_entries;
       le64 blob_id;
       le64 size;
};
```

A blob resource is a container for:

- a guest memory allocation (referred to as a ”guest-only blob resource”).
- a host memory allocation (referred to as a ”host-only blob resource”).
- a guest memory and host memory allocation (referred to as a ”default blob resource”).

The memory properties of the blob resource MUST be described by *blob_mem*, which MUST be non-zero.

For default and guest-only blob resources, *nr_entries* guest memory entries may be assigned to the resource. For default blob resources (i.e, when *blob_mem* is VIRTIO_GPU_BLOB_MEM_HOST3D_-GUEST), these memory entries are used as a shadow buffer for the host memory. To facilitate drivers that support swap-in and swap-out, *nr_entries* may be zero and VIRTIO_GPU_CMD_RESOURCE_-ATTACH_BACKING may be subsequently used. VIRTIO_GPU_CMD_RESOURCE_DETACH_BACK-ING may be used to unassign memory entries.

blob_mem can only be VIRTIO_GPU_BLOB_MEM_HOST3D and VIRTIO_GPU_BLOB_MEM_HOST3D_-GUEST if VIRTIO_GPU_F_VIRGL is supported. VIRTIO_GPU_BLOB_MEM_GUEST is valid regard-less whether VIRTIO_GPU_F_VIRGL is supported or not.

For VIRTIO_GPU_BLOB_MEM_HOST3D and VIRTIO_GPU_BLOB_MEM_HOST3D_GUEST, the virtio-gpu resource MUST be created from the rendering context local object identified by the *blob_id*. The actual allocation is done via VIRTIO_GPU_CMD_SUBMIT_3D.

The driver MUST inform the device if the blob resource is used for memory access, sharing between driver instances and/or sharing with other devices. This is done via the *blob_flags* field.

If VIRTIO_GPU_F_VIRGL is set, both VIRTIO_GPU_CMD_TRANSFER_TO_HOST_3D and VIRTIO_-GPU_CMD_TRANSFER_FROM_HOST_3D may be used to update the resource. There is no restric-tion on the image/buffer view the driver has on the blob resource.

VIRTIO_GPU_CMD_SET_SCANOUT_BLOB sets scanout parameters for a blob resource. Request data is *struct virtio_gpu_set_scanout_blob*. Response type is VIRTIO_GPU_RESP_OK_NODATA. Support is optional and negotiated using the VIRTIO_GPU_F_RESOURCE_BLOB feature flag.

```
struct virtio_gpu_set_scanout_blob {
       struct virtio_gpu_ctrl_hdr hdr;
       struct virtio_gpu_rect r;
       le32 scanout_id;
       le32 resource_id;
       le32 width;
       le32 height;
       le32 format;
       le32 padding;
```



```
        le32 strides[4];
        le32 offsets[4];
};
```

The rectangle *r* represents the portion of the blob resource being displayed. The rest is the metadata associated with the blob resource. The format MUST be one of *enum virtio_gpu_formats*. The format MAY be compressed with header and data planes.

5.7.6.9 Device Operation: controlq (3d)

These commands are supported by the device if the VIRTIO_GPU_F_VIRGL feature flag is set.

VIRTIO_GPU_CMD_CTX_CREATE creates a context for submitting an opaque command stream. Request data is *struct virtio_gpu_ctx_create*. Response type is VIRTIO_GPU_RESP_OK_NODATA.

```
#define VIRTIO_GPU_CONTEXT_INIT_CAPSET_ID_MASK 0x000000ff;
struct virtio_gpu_ctx_create {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 nlen;
        le32 context_init;
        char debug_name[64];
};
```

The implementation MUST create a context for the given *ctx_id* in the *hdr*. For debugging purposes, a *debug_name* and it's length *nlen* is provided by the driver. If VIRTIO_GPU_F_CONTEXT_INIT is supported, then lower 8 bits of *context_init* MAY contain the *capset_id* associated with context. In that case, then the device MUST create a context that can handle the specified command stream.

If the lower 8-bits of the *context_init* are zero, then the type of the context is determined by the device.

VIRTIO_GPU_CMD_CTX_DESTROY

VIRTIO_GPU_CMD_CTX_ATTACH_RESOURCE

VIRTIO_GPU_CMD_CTX_DETACH_RESOURCE Manage virtio-gpu 3d contexts.

VIRTIO_GPU_CMD_RESOURCE_CREATE_3D Create virtio-gpu 3d resources.

VIRTIO_GPU_CMD_TRANSFER_TO_HOST_3D

VIRTIO_GPU_CMD_TRANSFER_FROM_HOST_3D Transfer data from and to virtio-gpu 3d resources.

VIRTIO_GPU_CMD_SUBMIT_3D Submit an opaque command stream. The type of the command stream is determined when creating a context.

VIRTIO_GPU_CMD_RESOURCE_MAP_BLOB maps a host-only blob resource into an offset in the host visible memory region. Request data is *struct virtio_gpu_resource_map_blob*. The driver MUST not map a blob resource that is already mapped. Response type is VIRTIO_GPU_RESP_OK_MAP_INFO. Support is optional and negotiated using the VIRTIO_GPU_F_RESOURCE_BLOB feature flag and checking for the presence of the host visible memory region.

```
struct virtio_gpu_resource_map_blob {
        struct virtio_gpu_ctrl_hdr hdr;
        le32 resource_id;
        le32 padding;
        le64 offset;
};

#define VIRTIO_GPU_MAP_CACHE_MASK      0x0f
#define VIRTIO_GPU_MAP_CACHE_NONE      0x00
#define VIRTIO_GPU_MAP_CACHE_CACHED    0x01
#define VIRTIO_GPU_MAP_CACHE_UNCACHED  0x02
#define VIRTIO_GPU_MAP_CACHE_WC        0x03
struct virtio_gpu_resp_map_info {
        struct virtio_gpu_ctrl_hdr hdr;
        u32 map_info;
        u32 padding;
};
```

VIRTIO_GPU_CMD_RESOURCE_UNMAP_BLOB unmaps a host-only blob resource from the host visible memory region. Request data is *struct virtio_gpu_resource_unmap_blob*. Response type is VIRTIO_GPU_RESP_OK_NODATA. Support is optional and negotiated using the VIRTIO_GPU_F_RESOURCE_BLOB feature flag and checking for the presence of the host visible memory region.

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

5.7.6.10 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, AEAD, AKCIPHER and IPSEC. 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. The crypto device may have administration command interface through which IPsec service capabilities and resources are configured.

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.

VIRTIO_CRYPTO_F_AKCIPHER_STATELESS_MODE (5) stateless mode requests are supported by the AKCIPHER 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.

VIRTIO_CRYPTO_F_AKCIPHER_STATELESS_MODE Requires VIRTIO_CRYPTO_F_REVISION_1.

5.9.4 Supported crypto services

The following crypto services are defined:

```
/* CIPHER (Symmetric Key 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
/* AKCIPHER (Asymmetric Key Cipher) service */
#define VIRTIO_CRYPTO_SERVICE_AKCIPHER 4
/* IPSEC service */
#define VIRTIO_CRYPTO_SERVICE_IPSEC  5
```

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 5.9.9.2.1.

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 5.9.9.2.1.

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 5.9.9.2.1.

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 5.9.9.2.1.

5.9.4.5 AKCIPHER services

The following AKCIPHER algorithms are defined:

```
#define VIRTIO_CRYPTO_NO_AKCIPHER 0
#define VIRTIO_CRYPTO_AKCIPHER_RSA   1
#define VIRTIO_CRYPTO_AKCIPHER_ECDSA 2
```

The above constants have two usages:

1. As bit numbers, used to tell the driver which AKCIPHER algorithms are supported by the device, see 5.9.5.
2. As values, used to designate the algorithm in asymmetric crypto operation requests, see 5.9.9.2.1.

5.9.5 Device configuration layout

Crypto device configuration uses the following layout structure:

```
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 akcipher_algo;
    /* 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.

akcipher_algo AKCIPHER algorithms bit 0-31, see 5.9.4.5.

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 that the size of each crypto request's content is 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 that the *key_len* (CIPHER service or AEAD service) is 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 that the *auth_key_len* (MAC service) is 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 and driver capabilities

The crypto device has the following capabilities.

Identifier	Name	Description
0x0800	VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP	IPsec resource capability
0x0801	VIRTIO_CRYPTO_IPSEC_SA_CAP	IPsec Security Association(SA) capability

5.9.8 Device resource objects

The crypto device has the following resource objects.

type	Name	Description
0x0200	VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA	IPsec outbound SA resource object
0x0201	VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA	IPsec inbound SA resource object

5.9.9 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 5.9.9.2.1) 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).

5.9.9.1 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_KEY_REJECTED = 6,
    VIRTIO_CRYPTO_IPSEC_SA_SOFT_EXPIRY = 7,
    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_KEY_REJECTED: signature verification failed (only when AKCIPHER verification).
- VIRTIO_CRYPTO_IPSEC_SA_SOFT_EXPIRY: IPsec SA lifetime soft limits are reached. When VIRTIO_CRYPTO_IPSEC_SA_SOFT_EXPIRY occurs, the request is completed successfully, but one or all of the soft limits are reached. This is applicable only for IPsec service operations.
- VIRTIO_CRYPTO_ERR: any failure not mentioned above occurs.

5.9.9.2 Control Virtqueue

The driver uses the control virtqueue to send control commands to the device, such as session operations (See 5.9.9.2.1).

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)
#define VIRTIO_CRYPTO_AKCIPHER_CREATE_SESSION \
       VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x04)
#define VIRTIO_CRYPTO_AKCIPHER_DESTROY_SESSION \
       VIRTIO_CRYPTO_OPCDE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x05)
    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 parameters.

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

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

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_AKCIPHER_CREATE_SESSION then *op_flf* is struct virtio_crypto_akcipher_create_session_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_crypto_akcipher_create_session_flf is padded to 56 bytes if NOT negotiated, and *op_vlf* is struct virtio_crypto_akcipher_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.

5.9.9.2.1 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;
};
```

5.9.9.2.1.1 Session operation: HASH session

The fixed-length parameters 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;
};
```

5.9.9.2.1.2 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.

5.9.9.2.1.3 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.

5.9.9.2.1.4 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.

5.9.9.2.1.5 Session operation: AKCIPHER session

Due to the complexity of asymmetric key algorithms, different algorithms require different parameters. The following data structures are used as supplementary parameters to describe the asymmetric algorithm sessions.

For the RSA algorithm, the extra parameters are as follows:

```
struct virtio_crypto_rsa_session_para {
#define VIRTIO_CRYPTO_RSA_RAW_PADDING   0
#define VIRTIO_CRYPTO_RSA_PKCS1_PADDING 1
    le32 padding_algo;

#define VIRTIO_CRYPTO_RSA_NO_HASH   0
#define VIRTIO_CRYPTO_RSA_MD2       1
#define VIRTIO_CRYPTO_RSA_MD3       2
#define VIRTIO_CRYPTO_RSA_MD4       3
#define VIRTIO_CRYPTO_RSA_MD5       4
#define VIRTIO_CRYPTO_RSA_SHA1      5
#define VIRTIO_CRYPTO_RSA_SHA256    6
#define VIRTIO_CRYPTO_RSA_SHA384    7
#define VIRTIO_CRYPTO_RSA_SHA512    8
#define VIRTIO_CRYPTO_RSA_SHA224    9
    le32 hash_algo;
};
```

padding_algo specifies the padding method used by RSA sessions.

- If VIRTIO_CRYPTO_RSA_RAW_PADDING is specified, 1) *hash_algo* is ignored, 2) ciphertext and plaintext MUST be padded with leading zeros, 3) and RSA sessions with VIRTIO_CRYPTO_RSA_-RAW_PADDING MUST not be used for verification and signing operations.
- If VIRTIO_CRYPTO_RSA_PKCS1_PADDING is specified, EMSA-PKCS1-v1_5 padding method is used (see PKCS#1), *hash_algo* specifies how the digest of the data passed to RSA sessions is calculated when verifying and signing. It only affects the padding algorithm and is ignored during encryption and decryption.

The ECC algorithms such as the ECDSA algorithm, cannot use custom curves, only the following known curves can be used (see NIST-recommended curves).

```
#define VIRTIO_CRYPTO_CURVE_UNKNOWN   0
#define VIRTIO_CRYPTO_CURVE_NIST_P192 1
#define VIRTIO_CRYPTO_CURVE_NIST_P224 2
#define VIRTIO_CRYPTO_CURVE_NIST_P256 3
#define VIRTIO_CRYPTO_CURVE_NIST_P384 4
#define VIRTIO_CRYPTO_CURVE_NIST_P521 5
```

For the ECDSA algorithm, the extra parameters are as follows:

```
struct virtio_crypto_ecdsa_session_para {
    /* See VIRTIO_CRYPTO_CURVE_* above */
    le32 curve_id;
};
```

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

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

    /* See VIRTIO_CRYPTO_AKCIPHER_* above */
    le32 algo;
#define VIRTIO_CRYPTO_AKCIPHER_KEY_TYPE_PUBLIC 1
#define VIRTIO_CRYPTO_AKCIPHER_KEY_TYPE_PRIVATE 2
    le32 key_type;
    /* length of key */
    le32 key_len;

#define VIRTIO_CRYPTO_AKCIPHER_SESS_ALGO_SPEC_HDR_SIZE 44
    u8 algo_flf[VIRTIO_CRYPTO_AKCIPHER_SESS_ALGO_SPEC_HDR_SIZE];
};

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

algo decides the type used by *algo_flf*. *algo_flf* is fixed to 44 bytes and MUST contains of be one the following structures:

- struct virtio_crypto_rsa_session_para
- struct virtio_crypto_ecdsa_session_para

The length of *key* is specified in *key_len* in the struct virtio_crypto_akcipher_create_session_flf.

For the RSA algorithm, the key needs to be encoded according to PKCS#1. The private key is described with the RSAPrivateKey structure, and the public key is described with the RSAPublicKey structure. These ASN.1 structures are encoded in DER encoding rules (see rfc6025).

```
RSAPrivateKey ::= SEQUENCE {
    version          INTEGER,
    modulus          INTEGER,
    publicExponent   INTEGER,
    privateExponent  INTEGER,
    prime1           INTEGER,
    prime2           INTEGER,
    exponent1        INTEGER,
    exponent1        INTEGER,
    coefficient      INTEGER,
    otherPrimeInfos  OtherPrimeInfos OPTIONAL
}

OtherPrimeInfos ::= SEQUENCE SIZE(1...MAX) OF OtherPrimeInfo

OtherPrimeINfo ::= SEQUENCE {
    prime           INTEGER,
    exponent        INTEGER,
    coefficient     INTEGER
}

RSAPublicKey ::= SEQUENCE {
    modulus         INTEGER,
    publicExponent  INTEGER
}
```

For the ECDSA algorithm, the private key is encoded according to RFC5915, the private key of the ECDSA algorithm is described by the ASN.1 structure ECPrivateKey and encoded with DER encoding rules (see rfc6025).

```
ECPrivateKey ::= SEQUNCE {
    version         INTEGER,
    privateKey      OCTET STRING,
    parameters [0]  ECParameters {{ NamedCurve }} OPTIONAL,
    publicKey  [1]  BIT STRING OPTIONAL
}
```

The public key of the ECDSA algorithm is encoded according to SEC1, and the public key of ECDSA is described by the ASN.1 structure ECPoint. When initializing a session with ECDSA public key, the ECPoint is DER encoded and the *key* only contains the value part of ECPoint, that is, the header part of the OCTET STRING will be omitted (see rfc6025).

```
ECPoint ::= OCTET STRING
```

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

5.9.9.2.1.6 Driver Requirements: Session operation: create session

- The driver MUST set the *opcode* field based on service type: CIPHER, HASH, MAC, AEAD or AKCIPHER.
- 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 5.9.9.2.
- The driver MUST set the *reversed* field to zero.

5.9.9.2.1.7 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.

5.9.9.2.1.8 Driver Requirements: Session operation: destroy session

- The driver MUST set the *opcode* field based on service type: CIPHER, HASH, MAC, AEAD or AKCIPHER.
- The driver MUST set the *session_id* to a valid value assigned by the device when the session was created.

5.9.9.2.1.9 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.

5.9.9.3 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)
#define VIRTIO_CRYPTO_AKCIPHER_ENCRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x00)
#define VIRTIO_CRYPTO_AKCIPHER_DECRYPT \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x01)
#define VIRTIO_CRYPTO_AKCIPHER_SIGN \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x02)
#define VIRTIO_CRYPTO_AKCIPHER_VERIFY \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_AKCIPHER, 0x03)
#define VIRTIO_CRYPTO_IPSEC_OUTBOUND \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_IPSEC, 0x00)
#define VIRTIO_CRYPTO_IPSEC_INBOUND \
    VIRTIO_CRYPTO_OPCODE(VIRTIO_CRYPTO_SERVICE_IPSEC, 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*).

For VIRTIO_CRYPTO_IPSEC_OUTBOUND and VIRTIO_CRYPTO_IPSEC_INBOUND opcodes, *algo* is ignored.

For VIRTIO_CRYPTO_IPSEC_OUTBOUND opcode, *session_id* MUST be set to one of the resource objects *id* created using VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA resource type.

For VIRTIO_CRYPTO_IPSEC_INBOUND opcode, *session_id* MUST be set to one of the resource objects *id* created using VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA resource type.

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.
- If the opcode (in *header*) is VIRTIO_CRYPTO_AKCIPHER_ENCRYPT, VIRTIO_CRYPTO_AKCIPHER_-DECRYPT, VIRTIO_CRYPTO_AKCIPHER_SIGN or VIRTIO_CRYPTO_AKCIPHER_VERIFY then:
  - If VIRTIO_CRYPTO_F_AKCIPHER_STATELESS_MODE is negotiated, *op_flf* is struct virtio_crypto_-akcipher_data_flf_statless, and *op_vlf* is struct virtio_crypto_akcipher_data_vlf_stateless.
  - If VIRTIO_CRYPTO_F_AKCIPHER_STATELESS_MODE is NOT negotiated, *op_flf* is struct virtio_crypto_akcipher_data_flf if VIRTIO_CRYPTO_F_REVISION_1 is negotiated and struct virtio_-crypto_akcipher_data_flf is padded to 48 bytes if NOT negotiated, and *op_vlf* is struct virtio_-crypto_akcipher_data_vlf.
- If the the opcode (in *header*) is VIRTIO_CRYPTO_IPSEC_OUTBOUND or VIRTIO_CRYPTO_IPSEC_-INBOUND then:
  - *op_flf* is struct virtio_crypto_ipsec_data_flf and *op_vlf* is struct virtio_crypto_ipsec_data_vlf. Works only for session mode.

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;
};
```

5.9.9.4 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];
};
```

5.9.9.4.1 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.

5.9.9.4.2 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.

5.9.9.5 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.

5.9.9.5.1 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.

5.9.9.5.2 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.

5.9.9.6 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 ignored.

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 ignored.

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.

5.9.9.6.1 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.

5.9.9.6.2 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.

5.9.9.7 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];
};
```

5.9.9.7.1 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.

5.9.9.7.2 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.

5.9.9.8 AKCIPHER Service Operation

Session mode AKCIPHER requests are as follows:

```
struct virtio_crypto_akcipher_data_flf {
    /* length of source data */
    le32 src_data_len;
    /* length of dst data */
    le32 dst_data_len;
};

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

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

Each data request uses the virtio_crypto_akcipher_flf structure and the virtio_crypto_akcipher_data_vlf structure to store information used to run the AKCIPHER operations.

For encryption, decryption, and signing: *src_data* is the source data that will be processed, note that for signing operations, src_data stores the data to be signed, which usually is the digest of some data rather than the data itself. *src_data_len* is the length of source data. *dst_result* is the result data and *dst_data_len* is the length of it. Note that the length of the result is not always exactly equal to dst_data_len, the driver needs to check how many bytes the device has written and calculate the actual length of the result.

For verification: *src_data_len* refers to the length of the signature, and *dst_data_len* refers to the length of signed data, where the signed data is usually the digest of some data. *src_data* is spliced by the signature and the signed data, the src_data with the lower address stores the signature, and the higher address stores the signed data. *dst_data* is always empty for verification.

Different algorithms have different signature formats. For the RSA algorithm, the result is determined by the padding algorithm specified by *padding_algo* in structure virtio_crypto_rsa_session_para.

For the ECDSA algorithm, the signature is composed of the following ASN.1 structure (see RFC3279) and MUST be DER encoded (see rfc6025).

```
Ecdsa-Sig-Value ::= SEQUENCE {
    r INTEGER,
    s INTEGER
}
```

Stateless mode AKCIPHER service requests are as follows:

```
struct virtio_crypto_akcipher_data_flf_stateless {
    struct {
        /* See VIRTIO_CYRPTO_AKCIPHER* above */
        le32 algo;
        /* See VIRTIO_CRYPTO_AKCIPHER_KEY_TYPE_* above */
        le32 key_type;
        /* length of key */
        le32 key_len;

        /* algothrim specific parameters described above */
        union {
            struct virtio_crypto_rsa_session_para rsa;
            struct virtio_crypto_ecdsa_session_para ecdsa;
        } u;
    } sess_para;

    /* length of source data */
    le32 src_data_len;
    /* length of destination data */
    le32 dst_data_len;
};

struct virtio_crypto_akcipher_data_vlf_stateless {
    /* Device read only portion */
    u8 akcipher_key[key_len];

    /* Source data */
    u8 src_data[src_data_len];

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

In stateless mode, the format of key and signature, the meaning of src_data and dst_data, are all the same with session mode.

5.9.9.8.1 Driver Requirements: AKCIPHER 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_AKCIPHER_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_akcipher_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 one of VIRTIO_CRYPTO_-AKCIPHER_ENCRYPT, VIRTIO_CRYPTO_AKCIPHER_DESTROY_SESSION, VIRTIO_CRYPTO_-AKCIPHER_SIGN, and VIRTIO_CRYPTO_AKCIPHER_VERIFY.

5.9.9.8.2 Device Requirements: AKCIPHER Service Operation

- If the VIRTIO_CRYPTO_F_AKCIPHER_STATELESS_MODE feature bit is negotiated, the device MUST parse the virtio_crypto_akcipher_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 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_BADMSG if the verification result is incorrect.
  - VIRTIO_CRYPTO_INVSESS if the session ID invalid when in session mode.
  - VIRTIO_CRYPTO_KEY_REJECTED if the signature verification failed.
  - VIRTIO_CRYPTO_ERR if any failure not mentioned above occurs.

5.9.9.9 IPSEC Service Operation

A crypto device can support the processing of IPsec protocol operations. In addition to standard crypto processing, the IPsec protocol operations are also handled by the crypto device as a lookaside operation.

IPsec Inbound processing: The device performs decryption, authentication, integrity checking and remove additional headers, including tunnel header if in tunnel mode, as well as the ESP/AH header on the given packet(See IPSEC RFC). The resulting packet contains only the plain data.

The driver can request IPsec Inbound processing by

- Creating inbound SAs using the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA command.
- Setting the *opcode* in struct virtio_crypto_op_data_req to VIRTIO_CRYPTO_IPSEC_INBOUND.

IPsec Outbound processing: The device performs encryption, attach ICV, update/add IP header and add ESP/AH header/trailer. The resulting packet contains encrypted data along with the IPsec header and trailer.

The driver can request IPsec outbound processing by

- Creating outbound SAs using the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA command.
- Setting the *opcode* in struct virtio_crypto_op_data_req to VIRTIO_CRYPTO_IPSEC_OUTBOUND.

A crypto device can support number of IPsec SAs, allowing it to manage multiple secure connections simultaneously. See **??** for IPsec SA information.

The device and the driver indicate IPsec SA resource limits using the VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP capability specifying the limits on the number of IPsec outbound and inbound SA resource objects. The VIRTIO_CRYPTO_IPSEC_SA_CAP capability specifies the IPsec protocol capabilities supported by the device. The driver indicates the IPsec parameters by setting VIRTIO_CRYPTO_IPSEC_SA_CAP capability prior to adding any resource object.

The driver controls the IPsec SA resource object using administration commands described in 2.12.1.3.

5.9.9.9.1 Device and driver capabilities

5.9.9.9.1.1 VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP

The VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP capability indicates the IPsec SA resource limits. *cap_specific_data* is in the format *struct virtio_crypto_ipsec_resource_cap*.

```
struct virtio_crypto_ipsec_resource_cap {
        le32 inb_sa_limit;
        le32 outb_sa_limit;
};
```

inb_sa_limit, and *outb_sa_limit* denote the maximum number of IPsec security Associations (SAs) that can be utilized for IPsec inbound and outbound processing, respectively, which the device is capable of creating.

5.9.9.9.1.2 VIRTIO_CRYPTO_IPSEC_SA_CAP

The VIRTIO_CRYPTO_IPSEC_SA_CAP capability lists the supported IPsec modes along with the supported cryptographic, authentication algorithms and anti-replay window size for each IPsec mode. *cap_specific_data* is in the format *struct virtio_crypto_ipsec_sa_cap_data*.

```
struct virtio_crypto_ipsec_mode_cap {
        u8 mode;
        u8 reserved[3];
        le32 max_replay_win_sz;
        le32 options;
        le32 reserved1;
        le64 cipher_algo;
        le64 hmac_algo;
        le32 aead_algo;
        le32 max_cipher_key_len;
        le32 max_auth_key_len;
};

struct virtio_crypto_ipsec_sa_cap_data {
        u8 count;
        u8 reserved[7];
        struct virtio_crypto_ipsec_mode_cap cap_mode[];
};
```

count indicates number of valid entries in the *mode* array. *cap_mode[]* is an array of supported IPsec modes. Within each array entry:

mode specifies the IPsec mode, as defined in table 5.12. *max_replay_win_sz* specifies the maximum anti-replay window size the device supports. This field is applicable only for inbound operation.

options Each bit indicates the IPsec protocol options supported by the device, as defined in table 5.14.

cipher_algo CIPHER algorithms mask, see 5.9.4.1.

hmac_algo HMAC algorithms mask, see 5.9.4.3.

aead_algo AEAD algorithms mask, 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 authentication key supported by the device.

Table 5.12: IPsec Modes

Type	Name	Description
0x0	-	Reserved
0x1	VIRTIO_CRYPTO_IPSEC_MODE_ESP_-TUNNEL	IPsec ESP protocol in tunnel mode
0x2	VIRTIO_CRYPTO_IPSEC_MODE_ESP_-TRANSPORT	IPsec ESP protocol in transport mode
0x3	VIRTIO_CRYPTO_IPSEC_MODE_AH_TUN-NEL	IPsec AH protocol in tunnel mode
0x4	VIRTIO_CRYPTO_IPSEC_MODE_AH_-TRANSPORT	IPsec AH protocol in transport mode

See IPSEC for more information on tunnel and transport modes in ESP/AH IPsec processing.

Table 5.14: IPsec Options

Bit Number	Name	Description
0	VIRTIO_CRYPTO_IPSEC_ESN	Specifies whether extended sequence number is supported, as described in ESN
1	VIRTIO_CRYPTO_IPSEC_UDP_ENCAP	Specifies whether udp encapsulation is supported, as described in UDP Encapsulation, applicable only for ESP IPsec processing
2	VIRTIO_CRYPTO_IPSEC_COPY_DSCP	Specifies whether copy dscp is supported, as described in IPSEC
3	VIRTIO_CRYPTO_IPSEC_DEC_TTL	Specifies whether decrementing the time to live is supported, as described in IPSEC
4	VIRTIO_CRYPTO_IPSEC_COPY_DF	Specifies whether copy Don't Fragment bit is supported, as described in IPSEC
5	VIRTIO_CRYPTO_IPSEC_ECN	Specifies whether copy Explicit Congestion Notification is supported, as described in IPSEC
6	VIRTIO_CRYPTO_IPSEC_SA_LIFETIME	Specifies whether SA lifetime feature is supported, as described in IPSEC

5.9.9.9.2 Resource objects

5.9.9.9.2.1 VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA

A driver can have outbound SAs between 0 and *outb_sa_limit*, as specified by the capability VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP. For the IPsec outbound SA resource object *resource_obj_specific_data* is in the format *struct virtio_crypto_resource_obj_ipsec_sa_256b_key*.

```
struct in_addr {
        le32 s_addr;
};

struct in6_addr {
        u8 s6_u8[16];
};

struct virtio_crypto_ipsec_tunnel_param {
        /* Tunnel type: IPv4 or IPv6 */
        u8 type;
        u8 reserved[3];
        union {
                /* IPv4 tunnel header parameters */
                struct {
                        /* IPv4 source address */
                        struct in_addr src_ip;
                        /* IPv4 destination address */
                        struct in_addr dst_ip;
                        /* IPv4 Differentiated Services Code Point */
                        u8 dscp;
                        /* IPv4 Don't Fragment bit */
                        u8 df;
                        /* IPv4 Time To Live */
                        u8 ttl;
                        u8 reserved1;
                } ipv4;
                /* IPv6 tunnel header parameters */
                struct {
                        /* IPv6 source address */
                        struct in6_addr src_addr;
```

```
                        /* IPv6 destination address */
                        struct in6_addr dst_addr;
                        /* IPv6 flow label */
                        le32 flabel;
                        /* IPv6 hop limit */
                        u8 hlimit;
                        /* IPv6 Differentiated Services Code Point */
                        u8 dscp;
                        u8 reserved2[2];
                } ipv6;
        };
};

struct virtio_crypto_ipsec_lifetime {
        le64 packets_soft_limit;
        le64 bytes_soft_limit;
        le64 packets_hard_limit;
        le64 bytes_hard_limit;
};

struct virtio_crypto_resource_obj_ipsec_sa_256b_key {
        u8 mode;
        u8 direction;
        u8 reserved[2];
        le32 obj_id;
        le32 spi;
        le32 salt;
        le64 options;
        struct virtio_crypto_ipsec_tunnel_param param;
        le16 udp_sport;
        le16 udp_dport;
        le32 replay_win_sz;
        le64 cipher_algo;
        struct {
            le16 length;
            le16 reserved1;
            u8 data[32];
        } cipher_key;
        le64 auth_algo;
        struct {
            le16 length;
            le16 reserved2;
            u8 data[32];
        } auth_key;
        struct virtio_crypto_ipsec_lifetime life;
}
```

mode specifies the mode of the IPsec SA, see 5.12.

direction specifies IPsec SA direction. *obj_id* specifies the object id of the SA that can be used to retrieve driver-defined data associated with the IPsec SA. *spi* is the Security Parameter Index(SPI) used to uniquely identify the IPsec SA. *salt* is the 32 bit salt value used in the cryptographic operations.

options specifies the Options for configuring the IPsec SA, see 5.14.

param specifies the parameters for IPsec tunnel mode. *udp_sport* is the source port for UDP encapsulation. *udp_dport* is the destination port for UDP encapsulation. *replay_win_sz* is the anti-replay window size to enable sequence replay attack handling, replay checking is disabled if the window size is 0.

cipher_algo is the cipher algorithm identifier see 5.9.4.1 *cipher_key* specifies the cipher key and its length. *auth_algo* is the Authentication algorithm identifier *auth_key* specifies the authentication key data and its length. *life* configures soft and hard lifetime of an IPsec SA. The Lifetime of an IPsec SA specifies the maximum number of packets or bytes that can be processed. IPsec operations starts failing once any hard limit is reached. Soft limits generate a warning status when the SA is approaching its hard lifetime limits.

Table 5.16: IPsec Direction

Type	Name	Description
0x0	-	Reserved
0x1	VIRTIO_CRYPTO_IPSEC_DIR_OUTBOUND	IPsec direction outbound
0x2	VIRTIO_CRYPTO_IPSEC_DIR_INBOUND	IPsec direction inbound

5.9.9.9.2.2 VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA

A driver can have inbound SAs between 0 and *inb_sa_limit*, as specified by the capability VIRTIO_CRYPTO_-IPSEC_RESOURCE_CAP. For the IPsec inbound SA resource object *resource_obj_specific_data* is in the format *struct virtio_crypto_resource_obj_ipsec_sa*.

5.9.9.9.3 Data processing

Data requests for IPsec processing are as follows:

```
struct virtio_crypto_ipsec_data_flf {
        /* length of source data, full IP/IPsec packet */
        le32 src_data_len;
        /* length of dst data */
        le32 dst_data_len;
};

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

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

Each data request uses the virtio_crypto_ipsec_data_flf structure and the virtio_crypto_ipsec_data_vlf structure to store information used to run the IPSEC operations.

For IPsec encryption: *src_data* is the full IP packet that will be processed. *src_data_len* is the length of source data. *dst_result* is the result ESP encrypted packet and *dst_data_len* is the length of it. Please note, dst_data_len MUST include additional header and trailer lengths.

For IPsec decryption: *src_data* is the IPsec packet that will be processed. *src_data_len* is the length of source data. *dst_result* is the result plain IP packet and *dst_data_len* is the length of it.

5.9.9.9.4 Device Requirements: IPsec Service Operation

When the device supports IPsec operations,

- the device MUST set VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP, VIRTIO_CRYPTO_IPSEC_SA_-CAP capability in the *supported_caps* in the command VIRTIO_ADMIN_CMD_CAP_SUPPORT_QUERY.
- the device MUST support the administration commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_-CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_-OBJ_QUERY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY for the resource types VIRTIO_-CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA and VIRTIO_CRYPTO_RESOURCE_OBJ_-IPSEC_INBOUND_SA.

When any of the VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP or VIRTIO_CRYPTO_IPSEC_SA_CAP capability is disabled, the device MUST set *status* to VIRTIO_ADMIN_STATUS_Q_INVALID_OPCODE for the

commands VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_-MODIFY, VIRTIO_ADMIN_CMD_RESOURCE_OBJ_QUERY, and VIRTIO_ADMIN_CMD_RESOURCE_-OBJ_DESTROY for the resource types VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA and VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA.

The device MUST set *status* to VIRTIO_ADMIN_STATUS_EEXIST for the command VIRTIO_ADMIN_-CMD_RESOURCE_OBJ_CREATE when the resource *type* is VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_-OUTBOUND_SA or VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA, if the object already exists with the supplied *id*.

The device MUST fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with the *status* set to VIRTIO_ADMIN_STATUS_EINVAL, for the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_-SA object if,

- *id* is greater than or equal to *outb_sa_limit*.
- the supplied SA parameters, such as mode, options, cipher and authentication algorithms are not supported in the capabitlity VIRTIO_CRYPTO_IPSEC_SA_CAP.

The device MUST fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with the *status* set to VIRTIO_ADMIN_STATUS_EINVAL, for the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_-SA object if,

- *id* is greater than or equal to *inb_sa_limit*.
- the supplied SA parameters, such as mode, options, cipher and authentication algorithms are not supported in the capabitlity VIRTIO_CRYPTO_IPSEC_SA_CAP.

The device SHOULD maintain a table for subsequent lookups for inbound/outbound data processing with the corresponding SA based on the supplied *id*.

The device MUST allow recreating the resource objects using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE which was previously destroyed using the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_DESTROY respectively without undergoing a device reset.

The device MAY fail the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE with the *status* set to VIRTIO_ADMIN_STATUS_EINVAL for the VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_-SA or VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA commands if the resource object with the same *spi* already exists.

On device reset, the device MUST destroy all the resource objects which have been created.

The device MUST copy the result of IPsec operation in the *dst_data[]*. 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_BADMSG if the integrity check is failed for IPsec decryption.
- VIRTIO_CRYPTO_INVSESS if the session ID invalid.
- VIRTIO_CRYPTO_ERR if any failure not mentioned above occurs.
- VIRTIO_CRYPTO_IPSEC_SA_SOFT_EXPIRY if an IPsec SA reaches the SA soft expiry limit configured in *struct virtio_crypto_ipsec_lifetime*.

5.9.9.9.5 Driver Requirements: IPsec Service Operation

The driver MUST query the capabilities using VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY to discover the capability types the device offers.

The driver MUST get VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP and VIRTIO_CRYPTO_IPSEC_SA_-CAP if listed in VIRTIO_ADMIN_CMD_CAP_ID_LIST_QUERY command result, using VIRTION_ADMIN_-CMD_DEVICE_CAP_GET to discover the capabilities the device is able to offer. The driver MUST set VIRTIO_CRYPTO_IPSEC_RESOURCE_CAP and VIRTIO_CRYPTO_IPSEC_SA_CAP using VIRTIO_ADMIN_CMD_DEVICE_CAP_SET to indicate the device which capability the driver uses.

For the command VIRTIO_ADMIN_CMD_RESOURCE_OBJ_CREATE, when creating a resource VIRTIO_-CRYPTO_RESOURCE_OBJ_IPSEC_OUTBOUND_SA, the driver MUST set all the parameters in *struct virtio_crypto_resource_obj_ipsec_sa* with relevant values. And when create a resource object VIRTIO_-CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA, the driver MUST set all the parameters except *struct virtio_crypto_ipsec_tunnel_param*.

The driver MUST set *session_id* in struct virtio_crypto_op_header to a valid VIRTIO_CRYPTO_RESOURCE_-OBJ_IPSEC_OUTBOUND_SA or VIRTIO_CRYPTO_RESOURCE_OBJ_IPSEC_INBOUND_SA *id*. The driver MUST set the *opcode* field in struct virtio_crypto_op_header to one of VIRTIO_CRYPTO_IPSEC_OUT-BOUND and VIRTIO_CRYPTO_IPSEC_INBOUND.

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

VIRTIO_VSOCK_F_STREAM (0) stream socket type is supported.

VIRTIO_VSOCK_F_SEQPACKET (1) seqpacket socket type is supported.

VIRTIO_VSOCK_F_NO_IMPLIED_STREAM (2) stream socket type is not implied.

5.10.3.1 Driver Requirements: Feature bits

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

If no feature bit has been negotiated, the driver SHOULD act as if VIRTIO_VSOCK_F_STREAM has been negotiated.

If VIRTIO_VSOCK_F_SEQPACKET has been negotiated, but not VIRTIO_VSOCK_F_NO_IMPLIED_STREAM, the driver MAY act as if VIRTIO_VSOCK_F_STREAM has also been negotiated.

5.10.3.2 Device Requirements: Feature bits

The device SHOULD offer the VIRTIO_VSOCK_F_NO_IMPLIED_STREAM feature.

If no feature bit has been negotiated, the device SHOULD act as if VIRTIO_VSOCK_F_STREAM has been negotiated.

If VIRTIO_VSOCK_F_SEQPACKET has been negotiated, but not VIRTIO_VSOCK_F_NO_IMPLIED_STREAM, the device MAY act as if VIRTIO_VSOCK_F_STREAM has also been negotiated.

5.10.4 Device configuration layout

Socket device configuration uses the following layout structure:

```
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:

```
#define VIRTIO_VSOCK_OP_INVALID        0
/* Connect operations */
#define VIRTIO_VSOCK_OP_REQUEST        1
#define VIRTIO_VSOCK_OP_RESPONSE       2
#define VIRTIO_VSOCK_OP_RST            3
#define VIRTIO_VSOCK_OP_SHUTDOWN       4
/* To send payload */
#define VIRTIO_VSOCK_OP_RW             5
/* Tell the peer our credit info */
#define VIRTIO_VSOCK_OP_CREDIT_UPDATE  6
/* Request the peer to send the credit info to us */
#define VIRTIO_VSOCK_OP_CREDIT_REQUEST 7
```

len is the size of the payload, in bytes. However, the driver may provide buffer(s) for the payload that have a total size longer than *len*, in which case only the first *len* bytes will be used for the actual data.

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 stream and seqpacket sockets are supported. *type* is 1 (VIRTIO_VSOCK_TYPE_STREAM) for stream socket types, and 2 (VIRTIO_VSOCK_TYPE_SEQPACKET) for seqpacket socket types.

```
#define VIRTIO_VSOCK_TYPE_STREAM    1
#define VIRTIO_VSOCK_TYPE_SEQPACKET 2
```

Stream sockets provide in-order, guaranteed, connection-oriented delivery without message boundaries. Seqpacket sockets provide in-order, guaranteed, connection-oriented delivery with message and record 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 enqueues outgoing packets to 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 VIRTIO_VSOCK_SHUTDOWN_F_RECEIVE (bit 0) set indicates that the peer will not receive any more data and bit VIRTIO_VSOCK_SHUTDOWN_F_SEND (bit 1) set 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.

```
#define VIRTIO_VSOCK_SHUTDOWN_F_RECEIVE 0
#define VIRTIO_VSOCK_SHUTDOWN_F_SEND    1
```

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 Seqpacket Sockets

5.10.6.6.1 Message and record boundaries

Two types of boundaries are supported: message and record boundaries.

A message contains data sent in a single operation. A single message can be split into multiple RW packets. To provide message boundaries, last RW packet of each message has VIRTIO_VSOCK_SEQ_EOM bit (bit 0) set in the *flags* of packet's header.

Record is any number of subsequent messages, where last message is sent with POSIX MSG_EOR flag set. Record boundary means that receiver gets MSG_EOR flag set in the corresponding message where sender set it. To provide record boundaries, last RW packet of each record has VIRTIO_VSOCK_SEQ_EOR bit (bit 1) set in the *flags* of packet's header.

```
#define VIRTIO_VSOCK_SEQ_EOM (1 << 0)
#define VIRTIO_VSOCK_SEQ_EOR (1 << 1)
```

5.10.6.7 Device Events

Certain events are communicated by the device to the driver using the event virtqueue.

The event buffer is as follows:

```
#define 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*.

5.10.6.7.1 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_TRANSPORT_RESET event is received.

5.11 File System Device

The virtio file system device provides file system access. The device either directly manages a file system or it acts as a gateway to a remote file system. The details of how the device implementation accesses files are hidden by the device interface, allowing for a range of use cases.

Unlike block-level storage devices such as virtio block and SCSI, the virtio file system device provides file-level access to data. The device interface is based on the Linux Filesystem in Userspace (FUSE) protocol. This consists of requests for file system traversal and access the files and directories within it. The protocol details are defined by FUSE.

The device acts as the FUSE file system daemon and the driver acts as the FUSE client mounting the file system. The virtio file system device provides the mechanism for transporting FUSE requests, much like /dev/fuse in a traditional FUSE application.

This section relies on definitions from FUSE.

5.11.1 Device ID

26

5.11.2 Virtqueues

0 hiprio

1 notification queue

2...n request queues

The notification queue only exists if VIRTIO_FS_F_NOTIFICATION is set.

5.11.3 Feature bits

VIRTIO_FS_F_NOTIFICATION (0) Device has support for FUSE notify messages. The notification queue is virtqueue 1.

5.11.4 Device configuration layout

```
struct virtio_fs_config {
        char tag[36];
        le32 num_request_queues;
        le32 notify_buf_size;
};
```

The *tag* and *num_request_queues* fields are always available. The *notify_buf_size* field is only available when VIRTIO_FS_F_NOTIFICATION is set.

tag is the name associated with this file system. The tag is encoded in UTF-8 and padded with NUL bytes if shorter than the available space. This field is not NUL-terminated if the encoded bytes take up the entire field.

num_request_queues is the total number of request virtqueues exposed by the device. Each virtqueue offers identical functionality and there are no ordering guarantees between requests made available on different queues. Use of multiple queues is intended to increase performance.

notify_buf_size is the minimum number of bytes required for each buffer in the notification queue.

5.11.4.1 Driver Requirements: Device configuration layout

The driver MUST NOT write to device configuration fields.

The driver MAY use from one up to *num_request_queues* request virtqueues.

5.11.4.2 Device Requirements: Device configuration layout

The device MUST set *num_request_queues* to 1 or greater.

The device MUST set *notify_buf_size* to be large enough to hold any of the FUSE notify messages that this device emits.

5.11.5 Device Initialization

On initialization the driver first discovers the device's virtqueues.

The driver populates the notification queue with buffers for receiving FUSE notify messages if VIRTIO_FS_F_NOTIFICATION is set.

The FUSE session is started by sending a FUSE_INIT request as defined by the FUSE protocol on one request virtqueue. All virtqueues provide access to the same FUSE session and therefore only one FUSE_INIT request is required regardless of the number of available virtqueues.

5.11.6 Device Operation

Device operation consists of operating the virtqueues to facilitate file system access.

The FUSE request types are as follows:

- Normal requests are made available by the driver on request queues and are used by the device.
- High priority requests (FUSE_INTERRUPT, FUSE_FORGET, and FUSE_BATCH_FORGET) are made available by the driver on the hiprio queue so the device is able to process them even if the request queues are full.

FUSE notify messages are received on the notification queue if VIRTIO_FS_F_NOTIFICATION is set.

5.11.6.1 Device Operation: Request Queues

The driver enqueues normal requests on an arbitrary request queue. High priority requests are not placed on request queues. The device processes requests in any order. The driver is responsible for ensuring that ordering constraints are met by making available a dependent request only after its prerequisite request has been used.

Requests have the following format with endianness chosen by the driver in the FUSE_INIT request used to initiate the session as detailed below:

```
struct virtio_fs_req {
        // Device-readable part
        struct fuse_in_header in;
        u8 datain[];

        // Device-writable part
        struct fuse_out_header out;
        u8 dataout[];
};
```

Note that the words ”in” and ”out” follow the FUSE meaning and do not indicate the direction of data transfer under VIRTIO. ”In” means input to a request and ”out” means output from processing a request.

in is the common header for all types of FUSE requests.

datain consists of request-specific data, if any. This is identical to the data read from the /dev/fuse device by a FUSE daemon.

out is the completion header common to all types of FUSE requests.

dataout consists of request-specific data, if any. This is identical to the data written to the /dev/fuse device by a FUSE daemon.

For example, the full layout of a FUSE_READ request is as follows:

```
struct virtio_fs_read_req {
        // Device-readable part
        struct fuse_in_header in;
        union {
                struct fuse_read_in readin;
                u8 datain[sizeof(struct fuse_read_in)];
        };

        // Device-writable part
        struct fuse_out_header out;
        u8 dataout[out.len - sizeof(struct fuse_out_header)];
};
```

The FUSE protocol documented in FUSE specifies the set of request types and their contents.

The endianness of the FUSE protocol session is detectable by inspecting the uint32_t *in.opcode* field of the FUSE_INIT request sent by the driver to the device. This allows the device to determine whether the session is little-endian or big-endian. The next FUSE_INIT message terminates the current session and starts a new session with the possibility of changing endianness.

5.11.6.2 Device Operation: High Priority Queue

The hiprio queue follows the same request format as the request queues. This queue only contains FUSE_-INTERRUPT, FUSE_FORGET, and FUSE_BATCH_FORGET requests.

Interrupt and forget requests have a higher priority than normal requests. The separate hiprio queue is used for these requests to ensure they can be delivered even when all request queues are full.

5.11.6.2.1 Device Requirements: Device Operation: High Priority Queue

The device MUST NOT pause processing of the hiprio queue due to activity on a normal request queue.

The device MAY process request queues concurrently with the hiprio queue.

5.11.6.2.2 Driver Requirements: Device Operation: High Priority Queue

The driver MUST submit FUSE_INTERRUPT, FUSE_FORGET, and FUSE_BATCH_FORGET requests solely on the hiprio queue.

The driver MUST not submit normal requests on the hiprio queue.

The driver MUST anticipate that request queues are processed concurrently with the hiprio queue.

5.11.6.3 Device Operation: Notification Queue

The notification queue is populated with buffers by the driver and these buffers are used by the device to emit FUSE notify messages. Notification queue buffer layout is as follows:

```
struct virtio_fs_notify {
        // Device-writable part
        struct fuse_out_header out_hdr;
        char outarg[notify_buf_size - sizeof(struct fuse_out_header)];
};
```

outarg contains the FUSE notify message payload that depends on the type of notification being emitted.

If the driver provides notification queue buffers at a slower rate than the device emits FUSE notify messages then the virtqueue will eventually become empty. The behavior in response to an empty virtqueue depends on the FUSE notify message type. The following FUSE notify message types are supported:

- FUSE_NOTIFY_LOCK messages are delivered when buffers become available again. The device has resources for a certain number of lock requests. If the device runs out of resources new lock requests fail with ENOLCK.

5.11.6.3.1 Driver Requirements: Device Operation: Notification Queue

The driver MUST provide buffers of at least *notify_buf_size* bytes.

The driver SHOULD replenish notification queue buffers sufficiently quickly so that there is always at least one available buffer.

5.11.6.4 Device Operation: DAX Window

FUSE_READ and FUSE_WRITE requests transfer file contents between the driver-provided buffer and the device. In cases where data transfer is undesirable, the device can map file contents into the DAX window shared memory region. The driver then accesses file contents directly in device-owned memory without a data transfer.

The DAX Window is an alternative mechanism for accessing file contents. FUSE_READ/FUSE_WRITE requests and DAX Window accesses are possible at the same time. Providing the DAX Window is optional for devices. Using the DAX Window is optional for drivers.

Shared memory region ID 0 is called the DAX window. Drivers map this shared memory region with writeback caching as if it were regular RAM. The contents of the DAX window are undefined unless a mapping exists for that range.

The driver maps a file range into the DAX window using the FUSE_SETUPMAPPING request. Alignment constraints for FUSE_SETUPMAPPING and FUSE_REMOVEMAPPING requests are communicated during FUSE_INIT negotiation.

When a FUSE_SETUPMAPPING request perfectly overlaps a previous mapping, the previous mapping is replaced. When a mapping partially overlaps a previous mapping, the previous mapping is split into one or two smaller mappings. When a mapping is partially unmapped it is also split into one or two smaller mappings.

Establishing new mappings or splitting existing mappings consumes resources. If the device runs out of resources the FUSE_SETUPMAPPING request fails until resources are available again following FUSE_REMOVEMAPPING.

After FUSE_SETUPMAPPING has completed successfully the file range is accessible from the DAX window at the offset provided by the driver in the request. A mapping is removed using the FUSE_REMOVEMAPPING request.

Data is only guaranteed to be persistent when a FUSE_FSYNC request is used by the device after having been made available by the driver following the write.

5.11.6.4.1 Device Requirements: Device Operation: DAX Window

The device MAY provide the DAX Window to memory-mapped access to file contents. If present, the DAX Window MUST be shared memory region ID 0.

The device MUST support FUSE_READ and FUSE_WRITE requests regardless of whether the DAX Window is being used or not.

The device MUST allow mappings that completely or partially overlap existing mappings within the DAX window.

The device MUST reject mappings that would go beyond the end of the DAX window.

5.11.6.4.2 Driver Requirements: Device Operation: DAX Window

The driver SHOULD be prepared to find shared memory region ID 0 absent and fall back to FUSE_READ and FUSE_WRITE requests.

The driver MAY use both FUSE_READ/FUSE_WRITE requests and the DAX Window to access file contents.

The driver MUST NOT access DAX window areas that have not been mapped.

5.11.6.5 Security Considerations

The device provides access to a file system containing files owned by one or more POSIX user ids and group ids. The device has no secure way of differentiating between users originating requests via the driver. Therefore the device accepts the POSIX user ids and group ids provided by the driver and security is enforced by the driver rather than the device. It is nevertheless possible for devices to implement POSIX user id and group id mapping or whitelisting to control the ownership and access available to the driver.

File systems containing special files including device nodes and setuid executable files pose a security concern. These properties are defined by the file type and mode, which are set by the driver when creating new files or by changes at a later time. These special files present a security risk when the file system is shared with another machine. A setuid executable or a device node placed by a malicious machine make it possible for unprivileged users on other machines to elevate their privileges through the shared file system. This issue can be solved on some operating systems using mount options that ignore special files. It is also possible for devices to implement restrictions on special files by refusing their creation.

When the device provides shared access to a file system between multiple machines, symlink race conditions, exhausting file system capacity, and overwriting or deleting files used by others are factors to consider. These issues have a long history in multi-user operating systems and also apply to virtio-fs. They are typically managed at the file system administration level by providing shared access only to mutually trusted users.

Multiple machines sharing access to a file system are susceptible to timing side-channel attacks. By measuring the latency of accesses to file contents or file system metadata it is possible to infer whether other machines also accessed the same information. Short latencies indicate that the information was cached due to a previous access. This can reveal sensitive information, such as whether certain code paths were taken. The DAX Window provides direct access to file contents and is therefore a likely target of such attacks. These attacks are also possible with traditional FUSE requests. The safest approach is to avoid sharing file systems between untrusted machines.

5.11.6.6 Live migration considerations

When a driver is migrated to a new device it is necessary to consider the FUSE session and its state. The continuity of FUSE inode numbers (also known as nodeids) and fh values is necessary so the driver can continue operation without disruption.

It is possible to maintain the FUSE session across live migration either by transferring the state or by redirecting requests from the new device to the old device where the state resides. The details of how to achieve this are implementation-dependent and are not visible at the device interface level.

Maintaining version and feature information negotiated by FUSE_INIT is necessary so that no FUSE protocol feature changes are visible to the driver across live migration. The FUSE_INIT information forms part of the FUSE session state that needs to be transferred during live migration.

5.12 RPMB Device

virtio-rpmb is a virtio based RPMB (Replay Protected Memory Block) device. It is used as a tamper-resistant and anti-replay storage. The device is driven via requests including read, write, get write counter and program key, which are submitted via a request queue. This section relies on definitions from paragraph 6.6.22 of eMMC.

5.12.1 Device ID

28

5.12.2 Virtqueues

0 requestq

5.12.3 Feature bits

None.

5.12.4 Device configuration layout

All fields of this configuration are always available and read-only for the driver.

```
struct virtio_rpmb_config {
        u8 capacity;
        u8 max_wr_cnt;
        u8 max_rd_cnt;
}
```

capacity is the capacity of the device (expressed in 128KB units). The values MUST range between 0x00 and 0x80 inclusive.

max_wr_cnt and max_rd_cnt are the maximum numbers of RPMB block count (256B) that can be performed to device in one request. 0 implies no limitation.

5.12.5 Device Requirements: Device Initialization

1. The virtqueue is initialized.
2. The device capacity MUST be initialized to a multiple of 128Kbytes and up to 16Mbytes.

5.12.6 Device Operation

The operation of a virtio RPMB device is driven by the requests placed on the virtqueue. The type of request can be program key (VIRTIO_RPMB_REQ_PROGRAM_KEY), get write counter (VIRTIO_RPMB_REQ_GET_WRITE_COUNTER), write (VIRTIO_RPMB_REQ_DATA_WRITE), and read (VIRTIO_RPMB_REQ_DATA_READ). A program key or write request can also combine with a result read (VIRTIO_RPMB_REQ_RESULT_READ) for a returned result.

```
/* RPMB Request Types */
#define VIRTIO_RPMB_REQ_PROGRAM_KEY        0x0001
#define VIRTIO_RPMB_REQ_GET_WRITE_COUNTER  0x0002
#define VIRTIO_RPMB_REQ_DATA_WRITE         0x0003
#define VIRTIO_RPMB_REQ_DATA_READ          0x0004
#define VIRTIO_RPMB_REQ_RESULT_READ        0x0005

/* RPMB Response Types */
#define VIRTIO_RPMB_RESP_PROGRAM_KEY       0x0100
#define VIRTIO_RPMB_RESP_GET_COUNTER       0x0200
#define VIRTIO_RPMB_RESP_DATA_WRITE        0x0300
#define VIRTIO_RPMB_RESP_DATA_READ         0x0400
```

VIRTIO_RPMB_REQ_PROGRAM_KEY requests for authentication key programming. If VIRTIO_RPMB_REQ_RESULT_READ is requested, the device returns the RPMB frame with the response (VIRTIO_RPMB_RESP_PROGRAM_KEY), the calculated MAC and the result.

VIRTIO_RPMB_REQ_GET_WRITE_COUNTER requests for reading the write counter. The device returns the RPMB frame with the response (VIRTIO_RPMB_RESP_GET_COUNTER), the writer counter, a copy of the nonce received in the request, the calculated MAC and the result.

VIRTIO_RPMB_REQ_DATA_WRITE requests for authenticated data write. If VIRTIO_RPMB_REQ_RESULT_READ is requested, the device returns the RPMB data frame with the response (VIRTIO_RPMB_RESP_DATA_WRITE), the incremented counter value, the data address, the calculated MAC and the result.

VIRTIO_RPMB_REQ_DATA_READ requests for authenticated data read. The device returns the RPMB frame with the response (VIRTIO_RPMB_RESP_DATA_READ), the block count, a copy of the nonce received in the request, the address, the data, the calculated MAC and the result.

VIRTIO_RPMB_REQ_RESULT_READ requests for a returned result. It is used following with VIRTIO_RPMB_REQ_PROGRAM_KEY or VIRTIO_RPMB_REQ_DATA_WRITE request types for a returned result in one or multiple RPMB frames. If it's not requested, the device will not return result frame to the driver.

5.12.6.1 Device Operation: Request Queue

The request information is delivered in RPMB frame. The frame is in size of 512B.

```
struct virtio_rpmb_frame {
        u8 stuff[196];
        u8 key_mac[32];
        u8 data[256];
        u8 nonce[16];
        be32 write_counter;
        be16 address;
        be16 block_count;
        be16 result;
```

```
        be16 req_resp;
};

/* RPMB Operation Results */
#define VIRTIO_RPMB_RES_OK                      0x0000
#define VIRTIO_RPMB_RES_GENERAL_FAILURE         0x0001
#define VIRTIO_RPMB_RES_AUTH_FAILURE            0x0002
#define VIRTIO_RPMB_RES_COUNT_FAILURE           0x0003
#define VIRTIO_RPMB_RES_ADDR_FAILURE            0x0004
#define VIRTIO_RPMB_RES_WRITE_FAILURE           0x0005
#define VIRTIO_RPMB_RES_READ_FAILURE            0x0006
#define VIRTIO_RPMB_RES_NO_AUTH_KEY             0x0007
#define VIRTIO_RPMB_RES_WRITE_COUNTER_EXPIRED   0x0080
```

stuff Padding for the frame.

key_mac is the authentication key or the message authentication code (MAC) depending on the request/response type. If the request is VIRTIO_RPMB_REQ_PROGRAM_KEY, it's used as an authentication key. Otherwise, it's used as MAC. The MAC is calculated using HMAC SHA-256. It takes as input a key and a message. The key used for the MAC calculation is always the 256-bit RPMB authentication key. The message used as input to the MAC calculation is the concatenation of the fields in the RPMB frames excluding stuff bytes and the MAC itself.

data is used to be written or read via authenticated read/write access. It's fixed 256B.

nonce is a random number generated by the user for the read or get write counter requests and copied to the response by the device. It's used for anti-replay protection.

writer_counter is the counter value for the total amount of the successful authenticated data write requests.

address is the address of the data to be written to or read from the RPMB virtio device. It is the number of the accessed half sector (256B).

block_count is the number of blocks (256B) requested to be read/written. It's limited by *max_wr_cnt* or *max_rd_cnt*. For RPMB read request, one virtio buffer including request command and the subsequent [*block_count*] virtio buffers for response data are placed in the queue. For RPMB write request, [*block_count*] virtio buffers including request command and data are placed in the queue.

result includes information about the status of access made to the device. It is written by the device.

req_resp is the type of request or response, to/from the device.

5.12.6.1.1 Device Requirements: Device Operation: Program Key

If VIRTIO_RPMB_REQ_RESULT_READ is requested, the device SHOULD return the RPMB frame with the response, the calculated MAC and the result:

1. If the *block_count* is not set to 1 then VIRTIO_RPMB_RES_GENERAL_FAILURE SHOULD be responded as *result*.
2. If the programming of authentication key fails, then VIRTIO_RPMB_RES_WRITE_FAILURE SHOULD be responded as *result*.
3. If some other error occurs then returned result VIRTIO_RPMB_RES_GENERAL_FAILURE SHOULD be responded as *result*.
4. The *req_resp* value VIRTIO_RPMB_RESP_PROGRAM_KEY SHOULD be responded.

5.12.6.1.2 Device Requirements: Device Operation: Get Write Counter

If the authentication key is not yet programmed then VIRTIO_RPMB_RES_NO_AUTH_KEY SHOULD be returned in *result*.

If block count has not been set to 1 then VIRTIO_RPMB_RES_GENERAL_FAILURE SHOULD be responded as *result*.

The *req_resp* value VIRTIO_RPMB_RESP_GET_COUNTER SHOULD be responded.

5.12.6.1.3 Device Requirements: Device Operation: Data Write

If VIRTIO_RPMB_REQ_RESULT_READ is requested, the device SHOULD return the RPMB data frame with the response VIRTIO_RPMB_RESP_DATA_WRITE, the incremented counter value, the data address, the calculated MAC and the result:

1. If the authentication key is not yet programmed, then VIRTIO_RPMB_RES_NO_AUTH_KEY SHOULD be returned in *result*.
2. If block count is zero or greater than *max_wr_cnt* then VIRTIO_RPMB_RES_GENERAL_FAILURE SHOULD be responded.
3. The device MUST check whether the write counter has expired. If the write counter is expired then the *result* SHOULD be set to VIRTIO_RPMB_RES_WRITE_COUNTER_EXPIRED.
4. If there is an error in the address (out of range) then the *result* SHOULD be set to VIRTIO_RPMB_-RES_ADDR_FAILURE.
5. The device MUST calculate the MAC taking authentication key and frame as input, and compare this with the MAC in the request. If the two MAC's are different then VIRTIO_RPMB_RES_AUTH_FAILURE SHOULD be returned in *result*.
6. If the writer counter in the request is different from the one maintained by the device then VIRTIO_-RPMB_RES_COUNT_FAILURE SHOULD be returned in *result*.
7. If the MAC and write counter comparisons are matched then the write request is considered to be authenticated. The data from the request SHOULD be written to the address indicated in the request and the write counter SHOULD be incremented by 1.
8. If the write fails then the *result* SHOULD be VIRTIO_RPMB_RES_WRITE_FAILURE.
9. If some other error occurs during the writing procedure then the *result* SHOULD be VIRTIO_RPMB_-RES_GENERAL_FAILURE.
10. The *req_resp* value VIRTIO_RPMB_RESP_DATA_WRITE SHOULD be responded.

5.12.6.1.4 Device Requirements: Device Operation: Data Read

1. If the authentication key is not yet programmed then VIRTIO_RPMB_RES_NO_AUTH_KEY SHOULD be returned in *result*.
2. If block count is zero or greater than *max_rd_cnt* then VIRTIO_RPMB_RES_GENERAL_FAILURE SHOULD be responded as *result*.
3. If there is an error in the address (out of range) then the *result* SHOULD be set to VIRTIO_RPMB_-RES_ADDR_FAILURE.
4. If data fetch from addressed location inside the device fails then the *result* SHOULD be VIRTIO_-RPMB_RES_READ_FAILURE.
5. If some other error occurs during the read procedure then the *result* SHOULD be VIRTIO_RPMB_-RES_GENERAL_FAILURE.
6. The device SHOULD respond with *block_count* frames containing the data and *req_resp* value set to VIRTIO_RPMB_RESP_DATA_READ.

5.12.6.1.5 Device Requirements: Device Operation: Result Read

If the *block_count* has not been set to 1 of VIRTIO_RPMB_REQ_RESULT_READ request then VIRTIO_-RPMB_RES_GENERAL_FAILURE SHOULD be responded as *result*.

5.12.6.2 Driver Requirements: Device Operation

The RPMB frames MUST not be packed by the driver. The driver MUST configure, initialize and format virtqueue for the RPMB requests received from its caller then send it to the device.

5.12.6.3 Device Requirements: Device Operation

The virtio-rpmb device could be backed in a number of ways. It SHOULD keep consistent behaviors with hardware as described in paragraph 6.6.22 of eMMC. Some elements are maintained by the device:

1. The device MUST maintain a one-time programmable authentication key. It cannot be overwritten, erased or read. The key is used to authenticate the accesses when MAC is calculated. This key MUST be kept regardless of device reset or reboot.
2. The device MUST maintain a read-only monotonic write counter. It MUST be initialized to zero and added by one automatically along with successful write operation. The value cannot be reset. After the counter has reached its maximum value 0xFFFF FFFF, it will not be incremented anymore. This counter MUST be kept regardless of device reset or reboot.
3. The device MUST maintain the data for read/write via authenticated access.

5.13 IOMMU device

The virtio-iommu device manages Direct Memory Access (DMA) from one or more endpoints. It may act both as a proxy for physical IOMMUs managing devices assigned to the guest, and as virtual IOMMU managing emulated and paravirtualized devices.

The driver first discovers endpoints managed by the virtio-iommu device using platform specific mechanisms. It then sends requests to create virtual address spaces and virtual-to-physical mappings for these endpoints. In its simplest form, the virtio-iommu supports four request types:

1. Create a domain and attach an endpoint to it.
   ```
   attach(endpoint = 0x8, domain = 1)
   ```
2. Create a mapping between a range of guest-virtual and guest-physical address.
   ```
   map(domain = 1, virt_start = 0x1000, virt_end = 0x1fff, phys = 0xa000, flags = READ)
   ```

   Endpoint 0x8, for example a hardware PCI endpoint with BDF 00:01.0, can now read at addresses 0x1000-0x1fff. These accesses are translated into system-physical addresses by the IOMMU.
3. Remove the mapping.
   ```
   unmap(domain = 1, virt_start = 0x1000, virt_end = 0x1fff)
   ```

   Any access to addresses 0x1000-0x1fff by endpoint 0x8 would now be rejected.
4. Detach the device and remove the domain.
   ```
   detach(endpoint = 0x8, domain = 1)
   ```

5.13.1 Device ID

23

5.13.2 Virtqueues

0 requestq

1 eventq

5.13.3 Feature bits

VIRTIO_IOMMU_F_INPUT_RANGE (0) Available range of virtual addresses is described in *input_range*.

VIRTIO_IOMMU_F_DOMAIN_RANGE (1) The number of domains supported is described in *domain_range*.

VIRTIO_IOMMU_F_MAP_UNMAP (2) Map and unmap requests are available.[12]

VIRTIO_IOMMU_F_BYPASS (3) Endpoints that are not attached to a domain are in bypass mode.

VIRTIO_IOMMU_F_PROBE (4) The PROBE request is available.

VIRTIO_IOMMU_F_MMIO (5) The VIRTIO_IOMMU_MAP_F_MMIO flag is available.

VIRTIO_IOMMU_F_BYPASS_CONFIG (6) Field *bypass* of struct virtio_iommu_config determines whether endpoints that are not attached to a domain are in bypass mode. Flag VIRTIO_IOMMU_ATTACH_F_BYPASS determines whether endpoints that are attached to a domain are in bypass mode.

5.13.3.1 Driver Requirements: Feature bits

The driver SHOULD accept any of the VIRTIO_IOMMU_F_INPUT_RANGE, VIRTIO_IOMMU_F_DOMAIN_RANGE and VIRTIO_IOMMU_F_PROBE feature bits if offered by the device.

5.13.3.2 Device Requirements: Feature bits

The device SHOULD offer feature bit VIRTIO_IOMMU_F_MAP_UNMAP.

The VIRTIO_IOMMU_F_BYPASS_CONFIG feature supersedes VIRTIO_IOMMU_F_BYPASS. New devices SHOULD NOT offer VIRTIO_IOMMU_F_BYPASS. Devices SHOULD NOT offer both VIRTIO_IOMMU_F_BYPASS and VIRTIO_IOMMU_F_BYPASS_CONFIG.

5.13.4 Device configuration layout

The *page_size_mask* field is always present. Availability of the others all depend on feature bits described in 5.13.3.

```
struct virtio_iommu_config {
  le64 page_size_mask;
  struct virtio_iommu_range_64 {
    le64 start;
    le64 end;
  } input_range;
  struct virtio_iommu_range_32 {
    le32 start;
    le32 end;
  } domain_range;
  le32 probe_size;
  u8 bypass;
  u8 reserved[3];
};
```

5.13.4.1 Driver Requirements: Device configuration layout

When the VIRTIO_IOMMU_F_BYPASS_CONFIG feature is negotiated, the driver MAY write to *bypass*. The driver MUST NOT write to any other device configuration field.

The driver MUST NOT write a value different than 0 or 1 to *bypass*. The driver SHOULD ignore bits 1-7 of *bypass*.

5.13.4.2 Device Requirements: Device configuration layout

The device MUST set at least one bit in *page_size_mask*, describing the page granularity. The device MAY set more than one bit in *page_size_mask*.

If the device offers the VIRTIO_IOMMU_F_BYPASS_CONFIG feature, it MAY initialize the *bypass* field to 1. Field *bypass* SHOULD NOT change on device reset, but SHOULD be restored to its initial value on system reset.

12Future extensions may add different modes of operations. At the moment, only VIRTIO_IOMMU_F_MAP_UNMAP is supported.

The device MUST NOT present a value different than 0 or 1 in *bypass*.

5.13.5 Device initialization

When the device is reset, endpoints are not attached to any domain.

Future devices might support more modes of operation besides MAP/UNMAP. Drivers verify that devices set VIRTIO_IOMMU_F_MAP_UNMAP and fail gracefully if they don't.

5.13.5.1 Driver Requirements: Device Initialization

The driver MUST NOT negotiate VIRTIO_IOMMU_F_MAP_UNMAP if it is incapable of sending VIRTIO_-IOMMU_T_MAP and VIRTIO_IOMMU_T_UNMAP requests.

If the VIRTIO_IOMMU_F_PROBE feature is negotiated, the driver SHOULD send a VIRTIO_IOMMU_T_-PROBE request for each endpoint before attaching the endpoint to a domain.

5.13.6 Device operations

Driver send requests on the request virtqueue, notifies the device and waits for the device to return the request with a status in the used ring. All requests are split in two parts: one device-readable, one device-writable.

```
struct virtio_iommu_req_head {
  u8   type;
  u8   reserved[3];
};

struct virtio_iommu_req_tail {
  u8   status;
  u8   reserved[3];
};
```

Type may be one of:

```
#define VIRTIO_IOMMU_T_ATTACH     1
#define VIRTIO_IOMMU_T_DETACH     2
#define VIRTIO_IOMMU_T_MAP        3
#define VIRTIO_IOMMU_T_UNMAP      4
#define VIRTIO_IOMMU_T_PROBE      5
```

A few general-purpose status codes are defined here.

```
/* All good! Carry on. */
#define VIRTIO_IOMMU_S_OK         0
/* Virtio communication error */
#define VIRTIO_IOMMU_S_IOERR      1
/* Unsupported request */
#define VIRTIO_IOMMU_S_UNSUPP     2
/* Internal device error */
#define VIRTIO_IOMMU_S_DEVERR     3
/* Invalid parameters */
#define VIRTIO_IOMMU_S_INVAL      4
/* Out-of-range parameters */
#define VIRTIO_IOMMU_S_RANGE      5
/* Entry not found */
#define VIRTIO_IOMMU_S_NOENT      6
/* Bad address */
#define VIRTIO_IOMMU_S_FAULT      7
/* Insufficient resources */
#define VIRTIO_IOMMU_S_NOMEM      8
```

When the device fails to parse a request, for instance if a request is too small for its type and the device cannot find the tail, then it is unable to set *status*. In that case, it returns the buffers without writing to them.

Range limits of some request fields are described in the device configuration:

- *page_size_mask* contains the bitmask of all page sizes that can be mapped. The least significant bit set defines the page granularity of IOMMU mappings.

  The smallest page granularity supported by the IOMMU is one byte. It is legal for the driver to map one byte at a time if bit 0 of *page_size_mask* is set.

  Other bits in *page_size_mask* are hints and describe larger page sizes that the IOMMU device handles efficiently. For example, when the device stores mappings using a page table tree, it may be able to describe large mappings using a few leaf entries in intermediate tables, rather than using lots of entries in the last level of the tree. Creating mappings aligned on large page sizes can improve performance since they require fewer page table and TLB entries.

- If the VIRTIO_IOMMU_F_DOMAIN_RANGE feature is offered, *domain_range* describes the values supported in a *domain* field. If the feature is not offered, any *domain* value is valid.

- If the VIRTIO_IOMMU_F_INPUT_RANGE feature is offered, *input_range* contains the virtual address range that the IOMMU is able to translate. Any mapping request to virtual addresses outside of this range fails.

  If the feature is not offered, virtual mappings span over the whole 64-bit address space (`start = 0, end = 0xffffffff ffffffff`)

An endpoint is in bypass mode if:

- the VIRTIO_IOMMU_F_BYPASS_CONFIG feature is offered and:
  - config field *bypass* is 1 and the endpoint is not attached to a domain. This applies even if the driver does not accept the VIRTIO_IOMMU_F_BYPASS_CONFIG feature and the device initializes field *bypass* to 1.

    or

  - the endpoint is attached to a domain with VIRTIO_IOMMU_ATTACH_F_BYPASS.

  or

- the VIRTIO_IOMMU_F_BYPASS feature is negotiated and the endpoint is not attached to a domain.

All accesses from an endpoint in bypass mode are allowed and translated by the IOMMU using the identity function.

5.13.6.1 Driver Requirements: Device operations

The driver SHOULD set field *reserved* of struct virtio_iommu_req_head to zero and MUST ignore field *reserved* of struct virtio_iommu_req_tail.

When a device uses a buffer without having written to it (i.e. used length is zero), the driver SHOULD interpret it as a request failure.

If the VIRTIO_IOMMU_F_INPUT_RANGE feature is negotiated, the driver MUST NOT send requests with *virt_start* less than *input_range.start* or *virt_end* greater than *input_range.end*.

If the VIRTIO_IOMMU_F_DOMAIN_RANGE feature is negotiated, the driver MUST NOT send requests with *domain* less than *domain_range.start* or greater than *domain_range.end*.

5.13.6.2 Device Requirements: Device operations

The device SHOULD set *status* to VIRTIO_IOMMU_S_OK if a request succeeds.

If a request *type* is not recognized, the device SHOULD NOT write the buffer and SHOULD set the used length to zero.

The device MUST ignore field *reserved* of struct virtio_iommu_req_head and SHOULD set field *reserved* of struct virtio_iommu_req_tail to zero.

The device SHOULD NOT let unattached endpoints that are not in bypass mode access the guest-physical address space.

5.13.6.3 ATTACH request

```
struct virtio_iommu_req_attach {
  struct virtio_iommu_req_head head;
  le32 domain;
  le32 endpoint;
  le32 flags;
  u8   reserved[4];
  struct virtio_iommu_req_tail tail;
};

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

Attach an endpoint to a domain. *domain* uniquely identifies a domain within the virtio-iommu device. If the domain doesn't exist in the device, it is created. Semantics of the *endpoint* identifier are platform specific, but the following rules apply:

- The endpoint ID uniquely identifies an endpoint from the virtio-iommu point of view. Multiple endpoints whose DMA transactions are not translated by the same virtio-iommu device can have the same endpoint ID. Endpoints whose DMA transactions may be translated by the same virtio-iommu device have different endpoint IDs.
- On some platforms, it might not be possible to completely isolate two endpoints from each other. For example on a conventional PCI bus, endpoints can snoop DMA transactions from other endpoints on the same bus. Such limitations need to be communicated in a platform specific way.

Multiple endpoints can be attached to the same domain. An endpoint can be attached to a single domain at a time. Endpoints attached to different domains are isolated from each other.

When the VIRTIO_IOMMU_F_BYPASS_CONFIG is negotiated, the driver can set the VIRTIO_IOMMU_-ATTACH_F_BYPASS flag to create a bypass domain. Endpoints attached to this domain are in bypass mode.

5.13.6.3.1 Driver Requirements: ATTACH request

The driver SHOULD set *reserved* to zero.

The driver SHOULD ensure that endpoints that cannot be isolated from each other are attached to the same domain.

If the domain already exists and is a bypass domain, the driver SHOULD set the VIRTIO_IOMMU_ATTACH_-F_BYPASS flag. If the domain exists and is not a bypass domain, the driver SHOULD NOT set the VIRTIO_-IOMMU_ATTACH_F_BYPASS flag.

5.13.6.3.2 Device Requirements: ATTACH request

If the *reserved* field of an ATTACH request is not zero, the device MUST reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

If the device does not recognize a *flags* bit, it MUST reject the request and set *status* to VIRTIO_IOMMU_-S_INVAL.

If the endpoint identified by *endpoint* doesn't exist, the device MUST reject the request and set *status* to VIRTIO_IOMMU_S_NOENT.

If another endpoint is already attached to the domain identified by *domain*, then the device MAY attach the endpoint identified by *endpoint* to the domain. If it cannot do so, the device MUST reject the request and set *status* to VIRTIO_IOMMU_S_UNSUPP.

If the domain already exists and the VIRTIO_IOMMU_ATTACH_F_BYPASS flag is not consistent with that domain, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

If the endpoint identified by *endpoint* is already attached to another domain, then the device SHOULD first detach it from that domain and attach it to the one identified by *domain*. In that case the device SHOULD

behave as if the driver issued a DETACH request with this *endpoint*, followed by the ATTACH request. If the device cannot do so, it MUST reject the request and set *status* to VIRTIO_IOMMU_S_UNSUPP.

If properties of the endpoint (obtained with a PROBE request) are compatible with properties of other endpoints already attached to the requested domain, then the device SHOULD attach the endpoint. Otherwise the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_UNSUPP.

A device that does not reject the request MUST attach the endpoint.

5.13.6.4 DETACH request

```
struct virtio_iommu_req_detach {
  struct virtio_iommu_req_head head;
  le32 domain;
  le32 endpoint;
  u8   reserved[8];
  struct virtio_iommu_req_tail tail;
};
```

Detach an endpoint from a domain. When this request completes, the endpoint cannot access any mapping from that domain anymore. However the endpoint may then be in bypass mode and access the guest-physical address space.

After all endpoints have been successfully detached from a domain, it ceases to exist and its ID can be reused by the driver for another domain.

5.13.6.4.1 Driver Requirements: DETACH request

The driver SHOULD set *reserved* to zero.

5.13.6.4.2 Device Requirements: DETACH request

The device MUST ignore *reserved*.

If the endpoint identified by *endpoint* doesn't exist, then the device MUST reject the request and set *status* to VIRTIO_IOMMU_S_NOENT.

If the domain identified by *domain* doesn't exist, or if the endpoint identified by *endpoint* isn't attached to this domain, then the device MAY set the request *status* to VIRTIO_IOMMU_S_INVAL.

The device MUST ensure that after being detached from a domain, the endpoint cannot access any mapping from that domain.

5.13.6.5 MAP request

```
struct virtio_iommu_req_map {
  struct virtio_iommu_req_head head;
  le32  domain;
  le64  virt_start;
  le64  virt_end;
  le64  phys_start;
  le32  flags;
  struct virtio_iommu_req_tail tail;
};

/* Read access is allowed */
#define VIRTIO_IOMMU_MAP_F_READ   (1 << 0)
/* Write access is allowed */
#define VIRTIO_IOMMU_MAP_F_WRITE  (1 << 1)
/* Accesses are to memory-mapped I/O device */
#define VIRTIO_IOMMU_MAP_F_MMIO   (1 << 2)
```

Map a range of virtually-contiguous addresses to a range of physically-contiguous addresses of the same size. After the request succeeds, all endpoints attached to this domain can access memory in the range $[virt\_start; virt\_end]$ (inclusive). For example, if an endpoint accesses address $VA \in [virt\_start; virt\_end]$,

the device (or the physical IOMMU) translates the address: $PA = VA - virt\_start + phys\_start$. If the access parameters are compatible with *flags* (for instance, the access is write and *flags* are VIRTIO_IOMMU_MAP_F_READ | VIRTIO_IOMMU_MAP_F_WRITE) then the IOMMU allows the access to reach PA.

The range defined by *virt_start* and *virt_end* should be within the limits specified by *input_range*. Given $phys\_end = phys\_start + virt\_end - virt\_start$, the range defined by *phys_start* and phys_end should be within the guest-physical address space. This includes upper and lower limits, as well as any carving of guest-physical addresses for use by the host. Guest physical boundaries are set by the host in a platform specific way.

Availability and allowed combinations of *flags* depend on the underlying IOMMU architectures. VIRTIO_IOMMU_MAP_F_READ and VIRTIO_IOMMU_MAP_F_WRITE are usually implemented, although READ is sometimes implied by WRITE. In addition combinations such as "WRITE and not READ" might not be supported.

The VIRTIO_IOMMU_MAP_F_MMIO flag is a memory type rather than a protection flag. It is only available when the VIRTIO_IOMMU_F_MMIO feature has been negotiated. Accesses to the mapping are not speculated, buffered, cached, split into multiple accesses or combined with other accesses. It may be used, for example, to map Message Signaled Interrupt doorbells when a VIRTIO_IOMMU_RESV_MEM_T_MSI region isn't available. To trigger interrupts the endpoint performs a direct memory write to another peripheral, the IRQ chip.

This request is only available when VIRTIO_IOMMU_F_MAP_UNMAP has been negotiated.

5.13.6.5.1 Driver Requirements: MAP request

The driver SHOULD set undefined *flags* bits to zero.

The driver SHOULD NOT send MAP requests on a bypass domain.

virt_end MUST be strictly greater than *virt_start*.

The driver SHOULD set the VIRTIO_IOMMU_MAP_F_MMIO flag when the physical range corresponds to memory-mapped device registers. The physical range SHOULD have a single memory type: either normal memory or memory-mapped I/O.

If it intends to allow read accesses from endpoints attached to the domain, the driver MUST set the VIRTIO_IOMMU_MAP_F_READ flag.

If the VIRTIO_IOMMU_F_MMIO feature isn't negotiated, the driver MUST NOT use the VIRTIO_IOMMU_MAP_F_MMIO flag.

5.13.6.5.2 Device Requirements: MAP request

If *virt_start*, *phys_start* or (*virt_end* + 1) is not aligned on the page granularity, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_RANGE.

If a mapping already exists in the requested range, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

If the device doesn't recognize a *flags* bit, it MUST reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

If *domain* does not exist, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_NOENT.

If the domain is a bypass domain, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

The device MUST NOT allow writes to a range mapped without the VIRTIO_IOMMU_MAP_F_WRITE flag. However, if the underlying architecture does not support write-only mappings, the device MAY allow reads to a range mapped with VIRTIO_IOMMU_MAP_F_WRITE but not VIRTIO_IOMMU_MAP_F_READ.

5.13.6.6 UNMAP request

```
struct virtio_iommu_req_unmap {
  struct virtio_iommu_req_head head;
  le32  domain;
  le64  virt_start;
  le64  virt_end;
  u8    reserved[4];
  struct virtio_iommu_req_tail tail;
};
```

Unmap a range of addresses mapped with VIRTIO_IOMMU_T_MAP. We define here a mapping as a virtual region created with a single MAP request. All mappings covered by the range $[virt\_start; virt\_end]$ (inclusive) are removed.

The semantics of unmapping are specified in 5.13.6.6.1 and 5.13.6.6.2, and illustrated with the following requests, assuming each example sequence starts with a blank address space. We define two pseudocode functions `map(virt_start, virt_end) -> mapping` and `unmap(virt_start, virt_end)`.

```
(1) unmap(virt_start=0,
          virt_end=4)           -> succeeds, doesn't unmap anything

(2) a = map(virt_start=0,
            virt_end=9);
    unmap(0, 9)                 -> succeeds, unmaps a

(3) a = map(0, 4);
    b = map(5, 9);
    unmap(0, 9)                 -> succeeds, unmaps a and b

(4) a = map(0, 9);
    unmap(0, 4)                 -> fails, doesn't unmap anything

(5) a = map(0, 4);
    b = map(5, 9);
    unmap(0, 4)                 -> succeeds, unmaps a

(6) a = map(0, 4);
    unmap(0, 9)                 -> succeeds, unmaps a

(7) a = map(0, 4);
    b = map(10, 14);
    unmap(0, 14)                -> succeeds, unmaps a and b
```

As illustrated by example (4), partially removing a mapping isn't supported.

This request is only available when VIRTIO_IOMMU_F_MAP_UNMAP has been negotiated.

5.13.6.6.1 Driver Requirements: UNMAP request

The driver SHOULD set the *reserved* field to zero.

The range, defined by *virt_start* and *virt_end*, SHOULD cover one or more contiguous mappings created with MAP requests. The range MAY spill over unmapped virtual addresses.

The first address of a range MUST either be the first address of a mapping or be outside any mapping. The last address of a range MUST either be the last address of a mapping or be outside any mapping.

The driver SHOULD NOT send UNMAP requests on a bypass domain.

5.13.6.6.2 Device Requirements: UNMAP request

If the *reserved* field of an UNMAP request is not zero, the device MAY set the request *status* to VIRTIO_IOMMU_S_INVAL, in which case the device MAY perform the UNMAP operation.

If *domain* does not exist, the device SHOULD set the request *status* to VIRTIO_IOMMU_S_NOENT.

If the domain is a bypass domain, the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_-S_INVAL.

If a mapping affected by the range is not covered in its entirety by the range (the UNMAP request would split the mapping), then the device SHOULD set the request *status* to VIRTIO_IOMMU_S_RANGE, and SHOULD NOT remove any mapping.

If part of the range or the full range is not covered by an existing mapping, then the device SHOULD remove all mappings affected by the range and set the request *status* to VIRTIO_IOMMU_S_OK.

5.13.6.7 PROBE request

If the VIRTIO_IOMMU_F_PROBE feature bit is present, the driver sends a VIRTIO_IOMMU_T_PROBE request for each endpoint that the virtio-iommu device manages. This probe is performed before attaching the endpoint to a domain.

```
struct virtio_iommu_req_probe {
  struct virtio_iommu_req_head head;
  /* Device-readable */
  le32  endpoint;
  u8    reserved[64];

  /* Device-writable */
  u8    properties[probe_size];
  struct virtio_iommu_req_tail tail;
};
```

endpoint has the same meaning as in ATTACH and DETACH requests.

reserved is used as padding, so that future extensions can add fields to the device-readable part.

properties contains a list of properties of the *endpoint*, filled by the device. The length of the *properties* field is *probe_size* bytes. Each property is described with a struct virtio_iommu_probe_property header, which may be followed by a value of size *length*.

```
struct virtio_iommu_probe_property {
  le16 {
    type      : 12;
    reserved  : 4;
  };
  le16  length;
};
```

The driver allocates a buffer for the PROBE request, large enough to accommodate *probe_size* bytes of *properties*. It writes *endpoint* and adds the buffer to the request queue. The device fills the *properties* field with a list of properties for this endpoint.

The driver parses the first property by reading *type*, then *length*. If the driver recognizes *type*, it reads and handles the rest of the property. The driver then reads the next property, that is located $(\textit{length} + 4)$ bytes after the beginning of the first one, and so on. The driver parses all properties until it reaches an empty property (*type* is 0) or the end of *properties*.

Available property types are described in section 5.13.6.8.

5.13.6.7.1 Driver Requirements: PROBE request

The size of *properties* MUST be *probe_size* bytes.

The driver SHOULD set field *reserved* of the PROBE request to zero.

If the driver doesn't recognize the *type* of a property, it SHOULD ignore the property.

The driver SHOULD NOT deduce the property length from *type*.

The driver MUST ignore a property whose *reserved* field is not zero.

If the driver ignores a property, it SHOULD continue parsing the list.

5.13.6.7.2 Device Requirements: PROBE request

The device MUST ignore field *reserved* of a PROBE request.

If the endpoint identified by *endpoint* doesn't exist, then the device SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_NOENT.

If the device does not offer the VIRTIO_IOMMU_F_PROBE feature, and if the driver sends a VIRTIO_IOMMU_T_PROBE request, then the device SHOULD NOT write the buffer and SHOULD set the used length to zero.

The device SHOULD set field *reserved* of a property to zero.

The device MUST write the size of a property without the struct virtio_iommu_probe_property header, in bytes, into *length*.

When two properties follow each other, the device MUST put the second property exactly $(\textit{length} + 4)$ bytes after the beginning of the first one.

If the *properties* list is smaller than *probe_size*, the device SHOULD NOT write any property. It SHOULD reject the request and set *status* to VIRTIO_IOMMU_S_INVAL.

If the device doesn't fill all *probe_size* bytes with properties, it SHOULD fill the remaining bytes of *properties* with zeroes.

5.13.6.8 PROBE properties

```
#define VIRTIO_IOMMU_PROBE_T_RESV_MEM   1
```

5.13.6.8.1 Property RESV_MEM

The RESV_MEM property describes a chunk of reserved virtual memory. It may be used by the device to describe virtual address ranges that cannot be used by the driver, or that are special.

```
struct virtio_iommu_probe_resv_mem {
  struct virtio_iommu_probe_property head;
  u8    subtype;
  u8    reserved[3];
  le64  start;
  le64  end;
};
```

Fields *start* and *end* describe the range of reserved virtual addresses. *subtype* may be one of:

VIRTIO_IOMMU_RESV_MEM_T_RESERVED (0) These virtual addresses cannot be used in a MAP requests. The region is be reserved by the device, for example, if the platform needs to setup DMA mappings of its own.

VIRTIO_IOMMU_RESV_MEM_T_MSI (1) This region is a doorbell for Message Signaled Interrupts (MSIs). It is similar to VIRTIO_IOMMU_RESV_MEM_T_RESERVED, in that the driver cannot map virtual addresses described by the property.

In addition it provides information about MSI doorbells. If the endpoint doesn't have a VIRTIO_IOMMU_RESV_MEM_T_MSI property, then the driver creates an MMIO mapping to the doorbell of the MSI controller.

5.13.6.8.1.1 Driver Requirements: Property RESV_MEM

The driver SHOULD NOT map any virtual address described by a VIRTIO_IOMMU_RESV_MEM_T_RESERVED or VIRTIO_IOMMU_RESV_MEM_T_MSI property.

The driver MUST ignore *reserved*.

The driver SHOULD treat any *subtype* it doesn't recognize as if it was VIRTIO_IOMMU_RESV_MEM_T_RESERVED.

5.13.6.8.1.2 Device Requirements: Property RESV_MEM

The device SHOULD set *reserved* to zero.

The device SHOULD NOT present more than one VIRTIO_IOMMU_RESV_MEM_T_MSI property per endpoint.

The device SHOULD NOT present multiple RESV_MEM properties that overlap each other for the same endpoint.

The device SHOULD reject a MAP request that overlaps a RESV_MEM region.

The device SHOULD NOT allow accesses from the endpoint to RESV_MEM regions to affect any other component than the endpoint and the driver.

5.13.6.9 Fault reporting

The device can report translation faults and other significant asynchronous events on the event virtqueue. The driver initially populates the queue with device-writeable buffers. When the device needs to report an event, it fills a buffer and notifies the driver. The driver consumes the report and adds a new buffer to the virtqueue.

If no buffer is available, the device can either wait for one to be consumed, or drop the event.

```
struct virtio_iommu_fault {
  u8    reason;
  u8    reserved[3];
  le32  flags;
  le32  endpoint;
  le32  reserved1;
  le64  address;
};

#define VIRTIO_IOMMU_FAULT_F_READ     (1 << 0)
#define VIRTIO_IOMMU_FAULT_F_WRITE    (1 << 1)
#define VIRTIO_IOMMU_FAULT_F_ADDRESS  (1 << 8)
```

reason The reason for this report. It may have the following values:

VIRTIO_IOMMU_FAULT_R_UNKNOWN (0) An internal error happened, or an error that cannot be described with the following reasons.

VIRTIO_IOMMU_FAULT_R_DOMAIN (1) The endpoint attempted to access *address* without being attached to a domain.

VIRTIO_IOMMU_FAULT_R_MAPPING (2) The endpoint attempted to access *address*, which wasn't mapped in the domain or didn't have the correct protection flags.

flags Information about the fault context.

endpoint The endpoint causing the fault.

reserved* and *reserved1 Should be zero.

address If VIRTIO_IOMMU_FAULT_F_ADDRESS is set, the address causing the fault.

When the fault is reported by a physical IOMMU, the fault reasons may not match exactly the reason of the original fault report. The device does its best to find the closest match.

If the device encounters an internal error that wasn't caused by a specific endpoint, it is unlikely that the driver would be able to do anything else than print the fault and stop using the device, so reporting the fault on the event queue isn't useful. In that case, we recommend using the DEVICE_NEEDS_RESET status bit.

5.13.6.9.1 Driver Requirements: Fault reporting

If the *reserved* field is not zero, the driver MUST ignore the fault report.

The driver MUST ignore *reserved1*.

The driver MUST ignore undefined *flags*.

If the driver doesn't recognize *reason*, it SHOULD treat the fault as if it was VIRTIO_IOMMU_FAULT_R_-UNKNOWN.

5.13.6.9.2 Device Requirements: Fault reporting

The device SHOULD set *reserved* and *reserved1* to zero.

The device SHOULD set undefined *flags* to zero.

The device SHOULD write a valid endpoint ID in *endpoint*.

The device MAY omit setting VIRTIO_IOMMU_FAULT_F_ADDRESS and writing *address* in any fault report, regardless of the *reason*.

If a buffer is too small to contain the fault report[13], the device SHOULD NOT use multiple buffers to describe it. The device MAY fall back to using an older fault report format that fits in the buffer.

5.14 Sound Device

The virtio sound card is a virtual audio device supporting input and output PCM streams.

A device is managed by control requests and can send various notifications through dedicated queues. A driver can transmit PCM frames using message-based transport or shared memory.

A small part of the specification reuses existing layouts and values from the High Definition Audio specification (HDA). It allows to provide the same functionality and assist in two possible cases:

1. implementation of a universal sound driver,
2. implementation of a sound driver as part of the High Definition Audio subsystem.

5.14.1 Device ID

25

5.14.2 Virtqueues

0 controlq

1 eventq

2 txq

3 rxq

The control queue is used for sending control messages from the driver to the device.

The event queue is used for sending notifications from the device to the driver.

The tx queue is used to send PCM frames for output streams.

The rx queue is used to receive PCM frames from input streams.

5.14.3 Feature Bits

VIRTIO_SND_F_CTLS (0) Device supports control elements.

13This would happen for example if the device implements a more recent version of this specification, whose fault report contains additional fields.

5.14.4 Device Configuration Layout

```
struct virtio_snd_config {
    le32 jacks;
    le32 streams;
    le32 chmaps;
    le32 controls;
};
```

A configuration space contains the following fields:

jacks (driver-read-only) indicates a total number of all available jacks.

streams (driver-read-only) indicates a total number of all available PCM streams.

chmaps (driver-read-only) indicates a total number of all available channel maps.

controls (driver-read-only) indicates a total number of all available control elements if VIRTIO_SND_F_-CTLS has been negotiated.

5.14.5 Device Initialization

1. Configure the control, event, tx and rx queues.
2. Read the *jacks* field and send a control request to query information about the available jacks.
3. Read the *streams* field and send a control request to query information about the available PCM streams.
4. Read the *chmaps* field and send a control request to query information about the available channel maps.
5. If VIRTIO_SND_F_CTLS has been negotiated, read the *controls* field and send a control request to query information about the available control elements.
6. Populate the event queue with empty buffers.

5.14.5.1 Driver Requirements: Device Initialization

- The driver MUST NOT read the *controls* field if VIRTIO_SND_F_CTLS has not been negotiated.
- The driver MUST populate the event queue with empty buffers of at least the struct virtio_snd_event size.
- The driver MUST NOT put a device-readable buffers in the event queue.

5.14.6 Device Operation

All control messages are placed into the control queue and all notifications are placed into the event queue. They use the following layout structure and definitions:

```
enum {
    /* jack control request types */
    VIRTIO_SND_R_JACK_INFO = 1,
    VIRTIO_SND_R_JACK_REMAP,

    /* PCM control request types */
    VIRTIO_SND_R_PCM_INFO = 0x0100,
    VIRTIO_SND_R_PCM_SET_PARAMS,
    VIRTIO_SND_R_PCM_PREPARE,
    VIRTIO_SND_R_PCM_RELEASE,
    VIRTIO_SND_R_PCM_START,
    VIRTIO_SND_R_PCM_STOP,

    /* channel map control request types */
    VIRTIO_SND_R_CHMAP_INFO = 0x0200,

    /* control element request types */
    VIRTIO_SND_R_CTL_INFO = 0x0300,
```

```
    VIRTIO_SND_R_CTL_ENUM_ITEMS,
    VIRTIO_SND_R_CTL_READ,
    VIRTIO_SND_R_CTL_WRITE,
    VIRTIO_SND_R_CTL_TLV_READ,
    VIRTIO_SND_R_CTL_TLV_WRITE,
    VIRTIO_SND_R_CTL_TLV_COMMAND,

    /* jack event types */
    VIRTIO_SND_EVT_JACK_CONNECTED = 0x1000,
    VIRTIO_SND_EVT_JACK_DISCONNECTED,

    /* PCM event types */
    VIRTIO_SND_EVT_PCM_PERIOD_ELAPSED = 0x1100,
    VIRTIO_SND_EVT_PCM_XRUN,

    /* control element event types */
    VIRTIO_SND_EVT_CTL_NOTIFY = 0x1200,

    /* common status codes */
    VIRTIO_SND_S_OK = 0x8000,
    VIRTIO_SND_S_BAD_MSG,
    VIRTIO_SND_S_NOT_SUPP,
    VIRTIO_SND_S_IO_ERR
};

/* a common header */
struct virtio_snd_hdr {
    le32 code;
};

/* an event notification */
struct virtio_snd_event {
    struct virtio_snd_hdr hdr;
    le32 data;
};
```

A generic control message consists of a request part and a response part.

A request part has, or consists of, a common header containing the following device-readable field:

code specifies a device request type (VIRTIO_SND_R_*).

A response part has, or consists of, a common header containing the following device-writable field:

code indicates a device request status (VIRTIO_SND_S_*).

The status field can take one of the following values:

- VIRTIO_SND_S_OK - success.
- VIRTIO_SND_S_BAD_MSG - a control message is malformed or contains invalid parameters.
- VIRTIO_SND_S_NOT_SUPP - requested operation or parameters are not supported.
- VIRTIO_SND_S_IO_ERR - an I/O error occurred.

The request part may be followed by an additional device-readable payload, and the response part may be followed by an additional device-writable payload.

An event notification contains the following device-writable fields:

hdr indicates an event type (VIRTIO_SND_EVT_*).

data indicates an optional event data.

For all entities involved in the exchange of audio data, the device uses one of the following data flow directions:

```
enum {
    VIRTIO_SND_D_OUTPUT = 0,
    VIRTIO_SND_D_INPUT
};
```

5.14.6.1 Item Information Request

A special control message is used to request information about any kind of configuration item. The request part uses the following structure definition:

```
struct virtio_snd_query_info {
    struct virtio_snd_hdr hdr;
    le32 start_id;
    le32 count;
    le32 size;
};
```

The request contains the following device-readable fields:

hdr specifies a particular item request type (VIRTIO_SND_R_*_INFO).

start_id specifies the starting identifier for the item (the range of available identifiers is limited by the total number of particular items that is indicated in the device configuration space).

count specifies the number of items for which information is requested (the total number of particular items is indicated in the device configuration space).

size specifies the size of the structure containing information for one item (used for backward compatibility).

The response consists of the virtio_snd_hdr structure (contains the request status code), followed by the device-writable information structures of the item. Each information structure begins with the following common header:

```
struct virtio_snd_info {
    le32 hda_fn_nid;
};
```

The header contains the following field:

hda_fn_nid indicates a function group node identifier (see HDA, section 7.1.2). This field can be used to link together different types of resources (e.g. jacks with streams and channel maps with streams).

5.14.6.2 Driver Requirements: Item Information Request

- The driver MUST NOT set *start_id* and *count* such that *start_id* + *count* is greater than the total number of particular items that is indicated in the device configuration space.
- The driver MUST provide a buffer of sizeof(struct virtio_snd_hdr) + *count* * *size* bytes for the response.

5.14.6.3 Relationships with the High Definition Audio Specification

The High Definition Audio specification introduces the codec as part of the hardware that implements some of the functionality. The codec architecture and capabilities are described by tree structure of special nodes each of which is either a function module or a function group (see HDA for details).

The virtio sound specification assumes that a single codec is implemented in the device. Function module nodes are simulated by item information structures, and function group nodes are simulated by the *hda_fn_nid* field in each such structure.

5.14.6.4 Jack Control Messages

A jack control request has, or consists of, a common header with the following layout structure:

```
struct virtio_snd_jack_hdr {
    struct virtio_snd_hdr hdr;
    le32 jack_id;
};
```

The header consists of the following device-readable fields:

hdr specifies a request type (VIRTIO_SND_R_JACK_*).

jack_id specifies a jack identifier from 0 to *jacks* - 1.

5.14.6.4.1 VIRTIO_SND_R_JACK_INFO

Query information about the available jacks.

The request consists of the virtio_snd_query_info structure (see Item Information Request). The response consists of the virtio_snd_hdr structure, followed by the following jack information structures:

```
/* supported jack features */
enum {
    VIRTIO_SND_JACK_F_REMAP = 0
};

struct virtio_snd_jack_info {
    struct virtio_snd_info hdr;
    le32 features; /* 1 << VIRTIO_SND_JACK_F_XXX */
    le32 hda_reg_defconf;
    le32 hda_reg_caps;
    u8 connected;

    u8 padding[7];
};
```

The structure contains the following device-writable fields:

features indicates a supported feature bit map:

- VIRTIO_SND_JACK_F_REMAP - jack remapping support.

hda_reg_defconf indicates a pin default configuration value (see HDA, section 7.3.3.31).

hda_reg_caps indicates a pin capabilities value (see HDA, section 7.3.4.9).

connected indicates the current jack connection status (1 - connected, 0 - disconnected).

5.14.6.4.1.1 Device Requirements: Jack Information

- The device MUST NOT set undefined feature values.
- The device MUST initialize the *padding* bytes to 0.

5.14.6.4.2 VIRTIO_SND_R_JACK_REMAP

If the VIRTIO_SND_JACK_F_REMAP feature bit is set in the jack information, then the driver can send a control request to change the association and/or sequence number for the specified jack ID.

The request uses the following structure and layout definitions:

```
struct virtio_snd_jack_remap {
    struct virtio_snd_jack_hdr hdr; /* .code = VIRTIO_SND_R_JACK_REMAP */
    le32 association;
    le32 sequence;
};
```

The request contains the following device-readable fields:

association specifies the selected association number.

sequence specifies the selected sequence number.

5.14.6.5 Jack Notifications

Jack notifications consist of a virtio_snd_event structure, which has the following device-writable fields:

hdr indicates a jack event type:

- VIRTIO_SND_EVT_JACK_CONNECTED - an external device has been connected to the jack.

- VIRTIO_SND_EVT_JACK_DISCONNECTED - an external device has been disconnected from the jack.

data indicates a jack identifier from 0 to *jacks* - 1.

5.14.6.6 PCM Control Messages

A PCM control request has, or consists of, a common header with the following layout structure:

```
struct virtio_snd_pcm_hdr {
    struct virtio_snd_hdr hdr;
    le32 stream_id;
};
```

The header consists of the following device-readable fields:

hdr specifies request type (VIRTIO_SND_R_PCM_*).

stream_id specifies a PCM stream identifier from 0 to *streams* - 1.

5.14.6.6.1 PCM Command Lifecycle

A PCM stream has the following command lifecycle:

1. SET PARAMETERS

   The driver negotiates the stream parameters (format, transport, etc) with the device.

   Possible valid transitions: set parameters, prepare.

2. PREPARE

   The device prepares the stream (allocates resources, etc).

   Possible valid transitions: set parameters, prepare, start, release.

3. Output only: the driver transfers data for pre-buffing.

4. START

   The device starts the stream (unmute, putting into running state, etc).

   Possible valid transitions: stop.

5. The driver transfers data to/from the stream.

6. STOP

   The device stops the stream (mute, putting into non-running state, etc).

   Possible valid transitions: start, release.

7. RELEASE

   The device releases the stream (frees resources, etc).

   Possible valid transitions: set parameters, prepare.

5.14.6.6.2 VIRTIO_SND_R_PCM_INFO

Query information about the available streams.

The request consists of the virtio_snd_query_info structure (see Item Information Request). The response consists of the virtio_snd_hdr structure, followed by the following stream information structures:

```
/* supported PCM stream features */
enum {
    VIRTIO_SND_PCM_F_SHMEM_HOST = 0,
    VIRTIO_SND_PCM_F_SHMEM_GUEST,
    VIRTIO_SND_PCM_F_MSG_POLLING,
```

```
    VIRTIO_SND_PCM_F_EVT_SHMEM_PERIODS,
    VIRTIO_SND_PCM_F_EVT_XRUNS
};

/* supported PCM sample formats */
enum {
    /* analog formats (width / physical width) */
    VIRTIO_SND_PCM_FMT_IMA_ADPCM = 0,   /*  4 /  4 bits */
    VIRTIO_SND_PCM_FMT_MU_LAW,          /*  8 /  8 bits */
    VIRTIO_SND_PCM_FMT_A_LAW,           /*  8 /  8 bits */
    VIRTIO_SND_PCM_FMT_S8,              /*  8 /  8 bits */
    VIRTIO_SND_PCM_FMT_U8,              /*  8 /  8 bits */
    VIRTIO_SND_PCM_FMT_S16,             /* 16 / 16 bits */
    VIRTIO_SND_PCM_FMT_U16,             /* 16 / 16 bits */
    VIRTIO_SND_PCM_FMT_S18_3,           /* 18 / 24 bits */
    VIRTIO_SND_PCM_FMT_U18_3,           /* 18 / 24 bits */
    VIRTIO_SND_PCM_FMT_S20_3,           /* 20 / 24 bits */
    VIRTIO_SND_PCM_FMT_U20_3,           /* 20 / 24 bits */
    VIRTIO_SND_PCM_FMT_S24_3,           /* 24 / 24 bits */
    VIRTIO_SND_PCM_FMT_U24_3,           /* 24 / 24 bits */
    VIRTIO_SND_PCM_FMT_S20,             /* 20 / 32 bits */
    VIRTIO_SND_PCM_FMT_U20,             /* 20 / 32 bits */
    VIRTIO_SND_PCM_FMT_S24,             /* 24 / 32 bits */
    VIRTIO_SND_PCM_FMT_U24,             /* 24 / 32 bits */
    VIRTIO_SND_PCM_FMT_S32,             /* 32 / 32 bits */
    VIRTIO_SND_PCM_FMT_U32,             /* 32 / 32 bits */
    VIRTIO_SND_PCM_FMT_FLOAT,           /* 32 / 32 bits */
    VIRTIO_SND_PCM_FMT_FLOAT64,         /* 64 / 64 bits */
    /* digital formats (width / physical width) */
    VIRTIO_SND_PCM_FMT_DSD_U8,          /*  8 /  8 bits */
    VIRTIO_SND_PCM_FMT_DSD_U16,         /* 16 / 16 bits */
    VIRTIO_SND_PCM_FMT_DSD_U32,         /* 32 / 32 bits */
    VIRTIO_SND_PCM_FMT_IEC958_SUBFRAME  /* 32 / 32 bits */
};

/* supported PCM frame rates */
enum {
    VIRTIO_SND_PCM_RATE_5512 = 0,
    VIRTIO_SND_PCM_RATE_8000,
    VIRTIO_SND_PCM_RATE_11025,
    VIRTIO_SND_PCM_RATE_16000,
    VIRTIO_SND_PCM_RATE_22050,
    VIRTIO_SND_PCM_RATE_32000,
    VIRTIO_SND_PCM_RATE_44100,
    VIRTIO_SND_PCM_RATE_48000,
    VIRTIO_SND_PCM_RATE_64000,
    VIRTIO_SND_PCM_RATE_88200,
    VIRTIO_SND_PCM_RATE_96000,
    VIRTIO_SND_PCM_RATE_176400,
    VIRTIO_SND_PCM_RATE_192000,
    VIRTIO_SND_PCM_RATE_384000,
    VIRTIO_SND_PCM_RATE_12000,
    VIRTIO_SND_PCM_RATE_24000
};

struct virtio_snd_pcm_info {
    struct virtio_snd_info hdr;
    le32 features; /* 1 << VIRTIO_SND_PCM_F_XXX */
    le64 formats; /* 1 << VIRTIO_SND_PCM_FMT_XXX */
    le64 rates; /* 1 << VIRTIO_SND_PCM_RATE_XXX */
    u8 direction;
    u8 channels_min;
    u8 channels_max;

    u8 padding[5];
};
```

The structure contains the following device-writable fields:

features indicates a bit map of the supported features, which can be negotiated by setting the stream parameters:

- VIRTIO_SND_PCM_F_SHMEM_HOST - supports sharing a host memory with a guest,
- VIRTIO_SND_PCM_F_SHMEM_GUEST - supports sharing a guest memory with a host,
- VIRTIO_SND_PCM_F_MSG_POLLING - supports polling mode for message-based transport,
- VIRTIO_SND_PCM_F_EVT_SHMEM_PERIODS - supports elapsed period notifications for shared memory transport,
- VIRTIO_SND_PCM_F_EVT_XRUNS - supports underrun/overrun notifications.

formats indicates a supported sample format bit map.

rates indicates a supported frame rate bit map.

direction indicates the direction of data flow (VIRTIO_SND_D_*).

channels_min indicates a minimum number of supported channels.

channels_max indicates a maximum number of supported channels.

Only interleaved samples are supported.

5.14.6.6.2.1 Device Requirements: Stream Information

- The device MUST NOT set undefined feature, format, rate and direction values.
- The device MUST initialize the *padding* bytes to 0.

5.14.6.6.3 VIRTIO_SND_R_PCM_SET_PARAMS

Set selected stream parameters for the specified stream ID.

The request uses the following structure and layout definitions:

```
struct virtio_snd_pcm_set_params {
    struct virtio_snd_pcm_hdr hdr; /* .code = VIRTIO_SND_R_PCM_SET_PARAMS */
    le32 buffer_bytes;
    le32 period_bytes;
    le32 features; /* 1 << VIRTIO_SND_PCM_F_XXX */
    u8 channels;
    u8 format;
    u8 rate;

    u8 padding;
};
```

The request contains the following device-readable fields:

buffer_bytes specifies the size of the hardware buffer used by the driver.

period_bytes specifies the size of the hardware period used by the driver.

features specifies a selected feature bit map:

- VIRTIO_SND_PCM_F_SHMEM_HOST - use shared memory allocated by the host (is a place-holder and MUST NOT be selected at the moment),
- VIRTIO_SND_PCM_F_SHMEM_GUEST - use shared memory allocated by the guest (is a place-holder and MUST NOT be selected at the moment),
- VIRTIO_SND_PCM_F_MSG_POLLING - suppress available buffer notifications for tx and rx queues (device should poll virtqueue),
- VIRTIO_SND_PCM_F_EVT_SHMEM_PERIODS - enable elapsed period notifications for shared memory transport,
- VIRTIO_SND_PCM_F_EVT_XRUNS - enable underrun/overrun notifications.

channels specifies a selected number of channels.

format specifies a selected sample format (VIRTIO_SND_PCM_FMT_*).

rate specifies a selected frame rate (VIRTIO_SND_PCM_RATE_*).

5.14.6.6.3.1 Device Requirements: Stream Parameters

- If the device has an intermediate buffer, its size MUST be no less than the specified *buffer_bytes* value.

5.14.6.6.3.2 Driver Requirements: Stream Parameters

- *period_bytes* MUST be a divider *buffer_bytes*, i.e. *buffer_bytes* % *period_bytes* == 0.
- The driver MUST NOT set undefined feature, format and rate values.
- The driver MUST NOT set the feature, format, and rate that are not specified in the stream configuration.
- The driver MUST NOT set the *channels* value as less than the *channels_min* or greater than the *channels_max* values specified in the stream configuration.
- The driver MUST NOT set the VIRTIO_SND_PCM_F_SHMEM_HOST and VIRTIO_SND_PCM_F_SHMEM_GUEST bits at the same time.
- The driver MUST initialize the *padding* byte to 0.
- If the bits associated with the shared memory are not selected, the driver MUST use the tx and rx queues for data transfer (see PCM I/O Messages).

5.14.6.6.4 VIRTIO_SND_R_PCM_PREPARE

Prepare a stream with specified stream ID.

5.14.6.6.5 VIRTIO_SND_R_PCM_RELEASE

Release a stream with specified stream ID.

5.14.6.6.5.1 Device Requirements: Stream Release

- The device MUST complete all pending I/O messages for the specified stream ID.
- The device MUST NOT complete the control request while there are pending I/O messages for the specified stream ID.

5.14.6.6.6 VIRTIO_SND_R_PCM_START

Start a stream with specified stream ID.

5.14.6.6.7 VIRTIO_SND_R_PCM_STOP

Stop a stream with specified stream ID.

5.14.6.7 PCM Notifications

The device can announce support for different PCM events using feature bits in the stream information structure. To enable notifications, the driver must negotiate these features using the set stream parameters request (see 5.14.6.6.3).

PCM stream notifications consist of a virtio_snd_event structure, which has the following device-writable fields:

hdr indicates a PCM stream event type:

- VIRTIO_SND_EVT_PCM_PERIOD_ELAPSED - a hardware buffer period has elapsed, the period size is controlled using the *period_bytes* field.

- VIRTIO_SND_EVT_PCM_XRUN - an underflow for the output stream or an overflow for the input stream has occurred.

data indicates a PCM stream identifier from 0 to *streams* - 1.

5.14.6.8 PCM I/O Messages

An I/O message consists of the header part, followed by the buffer, and then the status part.

```
/* an I/O header */
struct virtio_snd_pcm_xfer {
    le32 stream_id;
};

/* an I/O status */
struct virtio_snd_pcm_status {
    le32 status;
    le32 latency_bytes;
};
```

The header part consists of the following device-readable field:

stream_id specifies a PCM stream identifier from 0 to *streams* - 1.

The status part consists of the following device-writable fields:

status contains VIRTIO_SND_S_OK if an operation is successful, and VIRTIO_SND_S_IO_ERR otherwise.

latency_bytes indicates the current device latency.

Since all buffers in the queue (with one exception) should be of the size *period_bytes*, the completion of such an I/O request can be considered an elapsed period notification.

5.14.6.8.1 Output Stream

In case of an output stream, the header is followed by a device-readable buffer containing PCM frames for writing to the device. All messages are placed into the tx queue.

5.14.6.8.1.1 Device Requirements: Output Stream

- The device MUST NOT complete the I/O request until the buffer is totally consumed.

5.14.6.8.1.2 Driver Requirements: Output Stream

- The driver SHOULD populate the tx queue with *period_bytes* sized buffers. The only exception is the end of the stream.
- The driver MUST NOT place device-writable buffers into the tx queue.

5.14.6.8.2 Input Stream

In case of an input stream, the header is followed by a device-writable buffer being populated with PCM frames from the device. All messages are placed into the rx queue.

A used descriptor specifies the length of the buffer that was written by the device. It should be noted that the length value contains the size of the virtio_snd_pcm_status structure plus the size of the recorded frames.

5.14.6.8.2.1 Device Requirements: Input Stream

- The device MUST NOT complete the I/O request until the buffer is full. The only exception is the end of the stream.

5.14.6.8.2.2 Driver Requirements: Input Stream

- The driver SHOULD populate the rx queue with *period_bytes* sized empty buffers before starting the stream.
- The driver MUST NOT place device-readable buffers into the rx queue.

5.14.6.9 Channel Map Control Messages

A device can provide one or more channel maps assigned to all streams with the same data flow direction in the same function group.

5.14.6.9.1 VIRTIO_SND_R_CHMAP_INFO

Query information about the available channel maps.

The request consists of the virtio_snd_query_info structure (see Item Information Request). The response consists of the virtio_snd_hdr structure, followed by the following channel map information structures:

```
/* standard channel position definition */
enum {
    VIRTIO_SND_CHMAP_NONE = 0,  /* undefined */
    VIRTIO_SND_CHMAP_NA,        /* silent */
    VIRTIO_SND_CHMAP_MONO,      /* mono stream */
    VIRTIO_SND_CHMAP_FL,        /* front left */
    VIRTIO_SND_CHMAP_FR,        /* front right */
    VIRTIO_SND_CHMAP_RL,        /* rear left */
    VIRTIO_SND_CHMAP_RR,        /* rear right */
    VIRTIO_SND_CHMAP_FC,        /* front center */
    VIRTIO_SND_CHMAP_LFE,       /* low frequency (LFE) */
    VIRTIO_SND_CHMAP_SL,        /* side left */
    VIRTIO_SND_CHMAP_SR,        /* side right */
    VIRTIO_SND_CHMAP_RC,        /* rear center */
    VIRTIO_SND_CHMAP_FLC,       /* front left center */
    VIRTIO_SND_CHMAP_FRC,       /* front right center */
    VIRTIO_SND_CHMAP_RLC,       /* rear left center */
    VIRTIO_SND_CHMAP_RRC,       /* rear right center */
    VIRTIO_SND_CHMAP_FLW,       /* front left wide */
    VIRTIO_SND_CHMAP_FRW,       /* front right wide */
    VIRTIO_SND_CHMAP_FLH,       /* front left high */
    VIRTIO_SND_CHMAP_FCH,       /* front center high */
    VIRTIO_SND_CHMAP_FRH,       /* front right high */
    VIRTIO_SND_CHMAP_TC,        /* top center */
    VIRTIO_SND_CHMAP_TFL,       /* top front left */
    VIRTIO_SND_CHMAP_TFR,       /* top front right */
    VIRTIO_SND_CHMAP_TFC,       /* top front center */
    VIRTIO_SND_CHMAP_TRL,       /* top rear left */
    VIRTIO_SND_CHMAP_TRR,       /* top rear right */
    VIRTIO_SND_CHMAP_TRC,       /* top rear center */
    VIRTIO_SND_CHMAP_TFLC,      /* top front left center */
    VIRTIO_SND_CHMAP_TFRC,      /* top front right center */
    VIRTIO_SND_CHMAP_TSL,       /* top side left */
    VIRTIO_SND_CHMAP_TSR,       /* top side right */
    VIRTIO_SND_CHMAP_LLFE,      /* left LFE */
    VIRTIO_SND_CHMAP_RLFE,      /* right LFE */
    VIRTIO_SND_CHMAP_BC,        /* bottom center */
    VIRTIO_SND_CHMAP_BLC,       /* bottom left center */
    VIRTIO_SND_CHMAP_BRC        /* bottom right center */
};

/* maximum possible number of channels */
#define VIRTIO_SND_CHMAP_MAX_SIZE 18

struct virtio_snd_chmap_info {
    struct virtio_snd_info hdr;
    u8 direction;
    u8 channels;
    u8 positions[VIRTIO_SND_CHMAP_MAX_SIZE];
};
```

The structure contains the following device-writable fields:

direction indicates the direction of data flow (VIRTIO_SND_D_*).

channels indicates the number of valid channel position values.

positions indicates channel position values (VIRTIO_SND_CHMAP_*).

5.14.6.9.1.1 Device Requirements: Channel Map Information

- The device MUST NOT set undefined direction values.
- The device MUST NOT set the channels value to more than VIRTIO_SND_CHMAP_MAX_SIZE.

5.14.6.10 Control Elements

Control elements can be used to set the volume level, mute/unmute the audio signal, switch different modes/states of the virtual sound device, etc. Design of virtual audio controls is based on and derived from ALSA audio controls.

The device informs about the support of audio controls by setting the VIRTIO_SND_F_CTLS feature bit. If VIRTIO_SND_F_CTLS has been negotiated, the following messages are available for manipulation of control elements.

A control request has, or consists of, a common header with the following layout structure:

```
struct virtio_snd_ctl_hdr {
    struct virtio_snd_hdr hdr;
    le32 control_id;
};
```

The header consists of the following device-readable fields:

hdr specifies request type (VIRTIO_SND_R_CTL_*).

control_id specifies a control element identifier from 0 to *virtio_snd_config::controls* - 1.

5.14.6.10.1 Query information

The VIRTIO_SND_R_CTL_INFO control message is used to query basic information about the available control elements.

The request consists of the virtio_snd_query_info structure (see Item Information Request). The response consists of the virtio_snd_hdr structure, followed by the following control element information structures:

```
enum {
    VIRTIO_SND_CTL_ROLE_UNDEFINED = 0,
    VIRTIO_SND_CTL_ROLE_VOLUME,
    VIRTIO_SND_CTL_ROLE_MUTE,
    VIRTIO_SND_CTL_ROLE_GAIN
};

enum {
    VIRTIO_SND_CTL_TYPE_BOOLEAN = 0,
    VIRTIO_SND_CTL_TYPE_INTEGER,
    VIRTIO_SND_CTL_TYPE_INTEGER64,
    VIRTIO_SND_CTL_TYPE_ENUMERATED,
    VIRTIO_SND_CTL_TYPE_BYTES,
    VIRTIO_SND_CTL_TYPE_IEC958
};

enum {
    VIRTIO_SND_CTL_ACCESS_READ = 0,
    VIRTIO_SND_CTL_ACCESS_WRITE,
    VIRTIO_SND_CTL_ACCESS_VOLATILE,
    VIRTIO_SND_CTL_ACCESS_INACTIVE,
    VIRTIO_SND_CTL_ACCESS_TLV_READ,
    VIRTIO_SND_CTL_ACCESS_TLV_WRITE,
```

```
    VIRTIO_SND_CTL_ACCESS_TLV_COMMAND
};

struct virtio_snd_ctl_info {
    struct virtio_snd_info hdr;
    le32 role;
    le32 type;
    le32 access; /* 1 << VIRTIO_SND_CTL_ACCESS_XXX */
    le32 count;
    le32 index;
    u8 name[44];
    union {
        struct {
            le32 min;
            le32 max;
            le32 step;
        } integer;
        struct {
            le64 min;
            le64 max;
            le64 step;
        } integer64;
        struct {
            le32 items;
        } enumerated;
    } value;
};
```

The structure contains the following device-writable fields:

role indicates a role for the element. If the field value is not equal to UNDEFINED, then the least significant bit indicates the direction of data flow (VIRTIO_SND_D_*), and (*role* & 0xfffffffe) » 1 is equal to one of the following values (VIRTIO_SND_CTL_ROLE_*):

VOLUME is for a volume control.

MUTE is for a mute switch.

GAIN is for a gain control.

type indicates the element value type (VIRTIO_SND_CTL_TYPE_*):

BOOLEAN This is a special case of the INTEGER type, which can take only two values: 0 (off) and 1 (on).

INTEGER 32-bit integer values.

INTEGER64 64-bit integer values.

ENUMERATED The value is represented by an array of ASCII strings.

BYTES 8-bit integer values.

IEC958 This element is connected to the digital audio interface. The value is in the form of the virtio_snd_ctl_iec958 structure.

access indicates a bit mask describing access rights to the element (VIRTIO_SND_CTL_ACCESS_*):

READ It is possible to read the value.

WRITE It is possible to write (change) the value.

VOLATILE The value may be changed without a notification.

INACTIVE The element does actually nothing, but may be updated.

TLV_READ It is possible to read metadata.

TLV_WRITE It is possible to write (change) metadata.

TLV_COMMAND It is possible to execute a command for metadata.

count indicates the number of *type* members that represent the value of the element.

index indicates the index for an element with a non-unique *name*.

name indicates the name identifier string for the element.

value indicates some additional information about the value for certain types of elements:

integer

integer64 *min* and *max* indicate minimum and maximum element values. *step* indicates a fixed step size for changing the element value between minimum and maximum values. The special value 0 means that the step has variable size.

enumerated *items* indicates the number of items from which the element value can be selected.

To query an array of items for elements with the ENUMERATED type, an additional VIRTIO_SND_R_CTL_ENUM_ITEMS control message is used. The request consists of the virtio_snd_ctl_hdr structure. The response consists of the virtio_snd_hdr structure, followed by an array of size *value.enumerated.items*, consisting of the following structures:

```
struct virtio_snd_ctl_enum_item {
    u8 item[64];
};
```

The structure contains the only device-writable field:

item indicates the name of an available element option.

5.14.6.10.1.1 Device Requirements: Control Element Information

- The device MUST NOT set undefined *role*, *type* and *access* values.
- The device MUST set the *count* to a value other than zero. The maximum allowed value depends on the element type:

  BOOLEAN 128

  INTEGER 128

  INTEGER64 64

  ENUMERATED 128

  BYTES 512

  IEC958 1

- The device MUST set *name* and *item* fields to a non-empty 0-terminated ASCII strings.
- The device MUST ensure that the combination (*name*, *index*) is unique for each available element.

5.14.6.10.2 Value

If the element has VIRTIO_SND_CTL_ACCESS_READ access right, then the driver can issue VIRTIO_SND_R_CTL_READ request to the device to read the element's value.

If the element has VIRTIO_SND_CTL_ACCESS_WRITE access right, then the driver can issue VIRTIO_SND_R_CTL_WRITE request to the device to write the element's value.

The following structure and layout definitions are used in read and write requests:

```
struct virtio_snd_ctl_iec958 {
    u8 status[24];      /* AES/IEC958 channel status bits */
    u8 subcode[147];    /* AES/IEC958 subcode bits */
    u8 pad;             /* nothing */
    u8 dig_subframe[4]; /* AES/IEC958 subframe bits */
};
```

```
struct virtio_snd_ctl_value {
    union {
        le32 integer[128];
        le64 integer64[64];
        le32 enumerated[128];
        u8 bytes[512];
        struct virtio_snd_ctl_iec958 iec958;
    } value;
};
```

The element value structure consists of a single *value* union, which contains the following fields:

integer specifies values for an element of type BOOLEAN or INTEGER.

integer64 specifies values for an element of type INTEGER64.

enumerated specifies indexes of items for an element of type ENUMERATED.

bytes specifies values for an element of type BYTES.

iec958 specifies a value for an element of type IEC958.

For all types, except for IEC958, the array contains *count* values (the *count* is reported in the element information structure).

A read request consists of a (device-readable) virtio_snd_ctl_hdr structure containing request, followed by (device-writable) virtio_snd_hdr and virtio_snd_ctl_value structures, into which the status of the request and the current value of the element will be written.

A write request consists of (device-readable) virtio_snd_ctl_hdr and virtio_snd_ctl_value structures containing request and the new element value, followed by (device-writable) virtio_snd_hdr structure, into which the status of the request will be written.

5.14.6.10.2.1 Driver Requirements: Control Element Value

- The driver MUST NOT send READ request if the element does not have ACCESS_READ access right.
- The driver MUST NOT send WRITE request if the element does not have ACCESS_WRITE access right.

5.14.6.10.3 Metadata

Metadata can be used to provide additional (arbitrary) information about the element (e.g. dB range).

If the element has VIRTIO_SND_CTL_ACCESS_TLV_READ access right, then the driver can issue VIRTIO_SND_R_CTL_TLV_READ request to the device to read the element's metadata.

If the element has VIRTIO_SND_CTL_ACCESS_TLV_WRITE access right, then the driver can issue VIRTIO_SND_R_CTL_TLV_WRITE request to the device to write the element's metadata.

If the element has VIRTIO_SND_CTL_ACCESS_TLV_COMMAND access right, then the driver can issue VIRTIO_SND_R_CTL_TLV_COMMAND request to the device to execute a command for element's metadata.

All information related to metadata is presented in the form of TLV (Type-Length-Value):

```
struct virtio_snd_ctl_tlv {
    le32 type;
    le32 length;
    le32 value[];
};
```

The structure contains the following fields:

type specifies metadata type. ALSA defines several standard metadata types for control elements, details of which can be found in various ALSA documentation. Device implementers can also define their own types and formats.

length specifies the *value* length in bytes aligned to 4.

value contains metadata in form of an array of 4-byte integers.

A read request consists of a (device-readable) virtio_snd_ctl_hdr structure containing request, followed by (device-writable) virtio_snd_hdr and virtio_snd_ctl_tlv structures, into which the status of the request and element's metadata will be written.

A write and command requests consist of (device-readable) virtio_snd_ctl_hdr and virtio_snd_ctl_tlv structures containing request and element's metadata/command content, followed by (device-writable) virtio_snd_hdr structure, into which the status of the request will be written.

5.14.6.10.3.1 Device Requirements: Control Element Metadata

- For a read request, if there is not enough space in the *value* field, the device SHOULD write as much as it can and successfully complete the request.

5.14.6.10.3.2 Driver Requirements: Control Element Metadata

- The driver MUST NOT send TLV_READ request if the element does not have ACCESS_TLV_READ access right.
- The driver MUST NOT send TLV_WRITE request if the element does not have ACCESS_TLV_WRITE access right.
- The driver MUST NOT send TLV_COMMAND request if the element does not have ACCESS_TLV_COMMAND access right.
- The driver MUST NOT submit the *value* field larger than 65536 bytes.

5.14.6.10.4 Notifications

The notification uses the following structure and layout definitions:

```
enum {
    VIRTIO_SND_CTL_EVT_MASK_VALUE = 0,
    VIRTIO_SND_CTL_EVT_MASK_INFO,
    VIRTIO_SND_CTL_EVT_MASK_TLV
};

struct virtio_snd_ctl_event {
    struct virtio_snd_hdr hdr; /* .code = VIRTIO_SND_EVT_CTL_NOTIFY */
    le16 control_id;
    le16 mask; /* 1 << VIRTIO_SND_CTL_EVT_MASK_XXX */
};
```

The structure contains the following device-writable fields:

control_id indicates a control element identifier from 0 to *virtio_snd_config::controls* - 1.

mask indicates a bit mask describing the reason(s) for sending the notification (VIRTIO_SND_CTL_EVT_MASK_*):

VALUE means the element's value has changed.

INFO means the element's information has changed.

TLV means the element's metadata has changed.

5.14.6.10.4.1 Device Requirements: Control Element Notifications

- The device MUST NOT set undefined *mask* values.

5.15 Memory Device

The virtio memory device provides and manages a memory region in guest physical address space. This memory region is partitioned into memory blocks of fixed size that can either be in the state plugged or unplugged. Once plugged, a memory block can be used like ordinary RAM. The driver selects memory blocks to (un)plug and requests the device to perform the (un)plug.

The device requests the driver to plug a certain amount of memory, by setting the *requested_size* in the device configuration, which can change at runtime. It is up to the device driver to fulfill this request by (un)plugging memory blocks. Once the *plugged_size* is greater or equal to the *requested_size*, requests to plug memory blocks will be rejected by the device.

The device-managed memory region is split into two parts, the usable region and the unusable region. All memory blocks in the unusable region are unplugged and requests to plug them will be rejected. The device will grow the usable region to fit the *requested_size*. Usually, the usable region is bigger than the *requested_size* of the device, to give the driver some flexibility when selecting memory blocks to plug.

On initial start, and after a system reset, all memory blocks are unplugged. In corner cases, memory blocks might still be plugged after a system reset, and the driver usually requests to unplug all memory while initializing, before starting to select memory blocks to plug.

The device-managed memory region is not exposed as RAM via other firmware / hw interfaces (e.g., e820 on x86). The driver is responsible for deciding how plugged memory blocks will be used. A common use case is to expose plugged memory blocks to the operating system as system RAM, available for the page allocator.

Some platforms provide memory properties for system RAM that are usually queried and modified using special CPU instructions. Memory properties might be implicitly queried or modified on memory access. Memory properties can include advanced memory protection, access and change indication, or memory usage indication relevant in virtualized environments. [1][4] The device provides the exact same properties with the exact same semantics for plugged device memory as available for comparable RAM in the same configuration.

5.15.1 Device ID

24

5.15.2 Virtqueues

0 guest-request

5.15.3 Feature bits

VIRTIO_MEM_F_ACPI_PXM (0) The field *node_id* in the device configuration is valid and corresponds to an ACPI PXM.

VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE (1) The driver is not allowed to access unplugged memory. [1][5]

VIRTIO_MEM_F_PERSISTENT_SUSPEND (2) The driver can allow the guest to enter suspended state (deep sleep, suspend-to-RAM).

5.15.4 Device configuration layout

All fields of this configuration are always available and read-only for the driver.

1[4]For example, s390x provides storage keys for each 4 KiB page and may, depending on the configuration, provide storage attributes for each 4 KiB page.

1[5]On platforms with memory properties that might get modified implicitly on memory access, this feature is expected to be offered by the device.

```
struct virtio_mem_config {
  le64 block_size;
  le16 node_id;
  le8 padding[6];
  le64 addr;
  le64 region_size;
  le64 usable_region_size;
  le64 plugged_size;
  le64 requested_size;
};
```

block_size is the size and the alignment in bytes of a memory block. Cannot change.

node_id has no meaning without VIRTIO_MEM_F_ACPI_PXM. With VIRTIO_MEM_F_ACPI_PXM, this field is valid and corresponds to an ACPI PXM. Cannot change.

padding has no meaning and is reserved for future use.

addr is the guest physical address of the start of the device-managed memory region in bytes. Cannot change.

region_size is the size of device-managed memory region in bytes. Cannot change.

usable_region_size is the size of the usable device-managed memory region. Can grow up to *region_size*. Can only shrink due to VIRTIO_MEM_REQ_UNPLUG_ALL requests.

plugged_size is the amount of plugged memory in bytes within the usable device-managed memory region.

requested_size is the requested amount of plugged memory within the usable device-managed memory region.

5.15.4.1 Driver Requirements: Device configuration layout

The driver MUST NOT write to device configuration fields.

The driver MUST ignore the value of *padding*.

The driver MUST ignore the value of *node_id* without VIRTIO_MEM_F_ACPI_PXM.

5.15.4.2 Device Requirements: Device configuration layout

The device MAY change *usable_region_size* and *requested_size*.

The device MUST NOT change *block_size*, *node_id*, *addr*, and *region_size*, except during a system reset.

The device MUST change *plugged_size* to reflect the size of plugged memory blocks.

The device MUST set *usable_region_size* to *requested_size* or greater.

The device MUST set *block_size* to a power of two.

The device MUST set *addr*, *region_size*, *usable_region_size*, *plugged_size*, *requested_size* to multiples of *block_size*.

The device MUST set *region_size* to 0 or greater.

The device MUST NOT shrink *usable_region_size*, except when processing an UNPLUG ALL request, or during a system reset.

The device MUST send a configuration update notification when changing *usable_region_size* or *requested_size*, except when processing an UNPLUG ALL request.

The device SHOULD NOT send a configuration update notification when changing *plugged_size*.

The device MAY send a configuration update notification even if nothing changed.

5.15.5 Device Initialization

On initialization, the driver first discovers the device's virtqueues. It then reads the device configuration.

In case the driver detects that the device still has memory plugged (*plugged_size* in the device configuration is greater than 0), the driver will either try to re-initialize by issuing STATE requests, or try to unplug all memory before continuing. Special handling might be necessary in case some plugged memory might still be relevant (e.g., system dump, memory still in use after unloading the driver).

5.15.5.1 Driver Requirements: Device Initialization

The driver SHOULD accept VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE if it is offered and the driver supports it.

The driver SHOULD issue UNPLUG ALL requests until successful if the device still has memory plugged and the plugged memory is not in use.

5.15.5.2 Device Requirements: Device Initialization

A device MAY fail to operate further if VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE is not accepted.

The device MUST NOT change the state of memory blocks during device reset.

The device MUST NOT modify memory or memory properties of plugged memory blocks during device reset.

The device SHOULD offer VIRTIO_MEM_F_PERSISTENT_SUSPEND if the platform supports suspending (deep sleep, suspend-to-RAM) with plugged memory blocks.

5.15.6 Device Operation

The device notifies the driver about the amount of memory the device wants the driver to consume via the device. These resize requests from the device are communciated via the *requested_size* in the device configuration. The driver will react by requesting to (un)plug specific memory blocks, to make the *plugged_size* match the *requested_size* as close as possible.

The driver sends requests to the device on the guest-request virtqueue, notifies the device, and waits for the device to respond. Requests have a common header, defining the request type, followed by request-specific data. All requests are 24 bytes long and have the layout:

```
struct virtio_mem_req {
  le16 type;
  le16 padding[3];

  union {
    struct virtio_mem_req_plug plug;
    struct virtio_mem_req_unplug unplug;
    struct virtio_mem_req_state state;
  } u;
}
```

Possible request types are:

```
#define VIRTIO_MEM_REQ_PLUG            0
#define VIRTIO_MEM_REQ_UNPLUG          1
#define VIRTIO_MEM_REQ_UNPLUG_ALL      2
#define VIRTIO_MEM_REQ_STATE           3
```

Responses have a common header, defining the response type, followed by request-specific data. All responses are 10 bytes long and have the layout:

```
struct virtio_mem_resp {
  le16 type;
  le16 padding[3];

  union {
```

```
    struct virtio_mem_resp_state state;
  } u;
}
```

Possible response types, in general, are:

```
#define VIRTIO_MEM_RESP_ACK            0
#define VIRTIO_MEM_RESP_NACK           1
#define VIRTIO_MEM_RESP_BUSY           2
#define VIRTIO_MEM_RESP_ERROR          3
```

5.15.6.1 Driver Requirements: Device Operation

The driver MUST NOT write memory or modify memory properties of unplugged memory blocks.

The driver MUST NOT read memory or query memory properties of unplugged memory blocks outside *usable_region_size*.

The driver MUST NOT read memory or query memory properties of unplugged memory blocks inside *usable_region_size* via DMA.

If VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE has not been negotiated, the driver SHOULD NOT read memory or query memory properties of unplugged memory blocks inside *usable_region_size* via the CPU.

If VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE has been negotiated, the driver MUST NOT read memory or query memory properties of unplugged memory blocks.

The driver MUST NOT request unplug of memory blocks while corresponding memory or memory properties are still in use.

The driver SHOULD initialize memory blocks after plugging them, the content is undefined.

The driver SHOULD react to resize requests from the device (*requested_size* in the device configuration changed) by (un)plugging memory blocks.

The driver SHOULD only plug memory blocks it can actually use.

The driver MAY not reach the requested size (*requested_size* in the device configuration), for example, because it cannot free up any plugged memory blocks to unplug them, or it would not be able to make use of unplugged memory blocks after plugging them (e.g., alignment).

If VIRTIO_MEM_F_PERSISTENT_SUSPEND has not been negotiated, the driver MUST NOT allow the guest to enter a suspended state (deep sleep, suspend-to-RAM).

5.15.6.2 Device Requirements: Device Operation

The device MUST provide the exact same memory properties with the exact same semantics for device memory the platform provides in the same configuration for comparable RAM.

The device MAY modify memory of unplugged memory blocks or reset memory properties of such memory blocks to platform defaults at any time.

The device MUST NOT modify memory or memory properties of plugged memory blocks.

The device MUST allow the driver to read and write memory and to query and modify memory attributes of plugged memory blocks.

If VIRTIO_MEM_F_UNPLUGGED_INACCESSIBLE has not been negotiated, the device MUST allow the driver to read memory and to query memory properties of unplugged memory blocks inside *usable_region_size* via the CPU. [1][6]

The device MAY change the state of memory blocks during system resets.

The device SHOULD unplug all memory blocks during system resets.

[1][6] To allow for simplified dumping of memory. The CPU is expected to copy such memory to another location before starting DMA.

If VIRTIO_MEM_F_PERSISTENT_SUSPEND has been negotiated, the device MUST NOT not change the state of memory blocks when suspending or when waking up from suspended state (deep sleep, suspend-to-RAM).

5.15.6.3 PLUG request

Request to plug consecutive memory blocks that are currently unplugged.

The request-specific data in a PLUG request has the format:

```
struct virtio_mem_req_plug {
  le64 addr;
  le16 nb_blocks;
  le16 padding[3];
}
```

addr is the guest physical address of the first memory block. *nb_blocks* is the number of consecutive memory blocks

Responses don't have request-specific data defined.

5.15.6.3.1 Driver Requirements: PLUG request

The driver MUST ignore anything except the response type in a response.

5.15.6.3.2 Device Requirements: PLUG request

The device MUST ignore anything except the request type and the request-specific data in a request.

The device MUST ignore the *padding* in the request-specific data in a request.

The device MUST reject requests with VIRTIO_MEM_RESP_ERROR if *addr* is not aligned to the *block_size* in the device configuration, if *nb_blocks* is not greater than 0, or if any memory block outside of the usable device-managed memory region is covered by the request.

The device MUST reject requests with VIRTIO_MEM_RESP_ERROR if any memory block covered by the request is already plugged.

The device MAY reject requests with VIRTIO_MEM_RESP_BUSY if the request can currently not be processed.

The device MUST acknowledge requests with VIRTIO_MEM_RESP_ACK in case all memory blocks were successfully plugged. The device MUST reflect the change in the device configuration *plugged_size*.

5.15.6.4 UNPLUG request

Request to unplug consecutive memory blocks that are currently plugged.

The request-specific data in an UNPLUG request has the format:

```
struct virtio_mem_req_unplug {
  le64 addr;
  le16 nb_blocks;
  le16 padding[3];
}
```

addr is the guest physical address of the first memory block. *nb_blocks* is the number of consecutive memory blocks

Responses don't have request-specific data defined.

5.15.6.4.1 Driver Requirements: UNPLUG request

The driver MUST ignore anything except the response type in a response.

5.15.6.4.2 Device Requirements: UNPLUG request

The device MUST ignore anything except the request type and the request-specific data in a request.

The device MUST ignore the *padding* in the request-specific data in a request.

The device MUST reject requests with VIRTIO_MEM_RESP_ERROR if *addr* is not aligned to the *block_size* in the device configuration, if *nb_blocks* is not greater than 0, or if any memory block outside of the usable device-managed memory region is covered by the request.

The device MUST reject requests with VIRTIO_MEM_RESP_ERROR if any memory block covered by the request is already unplugged.

The device MAY reject requests with VIRTIO_MEM_RESP_BUSY if the request can currently not be processed.

The device MUST acknowledge requests with VIRTIO_MEM_RESP_ACK in case all memory blocks were successfully unplugged. The device MUST reflect the change in the device configuration *plugged_size*.

5.15.6.5 UNPLUG ALL request

Request to unplug all memory blocks the device has currently plugged. If successful, the *plugged_size* in the device configuration will be 0 and *usable_region_size* might have changed.

Requests don't have request-specific data defined, only the request type is relevant. Responses don't have request-specific data defined, only the response type is relevant.

5.15.6.5.1 Driver Requirements: UNPLUG request

The driver MUST ignore any data in a response except the response type.

5.15.6.5.2 Device Requirements: UNPLUG request

The device MUST ignore any data in a request except the request type.

The device MUST ignore the *padding* in the request-specific data in a request.

The device MAY reject requests with VIRTIO_MEM_RESP_BUSY if the request can currently not be processed.

The device MUST acknowledge requests with VIRTIO_MEM_RESP_ACK in case all memory blocks were successfully unplugged.

The device MUST set *plugged_size* to 0 in case the request is acknowledged with VIRTIO_MEM_RESP_ACK.

The device MAY modify *usable_region_size* before responding with VIRTIO_MEM_RESP_ACK.

5.15.6.6 STATE request

Request the state (plugged, unplugged, mixed) of consecutive memory blocks.

The request-specific data in a STATE request has the format:

```
struct virtio_mem_req_state {
  le64 addr;
  le16 nb_blocks;
  le16 padding[3];
};
```

addr is the guest physical address of the first memory block. *nb_blocks* is the number of consecutive memory blocks.

The request-specific data in a STATE response has the format:

```
struct virtio_mem_resp_state {
  le16 type;
};
```

Whereby *type* defines one of three different state types:

```
#define VIRTIO_MEM_STATE_PLUGGED        0
#define VIRTIO_MEM_STATE_UNPLUGGED      1
#define VIRTIO_MEM_STATE_MIXED          2
```

5.15.6.6.1 Driver Requirements: STATE request

The driver MUST ignore anything except the response type and the request-specific data in a response.

The driver MUST ignore the request-specific data in a response in case the response type is not VIRTIO_-MEM_RESP_ACK.

5.15.6.6.2 Device Requirements: STATE request

The device MUST ignore anything except the request type and the request-specific data in a request.

The device MUST ignore the *padding* in the request-specific data in a request.

The device MUST reject requests with VIRTIO_MEM_RESP_ERROR if *addr* is not aligned to the *block_size* in the device configuration, if *nb_blocks* is not greater than 0, or if any memory block outside of the usable device-managed memory region is covered by the request.

The device MUST acknowledge requests with VIRTIO_MEM_RESP_ACK, supplying the state of the memory blocks.

The device MUST set the state type in the response to VIRTIO_MEM_STATE_PLUGGED if all requested memory blocks are plugged. The device MUST set the state type in the response to VIRTIO_MEM_STATE_-UNPLUGGED if all requested memory blocks are unplugged. Otherwise, the device MUST set state type in the response to VIRTIO_MEM_STATE_MIXED.

5.16 I2C Adapter Device

virtio-i2c is a virtual I2C adapter device. It provides a way to flexibly organize and use the host I2C controlled devices from the guest. By attaching the host ACPI I2C controlled nodes to the virtual I2C adapter device, the guest can communicate with them without changing or adding extra drivers for these controlled I2C devices.

5.16.1 Device ID

34

5.16.2 Virtqueues

0 requestq

5.16.3 Feature bits

VIRTIO_I2C_F_ZERO_LENGTH_REQUEST (0) The device supports zero-length I2C request and *VIRTIO_I2C_FLAGS_M_RD* flag. It is mandatory to implement this feature.

Note: The *VIRTIO_I2C_FLAGS_M_RD* flag was not present in the initial implementation of the specification and the direction of the transfer (read or write) was inferred from the permissions (read-only or write-only) of the buffer itself. There is no need of having backwards compatibility for the older specification and so the *VIRTIO_I2C_FLAGS_FAIL_NEXT* feature is made mandatory. The driver should abort negotiation with the device, if the device doesn't offer this feature.

5.16.4 Device configuration layout

None currently defined.

5.16.5 Device Initialization

1. The virtqueue is initialized.

5.16.6 Device Operation

5.16.6.1 Device Operation: Request Queue

The driver enqueues requests to the virtqueue, and they are used by the device. The request is the representation of segments of an I2C transaction. Each request is of the form:

```
struct virtio_i2c_out_hdr {
        le16 addr;
        le16 padding;
        le32 flags;
};
```

```
struct virtio_i2c_in_hdr {
        u8 status;
};
```

```
struct virtio_i2c_req {
        struct virtio_i2c_out_hdr out_hdr;
        u8 buf[];
        struct virtio_i2c_in_hdr in_hdr;
};
```

The *addr* of the request is the address of the I2C controlled device. For 7-bit address mode (A0 ... A6) and 10-bit address mode (A0 ... A9), the format of *addr* is defined as follows:

Bits	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
7-bit address	0	0	0	0	0	0	0	0	A6	A5	A4	A3	A2	A1	A0	0
10-bit address	A7	A6	A5	A4	A3	A2	A1	A0	1	1	1	1	0	A9	A8	0

The *padding* is used to pad to full dword.

The *flags* of the request is defined as follows:

VIRTIO_I2C_FLAGS_FAIL_NEXT(0) is used to group the requests. For a group requests, a driver clears this bit on the final request and sets it on the other requests. If this bit is set and a device fails to process the current request, it needs to fail the next request instead of attempting to execute it.

VIRTIO_I2C_FLAGS_M_RD(1) is used to mark the request as READ or WRITE. If *VIRTIO_I2C_FLAGS_M_RD* bit is set in the *flags*, then the request is called a read request. If *VIRTIO_I2C_FLAGS_M_RD* bit is unset in the *flags*, then the request is called a write request.

Other bits of *flags* are currently reserved as zero for future feature extensibility.

The *buf* is optional and will not be present for a zero-length request, like the SMBus ”Quick” command. The *buf* contains one segment of an I2C transaction being read from or written to the device, based on the value of the *VIRTIO_I2C_FLAGS_M_RD* bit in the *flags* field.

The final *status* byte of the request is written by the device: either VIRTIO_I2C_MSG_OK for success or VIRTIO_I2C_MSG_ERR for error.

```
#define VIRTIO_I2C_MSG_OK     0
#define VIRTIO_I2C_MSG_ERR    1
```

The virtio I2C protocol supports write-read requests, i.e. an I2C write segment followed by a read segment (usually, the write segment provides the number of an I2C controlled device register to be read), by grouping a list of requests together using the *VIRTIO_I2C_FLAGS_FAIL_NEXT* flag.

5.16.6.2 Device Operation: Operation Status

addr, *flags*, and “length of *buf*” are determined by the driver, while *status* is determined by the processing of the device. A driver, for a write request, puts the data to be written to the device into the *buf*, while a device, for a read request, puts the data read from device into the *buf* according to the request from the driver.

A driver may send one request or multiple requests to the device at a time. The requests in the virtqueue are both queued and processed in order.

If a driver sends multiple requests at a time and a device fails to process some of them, then a device needs to set the *status* of the first failed request to be VIRTIO_I2C_MSG_ERR. For the remaining requests in the same group with the first failed one, a driver needs to treat them as VIRTIO_I2C_MSG_ERR, no matter what *status* of them, a device needs to fail them instead of attempting to execute them according to the VIRTIO_I2C_FLAGS_FAIL_NEXT bit.

5.16.6.3 Driver Requirements: Device Operation

A driver MUST accept the *VIRTIO_I2C_F_ZERO_LENGTH_REQUEST* feature and MUST abort negotiation with the device, if the device doesn’t offer this feature.

A driver MUST set *addr* and *flags* before sending the request.

A driver MUST set the reserved bits of *flags* to be zero.

A driver MUST NOT send the *buf*, for a zero-length request.

A driver MUST NOT use *buf*, for a read request, if the final *status* returned from the device is VIRTIO_I2C_-MSG_ERR.

A driver MUST set the *VIRTIO_I2C_FLAGS_M_RD* flag for a read operation, where the buffer is write-only for the device.

A driver MUST NOT set the *VIRTIO_I2C_FLAGS_M_RD* flag for a write operation, where the buffer is read-only for the device.

A driver MUST queue the requests in order if multiple requests are going to be sent at a time.

If multiple requests are sent at a time and some of them returned from the device have the *status* being VIRTIO_I2C_MSG_ERR, a driver MUST treat the first failed request and the remaining requests in the same group with the first failed one as VIRTIO_I2C_MSG_ERR.

5.16.6.4 Device Requirements: Device Operation

A device MUST offer the *VIRTIO_I2C_F_ZERO_LENGTH_REQUEST* feature and MUST reject any driver that doesn’t negotiate this feature.

A device SHOULD keep consistent behaviors with the hardware as described in I2C.

A device MUST NOT change the value of *addr*, and reserved bits of *flags*.

A device MUST not change the value of the *buf* for a write request.

A device MUST place one I2C segment of the “length of *buf*”, for the read request, into the *buf* according the driver’s request.

A device MUST guarantee the requests in the virtqueue being processed in order if multiple requests are received at a time.

If multiple requests are received at a time and processing of some of the requests fails, a device MUST set the *status* of the first failed request to be VIRTIO_I2C_MSG_ERR and MAY set the *status* of the remaining requests in the same group with the first failed one to be VIRTIO_I2C_MSG_ERR.

5.17 SCMI Device

An SCMI device implements the Arm System Control and Management Interface (SCMI). SCMI can be used for sensors, power state management, clock management and performance management among other things.

This section relies on definitions from the SCMI specification.

Virtio SCMI device and driver are mapped to SCMI platform and agent respectively. The device is visible to a particular SCMI agent. The device allows a guest to communicate as an SCMI agent using one or more SCMI protocols. The default SCMI protocols are defined in the SCMI specification. Virtio provides a transport medium for exchanging SCMI messages between the SCMI agent and platform. The virtio SCMI transport allows the queueing of multiple messages and responses.

SCMI FastChannels are not supported.

5.17.1 Device ID

32

5.17.2 Virtqueues

0 cmdq

1 eventq

The cmdq is used by the driver to send commands to the device. The device replies with responses (not delayed responses) over the cmdq.

The eventq is used by the device to send notifications and delayed responses. The eventq only exists if VIRTIO_SCMI_F_P2A_CHANNELS was negotiated.

5.17.3 Feature bits

VIRTIO_SCMI_F_P2A_CHANNELS (0) Device implements some SCMI notifications, or delayed responses.

VIRTIO_SCMI_F_SHARED_MEMORY (1) Device implements any SCMI statistics shared memory region.

VIRTIO_SCMI_F_P2A_CHANNELS is used to determine the existence of the eventq. The eventq is required for SCMI notifications and delayed responses.

VIRTIO_SCMI_F_SHARED_MEMORY is used to determine whether the device provides any SCMI statistics shared memory region. SCMI statistics shared memory regions are defined by some SCMI protocols.

The SCMI protocols provide the PROTOCOL_MESSAGE_ATTRIBUTES commands to inquire about the particular SCMI notifications and delayed responses implemented by the device. The SCMI protocols provide additional commands to detect other features implemented by the device.

5.17.3.1 Device Requirements: Feature bits

The device MUST offer VIRTIO_SCMI_F_P2A_CHANNELS if the device can implement at least one SCMI notification, or delayed response.

The device MUST offer VIRTIO_SCMI_F_SHARED_MEMORY if the device can implement at least one SCMI statistics shared memory region.

5.17.4 Device configuration layout

There is no configuration data for the device.

5.17.5 Device Initialization

The general requirements on device initialization apply.

5.17.6 Device Operation

The SCMI transport used for the device puts each SCMI message into a dedicated virtio buffer. The driver uses the cmdq for transmitting SCMI commands and receiving the corresponding SCMI responses. The device uses the eventq for transmitting SCMI notifications and delayed responses. Each message includes an SCMI protocol header and payload, as defined by the SCMI specification.

5.17.6.1 cmdq Operation

Each buffer in the cmdq holds a single SCMI command once the buffer has been made available. When the buffer has been marked as used, it contains the SCMI response. An arbitrary number of such SCMI messages can be in transit at the same time. Conceptually, each SCMI message in the cmdq uses its own SCMI A2P (agent to platform) channel.

The SCMI response is in the same virtio buffer as the corresponding SCMI command. The response contains the return values which SCMI specifies for each command, whether synchronous or asynchronous. Delayed responses are distinct SCMI messages transmitted over the eventq.

Buffers in the cmdq contain both the request and the response. A request has the following layout:

```
struct virtio_scmi_request {
        le32 hdr;
        u8 params[<actual parameters size>];
};
```

The virtio_scmi_request fields are interpreted as follows:

hdr (device-readable) contains the SCMI message header

params (device-readable) comprises the SCMI message parameters

A cmdq response has the following layout:

```
struct virtio_scmi_response {
        le32 hdr;
        u8 ret_values[<actual return values size>];
};
```

The virtio_scmi_response fields are interpreted as follows:

hdr (device-writable) contains the SCMI message header

ret_values (device-writable) comprises the SCMI message return values

If VIRTIO_SCMI_F_P2A_CHANNELS was not negotiated, the device responds to SCMI commands as if no SCMI notifications or delayed responses were implemented.

5.17.6.1.1 Device Requirements: cmdq Operation

The device MAY process available commands out of order and in parallel.

The device MUST process all available commands eventually, even in the case of bursts of multiple command messages.

If the *status* field in the *virtio_scmi_response ret_values* has a value other than SUCCESS, the device MUST set the size of *ret_values* to the size of the *status* field.

If the driver requests an SCMI notification or a delayed response and there are currently NOT enough available buffers in the eventq, the device SHOULD still return SCMI status code SUCCESS.

If VIRTIO_SCMI_F_P2A_CHANNELS was not negotiated, the device MUST deny any request for an SCMI notification or a delayed response by returning SCMI status code NOT_SUPPORTED.

If VIRTIO_SCMI_F_P2A_CHANNELS was not negotiated, the device MUST NOT indicate in the PROTOCOL_MESSAGE_ATTRIBUTES return values that any SCMI notification, or delayed response, is implemented.

5.17.6.1.2 Driver Requirements: cmdq Operation

Before sending a command, the driver MUST wait for responses to all commands whose completion the driver considers prerequisites to executing the command.

With every command message, the driver MUST provide enough device-writable memory to enable the device to return corresponding return values.

If VIRTIO_SCMI_F_P2A_CHANNELS was not negotiated, the driver MUST NOT request any SCMI notification, nor any delayed response.

5.17.6.2 Setting Up eventq Buffers

The driver has to populate the eventq before the device can use it.

5.17.6.2.1 Driver Requirements: Setting Up eventq Buffers

If VIRTIO_SCMI_F_P2A_CHANNELS was negotiated, the driver SHOULD populate the eventq with buffers.

The driver MUST NOT put device-readable descriptors into the eventq.

The driver MUST NOT put into the eventq any buffer which is smaller than the largest SCMI P2A (platform to agent) message which the driver will request.

5.17.6.3 eventq Operation

Each buffer in the eventq holds (once the buffer is marked as used) either a single SCMI notification, or a single SCMI delayed response. An arbitrary number of such SCMI messages can be in transit at the same time. Conceptually, each SCMI message transmitted over the eventq uses its own SCMI P2A (platform to agent) channel. Buffers in the eventq have the following layout:

```
struct virtio_scmi_event_msg {
        /* start of device-writable data */
        le32 hdr;
        u8 payload[<actual payload size>];
};
```

hdr (device-writable) contains the SCMI message header

payload (device-writable) comprises the SCMI message payload

5.17.6.3.1 Device Requirements: eventq Operation

If the device intends to send a notification and there are no available buffers in the eventq, the device MAY drop the notification, or send a corresponding notification later, once enough buffers become available.

The device MAY send the notification later if the events which cause the notification take place in quick succession.

If the device sends the notification later, the device MAY send the notification with updated data, unless the specific SCMI protocol disallows this.

If the device intends to send a notification and there are available buffers, but one of the buffers is too small to fit the notification, the device MAY omit the notification.

If the device intends to send a delayed response and there are no available buffers in the eventq, the device MUST send the corresponding delayed response once enough buffers become available.

If the *status* field in a delayed response *payload* has a value other than SUCCESS, the device MUST set the size of *payload* to the size of the *status* field.

5.17.6.4 Shared Memory Operation

Various SCMI protocols define statistics shared memory regions (for statistics and sensor values).

5.17.6.4.1 Device Requirements: Shared Memory Operation

If VIRTIO_SCMI_F_SHARED_MEMORY was negotiated, the device MAY implement an SCMI statistics shared memory region using a virtio shared memory region.

If the device implements a shared memory region, the device MUST assign the corresponding shmid as per the following table:

SCMI statistics shared memory region	Virtio shmid
Reserved (invalid)	0
Power state statistics shared memory region	1
Performance domain statistics shared memory region	2
Sensor Values Shared Memory	3
Reserved for future use	4 to 0x7F
Vendor-specific statistics shared memory regions	0x80 to 0xFF
Reserved for future use	0x100 and greater

5.18 GPIO Device

The Virtio GPIO device is a virtual General Purpose Input/Output device that supports a variable number of named I/O lines, which can be configured in input mode or in output mode with logical level low (0) or high (1).

5.18.1 Device ID

41

5.18.2 Virtqueues

0 requestq

1 eventq

The *eventq* virtqueue is available only if the *VIRTIO_GPIO_F_IRQ* feature has been negotiated.

5.18.3 Feature bits

VIRTIO_GPIO_F_IRQ (0) The device supports interrupts on GPIO lines.

5.18.4 Device configuration layout

GPIO device uses the following configuration structure layout:

```
struct virtio_gpio_config {
    le16 ngpio;
    u8 padding[2];
    le32 gpio_names_size;
};
```

ngpio is the total number of GPIO lines supported by the device.

padding has no meaning and is reserved for future use. This is set to zero by the device.

gpio_names_size is the size of the gpio-names memory block in bytes, which can be fetched by the driver using the *VIRTIO_GPIO_MSG_GET_LINE_NAMES* message. The device sets this to 0 if it doesn't support names for the GPIO lines.

5.18.5 Device Initialization

- The driver configures and initializes the *requestq* virtqueue.
- The driver configures and initializes the *eventq* virtqueue, if the *VIRTIO_GPIO_F_IRQ* feature has been negotiated.

5.18.6 Device Operation: requestq

The driver uses the *requestq* virtqueue to send messages to the device. The driver sends a pair of buffers, request (filled by driver) and response (to be filled by device later), to the device. The device in turn fills the response buffer and sends it back to the driver.

```
struct virtio_gpio_request {
    le16 type;
    le16 gpio;
    le32 value;
};
```

All the fields of this structure are set by the driver and read by the device.

type is the GPIO message type, i.e. one of *VIRTIO_GPIO_MSG_** values.

gpio is the GPIO line number, i.e. 0 <= *gpio* < *ngpio*.

value is a message specific value.

```
struct virtio_gpio_response {
    u8 status;
    u8 value;
};

/* Possible values of the status field */
#define VIRTIO_GPIO_STATUS_OK                   0x0
#define VIRTIO_GPIO_STATUS_ERR                  0x1
```

All the fields of this structure are set by the device and read by the driver.

status of the GPIO message, *VIRTIO_GPIO_STATUS_OK* on success and *VIRTIO_GPIO_STATUS_ERR* on failure.

value is a message specific value.

Following is the list of messages supported by the virtio gpio specification.

```
/* GPIO message types */
#define VIRTIO_GPIO_MSG_GET_LINE_NAMES          0x0001
#define VIRTIO_GPIO_MSG_GET_DIRECTION           0x0002
#define VIRTIO_GPIO_MSG_SET_DIRECTION           0x0003
#define VIRTIO_GPIO_MSG_GET_VALUE               0x0004
#define VIRTIO_GPIO_MSG_SET_VALUE               0x0005
#define VIRTIO_GPIO_MSG_SET_IRQ_TYPE            0x0006

/* GPIO Direction types */
#define VIRTIO_GPIO_DIRECTION_NONE              0x00
#define VIRTIO_GPIO_DIRECTION_OUT               0x01
#define VIRTIO_GPIO_DIRECTION_IN                0x02

/* GPIO interrupt types */
#define VIRTIO_GPIO_IRQ_TYPE_NONE               0x00
#define VIRTIO_GPIO_IRQ_TYPE_EDGE_RISING        0x01
#define VIRTIO_GPIO_IRQ_TYPE_EDGE_FALLING       0x02
#define VIRTIO_GPIO_IRQ_TYPE_EDGE_BOTH          0x03
#define VIRTIO_GPIO_IRQ_TYPE_LEVEL_HIGH         0x04
#define VIRTIO_GPIO_IRQ_TYPE_LEVEL_LOW          0x08
```

5.18.6.1 requestq Operation: Get Line Names

The driver sends this message to receive a stream of zero-terminated strings, where each string represents the name of a GPIO line, present in increasing order of the GPIO line numbers. The names of the GPIO lines are optional and may be present only for a subset of GPIO lines. If missing, then a zero-byte must be present for the GPIO line. If present, the name string must be zero-terminated and the name must be unique within a GPIO Device. The names of the GPIO lines are encoded in 7-bit ASCII.

These names of the GPIO lines should be most meaningful producer names for the system, such as name indicating the usage. For example ”MMC-CD”, ”Red LED Vdd” and ”ethernet reset” are reasonable line names as they describe what the line is used for, while ”GPIO0” is not a good name to give to a GPIO line.

Here is an example of how the gpio names memory block may look like for a GPIO device with 10 GPIO lines, where line names are provided only for lines 0 (”MMC-CD”), 5 (”Red LED Vdd”) and 7 (”ethernet reset”).

```
u8 gpio_names[] = {
    'M', 'M', 'C', '-', 'C', 'D', 0,
    0,
    0,
    0,
    0,
    'R', 'e', 'd', ' ', 'L', 'E', 'D', ' ', 'V', 'd', 'd', 0,
    0,
    'E', 't', 'h', 'e', 'r', 'n', 'e', 't', ' ', 'r', 'e', 's', 'e', 't', 0,
    0,
    0
};
```

The device sets the *gpio_names_size* to a non-zero value if this message is supported by the device, else it must be set to zero.

This message type uses different layout for the response structure, as the device needs to return the *gpio_names* array.

```
struct virtio_gpio_response_N {
    u8 status;
    u8 value[N];
};
```

The driver must allocate the *value[N]* buffer in the *struct virtio_gpio_response_N* for N bytes, where N = *gpio_names_size*.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_GET_LINE_NAMES	0	0

Response	*status*	*value[N]*	*Where N is*
	*VIRTIO_GPIO_STATUS_**	gpio-names	*gpio_names_size*

5.18.6.2 requestq Operation: Get Direction

The driver sends this message to request the device to return a line’s configured direction.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_GET_DIRECTION	line number	0

Response	*status*	*value*

	*VIRTIO_GPIO_STATUS_**	0 = none, 1 = output, 2 = input

5.18.6.3 requestq Operation: Set Direction

The driver sends this message to request the device to configure a line's direction. The driver can either set the direction to *VIRTIO_GPIO_DIRECTION_IN* or *VIRTIO_GPIO_DIRECTION_OUT*, which also activates the line, or to *VIRTIO_GPIO_DIRECTION_NONE*, which deactivates the line.

The driver should set the value of the GPIO line, using the *VIRTIO_GPIO_MSG_SET_VALUE* message, before setting the direction of the line to output to avoid any undesired behavior.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_SET_DIRECTION	line number	0 = none, 1 = output, 2 = input

Response	*status*	*value*
	*VIRTIO_GPIO_STATUS_**	0

5.18.6.4 requestq Operation: Get Value

The driver sends this message to request the device to return current value sensed on a line.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_GET_VALUE	line number	0

Response	*status*	*value*
	*VIRTIO_GPIO_STATUS_**	0 = low, 1 = high

5.18.6.5 requestq Operation: Set Value

The driver sends this message to request the device to set the value of a line. The line may already be configured for output or may get configured to output later, at which point this output value must be used by the device for the line.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_SET_VALUE	line number	0 = low, 1 = high

Response	*status*	*value*
	*VIRTIO_GPIO_STATUS_**	0

5.18.6.6 requestq Operation: Set IRQ Type

This request is allowed only if the *VIRTIO_GPIO_F_IRQ* feature has been negotiated.

The interrupt configuration is divided into two steps, enabling or disabling of the interrupt at the device and masking or unmasking of the interrupt for delivery at the driver. This request only pertains to enabling or disabling of the interrupt at the device, the masking and unmasking of the interrupt is handled by a separate request that takes place over the *eventq* virtqueue.

The driver sends the *VIRTIO_GPIO_MSG_SET_IRQ_TYPE* message over the *requestq* virtqueue to enable or disable interrupt for a GPIO line at the device.

The driver sends this message with trigger type set to any valid value other than *VIRTIO_GPIO_IRQ_-TYPE_NONE*, to enable the interrupt for a GPIO line, this doesn't unmask the interrupt for delivery at the driver though. For edge trigger type, the device should latch the interrupt events from this point onward and notify it to the driver once the interrupt is unmasked. For level trigger type, the device should notify the driver once the interrupt signal on a line is sensed and the interrupt is unmasked for the line.

The driver sends this message with trigger type set to *VIRTIO_GPIO_IRQ_TYPE_NONE*, to disable the interrupt for a GPIO line. The device should discard any latched interrupt event associated with it. In order to change the trigger type of an already enabled interrupt, the driver should first disable the interrupt and then re-enable it with appropriate trigger type.

The interrupts are masked at initialization and the driver unmasks them by queuing a pair of buffers, of type *virtio_gpio_irq_request* and *virtio_gpio_irq_response*, over the *eventq* virtqueue for a GPIO line. A separate pair of buffers must be queued for each GPIO line, the driver wants to configure for interrupts. Once an already enabled interrupt is unmasked by the driver, the device can notify the driver of an active interrupt signal on the GPIO line. This is done by updating the *struct virtio_gpio_irq_response* buffer's *status* with *VIRTIO_GPIO_IRQ_STATUS_VALID* and returning the updated buffers to the driver. The interrupt is masked automatically at this point until the buffers are available again at the device.

The interrupt for a GPIO line should not be unmasked before being enabled by the driver.

If the interrupt is disabled by the driver, by setting the trigger type to *VIRTIO_GPIO_IRQ_TYPE_NONE*, or the interrupt is unmasked without being enabled first, the device should return any unused pair of buffers for the GPIO line, over the *eventq* virtqueue, after setting the *status* field to *VIRTIO_GPIO_IRQ_STATUS_-INVALID*. This also masks the interrupt.

Request	*type*	*gpio*	*value*
	VIRTIO_GPIO_MSG_SET_-IRQ_TYPE	line number	one of *VIRTIO_GPIO_IRQ_-TYPE_**

Response	*status*	*value*
	*VIRTIO_GPIO_STATUS_**	0

5.18.6.7 requestq Operation: Message Flow

- The driver enqueues *struct virtio_gpio_request* and *virtio_gpio_response* buffers to the *requestq* virtqueue, after filling all fields of the *struct virtio_gpio_request* buffer as defined by the specific message type.
- The driver notifies the device of the presence of buffers on the *requestq* virtqueue.
- The device, after receiving the message from the driver, processes it and fills all the fields of the *struct virtio_gpio_response* buffer (received from the driver). The *status* must be set to *VIRTIO_GPIO_-STATUS_OK* on success and *VIRTIO_GPIO_STATUS_ERR* on failure.
- The device puts the buffers back on the *requestq* virtqueue and notifies the driver of the same.
- The driver fetches the buffers and processes the response received in the *virtio_gpio_response* buffer.
- The driver can send multiple messages in parallel for same or different GPIO line.

5.18.6.8 Driver Requirements: requestq Operation

- The driver MUST send messages on the *requestq* virtqueue.
- The driver MUST queue both *struct virtio_gpio_request* and *virtio_gpio_response* for every message sent to the device.
- The *struct virtio_gpio_request* buffer MUST be filled by the driver and MUST be read-only for the device.
- The *struct virtio_gpio_response* buffer MUST be filled by the device and MUST be writable by the device.

- The driver MAY send multiple messages for same or different GPIO lines in parallel.
- The driver MUST NOT send IRQ messages if the *VIRTIO_GPIO_F_IRQ* feature has not been negotiated.
- The driver MUST NOT send IRQ messages for a GPIO line configured for output.
- The driver MUST set the IRQ trigger type to *VIRTIO_GPIO_IRQ_TYPE_NONE* once it is done using the GPIO line configured for interrupts.
- In order to change the trigger type of an already enabled interrupt, the driver MUST first disable the interrupt and then re-enable it with appropriate trigger type.

5.18.6.9 Device Requirements: requestq Operation

- The device MUST set all the fields of the *struct virtio_gpio_response* before sending it back to the driver.
- The device MUST set all the fields of the *struct virtio_gpio_config* on receiving a configuration request from the driver.
- The device MUST set the *gpio_names_size* field as zero in the *struct virtio_gpio_config*, if it doesn't implement names for individual GPIO lines.
- The device MUST set the *gpio_names_size* field, in the *struct virtio_gpio_config*, with the size of gpio-names memory block in bytes, if the device implements names for individual GPIO lines. The strings MUST be zero-terminated and an unique (if available) within the GPIO device.
- The device MUST process multiple messages, for the same GPIO line, sequentially and respond to them in the order they were received on the virtqueue.
- The device MAY process messages, for different GPIO lines, out of order and in parallel, and MAY send message's response to the driver out of order.
- The device MUST discard all state information corresponding to a GPIO line, once the driver has requested to set its direction to *VIRTIO_GPIO_DIRECTION_NONE*.
- The device MUST latch an edge interrupt if the interrupt is enabled but still masked.
- The device MUST NOT latch an level interrupt if the interrupt is enabled but still masked.
- The device MUST discard any latched interrupt for a GPIO line, once interrupt is disabled for the same.

5.18.7 Device Operation: eventq

The *eventq* virtqueue is used by the driver to unmask the interrupts and used by the device to notify the driver of newly sensed interrupts. In order to unmask interrupt on a GPIO line, the driver enqueues a pair of buffers, *struct virtio_gpio_irq_request* (filled by driver) and *struct virtio_gpio_irq_response* (to be filled by device later), to the *eventq* virtqueue. A separate pair of buffers must be queued for each GPIO line, the driver wants to configure for interrupts. The device, on sensing an interrupt, returns the pair of buffers for the respective GPIO line, which also masks the interrupts. The driver can queue the buffers again to unmask the interrupt.

```
struct virtio_gpio_irq_request {
    le16 gpio;
};
```

This structure is filled by the driver and read by the device.

gpio is the GPIO line number, i.e. 0 <= *gpio* < *ngpio*.

```
struct virtio_gpio_irq_response {
    u8 status;
};

/* Possible values of the interrupt status field */
#define VIRTIO_GPIO_IRQ_STATUS_INVALID          0x0
```

```
#define VIRTIO_GPIO_IRQ_STATUS_VALID            0x1
```

This structure is filled by the device and read by the driver.

status of the interrupt event, *VIRTIO_GPIO_IRQ_STATUS_VALID* on interrupt and *VIRTIO_GPIO_IRQ_STATUS_INVALID* to return the buffers back to the driver after interrupt is disabled.

5.18.7.1 eventq Operation: Message Flow

- The virtio-gpio driver is requested by a client driver to enable interrupt for a GPIO line and configure it to a particular trigger type.
- The driver sends the *VIRTIO_GPIO_MSG_SET_IRQ_TYPE* message, over the *requestq* virtqueue, and the device configures the GPIO line for the requested trigger type and enables the interrupt. The interrupt is still masked for delivery though. The device shall latch the interrupt from now onward for edge trigger type.
- The driver unmasks the interrupt by queuing a pair of buffers to the *eventq* virtqueue for the GPIO line. The driver can do this before enabling the interrupt as well, though the interrupt must be both unmasked and enabled to get delivered at the driver.
- The driver notifies the device of the presence of new buffers on the *eventq* virtqueue. The interrupt is fully configured at this point.
- The device, on sensing an active interrupt on the GPIO line, finds the matching buffers (based on GPIO line number) from the *eventq* virtqueue and update its *struct virtio_gpio_irq_response* buffer's *status* with *VIRTIO_GPIO_IRQ_STATUS_VALID* and returns the pair of buffers to the device. This results in masking the interrupt as well.
- The device notifies the driver of the presence of returned buffers on the *eventq* virtqueue.
- If the GPIO line is configured for level interrupts, the device ignores any further interrupt signals on this GPIO line, until the interrupt is unmasked again by the driver by making the buffers available to the device. Once the interrupt is unmasked again and the interrupt on the line is still active, the device shall notify the driver again.
- If the GPIO line is configured for edge interrupts, the device latches the interrupt received for this GPIO line, until the interrupt is unmasked again by the driver by making the buffers available to the device. Once the interrupt is unmasked again and an interrupt was latched earlier, the device shall notify the driver again.
- The driver on receiving the notification from the device, processes the interrupt. The interrupt is masked at the device until the buffers are queued again by the driver.
- In a typical guest operating system kernel, the virtio-gpio driver notifies the client driver, that is associated with this GPIO line, to process the event. In the case of a level triggered interrupt, the client driver shall fully process and acknowledge the event at its source to return the line to its inactive state before the interrupt is unmasked again to avoid a spurious interrupt.
- Once the interrupt is handled, the driver may queue a pair of buffers for the GPIO line to unmask the interrupt again.
- The driver can also disable the interrupt by sending the *VIRTIO_GPIO_MSG_SET_IRQ_TYPE* message, with *VIRTIO_GPIO_IRQ_TYPE_NONE* trigger type. In that case, the device shall return the unused pair of buffers for the GPIO line after setting the *status* field with *VIRTIO_GPIO_IRQ_STATUS_INVALID*.

5.18.7.2 Driver Requirements: eventq Operation

- The driver MUST both enable and unmask the interrupt in order to get notified for the same.
- The driver MUST enable the interrupt before unmasking it.

- To unmask the interrupt, the driver MUST queue a separate pair of buffers to the *eventq* virtqueue for each GPIO line.
- The driver MUST NOT add multiple pairs of buffers for the same GPIO line on the *eventq* virtqueue.

5.18.7.3 Device Requirements: eventq Operation

- The device MUST NOT send an interrupt event to the driver for a GPIO line unless the interrupt has been both unmasked and enabled by the driver.
- On receiving *VIRTIO_GPIO_MSG_SET_IRQ_TYPE* message, with *VIRTIO_GPIO_IRQ_TYPE_NONE* trigger type, the device MUST return the buffers, if they were received earlier, after setting the *status* field to *VIRTIO_GPIO_IRQ_STATUS_INVALID*.

5.19 PMEM Device

The virtio pmem device is a persistent memory (NVDIMM) device that provides a virtio based asynchronous flush mechanism. This avoids the need for a separate page cache in the guest and keeps the page cache only in the host. Under memory pressure, the host makes use of efficient memory reclaim decisions for page cache pages of all the guests. This helps to reduce the memory footprint and fits more guests in the host system.

The virtio pmem device provides access to byte-addressable persistent memory. The persistent memory is a directly accessible range of system memory. Data written to this memory is made persistent by separately sending a flush command. Writes that have been flushed are preserved across device reset and power failure.

5.19.1 Device ID

27

5.19.2 Virtqueues

0 req_vq

5.19.3 Feature bits

VIRTIO_PMEM_F_SHMEM_REGION (0) The guest physical address range will be indicated as a shared memory region.

5.19.4 Device configuration layout

```
struct virtio_pmem_config {
  le64 start;
  le64 size;
};
```

start contains the physical address of the first byte of the persistent memory region, if VIRTIO_PMEM_F_-SHMEM_REGION has not been negotiated.

size contains the length of this address range, if VIRTIO_PMEM_F_SHMEM_REGION has not been negotiated.

5.19.5 Device Initialization

The device indicates the guest physical address to the driver in one of two ways:

1. As a physical address, using virtio_pmem_config.
2. As a shared memory region.

The driver determines the start address and size of the persistent memory region in preparation for reading or writing data.

The driver initializes req_vq in preparation for making flush requests.

5.19.5.1 Device Requirements: Device Initialization

If VIRTIO_PMEM_F_SHMEM_REGION has been negotiated, the device MUST indicate the guest physical address as a shared memory region. The device MUST use shared memory region ID 0. The device SHOULD set *start* and *size* to zero.

If VIRTIO_PMEM_F_SHMEM_REGION has not been negotiated, the device MUST indicate the guest physical address as a physical address. The device MUST set *start* to the absolute address and *size* to the size of the address range, in bytes.

5.19.5.2 Driver Requirements: Device Initialization

If VIRTIO_PMEM_F_SHMEM_REGION has been negotiated, the driver MUST query shared memory ID 0 for the physical address ranges, and MUST NOT use *start* or *stop*.

If VIRTIO_PMEM_F_SHMEM_REGION has not been negotiated, the driver MUST read the physical address ranges from *start* and *stop*.

5.19.6 Driver Operations

Requests have the following format:

```
struct virtio_pmem_req {
        le32 type;
};
```

type is the request command type.

Possible request types are:

```
#define VIRTIO_PMEM_REQ_TYPE_FLUSH      0
```

5.19.7 Device Operations

5.19.7.1 Device Requirements: Device Operation: Virtqueue flush

The device MUST ensure that all writes completed before a flush request persist across device reset and power failure before completing the flush request.

5.19.7.2 Device Operations

```
struct virtio_pmem_resp {
        le32 ret;
};
```

ret is the value which the device returns after command completion.

5.19.7.3 Device Requirements: Device Operation: Virtqueue return

The device MUST return ”0” for success and ”-1” for failure.

5.19.8 Possible security implications

There could be potential security implications depending on how memory mapped backing device is used. By default device emulation is done with SHARED memory mapping. There is a contract between driver and device to access shared memory region for read or write operations.

If a malicious driver or device maps the same memory region, the attacker can make use of known side channel attacks to predict the current state of data. If both attacker and victim somehow execute same shared code after a flush or evict operation, with difference in execution timing attacker could infer another device's data.

5.19.9 Countermeasures

5.19.9.1 With SHARED mapping

If a device's backing region is shared between multiple devices, this may act as a metric for side channel attacks. As a counter measure every device should have its own (not shared with another driver) SHARED backing memory.

5.19.9.2 With PRIVATE mapping

There maybe be chances of side channels attack with PRIVATE memory mapping similar to SHARED with read-only shared mappings. PRIVATE is not used for virtio pmem making this usecase irrelevant.

5.19.9.3 Workload specific mapping

When using SHARED mappings with a workload that is a single application inside the driver where the risk in sharing data is very low or nonexisting, the device sharing the same backing region with a SHARED mapping can be used as a valid configuration.

5.19.9.4 Prevent cache eviction

Don't allow device shared region eviction from driver filesystem trim or discard like commands with virtio pmem. This rules out any possibility of evict-reload cache side channel attacks if backing region is shared (SHARED) between mutliple devices. Though if we use per device backing file with shared mapping this countermeasure is not required.

5.20 CAN Device

virtio-can is a virtio based CAN (Controller Area Network) controller. It is used to give a virtual machine access to a CAN bus. The CAN bus might either be a physical CAN bus or a virtual CAN bus between virtual machines or a combination of both.

5.20.1 Device ID

36

5.20.2 Virtqueues

0 Txq

1 Rxq

2 Controlq

The *Txq* is used to send CAN packets to the CAN bus.

The *Rxq* is used to receive CAN packets from the CAN bus.

The *Controlq* is used to control the state of the CAN controller.

5.20.3 Feature bits

Device Types / CAN Device / Feature bits

Actual CAN controllers support Extended CAN IDs with 29 bits (CAN 2.0B) as well as Standard CAN IDs with 11 bits (CAN 2.0A). The support of CAN 2.0B Extended CAN IDs is considered as mandatory for this specification.

VIRTIO_CAN_F_CAN_CLASSIC (0) The device supports classic CAN frames with a maximum payload size of 8 bytes.

VIRTIO_CAN_F_CAN_FD (1) The device supports CAN FD frames with a maximum payload size of 64 bytes.

VIRTIO_CAN_F_RTR_FRAMES (2) The device supports RTR (remote transmission request) frames. RTR frames are only supported with classic CAN.

VIRTIO_CAN_F_LATE_TX_ACK (3) The virtio CAN device marks transmission requests from the *Txq* as used after the CAN message has been transmitted on the CAN bus. If this feature bit has not been negotiated, the device is allowed to mark transmission requests already as used when the CAN message has been scheduled for transmission but might not yet have been transmitted on the CAN bus.

5.20.3.1 Feature bit requirements

Some CAN feature bits require other CAN feature bits:

VIRTIO_CAN_F_RTR_FRAMES Requires VIRTIO_CAN_F_CAN_CLASSIC.

It is required that at least one of VIRTIO_CAN_F_CAN_CLASSIC and VIRTIO_CAN_F_CAN_FD is negotiated.

5.20.4 Device configuration layout

Device configuration fields are listed below, they are read-only for a driver. The *status* always exists. A single read-only bit (for the driver) is currently defined for *status*:

```
struct virtio_can_config {
#define VIRTIO_CAN_S_CTRL_BUSOFF (1 << 0)
        le16 status;
};
```

The bit VIRTIO_CAN_S_CTRL_BUSOFF in *status* is used to indicate the unsolicited CAN controller state change from started to stopped due to a detected bus off condition.

5.20.4.1 Driver Requirements: Device Initialization

The driver MUST populate the *Rxq* with empty device-writeable buffers of at least the size of struct virtio_can_rx, see section 5.20.5.3.

5.20.5 Device Operation

A device operation has an outcome which is described by one of the following values:

```
#define VIRTIO_CAN_RESULT_OK     0
#define VIRTIO_CAN_RESULT_NOT_OK 1
```

Other values are to be treated like VIRTIO_CAN_RESULT_NOT_OK.

5.20.5.1 Controller Mode

The general format of a request in the *Controlq* is

```
struct virtio_can_control_out {
#define VIRTIO_CAN_SET_CTRL_MODE_START  0x0201
#define VIRTIO_CAN_SET_CTRL_MODE_STOP   0x0202
        le16 msg_type;
};
```

To participate in bus communication the CAN controller is started by sending a VIRTIO_CAN_SET_CTRL_-MODE_START control message, to stop participating in bus communication it is stopped by sending a VIRTIO_CAN_SET_CTRL_MODE_STOP control message. Both requests are confirmed by the result of the operation.

```
struct virtio_can_control_in {
        u8 result;
};
```

If the transition succeeded the *result* is VIRTIO_CAN_RESULT_OK otherwise it is VIRTIO_CAN_RESULT_-NOT_OK. If a status update is necessary, the device updates the configuration *status* before marking the request used. As the configuration *status* change is caused by a request from the driver the device is allowed to omit the configuration change notification here. The device marks the request used when the CAN controller has finalized the transition to the requested controller mode.

On transition to the STOPPED state the device cancels all CAN messages already pending for transmission and marks them as used with *result* VIRTIO_CAN_RESULT_NOT_OK. In the STOPPED state the device marks messages received from the *Txq* as used with *result* VIRTIO_CAN_RESULT_NOT_OK without transmitting them to the CAN bus.

Initially the CAN controller is in the STOPPED state.

Control queue messages are processed in order.

5.20.5.2 Device Requirements: CAN Message Transmission

The driver transmits messages by placing outgoing CAN messages in the *Txq* virtqueue.

```
struct virtio_can_tx_out {
#define VIRTIO_CAN_TX 0x0001
        le16 msg_type;
        le16 length; /* 0..8 CC, 0..64 CAN-FD, 0..2048 CAN-XL, 12 bits */
        u8 reserved_classic_dlc; /* If CAN classic length = 8 then DLC can be 8..15 */
        u8 padding;
        le16 reserved_xl_priority; /* May be needed for CAN XL priority */
#define VIRTIO_CAN_FLAGS_FD             0x4000
#define VIRTIO_CAN_FLAGS_EXTENDED       0x8000
#define VIRTIO_CAN_FLAGS_RTR            0x2000
        le32 flags;
        le32 can_id;
        u8 sdu[];
};

struct virtio_can_tx_in {
        u8 result;
};
```

The length of the *sdu* is determined by the *length*.

The type of a CAN message identifier is determined by *flags*. The 3 most significant bits of *can_id* do not bear the information about the type of the CAN message identifier and are 0.

The device MUST reject any CAN frame type for which support has not been negotiated with VIRTIO_-CAN_RESULT_NOT_OK in *result* and MUST NOT schedule the message for transmission. A CAN frame with an undefined bit set in *flags* is treated like a CAN frame for which support has not been negotiated.

The device MUST reject any CAN frame for which *can_id* or *sdu* length are out of range or the CAN controller is in an invalid state with VIRTIO_CAN_RESULT_NOT_OK in *result* and MUST NOT schedule the message for transmission.

If the parameters are valid the message is scheduled for transmission.

If feature VIRTIO_CAN_F_CAN_LATE_TX_ACK has been negotiated the transmission request MUST be marked as used with *result* set to VIRTIO_CAN_OK after the CAN controller acknowledged the successful transmission on the CAN bus. If this feature bit has not been negotiated the transmission request MAY already be marked as used with *result* set to VIRTIO_CAN_OK when the transmission request has been processed by the virtio CAN device and send down the protocol stack being scheduled for transmission.

5.20.5.3 CAN Message Reception

Messages can be received by providing empty incoming buffers to the virtqueue *Rxq*.

```
struct virtio_can_rx {
#define VIRTIO_CAN_RX 0x0101
        le16 msg_type;
        le16 length; /* 0..8 CC, 0..64 CAN-FD, 0..2048 CAN-XL, 12 bits */
        u8 reserved_classic_dlc; /* If CAN classic length = 8 then DLC can be 8..15 */
        u8 padding;
        le16 reserved_xl_priority; /* May be needed for CAN XL priority */
        le32 flags;
        le32 can_id;
        u8 sdu[];
};
```

If the feature VIRTIO_CAN_F_CAN_FD has been negotiated the maximal possible *sdu* length is 64, if the feature has not been negotiated the maximal possible *sdu* length is 8.

The actual length of the *sdu* is determined by the *length*.

The type of a CAN message identifier is determined by *flags* in the same way as for transmitted CAN messages, see section 5.20.5.2. The 3 most significant bits of *can_id* do not bear the information about the type of the CAN message identifier and are 0. The flag bits are exactly the same as for the *flags* of struct virtio_can_tx_out.

5.20.5.4 BusOff Indication

There are certain error conditions so that the physical CAN controller has to stop participating in CAN communication on the bus. If such an error condition occurs the device informs the driver about the unsolicited CAN controller state change by setting the VIRTIO_CAN_S_CTRL_BUSOFF bit in the configuration *status* field.

After bus-off detection the CAN controller is in STOPPED state. The CAN controller does not participate in bus communication any more so all CAN messages pending for transmission are put into the used queue with *result* VIRTIO_CAN_RESULT_NOT_OK.

5.21 SPI Controller Device

The Virtio SPI (Serial Peripheral Interface) device is a virtual SPI controller that allows the driver to operate and use the SPI controller under the control of the host, either a physical SPI controller, or an emulated one.

The Virtio SPI device has a single virtqueue. SPI transfer requests are placed into the virtqueue by the driver, and are serviced by the device.

In a typical host and guest architecture with the Virtio SPI device, the Virtio SPI driver is the front-end running in the guest, and the Virtio SPI device is the back-end in the host.

5.21.1 Device ID

45

5.21.2 Virtqueues

0 requestq

5.21.3 Feature bits

None

5.21.4 Device configuration layout

All fields of this configuration are mandatory and read-only for the driver. The config space shows the features and settings supported by the device.

```
struct virtio_spi_config {
  u8 cs_max_number;
  u8 cs_change_supported;
  u8 tx_nbits_supported;
  u8 rx_nbits_supported;
  le32 bits_per_word_mask;
  le32 mode_func_supported;
  le32 max_freq_hz;
  le32 max_word_delay_ns;
  le32 max_cs_setup_ns;
  le32 max_cs_hold_ns;
  le32 max_cs_inactive_ns;
};
```

cs_max_number is the maximum number of chipselect the device supports.

Note: chipselect is an electrical signal, which is used to select the SPI peripherals connected to the controller.

cs_change_supported indicates if the device supports to toggle chipselect after each transfer in one message: 0: unsupported, chipselect will be kept in active state throughout the message transaction; 1: supported.

Note: Message here contains a sequence of SPI transfers.

tx_nbits_supported and *rx_nbits_supported* indicate the different n-bit transfer modes supported by the device, for writing and reading respectively. SINGLE is always supported. A set bit here indicates that the corresponding n-bit transfer is supported, otherwise not: bit 0: DUAL; bit 1: QUAD; bit 2: OCTAL; other bits are reserved and must be set as 0 by the device.

Note: The commonly used SPI n-bit transfer options are:

- SINGLE: 1-bit transfer
- DUAL: 2-bit transfer
- QUAD: 4-bit transfer
- OCTAL: 8-bit transfer

bits_per_word_mask is a mask indicating which values of bits_per_word are supported. If bit n of *bits_per_word_mask* is set, the bits_per_word with value (n+1) is supported. If *bits_per_word_mask* is 0, there is no limitation for bits_per_word.

Note: bits_per_word corresponds to *bits_per_word* in *struct virtio_spi_transfer_head*.

mode_func_supported indicates whether the following features are supported or not: bit 0-1: CPHA feature, 0b00: invalid, must support as least one CPHA setting. 0b01: supports CPHA=0 only; 0b10: supports CPHA=1 only; 0b11: supports CPHA=0 and CPHA=1;

bit 2-3: CPOL feature, 0b00: invalid, must support as least one CPOL setting. 0b01: supports CPOL=0 only; 0b10: supports CPOL=1 only; 0b11: supports CPOL=0 and CPOL=1;

bit 4: chipselect active high feature, 0 for unsupported and 1 for supported, chipselect active low must always be supported.

bit 5: LSB first feature, 0 for unsupported and 1 for supported, MSB first must always be supported.

bit 6: loopback mode feature, 0 for unsupported and 1 for supported, normal mode must always be supported.

Note: CPOL is clock polarity and CPHA is clock phase. If CPOL is 0, the clock idles at the logical low voltage, otherwise it idles at the logical high voltage. CPHA determines how data is outputted and sampled. If CPHA is 0, the first bit is outputted immediately when chipselect is active and subsequent bits are outputted on the clock edges when the clock transitions from active level to idle level. Data is sampled on the clock edges

when the clock transitions from idle level to active level. If CPHA is 1, the first bit is outputted on the fist clock edge after chipselect is active, subsequent bits are outputted on the clock edges when the clock transitions from idle level to active level. Data is sampled on the clock edges when the clock transitions from active level to idle level.

Note: LSB first indicates that data is transferred least significant bit first, and MSB first indicates that data is transferred most significant bit first.

max_freq_hz is the maximum clock rate supported in Hz unit, 0 means no limitation for transfer speed.

max_word_delay_ns is the maximum word delay supported in ns unit, 0 means word delay feature is unsupported.

Note: Just as one message contains a sequence of transfers, one transfer may contain a sequence of words.

max_cs_setup_ns is the maximum delay supported after chipselect is asserted, in ns unit, 0 means delay is not supported to introduce after chipselect is asserted.

max_cs_hold_ns is the maximum delay supported before chipselect is deasserted, in ns unit, 0 means delay is not supported to introduce before chipselect is deasserted.

max_cs_incative_ns is the maximum delay supported after chipselect is deasserted, in ns unit, 0 means delay is not supported to introduce after chipselect is deasserted.

5.21.5 Device Initialization

1. The Virtio SPI driver configures and initializes the virtqueue.

5.21.6 Device Operation

5.21.6.1 Device Operation: Request Queue

The Virtio SPI driver enqueues requests to the virtqueue, and they are used by the Virtio SPI device. Each request represents one SPI transfer and is of the form:

```
struct virtio_spi_transfer_head {
        u8 chip_select_id;
        u8 bits_per_word;
        u8 cs_change;
        u8 tx_nbits;
        u8 rx_nbits;
        u8 reserved[3];
        le32 mode;
        le32 freq;
        le32 word_delay_ns;
        le32 cs_setup_ns;
        le32 cs_delay_hold_ns;
        le32 cs_change_delay_inactive_ns;
};
```

```
struct virtio_spi_transfer_result {
        u8 result;
};
```

```
struct virtio_spi_transfer_req {
        struct virtio_spi_transfer_head head;
        u8 tx_buf[];
        u8 rx_buf[];
        struct virtio_spi_transfer_result result;
};
```

chip_select_id indicates the chipselect index to use for the SPI transfer.

bits_per_word indicates the number of bits in each SPI transfer word.

cs_change indicates whether to deselect device before starting the next SPI transfer, 0 means chipselect keep asserted and 1 means chipselect deasserted then asserted again.

tx_nbits and *rx_nbits* indicate n-bit transfer mode for writing and reading: 0,1: SINGLE; 2 : DUAL; 4 : QUAD; 8 : OCTAL; other values are invalid.

reserved is currently unused and might be used for further extensions in the future.

mode indicates some transfer settings. Bit definitions as follows: bit 0: CPHA, determines the timing (i.e. phase) of the data bits relative to the clock pulses. bit 1: CPOL, determines the polarity of the clock. bit 2: CS_HIGH, if 1, chipselect active high, else active low. bit 3: LSB_FIRST, determines per-word bits-on-wire, if 0, MSB first, else LSB first. bit 4: LOOP, if 1, device is in loopback mode, else normal mode.

freq indicates the SPI transfer speed in Hz.

word_delay_ns indicates delay to be inserted between consecutive words of a transfer, in ns unit.

cs_setup_ns indicates delay to be introduced after chipselect is asserted, in ns unit.

cs_delay_hold_ns indicates delay to be introduced before chipselect is deasserted, in ns unit.

cs_change_delay_inactive_ns indicates delay to be introduced after chipselect is deasserted and before next asserted, in ns unit.

tx_buf is the buffer for data sent to the device.

rx_buf is the buffer for data received from the device.

result is the transfer result, it may be one of the following values:

```
#define VIRTIO_SPI_TRANS_OK     0
#define VIRTIO_SPI_PARAM_ERR    1
#define VIRTIO_SPI_TRANS_ERR    2
```

VIRTIO_SPI_TRANS_OK indicates successful completion of the transfer.

VIRTIO_SPI_PARAM_ERR indicates a parameter error, which means the parameters in *struct virtio_spi_transfer_head* are not all valid, or some fields are set as non-zero values but the corresponding features are not supported by device. In particular, for full-duplex transfer, VIRTIO_SPI_PARAM_ERR can also indicate that *tx_buf* and *rx_buf* are not of the same length.

VIRTIO_SPI_TRANS_ERR indicates a transfer error, which means that the parameters are all valid but the transfer process failed.

5.21.6.2 Device Operation: Operation Status

Fields in *struct virtio_spi_transfer_head* are written by the Virtio SPI driver, while *result* in *struct virtio_spi_transfer_result* is written by the Virtio SPI device.

virtio-spi supports three transfer types:

- half-duplex read;
- half-duplex write;
- full-duplex transfer.

For half-duplex read and full-duplex transfer, *rx_buf* is filled by the Virtio SPI device and consumed by the Virtio SPI driver. For half-duplex write and full-duplex transfer, *tx_buf* is filled by the Virtio SPI driver and consumed by the Virtio SPI device.

5.21.6.3 Driver Requirements: Device Operation

For half-duplex read, the Virtio SPI driver MUST send *struct virtio_spi_transfer_head*, *rx_buf* and *struct virtio_spi_transfer_result* to the SPI Virtio Device in that order.

For half-duplex write, the Virtio SPI driver MUST send *struct virtio_spi_transfer_head*, *tx_buf* and *struct virtio_spi_transfer_result* to the SPI Virtio Device in that order.

For full-duplex transfer, the Virtio SPI driver MUST send *struct virtio_spi_transfer_head*, *tx_buf*, *rx_buf* and *struct virtio_spi_transfer_result* to the SPI Virtio Device in that order.

For full-duplex transfer, the Virtio SPI driver MUST guarantee that the buffer size of *tx_buf* and *rx_buf* is the same.

The Virtio SPI driver MUST not use *rx_buf* if the *result* returned from the Virtio SPI device is not VIRTIO_-SPI_TRANS_OK.

5.21.6.4 Device Requirements: Device Operation

The Virtio SPI device MUST set all the fields of *struct virtio_spi_config* before they are read by the Virtio SPI driver.

The Virtio SPI device MUST NOT change the data in *tx_buf*.

The Virtio SPI device MUST verify the parameters in *struct virtio_spi_transfer_head* after receiving the request, and MUST set *struct virtio_spi_transfer_result* as VIRTIO_SPI_PARAM_ERR if not all parameters are valid or some device unsupported features are set.

For full-duplex transfer, the Virtio SPI device MUST verify that the buffer size of *tx_buf* is equal to that of *rx_-buf*. If not, the Virtio SPI device MUST set *struct virtio_spi_transfer_result* as VIRTIO_SPI_PARAM_ERR.

5.22 Media Device

The virtio media device follows the same model (and structures) as V4L2. It can be used to virtualize cameras, codec devices, or any other device supported by V4L2. The complete definition of V4L2 structures and ioctls can be found under the V4L2 UAPI documentation.

V4L2 is a UAPI that allows a less privileged entity (user-space) to use video hardware exposed by a more privileged entity (the kernel). Virtio-media is an encapsulation of this API into virtio, turning it into a virtualization API for all classes of video devices supported by V4L2, where the device plays the role of the kernel and the driver the role of user-space.

The device is therefore responsible for presenting a virtual device that behaves like an actual V4L2 device, which the driver can control.

Note that virtio-media does not require the use of a V4L2 device driver or of Linux on any side - V4L2 is only used as a transport protocol, and both sides are free to convert it from/to any model that they wish to use.

5.22.1 Device ID

48

5.22.2 Virtqueues

0 commandq - used for driver commands and device responses to these commands.

1 eventq - used for events sent by the device to the driver.

5.22.3 Feature Bits

None

5.22.4 Device Configuration Layout

The video device configuration space uses the following layout:

```
struct virtio_media_config {
    le32 device_caps;
    le32 device_type;
    le8 card[32];
};
```

device_caps (driver-read-only) flags representing the device capabilities as used in struct v4l2_capability. It corresponds with the *device_caps* field in the *struct video_device*.

device_type (driver-read-only) informs the driver of the type of the video device. It corresponds with the *vfl_devnode_type* field of the device.

card (driver-read-only) name of the device, a NUL-terminated UTF-8 string. It corresponds with the *card* field of the *struct v4l2_capability*. If all the characters of the field are used, it does not need to be NUL-terminated.

5.22.5 Device Initialization

A driver executes the following sequence to initialize a device:

1. Read the *device_caps* and *device_type* fields from the configuration layout to identify the device.
2. Set up the *commandq* and *eventq* virtqueues.
3. May open a session (see Section 5.22.6.1.3) to use the device and send V4L2 ioctls in order to receive more information about the device, such as supported formats or controls.

5.22.6 Device Operation

The driver enqueues commands in the command queue for the device to process. The errors returned by each command are standard Linux kernel error codes. For instance, a driver sending a command that contains invalid options will receive *EINVAL* in return, after the device tries to process it.

The device enqueues events in the event queue for the driver to process.

5.22.6.1 Command Virtqueue

5.22.6.1.1 Device Operation: Command headers

```
#define VIRTIO_MEDIA_CMD_OPEN 1
#define VIRTIO_MEDIA_CMD_CLOSE 2
#define VIRTIO_MEDIA_CMD_IOCTL 3
#define VIRTIO_MEDIA_CMD_MMAP 4
#define VIRTIO_MEDIA_CMD_MUNMAP 5

/* Header for all virtio commands from the driver to the device on the commandq. */
struct virtio_media_cmd_header {
  le32 cmd;
  le32 __reserved;
};

/* Header for all virtio responses from the device to the driver on the commandq. */
struct virtio_media_resp_header {
  le32 status;
  le32 __reserved;
};
```

A command consists of a command header *virtio_media_cmd_header* containing the following device-readable field:

cmd specifies a device request type (VIRTIO_MEDIA_CMD_*).

A response consists of a response header *virtio_media_resp_header* containing the following device-writable field:

status indicates a device request status.

When the device executes the command successfully, the value of the status field is 0. Conversely, when the device fails to execute the command, the value of the status fields corresponds with one of the standard Linux error codes.

5.22.6.1.2 Driver Requirements: Device Operation: Command Virtqueue: Sessions

Sessions are how the device is multiplexed, allowing several distinct works to take place simultaneously. Before starting operation, the driver needs to open a session. This is equivalent to opening the */dev/videoX* file of the V4L2 device. Each session gets a unique ID assigned, which can be then used to perform actions on it.

5.22.6.1.3 Device Operation: Open device

VIRTIO_MEDIA_CMD_OPEN Command for creating a new session.

This is the equivalent of calling *open* on a V4L2 device node. The driver uses *virtio_media_cmd_open* to send an open request.

```
struct virtio_media_cmd_open {
    struct virtio_media_cmd_header hdr;
};
```

The device responds to *VIRTIO_MEDIA_CMD_OPEN* with *virtio_media_resp_open*.

```
struct virtio_media_resp_open {
    struct virtio_media_resp_header hdr;
    le32 session_id;
    le32 __reserved;
};
```

session_id identifies the current session, which is used for other commands, predominantly ioctls.

5.22.6.1.3.1 Device Requirements: Device Operation: Open device

Upon success, the device MUST set a *session_id* in *virtio_media_resp_open* to an integer that is NOT used by any other open session.

5.22.6.1.4 Device Operation: Close device

VIRTIO_MEDIA_CMD_CLOSE Command for closing an active session.

This is the equivalent of calling *close* on a previously opened V4L2 device node. All resources associated with this session will be freed.

This command does not require a response from the device.

```
struct virtio_media_cmd_close {
    struct virtio_media_cmd_header hdr;
    le32 session_id;
    le32 __reserved;
};
```

session_id identifies the session to close.

5.22.6.1.4.1 Driver Requirements: Device Operation: Close device

The session ID SHALL NOT be used again after queueing this command, until it has been obtained again through a subsequent *VIRTIO_MEDIA_CMD_OPEN* call.

5.22.6.1.5 Device Operation: V4L2 ioctls

VIRTIO_MEDIA_CMD_IOCTL Command for executing an ioctl on an open session.

This command tells the device to run one of the ‘VIDIOC_*‘ ioctls on the session identified by *session_id*.

```
struct virtio_media_cmd_ioctl {
    struct virtio_media_cmd_header hdr;
    le32 session_id;
    le32 code;
    /* Followed by the relevant ioctl command payload as defined in the macro */
};
```

session_id identifies the session to run the ioctl on.

code specifies the code of the *VIDIOC_** ioctl to run.

The code is extracted from the videodev2.h, header file. The file defines the ioctl's codes, type of payload, and direction. The code consists of the second argument of the *_IO** macro.

For example, the *VIDIOC_G_FMT* is defined as follows:

```
#define VIDIOC_G_FMT _IOWR('V',  4, struct v4l2_format)
```

This means that its ioctl code is *4*, its payload is a *struct v4l2_format*, and its direction is *WR* (i.e., the payload is written by both the driver and the device). See Section 5.22.6.1.5.1 for more information about the direction of ioctls.

The payload struct layout always matches the 64-bit, little-endian representation of the corresponding V4L2 structure.

The device responds to *VIRTIO_MEDIA_CMD_IOCTL* with *virtio_media_resp_ioctl*.

```
struct virtio_media_resp_ioctl {
    struct virtio_media_resp_header hdr;
    /* Followed by the ioctl response payload as defined in the macro */
};
```

5.22.6.1.5.1 Ioctls payload

Each ioctl has a payload, which is defined by the third argument of the *_IO** macro.

The payload of an ioctl in the descriptor chain follows the command structure, the response structure, or both depending on the direction:

- **_IOR** is read-only for the driver, meaning the payload follows the response in the device-writable section of the descriptor chain.
- **_IOW** is read-only for the device, meaning the payload follows the command in the device-readable section of the descriptor chain.
- **_IOWR** is writable by both the device and driver, meaning the payload must follow both the command in the device-readable section of the descriptor chain, and the response in the device-writable section.

A possible optimization for *WR* ioctls is to provide the payload using descriptors that both point to the same buffer. This mimics the behavior of V4L2 ioctls where the data is only passed once and used as both input and output by the kernel.

5.22.6.1.5.2 Device Requirements: Device Operation: V4L2 ioctls

In case of success of a device-writable ioctl, the device MUST always write the payload in the device-writable part of the descriptor chain.

In case of failure of a device-writable ioctl, the device is free to write the payload in the device-writable part of the descriptor chain or not. Some errors may still result in the payload being updated, and in this case the device is expected to write the updated payload.

5.22.6.1.5.3 Driver Requirements: Device Operation: V4L2 ioctls

For most V4L2 structures, the size is identical for both 32 and 64 bits versions. If the payload struct layout size differs for 32 and 64 bits, the driver MUST translate them to its the 64-bit, little-endian representation.

If the device has not written the payload after an error (i.e., only the header is returned), the driver MUST assume that the payload has not been modified.

5.22.6.1.5.4 Handling of pointers to data in ioctl payload

A few structures used as ioctl payloads contain pointers to further data needed for the ioctl. There are notably:

- The *planes* pointer of struct v4l2_buffer, which size is determined by the length member.
- The *controls* pointer of struct v4l2_ext_controls, which size is determined by the count member.

If the size of the pointed area is non-zero, then the main payload is immediately followed by the pointed data in their order of appearance in the structure.

5.22.6.1.5.5 Device Requirements: Handling of pointers to data in ioctl payload

The pointer value, when its area is non-zero, is ignored by the device, which MUST return the value initially passed by the driver.

5.22.6.1.5.6 Handling of pointers to userspace memory in ioctl payload

A few pointers (used for *SHARED_PAGES* memory type, see 5.22.6.1.8.2) are special in that they point to userspace memory in the original V4L2 specification. They are:

- The *m.userptr* member of *struct v4l2_buffer* and struct v4l2_plane (technically an unsigned long, but designated a userspace address).
- The *ptr* member of *struct v4l2_ext_ctrl*.

These pointers may cover large areas of scattered memory, which has the potential to require more descriptors than the virtio queue can provide. For these particular pointers only, a list of *struct virtio_media_sg_entry* that covers the needed amount of memory for the pointer is used instead of using descriptors to map the pointed memory directly.

```
struct virtio_media_sg_entry {
  le64 start;
  le32 len;
  le32 __reserved;
};
```

For each such pointer to read, the device reads as many SG entries as needed to cover the length of the pointed buffer, as described by its parent structure (*length* member of *struct v4l2_buffer* or *struct v4l2_plane* for buffer memory, and *size* member of *struct v4l2_ext_control* for control data).

Since the device never needs to modify the list of SG entries, it is only provided by the driver in the device-readable section of the descriptor chain, and not repeated in the device-writable section, even for WR ioctls.

5.22.6.1.5.7 Unsupported ioctls

A few ioctls are replaced by other, more suitable mechanisms.

- *VIDIOC_QUERYCAP* is replaced by reading the configuration area (see 5.22.4).
- *VIDIOC_DQBUF* and *VIDIOC_DQEVENT* are replaced by a dedicated event (see 5.22.6.2).
- *VIDIOC_G_JPEGCOMP* and *VIDIOC_S_JPEGCOMP* are deprecated and replaced by the controls of the JPEG class.
- *VIDIOC_LOG_STATUS* is a driver-only operation and shall not be implemented by the device.

5.22.6.1.5.8 Device Requirements: Device Operation: Unsupported ioctls

When a request is not supported, the device MUST return *ENOTTY*, which corresponds to the response for unknown ioctls.

5.22.6.1.6 Device Operation: Mapping a MMAP buffer

VIRTIO_MEDIA_CMD_MMAP Command for mapping a MMAP buffer into the driver's address space.

Shared memory region ID 0 is used to map MMAP buffers with the *VIRTIO_MEDIA_CMD_MMAP* command.

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

struct virtio_media_cmd_mmap {
  struct virtio_media_cmd_header hdr;
  le32 session_id;
  le32 flags;
  le32 offset;
};
```

session_id identifies the session which the mapped buffer pertains to.

flags is the set of flags for the mapping. *VIRTIO_MEDIA_MMAP_FLAG_RW* can be set if a read-write mapping is desired. Without this flag the mapping will be read-only.

offset corresponds to the *mem_offset* field of the *union v4l2_plane* for the plane to map. This field can be obtained using the *VIDIOC_QUERYBUF* ioctl.

The device responds to *VIRTIO_MEDIA_CMD_MMAP* with *virtio_media_resp_mmap*.

```
struct virtio_media_resp_mmap {
    struct virtio_media_resp_header hdr;
    le64 driver_addr;
    le64 len;
};
```

driver_addr offset into SHM region ID 0 of the start of the mapping.

len length of the mapping as indicated by the *struct v4l2_plane* the buffer belongs to.

5.22.6.1.6.1 Device Requirements: Device Operation: Mapping a MMAP buffer

The *len* parameter of the *virtio_media_resp_mmap* response sent by the device MUST always be equal to the length of the buffer.

5.22.6.1.7 Device Operation: Unmapping a MMAP buffer

VIRTIO_MEDIA_CMD_MUNMAP unmaps a MMAP buffer previously mapped using *VIRTIO_MEDIA_CMD_-MMAP*.

```
struct virtio_media_cmd_munmap {
    struct virtio_media_cmd_header hdr;
    le64 driver_addr;
};
```

driver_addr offset into SHM region ID 0 previously returned by *VIRTIO_MEDIA_CMD_MMAP* at which the buffer has been previously mapped.

The device responds to *VIRTIO_MEDIA_CMD_MUNMAP* with *virtio_media_resp_munmap*.

```
struct virtio_media_resp_munmap {
    struct virtio_media_resp_header hdr;
};
```

5.22.6.1.7.1 Device Requirements: Device Operation: Unmapping a MMAP buffer

The device MUST keep mappings performed using *VIRTIO_MEDIA_CMD_MMAP* valid until *VIRTIO_MEDIA_CMD_MUNMAP* is called, even if the buffers or session they belong to are released or closed by the driver.

5.22.6.1.8 Device Operation: Memory Types

The semantics of the three V4L2 memory types (*MMAP*, *USERPTR* and *DMABUF*) can easily be mapped to both driver and device context.

```
enum virtio_media_memory {
    VIRTIO_MEDIA_MMAP = V4L2_MEMORY_MMAP,
    VIRTIO_MEDIA_SHARED_PAGES = V4L2_MEMORY_USERPTR,
    VIRTIO_MEDIA_OBJECT = V4L2_MEMORY_DMABUF,
};
```

5.22.6.1.8.1 MMAP

MMAP memory type is the semantic equivalent of *V4L2_MEMORY_MMAP* in regular V4L2.

In virtio-media, *MMAP* buffers are provisioned by the device, just like they are by the kernel in regular V4L2. Similarly to how userspace can map a *MMAP* buffer into its address space using mmap and munmap, the virtio-media driver can map device buffers into the driver space by queueing the *struct virtio_media_cmd_mmap* and *struct virtio_media_cmd_munmap* commands to the commandq.

5.22.6.1.8.2 SHARED_PAGES

SHARED_PAGES memory type is the semantic equivalent of *V4L2_MEMORY_USERPTR* in regular V4L2.

In virtio-media, *SHARED_PAGES* buffers are provisioned by the driver, and use guest physical addresses. Instances of *struct v4l2_buffer* and *struct v4l2_plane* of this memory type are followed by a list of *struct virtio_media_sg_entry*. For more information, see 5.22.6.1.5.6

5.22.6.1.8.3 Device Requirements: Device Operation: Shared Pages

The device MUST not alter the pointer values provided by the driver, i.e. the *m.userptr* member of *struct v4l2_buffer* and *struct v4l2_plane* MUST be returned to the driver with the same value as it was provided.

5.22.6.1.8.4 VIRTIO_OBJECT

VIRTIO_OBJECT memory type is the semantic equivalent of *V4L2_MEMORY_DMABUF* in regular V4L2.

In virtio-media, *VIRTIO_OBJECT* buffers are provisioned by a virtio object, just like they are by a *DMABUF* in regular V4L2. Virtio objects are 16-bytes UUIDs and do not fit in the placeholders for file descriptors, so they follow their embedding data structure as needed. For example, in multi-planar buffers, *struct v4l2_plane* structures are located after the *struct v4l2_buffer* in the memory layout. Then, after the last plane, the memory contains an array of UUIDs in which the first element corresponds with the first plane, and so on.

Conversely to *SHARED_PAGES* buffers, which SG lists are never modified by the device, UUIDs of virtio objects need to be added in both the device-readable and device-writable section of the descriptor chain so the device can update them if needed.

Device-allocated buffers with the *VIRTIO_MEDIA_MMAP* memory type may also be exported as virtio objects for use with another virtio device using the *VIDIOC_EXPBUF* ioctl. The fd placefolder of *v4l2_exportbuffer* means that space for the UUID needs to be reserved right after that structure.

5.22.6.1.8.5 Device Requirements: Device Operation: Virtio Object

The device MUST leave the *fd* placeholder of the V4L2 structure unchanged.

5.22.6.2 Event Virtqueue

Events are asynchronous notifications to the driver. In virtio-media, they are used as a replacement for the *VIDIOC_DQBUF* and *VIDIOC_DQEVENT* ioctls and the polling mechanism, which would be impractical to implement on top of virtio.

5.22.6.2.1 Device Operation: Event header

```
#define VIRTIO_MEDIA_EVT_ERROR 0
#define VIRTIO_MEDIA_EVT_DQBUF 1
#define VIRTIO_MEDIA_EVT_EVENT 2

/* Header for events queued by the device for the driver on the eventq. */
struct virtio_media_event_header {
    le32 event;
    le32 session_id;
};
```

event one of *VIRTIO_MEDIA_EVT_*.

session_id ID of the session the event applies to.

5.22.6.2.2 Device Operation: Device-side error

VIRTIO_MEDIA_EVT_ERROR Unrecoverable session error. Upon emitting this event, the device considers the session mentioned in the header to be invalid, and returns an error to all future commands referring to it. Upon receiving this event, the driver stops using the session, and shall close it as soon as possible.

```
struct virtio_media_event_error {
    struct virtio_media_event_header hdr;
    le32 errno;
    le32 __reserved;
};
```

errno error code describing the kind of error that occurred.

5.22.6.2.2.1 Device Requirements: Device Operation: Device-side error

After an error is signaled, when the device considers the session as non-existing, the device MUST NOT recycle the session ID until the driver has explicitly closed it.

5.22.6.2.2.2 Driver Requirements: Device Operation: Device-side error

Upon receiving an error event for a session, the driver MUST explicitly close the session using a *VIRTIO_MEDIA_CMD_CLOSE* command.

5.22.6.2.3 Device Operation: Dequeue buffer

VIRTIO_MEDIA_EVT_DQBUF signals that a buffer is not being used anymore by the device and is returned to the driver.

Every time a buffer previously queued (i.e., using the *VIDIOC_QBUF* ioctl) is done being processed, the device queues a *struct virtio_media_event_dqbuf* event on the eventq, signifying that the buffer may be used again by the driver. This is like an implicit *VIDIOC_DQBUF* ioctl.

```
#define VIRTIO_MEDIA_MAX_PLANES 8

struct virtio_media_event_dqbuf {
    struct virtio_media_event_header hdr;
    struct v4l2_buffer buffer;
    struct v4l2_plane planes[VIRTIO_MEDIA_MAX_PLANES];
};
```

buffer *struct v4l2_buffer* describing the buffer that has been dequeued.

planes array of *struct v4l2_plane* containing the plane information for multi-planar buffers.

Note that in the case of a *SHARED_PAGES* buffer, the *struct v4l2_buffer* used as event payload is not followed by the buffer's SG entries: since that memory is the same that the driver submitted with the *VIDIOC_QBUF*, it would be redundant to have it here.

5.22.6.2.3.1 Driver Requirements: Device Operation: Dequeue buffer

Pointer values in the *struct v4l2_buffer* and *struct v4l2_plane* are meaningless and MUST be ignored by the driver. It is recommended that the device sets them to NULL in order to avoid leaking potential device addresses.

5.22.6.2.4 Device Operation: Emit an event

VIRTIO_MEDIA_EVT_EVENT Signals that a V4L2 event has been emitted for a session.

Every time an event for which the driver has been previously subcribed to (i.e., using the *VIDIOC_SUBSCRIBE_EVENT* ioctl) is signaled, the device queues a *struct virtio_media_event_event* event on the eventq. This is like an implicit *VIDIOC_DQEVENT* ioctl.

```
struct virtio_media_event_event {
    struct virtio_media_event_header hdr;
    struct v4l2_event event;
};
```

event *struct v4l2_event* describing the event that occurred.

5.23 RTC Device

The RTC (Real Time Clock) device provides information about current time. The device can provide different clocks, e.g. for the UTC or TAI time standards, or for physical time elapsed since some past epoch. The driver reads the clocks with simple or more accurate methods. Optionally, the driver can set an alarm.

5.23.1 Device ID

17

5.23.2 Virtqueues

0 requestq

1 alarmq

The driver enqueues requests to the requestq.

Through the alarmq, the device notifies the driver about alarm expirations. The alarmq exists only if VIRTIO_RTC_F_ALARM has been negotiated.

5.23.3 Feature bits

VIRTIO_RTC_F_ALARM (0) Device supports alarm.

VIRTIO_RTC_F_ALARM determines whether the device supports setting an alarm for some of the clocks.

5.23.3.1 Device Requirements: Feature bits

The device SHOULD offer VIRTIO_RTC_F_ALARM if the device can support setting an alarm for any of its clocks.

5.23.4 Device configuration layout

None currently defined.

5.23.5 Device Initialization

The device determines the set of clocks. The device provides zero or more clocks.

5.23.6 Device Operation

The driver makes a request available in the requestq. The device fills in the response and uses the buffer. The requestq uses common request and response headers.

```
/* common request header */
struct virtio_rtc_req_head {
        le16 msg_type;
        u8 reserved[6];
};

/* common response header */
struct virtio_rtc_resp_head {
        u8 status;
        u8 reserved[7];
};
```

The *msg_type* field identifies the message type.

The *status* field indicates whether the device successfully executed the request. The device sets the *status* field to one of the following values:

```
#define VIRTIO_RTC_S_OK         0
#define VIRTIO_RTC_S_EOPNOTSUPP 2
#define VIRTIO_RTC_S_ENODEV     3
#define VIRTIO_RTC_S_EINVAL     4
#define VIRTIO_RTC_S_EIO        5
```

VIRTIO_RTC_S_OK indicates that the device successfully executed the request. If a driver only makes requests according to the device capabilities detected by the driver, an error-free device will always set status VIRTIO_RTC_S_OK.

If *status* is not VIRTIO_RTC_S_OK, the value of other response fields is undefined.

VIRTIO_RTC_S_EOPNOTSUPP indicates that the device could not execute the specific request due to an implementation limitation. The device also returns status VIRTIO_RTC_S_EOPNOTSUPP for requests with unknown values in the *msg_type* or *hw_counter* fields.

VIRTIO_RTC_S_ENODEV indicates that the *clock_id* field value supplied with the request does not identify a clock.

VIRTIO_RTC_S_EINVAL indicates one or more of the following conditions:

- The driver request values are not allowed by the specification.
- The device read-only buffer is too small to fit the request.
- The device write-only buffer is too small to fit the response.

VIRTIO_RTC_S_EIO indicates that the device did not execute the request due to an error which was not caused by invalid input from the driver.

All *reserved* fields are written as zero.

The set of clocks does not change after feature negotiation completion, until device reset. The set of clocks should not change on device reset either (similar to negotiated features). Clock identifiers are zero-based, dense indices. In request structures and notification structures, all fields named *clock_id* contain clock identifiers.

5.23.6.1 Driver Requirements: Device Operation

The driver MUST interpret response fields other than the *struct virtio_rtc_resp_head* field *status* only when *status* is VIRTIO_RTC_S_OK.

The driver MUST set *reserved* fields in a device-readable buffer to zero.

The driver MUST NOT set bits in *flags* fields in a device-readable buffer which are not allowed according to the negotiated features.

The driver MUST NOT interpret *reserved* fields in a device-writable buffer.

The driver MUST only interpret these bits in *flags* fields in a device-writable buffer which are allowed according to the negotiated features.

The driver MUST allocate enough space for the response in a device-writable requestq buffer.

5.23.6.2 Device Requirements: Device Operation

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_OK if the device successfully executed the request.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to a status other than VIRTIO_RTC_S_OK if the device did not successfully execute the request.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EOPNOTSUPP if the device could not execute the specific request due to an implementation limitation.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EOPNOTSUPP for a request with a value of the *msg_type* field which is not described in this specification.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EOPNOTSUPP for a request with a value of the *hw_counter* field which is neither described in this specification nor otherwise known to the implementation.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_ENODEV if the *clock_id* field value supplied with the request does not identify a clock.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EINVAL if the request values are inconsistent with the specification and if the inconsistence is not described by the requirements which stipulate status VIRTIO_RTC_S_EOPNOTSUPP or VIRTIO_RTC_S_ENODEV.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EINVAL if the request specified in the request header through the *msg_type* field does not fit into the device read-only buffer.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EINVAL if the response specified in the request header through the *msg_type* field does not fit into the device write-only buffer, unless the *status* field does not fit into the device write-only buffer.

For *struct virtio_rtc_resp_head*, the device MUST NOT set the *status* field if the *status* field does not fit into the device write-only buffer.

For *struct virtio_rtc_resp_head*, the device MUST set the *status* field to VIRTIO_RTC_S_EIO if none of the previous requirements in this document stipulated another *status*.

If the device read-only buffer is bigger than the size of the request specified in the request header, the device MUST ignore the additional space.

If the device write-only buffer is bigger than the size of the response corresponding to the request header, the device MUST ignore the additional space.

The device MUST set *reserved* fields in a device-writable buffer to zero.

The device MUST NOT set bits in *flags* fields in a device-writable buffer which are not allowed according to the negotiated features.

During any period where the device remains live (keeps the DRIVER_OK *device status* bit set), the device MUST emit the same response for all repetitions of any specific request of type VIRTIO_RTC_REQ_CFG, VIRTIO_RTC_REQ_CLOCK_CAP, or VIRTIO_RTC_REQ_CROSS_CAP.

Whenever the device has a specific set of negotiated features, the device SHOULD emit the same response for all repetitions of any specific request of type VIRTIO_RTC_REQ_CFG, VIRTIO_RTC_REQ_CLOCK_CAP, or VIRTIO_RTC_REQ_CROSS_CAP, irrespective of any intermediate device resets.[17]

The device MUST use non-negative integers, which are smaller than the number of clocks, as clock identifiers.

5.23.6.3 Common Definitions

This section makes common definitions.

5.23.6.3.1 Clock Types

The following clock types are defined:

```
#define VIRTIO_RTC_CLOCK_UTC                    0
#define VIRTIO_RTC_CLOCK_TAI                    1
#define VIRTIO_RTC_CLOCK_MONOTONIC              2
#define VIRTIO_RTC_CLOCK_UTC_SMEARED            3
#define VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED      4
```

VIRTIO_RTC_CLOCK_UTC uses the UTC (Coordinated Universal Time) time standard. This clock uses the time epoch of January 1, 1970, 00:00 UTC. This is the same epoch as *Unix time*. The clock's seconds since the epoch are related to UTC time as defined by EPOCH.

This clock observes positive and negative leap seconds as announced by standard bodies. At the start of leap seconds, the clock steps accordingly.

VIRTIO_RTC_CLOCK_TAI uses the TAI (International Atomic Time) time standard. This clock uses the time epoch of January 1, 1970, 00:00 TAI.

VIRTIO_RTC_CLOCK_MONOTONIC uses monotonic physical time (SI seconds subdivisions) since some unspecified epoch. The epoch is before or during device reset.

VIRTIO_RTC_CLOCK_UTC_SMEARED deviates from the UTC standard by smearing time in the vicinity of a leap second. This avoids clock steps due to UTC leap seconds. Otherwise, this clock is similar to VIRTIO_RTC_CLOCK_UTC.

VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED This clock either

- deviates from the UTC standard by smearing time in the vicinity of a leap second (similar to VIRTIO_RTC_CLOCK_UTC_SMEARED), or
- steps at the start of leap seconds like VIRTIO_RTC_CLOCK_UTC.

A clock of type VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED can change this behavior for every leap second.

In the following, *UTC-like clock* designates any clock of type VIRTIO_RTC_CLOCK_UTC, VIRTIO_RTC_CLOCK_UTC_SMEARED, or VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED.

Additional clock types may be standardized in the future. Implementation-specific definitions of clock types are not recommended and are reserved for experimental implementations. Implementation-specific definitions use ids between 0xF0 and 0xFF.

5.23.6.3.2 Smearing Variants

Leap second *smearing variants* describe the deviation from the UTC standard in the vicinity of a leap second. The following smearing variants are currently defined:

[17] Failure to do so would interfere with resuming from suspend and error recovery.

```
#define VIRTIO_RTC_SMEAR_UNSPECIFIED    0
#define VIRTIO_RTC_SMEAR_NOON_LINEAR    1
#define VIRTIO_RTC_SMEAR_UTC_SLS        2
```

VIRTIO_RTC_SMEAR_UNSPECIFIED means that it is unspecified how time is smeared in the vicinity of leap seconds.

VIRTIO_RTC_SMEAR_NOON_LINEAR specifies a linear smear from noon prior to the leap second until noon after the leap second.

VIRTIO_RTC_SMEAR_UTC_SLS specifies a linear smear as per the UTC-SLS proposal.

Clocks of type VIRTIO_RTC_CLOCK_UTC_SMEARED always behave according to a smearing variant. The smearing variant does not change over the clock's lifetime.

For clocks of type VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED, it is unspecified whether leap seconds are smeared, and how leap seconds are smeared.

Additional smearing variants may be standardized in the future. Implementation-specific definitions of smearing variants are not recommended and are reserved for experimental implementations. Implementation-specific definitions use ids greater than or equal to 0xF0.

In the following, *leap smearing clock* designates any of the following clocks:

- any clock of type VIRTIO_RTC_CLOCK_UTC_SMEARED
- any clock of type VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED at any time when the clock is smearing a leap second.

5.23.6.3.3 Hardware Counters

The following hardware counter identifiers are specified:

```
/* Arm Generic Timer Counter-timer Virtual Count Register (CNTVCT_EL0) */
#define VIRTIO_RTC_COUNTER_ARM_VCT      0
/* x86 Time-Stamp Counter */
#define VIRTIO_RTC_COUNTER_X86_TSC      1
/* Invalid */
#define VIRTIO_RTC_COUNTER_INVALID   0xFF
```

Additional hardware counter identifiers may be standardized in the future. Implementation-specific hardware counter identifiers are not recommended and are reserved for experimental implementations. Implementation-specific hardware counter identifiers have values between 0xF0 and 0xFE.

5.23.6.4 Control Requests

Through *control requests*, the driver requests information about the device capabilities. The driver enqueues control requests in the requestq.

VIRTIO_RTC_REQ_CFG discovers the number of clocks.

```
#define VIRTIO_RTC_REQ_CFG 0x1000 /* message type */

struct virtio_rtc_req_cfg {
        struct virtio_rtc_req_head head;
        /* no request params */
};

struct virtio_rtc_resp_cfg {
        struct virtio_rtc_resp_head head;
        le16 num_clocks;
        u8 reserved[6];
};
```

The *num_clocks* field contains the number of clocks. A device provides zero or more clocks. Valid clock ids are those smaller than *num_clocks*.

VIRTIO_RTC_REQ_CLOCK_CAP discovers the capabilities of the clock identified by the *clock_id* field.

```
#define VIRTIO_RTC_REQ_CLOCK_CAP 0x1001 /* message type */

struct virtio_rtc_req_clock_cap {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        u8 reserved[6];
};

struct virtio_rtc_resp_clock_cap {
        struct virtio_rtc_resp_head head;
        u8 type;
        u8 leap_second_smearing;
        u8 flags;
        u8 reserved[5];
};
```

The *type* field identifies the clock type. A device provides zero or more clocks for a clock type.

Clocks of type VIRTIO_RTC_CLOCK_UTC_SMEARED indicate the *smearing variant* through the *leap_second_smearing* field. All other clocks set *leap_second_smearing* to VIRTIO_RTC_SMEAR_UNSPECIFIED.

The *flags* field provides the following information:

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

If VIRTIO_RTC_F_ALARM has been negotiated, the VIRTIO_RTC_FLAG_ALARM_CAP flag indicates that the clock supports an alarm.

VIRTIO_RTC_REQ_CROSS_CAP discovers whether the device supports cross-timestamping for a particular pair of clock and hardware counter.

```
#define VIRTIO_RTC_REQ_CROSS_CAP 0x1002 /* message type */

struct virtio_rtc_req_cross_cap {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        u8 hw_counter;
        u8 reserved[5];
};


struct virtio_rtc_resp_cross_cap {
        struct virtio_rtc_resp_head head;
#define VIRTIO_RTC_FLAG_CROSS_CAP       (1 << 0)
        u8 flags;
        u8 reserved[7];
};
```

The *clock_id* field identifies the clock, and the *hw_counter* field identifies the hardware counter, for which cross-timestamp support is probed. The device sets the VIRTIO_RTC_FLAG_CROSS_CAP flag in the *flags* field if the clock supports cross-timestamping for the particular clock and hardware counter, and clears the flag otherwise.

5.23.6.4.1 Driver Requirements: Control Requests

For VIRTIO_RTC_REQ_CROSS_CAP, the driver MUST set *hw_counter* to one of the hardware counter identifiers defined in this specification, or to a value between 0xF0 and 0xFE.

5.23.6.4.2 Device Requirements: Control Requests

For any clock of type VIRTIO_RTC_CLOCK_UTC, the device MUST use the UTC time standard (Coordinated Universal Time).

For any clock of type VIRTIO_RTC_CLOCK_UTC_SMEARED or VIRTIO_RTC_CLOCK_UTC_MAYBE_-SMEARED, the device MUST use the UTC time standard, insofar as the following requirements do not say otherwise.

For any UTC-like clock, the device MUST use the time epoch of January 1, 1970, 00:00 UTC.

For any UTC-like clock, the device MUST count seconds since the epoch according to EPOCH.

For any clock of type VIRTIO_RTC_CLOCK_UTC, the device MUST apply a positive leap second according to the UTC time standard by instantaneously stepping the clock backwards by 1 s at the start of the leap second.

For any clock of type VIRTIO_RTC_CLOCK_UTC, the device MUST apply a negative leap second according to the UTC time standard by instantaneously stepping the clock forward by 1 s at the start of the leap second.

For any leap smearing clock, the device MUST NOT step the clock due to a leap second.

For any leap smearing clock, on a positive leap second, the device MUST slow down the clock during part of the day containing the leap second and/or part of the day after the leap second.

For any leap smearing clock, on a negative leap second, the device MUST speed up the clock during part of the day containing the leap second and/or part of the day after the leap second.

For any clock with smearing variant VIRTIO_RTC_SMEAR_NOON_LINEAR, on a leap second, the device MUST change the frequency of the clock exactly from noon prior to the leap second until noon after the leap second.

For any clock with smearing variant VIRTIO_RTC_SMEAR_NOON_LINEAR, while changing the frequency of the clock due to a positive leap second, the device MUST decrease the frequency of the clock by $1/86400$.

For any clock with smearing variant VIRTIO_RTC_SMEAR_NOON_LINEAR, while changing the frequency of the clock due to a negative leap second, the device MUST increase the frequency of the clock by $1/86400$.

For any clock with smearing variant VIRTIO_RTC_SMEAR_UTC_SLS, on a leap second, the device MUST change the frequency of the clock exactly during the last 1000 seconds of the day with the leap second.

For any clock with smearing variant VIRTIO_RTC_SMEAR_UTC_SLS, while changing the frequency of the clock due to a positive leap second, the device MUST decrease the frequency of the clock by 0.1%.

For any clock with smearing variant VIRTIO_RTC_SMEAR_UTC_SLS, while changing the frequency of the clock due to a negative leap second, the device MUST increase the frequency of the clock by 0.1%.

For any clock of type VIRTIO_RTC_CLOCK_UTC_MAYBE_SMEARED, the device MAY deviate from the UTC standard with respect to leap second introduction.

For any clock of type VIRTIO_RTC_CLOCK_TAI, the device MUST use the TAI time standard (International Atomic Time).

For any clock of type VIRTIO_RTC_CLOCK_TAI, the device MUST use the time epoch of January 1, 1970, 00:00 TAI.

For any clock of type VIRTIO_RTC_CLOCK_MONOTONIC, the device MUST use SI seconds subdivisions.

For any clock of type VIRTIO_RTC_CLOCK_MONOTONIC, the device MUST use an epoch at a time instant before or during device reset.

For VIRTIO_RTC_REQ_CLOCK_CAP, and clock types other than VIRTIO_RTC_CLOCK_UTC_SMEARED, the device MUST set the *leap_second_smearing* field to VIRTIO_RTC_SMEAR_UNSPECIFIED.

For VIRTIO_RTC_REQ_CLOCK_CAP, and clock type VIRTIO_RTC_CLOCK_UTC_SMEARED, the device MUST set the *leap_second_smearing* field to VIRTIO_RTC_SMEAR_UNSPECIFIED, VIRTIO_RTC_-SMEAR_NOON_LINEAR, VIRTIO_RTC_SMEAR_UTC_SLS, or to a value greater than or equal to 0xF0.

The device SHOULD set the VIRTIO_RTC_FLAG_CROSS_CAP flag in the VIRTIO_RTC_REQ_CROSS_-CAP response if and only if the device would set status VIRTIO_RTC_S_OK for a VIRTIO_RTC_REQ_-READ_CROSS response with the same *hw_counter* and *clock_id* request values.

5.23.6.5 Read Requests

Through *read requests*, the driver requests clock readings from the device. The driver enqueues read requests in the requestq. The device obtains device-side clock readings and forwards these clock readings to the driver. The driver may enhance and interpret the clock readings through methods which are beyond the scope of this specification.

Once DRIVER_OK has been set, the device should support reading every clock, even when a clock may yet have to be aligned to reference time sources.

In general,

- clocks may jump backwards or forward, and
- the clock frequency may change. Clocks may be *slewed*, i.e. clocks may run at a frequency other than their current best frequency estimate.

As long as a clock does not jump backwards, the driver clock readings through read request responses increase monotonically:

- As long as a clock does not jump backwards in-between device-side clock readings, the driver-side readings for that clock increase monotonically as well, in the order in which the driver marks read requests as available.
- The device marks buffers with read requests for the same clock as used in the order in which the buffers are available.

For a clock of type VIRTIO_RTC_CLOCK_MONOTONIC, the device always returns monotonically increasing clock readings through read request responses.

The unit of all *clock_reading* fields is 1 nanosecond.[18]

VIRTIO_RTC_REQ_READ reads the clock identified by the *clock_id* field. The device supports this request for every clock.

```
#define VIRTIO_RTC_REQ_READ 0x0001 /* message type */

struct virtio_rtc_req_read {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        u8 reserved[6];
};

struct virtio_rtc_resp_read {
        struct virtio_rtc_resp_head head;
        le64 clock_reading;
};
```

clock_reading is a device-side clock reading obtained after the buffer was marked as available.

VIRTIO_RTC_REQ_READ_CROSS returns a cross-timestamp for the clock identified by the *clock_id* field.[19] This request may yield better performance than using VIRTIO_RTC_REQ_READ.

The driver can determine whether the device supports VIRTIO_RTC_REQ_READ_CROSS for a specific clock and *hw_counter* through VIRTIO_RTC_REQ_CROSS_CAP.

```
#define VIRTIO_RTC_REQ_READ_CROSS 0x0002 /* message type */

struct virtio_rtc_req_read_cross {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        u8 hw_counter;
        u8 reserved[5];
};
```

[18] For time epochs in year 1970 or later, this means that time until at least year 2553 can be represented in the *le64 clock_reading* fields.

[19] Cross-timestamping is similar to the ptp_kvm mechanism in the Linux kernel.

```
struct virtio_rtc_resp_read_cross {
        struct virtio_rtc_resp_head head;
        le64 clock_reading;
        le64 counter_cycles;
};
```

The *hw_counter* field specifies the hardware counter for which the driver requests a cross-timestamp.

Cross-timestamping returns a *clock_reading*, and an associated hardware counter value, *counter_cycles*. The *counter_cycles* field is the approximate or precise value which the driver would have read at the *clock_reading* time instant from the hardware counter identified by *hw_counter*.

To determine the *counter_cycles* value, the device converts the hardware counter value the device has read, accounting for any differences in counter offsets or counter multipliers between device and driver at the time of the reading.

In case hardware counter reads differ among CPUs used by the driver, the device should assume that the driver reads the hardware counter from the CPU which the driver enumerates as the first.

The hardware counter identifiers are defined in 5.23.6.3.3.

5.23.6.5.1 Driver Requirements: Read Requests

For VIRTIO_RTC_REQ_READ_CROSS, the driver MUST set *hw_counter* to one of the hardware counter identifiers defined in this specification, or to a value between 0xF0 and 0xFE.

5.23.6.5.2 Device Requirements: Read Requests

After DRIVER_OK has been set, the device SHOULD continuously support reading of all clocks.

For any two read requests to the same clock, the device MUST either obtain the *clock_reading* response value for the request which the driver makes available first before obtaining the *clock_reading* response value for the other request, or the device MUST return the same *clock_reading* values.

For any clock C, the device MUST mark all read requests reading C as used in the total order in which the driver marked these requests as available.

For any clock C of type VIRTIO_RTC_CLOCK_MONOTONIC and read requests *A* and *B* which read C, *A* being the request which the driver marks as available before *B*, the device MUST set the *clock_reading* response value for request *B* to a value greater than or equal to the *clock_reading* response value for request *A*.

For every clock, the device MUST support VIRTIO_RTC_REQ_READ.

For VIRTIO_RTC_REQ_READ and for any clock type listed in this specification, the device MUST use the nanosecond as unit for the *clock_reading* field.

For read requests, the device MUST obtain the *clock_reading* response value after the driver made the read request available.

For VIRTIO_RTC_REQ_READ_CROSS, the device MUST set *counter_cycles* to a value which approximates the value which the driver would have read from the hardware counter identified by *hw_counter* at the time instant when the device read the *clock_reading* value.

For VIRTIO_RTC_REQ_READ_CROSS, the device SHOULD assume that the driver reads the hardware counter identified by *hw_counter* through the CPU which the driver enumerates as the first.

For VIRTIO_RTC_REQ_READ_CROSS, the device MUST set *status* to a value other than VIRTIO_RTC_S_OK if the device cannot determine the approximate value which the driver would have read from the hardware counter identified by *hw_counter* at the time instant when the device read the *clock_reading* value.

If two VIRTIO_RTC_REQ_READ_CROSS requests read the same clock and the same hardware counter, and one request is made available before the other, the device MUST either

- set the later request's *counter_cycles* response to a value that the hardware counter shows after the earlier request's *counter_cycles* response, or
- set the same *counter_cycles* value in both responses.

For VIRTIO_RTC_REQ_READ_CROSS and for any clock type listed in this specification, the device MUST use the nanosecond as unit for the *clock_reading* field.

If the device sent an alarm notification for clock C with alarm time A, the device MUST, for all read requests of C which the driver marks as available after the notification, return a *clock_reading* which does not precede A (except if C stepped backwards to before A).

5.23.6.6 Alarm Operation

Through the optional alarm feature, the driver can set an alarm time. On alarm expiration, the device notifies the driver. On alarm expiration, the device may also wake up the driver, while the driver is in a sleep state, or while the driver is powered off. How this is done is beyond the scope of the specification. The driver can set one alarm time per clock, if the clock supports this.

The device may retain alarm times across device resets.[20]

If VIRTIO_RTC_F_ALARM has been negotiated, the device supports the alarm feature and the associated alarmq for notifications from the device. In addition, if the driver previously set an alarm time, even if the device

- no longer is live and/or
- no longer has negotiated VIRTIO_RTC_F_ALARM,

the device may still execute implementation-specific actions on alarm expiration.

An alarm expires in any of the following cases:

- When the associated clock progresses (also: steps) from a time prior to the alarm time to the alarm time, or to a time after the alarm time, while the alarm is enabled,
- when the driver sets an alarm time which is not in the future, while also setting the alarm to enabled,
- when the driver sets the alarm to enabled, and the alarm time is not in the future,
- when the device is reset, if the alarm time is retained and not in the future, and if the alarm is enabled.[21]

When an alarm expires, the driver can disable it. Otherwise, the alarm expires each time when one of the above expiration events occurs, even if it occurred before.[22]

On alarm expiration, the device executes the alarm actions. The alarm actions are:

- The device notifies the driver through the alarmq. If the device is not live, or no buffers are available in the alarmq, the device will notify once the device is live and buffers are available.
- Optionally, the device executes other, implementation-specific, actions. The device may execute those immediately, regardless of the device state.

An alarm *expiration* becomes obsolete on any of the following events:

- The driver disables the alarm.
- The driver sets an alarm time.
- The clock jumps backwards, before the alarm time.
- Another alarm expiration event happens.

20 Drivers may reset the device on boot or on resume from sleep state. It can make sense for the device to retain the alarm time then, similar to other alarm clocks.

21 The device is always responsible for detecting alarm expiration events. This avoids that the driver needs to reason about when it shall poll for alarm expiration.

22 This avoids that the driver may miss an alarm when the clock steps backwards after alarm expiration, but before the driver has resumed operation. This also facilitates distinct drivers using the same device, e.g. a driver in the bootloader, and a driver in the OS.

If an alarm expiration becomes obsolete, it is unspecified which alarm actions the device executes for this alarm expiration. When the driver disables an alarm, the device stops any alarm action for this alarm before using the buffer.

The device supports all alarm time values which the driver can request through alarm control requests. Initially, the alarm time is 0, and the alarm is disabled.

Alarms set prior to reset may cause unwanted alarm expiration notifications, and information leakage, after the reset. To prevent both issues, the driver can do the following after the reset, for each clock which supports alarm:

1. Make a VIRTIO_RTC_REQ_SET_ALARM request available, with *alarm_time* set to 0, and *flags* set to 0.
2. Wait until the device uses the buffer, with status VIRTIO_RTC_S_OK.

To prevent the above issues, the driver also marks buffers in the alarmq as available only after completing the above steps for all clocks.

5.23.6.6.1 Device Requirements: Alarm Operation

The device MAY retain both alarm time and alarm enabled status of a clock across a device reset.

If the device did not retain alarm time and alarm enabled status of a clock across a device reset, the device MUST initialize alarm time to 0.

If the device did not retain alarm time and alarm enabled status of a clock across a device reset, the device MUST disable the alarm.

If VIRTIO_RTC_F_ALARM has been negotiated, the device MUST support the alarm messages, VIRTIO_RTC_REQ_READ_ALARM, VIRTIO_RTC_REQ_SET_ALARM, VIRTIO_RTC_REQ_SET_ALARM_ENABLED, and VIRTIO_RTC_NOTIF_ALARM, for one or more clocks.

If VIRTIO_RTC_F_ALARM has not been negotiated, the device MUST NOT support the alarm messages.

The device MUST set the VIRTIO_RTC_FLAG_ALARM_CAP flag in *struct virtio_rtc_resp_clock_cap.flags* if the respective clock supports alarm messages, and clear the flag otherwise.

The device MUST consider it an alarm expiration event when the associated clock progresses (also: steps) from a time prior to the alarm time to the alarm time, or to a time after the alarm time, while the alarm is enabled.

The device MUST consider it an alarm expiration event when the driver sets an alarm time which the associated clock has already reached or passed, while also setting the alarm to enabled.

The device MUST consider it an alarm expiration event when the driver sets the alarm to enabled, if the clock has already reached or passed the alarm time.

If the device retained alarm time and alarm enabled status of a clock across a device reset, and the clock has already reached or passed the alarm time, the device MUST consider this device reset an alarm expiration event, if the alarm is enabled.

If an alarm expiration event E happens, the device MUST start serving the alarm expiration event E.

If the device is currently serving an alarm expiration event E, the device MUST use a single VIRTIO_RTC_NOTIF_ALARM notification for E, as soon as an alarmq buffer is available for this purpose.

While the device is serving an alarm expiration event, the device MAY execute implementation-specific alarm actions.

The device MAY ignore the device status when executing implementation-specific alarm actions.

The device MAY ignore whether VIRTIO_RTC_F_ALARM has been negotiated when executing implementation-specific alarm actions.

If the driver successfully disables an alarm for clock C with request VIRTIO_RTC_REQ_SET_ALARM or VIRTIO_RTC_REQ_SET_ALARM_ENABLED, the device MUST stop serving any previous alarm expiration event for C before the device uses the response buffer.

If the driver successfully requests VIRTIO_RTC_REQ_SET_ALARM, or VIRTIO_RTC_REQ_SET_ALARM_-ENABLED, for clock C, keeping the alarm enabled, the device MAY stop serving any previous alarm expiration event for C.

After a clock C stepped to a time previous to C's alarm time, the device MAY stop serving any previous alarm expiration event for C.

If an alarm expiration event happens for clock C, the device MAY stop serving any previous alarm expiration event for C.

5.23.6.6.2 Alarm Control Requests

If VIRTIO_RTC_F_ALARM has been negotiated,

- the driver can determine if a clock supports an alarm through the VIRTIO_RTC_FLAG_ALARM_CAP flag in the VIRTIO_RTC_REQ_CLOCK_CAP response,
- the driver can enqueue the alarm control requests into the requestq: VIRTIO_RTC_REQ_READ_-ALARM, VIRTIO_RTC_REQ_SET_ALARM, and VIRTIO_RTC_REQ_SET_ALARM_ENABLED.

The unit of all *alarm_time* fields is 1 nanosecond.

VIRTIO_RTC_REQ_READ_ALARM reads the current alarm.

```
#define VIRTIO_RTC_REQ_READ_ALARM 0x1003 /* message type */

struct virtio_rtc_req_read_alarm {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        u8 reserved[6];
};

struct virtio_rtc_resp_read_alarm {
        struct virtio_rtc_resp_head head;
        le64 alarm_time;
#define VIRTIO_RTC_FLAG_ALARM_ENABLED   (1 << 0)
        u8 flags;
        u8 reserved[7];
};
```

clock_id identifies the alarm through its associated clock. The *alarm_time* field returns the alarm time. In the *flags* field, VIRTIO_RTC_FLAG_ALARM_ENABLED indicates whether the alarm is enabled.

VIRTIO_RTC_REQ_SET_ALARM sets the alarm.

```
#define VIRTIO_RTC_REQ_SET_ALARM 0x1004 /* message type */

struct virtio_rtc_req_set_alarm {
        struct virtio_rtc_req_head head;
        le64 alarm_time;
        le16 clock_id;
        /* flag: VIRTIO_RTC_FLAG_ALARM_ENABLED */
        u8 flags;
        u8 reserved[5];
};

struct virtio_rtc_resp_set_alarm {
        struct virtio_rtc_resp_head head;
        /* no response params */
};
```

clock_id identifies the alarm through its associated clock. The *alarm_time* field sets the alarm time. If VIRTIO_RTC_FLAG_ALARM_ENABLED is set in the *flags* field, the device enables the alarm; otherwise, the device disables the alarm.

VIRTIO_RTC_REQ_SET_ALARM_ENABLED enables or disables the alarm.

```
#define VIRTIO_RTC_REQ_SET_ALARM_ENABLED 0x1005 /* message type */

struct virtio_rtc_req_set_alarm_enabled {
        struct virtio_rtc_req_head head;
        le16 clock_id;
        /* flag: VIRTIO_RTC_FLAG_ALARM_ENABLED */
        u8 flags;
        u8 reserved[5];
};

struct virtio_rtc_resp_set_alarm_enabled {
        struct virtio_rtc_resp_head head;
        /* no response params */
};
```

clock_id identifies the alarm through its associated clock. If VIRTIO_RTC_FLAG_ALARM_ENABLED is set in the *flags* field, the device enables the alarm; otherwise, the device disables the alarm.

When processing this request, the device retains the alarm time.

5.23.6.6.2.1 Driver Requirements: Alarm Control Requests

For VIRTIO_RTC_REQ_SET_ALARM and for any clock type listed in this specification, the driver MUST use the nanosecond as unit for the *alarm_time* field.

5.23.6.6.2.2 Device Requirements: Alarm Control Requests

If VIRTIO_RTC_F_ALARM has not been negotiated, the device MUST set status VIRTIO_RTC_S_ENODEV for VIRTIO_RTC_REQ_READ_ALARM, VIRTIO_RTC_REQ_SET_ALARM, and VIRTIO_RTC_REQ_SET_-ALARM_ENABLED.

If the clock does not support alarm messages, the device MUST set status VIRTIO_RTC_S_ENODEV for VIRTIO_RTC_REQ_READ_ALARM, VIRTIO_RTC_REQ_SET_ALARM, and VIRTIO_RTC_REQ_SET_ALARM_-ENABLED.

For VIRTIO_RTC_REQ_READ_ALARM, the device MUST set the *alarm_time* field to the alarm time.

For VIRTIO_RTC_REQ_READ_ALARM, the device MUST set the VIRTIO_RTC_FLAG_ALARM_ENABLED flag in the *flags* field if the alarm is enabled, and clear the flag otherwise.

For VIRTIO_RTC_REQ_READ_ALARM and for any clock type listed in this specification, the device MUST use the nanosecond as unit for the *alarm_time* field.

For VIRTIO_RTC_REQ_SET_ALARM, the device MUST accept any *alarm_time* value.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM, the device MUST set the alarm time to the time represented by the *alarm_time* field.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM, the device MUST enable the alarm if VIRTIO_RTC_FLAG_ALARM_ENABLED is set in the *flags* field.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM, the device MUST disable the alarm if VIRTIO_RTC_FLAG_ALARM_ENABLED is cleared in the *flags* field.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM_ENABLED, the device MUST enable the alarm if VIRTIO_RTC_FLAG_ALARM_ENABLED is set in the *flags* field.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM_ENABLED, the device MUST disable the alarm if VIRTIO_RTC_FLAG_ALARM_ENABLED is cleared in the *flags* field.

If the device sets status VIRTIO_RTC_S_OK for VIRTIO_RTC_REQ_SET_ALARM_ENABLED, the device MUST retain the alarm time.

5.23.6.6.3 Alarm Notifications

If the alarmq is present, the driver should make buffers available in the alarmq, which the device uses for alarm notifications. All alarmq fields are device-writable. The alarmq uses a common notification header.

```
/* common notification header */
struct virtio_rtc_notif_head {
        le16 msg_type;
        u8 reserved[6];
};
```

The *msg_type* field identifies the message type.

VIRTIO_RTC_NOTIF_ALARM notifies the driver about an alarm expiration.

```
#define VIRTIO_RTC_NOTIF_ALARM 0x2000 /* message type */

struct virtio_rtc_notif_alarm {
        struct virtio_rtc_notif_head head;
        le16 clock_id;
        u8 reserved[6];
};
```

clock_id identifies the expired alarm through its associated clock.

5.23.6.6.3.1 Driver Requirements: Alarm Notifications

If VIRTIO_RTC_F_ALARM has been negotiated, the driver SHOULD populate the alarmq with buffers.

The driver MUST allocate enough space for any alarmq notification in the device-writable part of an alarmq buffer.

If the driver receives a VIRTIO_RTC_NOTIF_ALARM notification, the driver SHOULD read the associated clock instead of assuming that the alarm time which the driver set last has been reached.

5.23.6.6.3.2 Device Requirements: Alarm Notifications

The device MUST NOT use a VIRTIO_RTC_NOTIF_ALARM notification for a clock which does not support alarm messages.

6 Reserved Feature Bits

Currently these device-independent feature bits are defined:

VIRTIO_F_INDIRECT_DESC (28) Negotiating this feature indicates that the driver can use descriptors with the VIRTQ_DESC_F_INDIRECT flag set, as described in 2.7.5.3 Indirect Descriptors and 2.8.7 Indirect Flag: Scatter-Gather Support.

VIRTIO_F_EVENT_IDX(29) This feature enables the *used_event* and the *avail_event* fields as described in 2.7.7, 2.7.8 and 2.8.10.

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 2.8 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 2.9 Driver Notifications.

VIRTIO_F_NOTIF_CONFIG_DATA(39) This feature indicates that the driver uses the data provided by the device as a virtqueue identifier in available buffer notifications. As mentioned in section 2.9, when the driver is required to send an available buffer notification to the device, it sends the virtqueue index to be notified. The method of delivering notifications is transport specific. With the PCI transport, the device can optionally provide a per-virtqueue value for the driver to use in driver notifications, instead of the

virtqueue index. Some devices may benefit from this flexibility by providing, for example, an internal virtqueue identifier, or an internal offset related to the virtqueue index.

This feature indicates the availability of such value. The definition of the data to be provided in driver notification and the delivery method is transport specific. For more details about driver notifications over PCI see 4.1.5.2.

VIRTIO_F_RING_RESET(40) This feature indicates that the driver can reset a queue individually. See 2.6.1.

VIRTIO_F_ADMIN_VQ(41) This feature indicates that the device exposes one or more administration virtqueues. At the moment this feature is only supported for devices using 4.1 Virtio Over PCI Bus as the transport and is reserved for future use for devices using other transports (see 2.2.1 and 2.2.2 for handling features reserved for future use.

VIRTIO_F_SUSPEND(43) This feature indicates that the driver can suspend the device by set the SUSPEND bit to 1. See 2.1.

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.

A driver SHOULD accept VIRTIO_F_NOTIF_CONFIG_DATA if it is offered.

6.2 Device Requirements: Reserved Feature Bits

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

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 2.7.4.3 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 four 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, 7.2.10, 7.2.11, 7.2.12, 7.2.13, 7.2.14, 7.2.15, 7.2.16, 7.2.17, 7.2.18, 7.2.19, 7.2.20, 7.2.21, 7.2.22, 7.2.23, 7.2.24 or 7.2.25.
- Clause 7.4.

Device A device MUST conform to four 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, 7.3.10, 7.3.11, 7.3.12, 7.3.13, 7.3.14, 7.3.15, 7.3.16, 7.3.17, 7.3.18, 7.3.19, 7.3.20, 7.3.21, 7.3.22, 7.3.23, 7.3.24, 7.3.25 or 7.3.26.
- Clause 7.4.

7.2 Clause 1: Driver Conformance

A driver MUST conform to the following normative statements:

- 2.1.1
- 2.2.1
- 2.4.2
- 2.5.1
- 2.7.1
- 2.6.1.1.2
- 2.7.4.2
- 2.7.5.2
- 2.7.5.3.1
- 2.7.7.1
- 2.7.6.1
- 2.7.8.3
- 2.7.10.1

- 2.7.13.3.1
- 2.7.13.4.1
- 2.8.11
- 2.8.16
- 2.8.17
- 2.8.19
- 2.8.21.1.1
- 2.8.21.3.1
- 3.1.1
- 3.3.1
- 6.1
- 2.12.1.2.5
- 2.12.1.3.6
- 2.12.1.4.4.2

7.2.1 Clause 2: 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
- 4.1.4.9.2
- 4.1.5.1.2.2
- 4.1.5.4.2

7.2.2 Clause 3: 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 Clause 4: 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

- 4.3.3.3.2

7.2.4 Clause 5: Network Driver Conformance

A network driver MUST conform to the following normative statements:

- 5.1.4.2
- 5.1.9.2.1
- 5.1.9.3.1
- 5.1.9.4.2
- 5.1.9.5.1.2
- 5.1.9.5.2.2
- 5.1.9.5.4.1
- 5.1.9.5.6.1
- 5.1.9.5.9.2
- 5.1.9.5.7.2
- 5.1.9.5.10.5
- 5.1.9.4.4.5
- 5.1.9.5.11.13
- 5.1.9.5.8.4
- 5.1.9.6.5
- 5.1.9.7.5

7.2.5 Clause 6: Block Driver Conformance

A block driver MUST conform to the following normative statements:

- 5.2.5.1
- 5.2.6.1

7.2.6 Clause 7: 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 Clause 8: Entropy Driver Conformance

An entropy driver MUST conform to the following normative statements:

- 5.4.6.1

7.2.8 Clause 9: 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

- 5.5.6.5.1
- 5.5.6.6.1
- 5.5.6.7.1

7.2.9 Clause 10: 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 Clause 11: Input Driver Conformance

An input driver MUST conform to the following normative statements:

- 5.8.5.1
- 5.8.6.1

7.2.11 Clause 12: Crypto Driver Conformance

A Crypto driver MUST conform to the following normative statements:

- 5.9.5.2
- 5.9.6.1
- 5.9.9.2.1.6
- 5.9.9.2.1.8
- 5.9.9.4.1
- 5.9.9.5.1
- 5.9.9.6.1
- 5.9.9.7.1
- 5.9.9.9.5

7.2.12 Clause 13: Socket Driver Conformance

A socket driver MUST conform to the following normative statements:

- 5.10.3.1
- 5.10.6.3.1
- 5.10.6.4.1
- 5.10.6.7.1

7.2.13 Clause 14: File System Driver Conformance

A file system driver MUST conform to the following normative statements:

- 5.11.4.1
- 5.11.6.2.2
- 5.11.6.3.1
- 5.11.6.4.2

7.2.14 Clause 15: RPMB Driver Conformance

A RPMB driver MUST conform to the following normative statements:

- 5.12.6.2

7.2.15 Clause 16: IOMMU Driver Conformance

An IOMMU driver MUST conform to the following normative statements:

- 5.13.3.1
- 5.13.4.1
- 5.13.5.1
- 5.13.6.1
- 5.13.6.3.1
- 5.13.6.4.1
- 5.13.6.5.1
- 5.13.6.6.1
- 5.13.6.7.1
- 5.13.6.8.1.1
- 5.13.6.9.1

7.2.16 Clause 17: Sound Driver Conformance

A sound driver MUST conform to the following normative statements:

- 5.14.5.1
- 5.14.6.2
- 5.14.6.6.3.2
- 5.14.6.8.1.2
- 5.14.6.8.2.2
- 5.14.6.10.2.1
- 5.14.6.10.3.2

7.2.17 Clause 18: Memory Driver Conformance

A memory driver MUST conform to the following normative statements:

- 5.15.4.1
- 5.15.5.1
- 5.15.6.1
- 5.15.6.3.1
- 5.15.6.4.1
- 5.15.6.5.1
- 5.15.6.6.1

7.2.18 Clause 19: I2C Adapter Driver Conformance

An I2C Adapter driver MUST conform to the following normative statements:

- 5.16.6.3

7.2.19 Clause 20: SCMI Driver Conformance

An SCMI driver MUST conform to the following normative statements:

- 5.17.6.1.2
- 5.17.6.2.1

7.2.20 Clause 21: GPIO Driver Conformance

A General Purpose Input/Output (GPIO) driver MUST conform to the following normative statements:

- 5.18.6.8
- 5.18.7.2

7.2.21 Clause 22: PMEM Driver Conformance

A PMEM driver MUST conform to the following normative statements:

- 5.19.5.2

7.2.22 Clause 23: CAN Driver Conformance

A CAN driver MUST conform to the following normative statements:

- 5.20.4.1

7.2.23 Clause 24: SPI Controller Driver Conformance

An SPI Controller driver MUST conform to the following normative statements:

- 5.21.6.3

7.2.24 Clause 25: Media Driver Conformance

A Media driver MUST conform to the following normative statements:

- 5.22.6.1.2
- 5.22.6.1.4.1
- 5.22.6.1.5.3
- 5.22.6.2.2.2
- 5.22.6.2.3.1

7.2.25 Clause 26: RTC Driver Conformance

An RTC driver MUST conform to the following normative statements:

- 5.23.6.1
- 5.23.6.4.1
- 5.23.6.5.1
- 5.23.6.6.2.1
- 5.23.6.6.3.1

7.3 Clause 27: Device Conformance

A device MUST conform to the following normative statements:

- 2.1.2
- 2.2.2
- 2.4.1
- 2.5.2
- 2.6.1.1.1
- 2.7.4.1
- 2.7.5.1
- 2.7.5.3.2
- 2.7.7.2
- 2.7.8.2
- 2.7.10.2
- 2.8.12
- 2.8.15
- 2.8.18
- 2.10.2
- 6.2
- 2.12.1.2.4
- 2.12.1.3.5
- 2.12.1.4.4.1

7.3.1 Clause 28: 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
- 4.1.4.9.1
- 4.1.4.11.0.1
- 4.1.5.1.2.1
- 4.1.5.3.1
- 4.1.5.4.1

7.3.2 Clause 29: MMIO Device Conformance

An MMIO device MUST conform to the following normative statements:

- 4.2.2.1

7.3.3 Clause 30: 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
- 4.3.3.3.1

7.3.4 Clause 31: Network Device Conformance

A network device MUST conform to the following normative statements:

- 5.1.4.1
- 5.1.9.2.2
- 5.1.9.3.2
- 5.1.9.4.1
- 5.1.9.5.1.1
- 5.1.9.5.2.1
- 5.1.9.5.3.1
- 5.1.9.5.4.2
- 5.1.9.5.6.2
- 5.1.9.5.7.3
- 5.1.9.5.10.6
- 5.1.9.4.4.4
- 5.1.9.5.11.12
- 5.1.9.5.8.3
- 5.1.9.6.4
- 5.1.9.7.4

7.3.5 Clause 32: Block Device Conformance

A block device MUST conform to the following normative statements:

- 5.2.5.2
- 5.2.6.2

7.3.6 Clause 33: 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 Clause 34: Entropy Device Conformance

An entropy device MUST conform to the following normative statements:

- 5.4.6.2

7.3.8 Clause 35: 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
- 5.5.6.5.2
- 5.5.6.6.2
- 5.5.6.7.2

7.3.9 Clause 36: 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 Clause 37: GPU Device Conformance

A GPU device MUST conform to the following normative statements:

- 5.7.5
- 5.7.6.5

7.3.11 Clause 38: Input Device Conformance

An input device MUST conform to the following normative statements:

- 5.8.5.2
- 5.8.6.2

7.3.12 Clause 39: Crypto Device Conformance

A Crypto device MUST conform to the following normative statements:

- 5.9.5.1
- 5.9.9.2.1.7
- 5.9.9.2.1.9
- 5.9.9.4.2

- 5.9.9.5.2
- 5.9.9.6.2
- 5.9.9.7.2
- 5.9.9.9.4

7.3.13 Clause 40: Socket Device Conformance

A socket device MUST conform to the following normative statements:

- 5.10.3.2
- 5.10.6.3.2
- 5.10.6.4.2

7.3.14 Clause 41: File System Device Conformance

A file system device MUST conform to the following normative statements:

- 5.11.4.2
- 5.11.6.2.1
- 5.11.6.4.1

7.3.15 Clause 42: RPMB Device Conformance

An RPMB device MUST conform to the following normative statements:

- 5.12.5
- 5.12.6.1.1
- 5.12.6.1.2
- 5.12.6.1.3
- 5.12.6.1.4
- 5.12.6.1.5
- 5.12.6.3

7.3.16 Clause 43: IOMMU Device Conformance

An IOMMU device MUST conform to the following normative statements:

- 5.13.3.2
- 5.13.4.2
- 5.13.6.2
- 5.13.6.3.2
- 5.13.6.4.2
- 5.13.6.5.2
- 5.13.6.6.2
- 5.13.6.7.2
- 5.13.6.8.1.2
- 5.13.6.9.2

7.3.17 Clause 44: Sound Device Conformance

A sound device MUST conform to the following normative statements:

- 5.14.6.4.1.1
- 5.14.6.6.2.1
- 5.14.6.6.3.1
- 5.14.6.6.5.1
- 5.14.6.8.1.1
- 5.14.6.8.2.1
- 5.14.6.9.1.1
- 5.14.6.10.1.1
- 5.14.6.10.3.1
- 5.14.6.10.4.1

7.3.18 Clause 45: Memory Device Conformance

A memory device MUST conform to the following normative statements:

- 5.15.4.2
- 5.15.5.2
- 5.15.6.2
- 5.15.6.3.2
- 5.15.6.4.2
- 5.15.6.5.2
- 5.15.6.6.2

7.3.19 Clause 46: I2C Adapter Device Conformance

An I2C Adapter device MUST conform to the following normative statements:

- 5.16.6.4

7.3.20 Clause 47: SCMI Device Conformance

An SCMI device MUST conform to the following normative statements:

- 5.17.3.1
- 5.17.6.1.1
- 5.17.6.3.1
- 5.17.6.4.1

7.3.21 Clause 48: GPIO Device Conformance

A General Purpose Input/Output (GPIO) device MUST conform to the following normative statements:

- 5.18.6.9
- 5.18.7.3

7.3.22 Clause 49: PMEM Device Conformance

A PMEM device MUST conform to the following normative statements:

- 5.19.5.1
- 5.19.7.1
- 5.19.7.3

7.3.23 Clause 50: CAN Device Conformance

A CAN device MUST conform to the following normative statements:

- 5.20.5.2

7.3.24 Clause 51: SPI Controller Device Conformance

An SPI Controller device MUST conform to the following normative statements:

- 5.21.6.4

7.3.25 Clause 52: Media Device Conformance

A Media device MUST conform to the following normative statements:

- 5.22.6.1.3.1
- 5.22.6.1.5.2
- 5.22.6.1.5.5
- 5.22.6.1.5.8
- 5.22.6.1.6.1
- 5.22.6.1.7.1
- 5.22.6.1.8.3
- 5.22.6.1.8.5
- 5.22.6.2.2.1

7.3.26 Clause 53: RTC Device Conformance

An RTC device MUST conform to the following normative statements:

- 5.23.3.1
- 5.23.6.2
- 5.23.6.4.2
- 5.23.6.5.2
- 5.23.6.6.1
- 5.23.6.6.2.2
- 5.23.6.6.3.2

7.4 Clause 54: Legacy Interface: Transitional Device and Transitional Driver Conformance

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

An implementation MAY choose to implement OPTIONAL support for the legacy interface, including support for legacy drivers or devices, by 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 2.5.3
- Section 2.5.4
- Section 2.7.2
- Section 2.7.3
- Section 2.7.4.3
- Section 2.12.1.1.6
- Section 2.12.1.1.7
- Section 3.1.2
- Section 4.1.2.3
- Section 4.1.4.10
- 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 5.1.9.1
- Section 5.1.9.5.2.3
- Section 5.1.9.5.3.2
- Section 5.1.9.5.6.3
- Section 5.1.9.5.9.3
- 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/v1.3/csd01/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-comment@lists.linux.dev (after subscribing through virtio-comment+subscribe@lists.linux.dev) 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!

B.6 How to define a new device part?

Few considerations are necessary when creating new device part or when extending the device part structure. If the new field is generic for all the device types or most of the device types, a new device part should be defined as common device part by claiming a new *type* value. If the new field is unique to a device type, a new device specific device part should be added. range.

B.7 When to define a new device part?

When the device part for a specific field does not exists, one should define a new device part.

B.8 How to avoid device part duplication with existing structure?

Each device should reuse any existing field definition that may exists as part of device control virtqueue or any other existing structure definition.

B.9 How to extend the existing device part definition?

When a field is missing in already defined device part, a new field should be added at the end of the existing device part. New field MUST not be added at beginning or in the middle of the device part structure. Any field which is already present MUST NOT be removed.

Appendix C. Creating New Transports

Devices and drivers utilize various transport methods to facilitate communication, such as PCI, MMIO, or Channel I/O. These transport methods determine aspects of the interaction between the device and the driver, including device discovery, capability exchange, interrupt handling, and data transfer. For instance, in a host/guest architecture, the host might expose a device to the guest via a virtual PCI bus, and the guest would use a PCI device driver to interface with the device.

This section outlines the mandatory requirements that a transport method implements.

A transport provides a mechanism to implement configuration space for the device.

A transport provides a mechanism for the driver to identify the device type.

A transport provides a mechanism for the driver to read the device's FEATURES_OK and DEVICE_NEEDS_-RESET status bits.

A transport provides a mechanism for the driver to modify the device's status.

A transport provides a mechanism for the driver to read and modify the device's feature bits.

A transport allows one or more virtqueues to be implemented by the device. The number of virtqueues is device specific and not specified by the transport.

A transport provides a mechanism for the driver to communicate virtqueue configuration and memory location to the device.

A transport provides a mechanism for the device to send device notifications to the driver, such as used buffer notifications.

A transport provides a mechanism for the driver to send driver notifications to the device, such as available buffer notifications.

A transport provides a mechanism for the driver to initiate a device reset.

Appendix D. Acknowledgements

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants

Alvaro Karsz, SolidRun
Anton Yakovlev, OpenSynergy
Cornelia Huck, Red Hat
David Edmondson, Oracle
David Hildenbrand, Red Hat
Dmitry Fomichev, Western Digital
Dust Li, Alibaba
Enrico Granata, Google
Haixu Cui, Quic Inc
Halil Pasic, IBM
Heng Qi, Alibaba
Hrishivarya Bhageeradhan, OpenSynergy
Jan Kiszka, Siemens
Jiri Pirko, Nvidia
Laura Loghin, Amazon
Lei He, Bytedance
Lingshan Zhu, Intel
Matti Moell, OpenSynergy
Michael S. Tsirkin, Red Hat
Mihai Carabas, Oracle
Parav Pandit, Nvidia
Ran Koren, Nvidia
Satananda Burla, Marvell
Shahaf Shuler, Nvidia
Si-Wei Liu, Oracle
Stefan Hajnoczi, Red Hat
Stefano Garzarella, Red Hat
Xuan Zhuo, Alibaba
Yuri Benditovich, Red Hat / Daynix
Zhenwei Pi, Bytedance

The following non-members have provided valuable feedback on this specification and are gratefully acknowledged:

Reviewers

Damien Le Moal, Western Digital
Hans Holmberg, Western Digital
Hans Zhang, Alibaba
He Rongguang, Alibaba
Helin Guo, Alibaba
Jiang Liu, Alibaba
Matias Bjørling, Western Digital

Max Gurtovoy, Nvidia
Niklas Cassel, Western Digital
Tony Lu, Alibaba

The following individuals have participated in the creation of previous versions of this specification and are gratefully acknowledged:

Participants

Alexander Duyck, Intel
Alex Bennée, Linaro
Allen Chia, Oracle
Amit Shah, Red Hat
Amos Kong, Red Hat
Anthony Liguori, IBM
Anton Yakovlev, OpenSynergy
Arseny Krasnov, Kaspersky Lab
Bruce Rogers, SUSE
Bryan Venteicher, NetApp
Chandra Thyamagondlu, Xilinx
Chet Ensign, OASIS
Cornelia Huck, Red Hat
Cunming, Liang, Intel
Damjan, Marion, Cisco
Daniel Kiper, Oracle
David Hildenbrand, Red Hat
David Stevens, Chromium
Dr. David Alan Gilbert, Red Hat
Enrico Granata, Google
Eugenio Pérez, Red Hat
Fang Chen, Huawei
Fang You, Huawei
Felipe Franciosi, Nutanix
Gaetan Harter, OpenSynergy
Geoff Brown, M2Mi
Gerd Hoffmann, Red Hat
Gershon Janssen, Individual Member
Grant Likely, ARM
Gurchetan Singh, Chromium
Haggai Eran, Mellanox
Halil Pasic, IBM
Hao Chen, Google
Huang Yang, Intel
James Bottomley, Parallels IP Holdings GmbH
Jani Kokkonen, Huawei
Jan Kiszka, Siemens AG
Jean-Philippe Brucker, Linaro
Jens Freimann, Red Hat
Jian Zhou, Huawei
Jiang Wang, Bytedance
Jie Deng, Intel
Joel Nider, Individual
Johannes Berg, Intel
Junji Wei, Bytedance
Karen Xie, Xilinx
Keiichi Watanabe, Chromium
Kumar Sanghvi, Xilinx

Lei Gong, Huawei
Lior Narkis, Mellanox
Luiz Capitulino, Red Hat
Marc-André Lureau, Red Hat
Marcel Holtmann, Individual
Mark Gray, Intel
Michael S. Tsirkin, Red Hat
Mihai Carabas, Oracle
Nikos Dragazis, Arrikto
Nishank Trivedi, NetApp
Pankaj Gupta, Red Hat
Paolo Bonzini, Red Hat
Paul Mundt, Huawei
Pawel Moll, ARM
Peng Long, Huawei
Peter Hilber, OpenSynergy
Petre Eftime, Amazon
Philipp Hahn, Univention
Piotr Uminski, Intel
Qian Xum, Intel
Richard Sohn, Alcatel-Lucent
Rob Bradford, Intel
Rusty Russell, IBM
Sasha Levin, Oracle
Sergey Tverdyshev, Thales e-Security
Stefan Fritsch, Individual
Stefan Hajnoczi, Red Hat
Sundar Mohan, Xilinx
Taylor Stark, Microsoft
Tiwei Bie, Intel
Tom Lyon, Samya Systems, Inc.
Victor Kaplansky, Red Hat
Vijay Balakrishna, Oracle
Viresh Kumar, Linaro
Vitaly Mireyno, Marvell
Wei Wang, Intel
Xin Zeng, Intel
Yadong Qi, Intel
Yoni Bettan, Red Hat
Yuri Benditovich, Red Hat / Daynix

The following non-members have provided valuable feedback on previous versions of this specification and are gratefully acknowledged:

Reviewers

Aaron Conole, Red Hat
Adam Tao, Huawei
Alexander Duyck, Intel
Andreas Pape, ADITG/ESB
Andrew Thornton, Google
Arnd Bergmann, Individual
Arun Subbarao, LynuxWorks
Baptiste Reynal, Virtual Open Systems
Bharat Bhushan, NXP Semiconductors
Bing Zhu, Intel
Brian Foley, ARM

Chandra Thyamagondlu, Xilinx
Changpeng Liu, Intel
Christian Pinto, Virtual Open Systems
Christoffer Dall, ARM
Christoph Hellwig, Individual
Christophe de Dinechin, Red Hat
Christian Borntraeger, IBM
Daniel Marcovitch, Mellanox
David Alan Gilbert, Red Hat
David Hildenbrand, Red Hat
David Riddoch, Solarflare
Denis V. Lunev, OpenVZ
Dmitry Fleytman, Red Hat
Don Wallwork, Broadcom
Eduardo Habkost, Red Hat
Emily Drea, ARM
Eric Auger, Red Hat
Fam Zheng, Red Hat
Francesco Fusco, Red Hat
Frank Yang, Google
Gil Savir, Intel
Gonglei (Arei), Huawei
Greg Kurz, IBM
Hannes Reiencke, SUSE
Ian Campbell, Docker
Ilya Lesokhin, Mellanox
Jacques Durand, Fujutsu
Jakub Jermar, Kernkonzept
Jan Scheurich, Ericsson
Jason Baron, Akamai
Jason Wang, Red Hat
Jean-Philippe Brucker, ARM
Jens Freimann, Red Hat
Jianfeng Tan, intel
Jonathan Helman, Oracle
Karandeep Chahal, DDN
Kevin Lo, MSI
Kevin Tian, Intel
Kully Dhanoa, Intel
Laura Novich, Red Hat
Ladi Prosek, Red Hat
Lars Ganrot, Napatech
Linus Walleij, Linaro
Longpeng (Mike), Huawei
Mario Torrecillas Rodriguez, ARM
Mark Rustad, Intel
Matti Möll, OpenSynergy
Maxime Coquelin, Red Hat
Namhyung Kim, LG
Ola Liljedahl, ARM
Pankaj Gupta, Red Hat
Paolo Bonzini, Red Hat
Patrick Durusau, OASIS
Pierre Pfister, Cisco
Pranavkumar Sawargaonkar, Linaro
Rauchfuss Holm, Huawei
Rob Miller, Broadcom

Roman Kiryanov, Google
Robin Cover, OASIS
Roger S Chien, Intel
Ruchika Gupta, Linaro
Sameeh Jubran, Red Hat / Daynix
Si-Wei Liu, Oracle
Sridhar Samudrala, Intel
Stefan Fritsch, Individual
Stefano Garzarella, Red Hat
Steven Luong, Cisco
Thomas Huth, Red Hat
Tiwei Bie, Intel
Tomáš Golembiovský, Red Hat
Tomas Winkler, Intel
Venu Busireddy, Oracle
Victor Kaplansky, Red Hat
Vijayabhaskar Balakrishna, Oracle
Vlad Yasevich, Red Hat
Yan Vugenfirer, Red Hat / Daynix
Wei Xu, Red Hat
Will Deacon, ARM
Willem de Bruijn, Google
Yang Huang, Intel
Yuanhan Liu, Intel
Yuri Benditovich, Red Hat / Daynix
Zhi Yong Wu, IBM
Zhoujian, Huawei

Appendix E. Revision History

The following changes have been made since version 1.2 of this specification:

Revision	Date	Editor	Changes Made
5da7c1414e7e	13 Jun 2022	Stefan Hajnoczi	virtio-blk: document that the capacity field can change Block devices can change size during operation. A configuration change notification is sent by the device and the driver detects that the field has changed. Document this behavior that has already been implemented in Linux and QEMU since 2011. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/136 Signed-off-by: Stefan Hajnoczi <stefanha@redhat.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.2.6.
ad2e1674bb69	13 Jun 2022	Laura Loghin	vsock: add documentation about len header field Fixes: https://github.com/oasis-tcs/virtio-spec/issues/137 Reviewed-by: Stefano Garzarella <sgarzare@redhat.com> Signed-off-by: Laura Loghin <lauralg@amazon.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.10.6.
fca015771bc9	13 Jun 2022	Xuan Zhuo	virtio-net: support reset queue A separate reset queue function introduced by Virtqueue Reset. However, it is currently not defined what to do if the destination queue is being reset when virtio-net is steering in multi-queue mode. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/138 Reviewed-by: Jason Wang <jasowang@redhat.com> Signed-off-by: Xuan Zhuo <xuanzhuo@linux.alibaba.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.9.5.6, and 5.1.9.5.7.1.
6328f51e21b5	24 Jun 2022	Yuri Benditovich	virtio-net: define guest USO features Fixes: https://github.com/oasis-tcs/virtio-spec/issues/120 Add definition for large UDP packets device-to-driver. Signed-off-by: Yuri Benditovich <yuri.benditovich@daynix.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.3, 5.1.3.1, 5.1.5, 5.1.9.3, 5.1.9.4, and 5.1.9.5.9.1.

Revision	Date	Editor	Changes Made
49ff7805924c	24 Jun 2022	Anton Yakovlev	virtio-snd: add support for audio controls This patch extends the virtio sound device specification by adding support for audio controls. Audio controls can be used to set the volume level, mute/unmute the audio signal, switch different modes/states of the virtual sound device, etc. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/107 Signed-off-by: Anton Yakovlev <anton.yakovlev@opensynergy.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.2.16, 7.3.17, 5.14.3, 5.14.4, 5.14.6, and 5.14.6.10.
4d9068effa81	11 Jul 2022	Alvaro Karsz	Introduction of Virtio Network device notifications coalescing feature. Control a network device notifications coalescing parameters using the control virtqueue. A new control class was added: VIRTIO_NET_CTRL_NOTF_COAL. This class provides 2 commands: • VIRTIO_NET_CTRL_NOTF_COAL_TX_SET: Ask the network device to change the tx_usecs and tx_max_packets parameters. – tx_usecs: Maximum number of usecs to delay a TX notification. – tx_max_packets: Maximum number of packets to send before a TX notification. • VIRTIO_NET_CTRL_NOTF_COAL_RX_SET: Ask the network device to change the rx_usecs and rx_max_packets parameters. – rx_usecs: Maximum number of usecs to delay a RX notification. – rx_max_packets: Maximum number of packets to receive before a RX notification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/141 Reviewed-by: Jason Wang <jasowang@redhat.com> Signed-off-by: Alvaro Karsz <alvaro.karsz@solid-run.com> [CH: fixed commit message] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.2.4, 7.3.4, 5.1.3, 5.1.3.1, and 5.1.9.5.10.
abbe8afda8db	03 Aug 2022	Lei He	virtio-crypto: introduce akcipher service Introduce akcipher (asymmetric key cipher) service type, several asymmetric algorithms and relevent information: - RSA(padding algorithm, ASN.1 schema definition) - ECDSA(ECC algorithm) Fixes: https://github.com/oasis-tcs/virtio-spec/issues/129 Signed-off-by: zhenwei pi <pizhenwei@bytedance.com> Signed-off-by: Lei He <helei.sig11@bytedance.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 1.1, 5.9, 5.9.3, 5.9.3.1, 5.9.4, 5.9.4.5, 5.9.5, 5.9.9.1, 5.9.9.2, 5.9.9.2.1.5, 5.9.9.3, and 5.9.9.8.

Revision	Date	Editor	Changes Made
26ed30ccb049	03 Aug 2022	Stefano Garzarella	virtio-vsock: add VIRTIO_VSOCK_F_NO_IMPLIED_STREAM feature bit Initially virtio-vsock only supported the stream type, which is why there was no feature. Later we added the seqpacket type and in the future we may have other types (e.g. datagram). seqpacket is an extension of stream, so it might be implied that if seqpacket is supported, stream is too, but this might not be true for other types. As we discussed here [1] should be better to add a new VIRTIO_VSOCK_-F_NO_IMPLIED_STREAM feature bit to avoid this implication. Let's also add normative sections to better define the behavior when VIR-TIO_VSOCK_F_NO_IMPLIED_STREAM is negotiated or not. [1] http://markmail.org/message/2s3qd74drgjxkvte Fixes: https://github.com/oasis-tcs/virtio-spec/issues/142 Suggested-by: Michael S. Tsirkin <mst@redhat.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Stefano Garzarella <sgarzare@redhat.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.2.12, 7.3.13, and 5.10.3.
a7251b0cb4d9	14 Nov 2022	Hrishivarya Bhageer-adhan	content: reserve device ID 43 for Camera device The virtio-camera device allows to stream a camera video with ability to change controls, formats and get camera captures. This patch is to reserve the next available device ID for virtio-camera. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/148 Signed-off-by: Hrishivarya Bhageeradhan <hrishiv-arya.bhageeradhan@opensynergy.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.

Revision	Date	Editor	Changes Made
b4e8efa0fa6c	05 Dec 2022	Dmitry Fomichev	virtio-blk: add zoned block device specification Introduce support for Zoned Block Devices to virtio. Zoned Block Devices (ZBDs) aim to achieve a better capacity, latency and/or cost characteristics compared to commonly available block devices by getting the entire LBA space of the device divided to block regions that are much larger than the LBA size. These regions are called zones and they can only be written sequentially. More details about ZBDs can be found at https://zonedstorage.io/docs/introduction/zoned-storage . In its current form, the virtio protocol for block devices (virtio-blk) is not aware of ZBDs but it allows the driver to successfully scan a host-managed drive provided by the virtio block device. As the result, the host-managed drive is recognized by virtio driver as a regular, non-zoned drive that will operate erroneously under the most common write workloads. Host-aware ZBDs are currently usable, but their performance may not be optimal because the driver can only see them as non-zoned block devices. To fix this, the virtio-blk protocol needs to be extended to add the capabilities to convey the zone characteristics of ZBDs at the device side to the driver and to provide support for ZBD-specific commands - Report Zones, four zone operations (Open, Close, Finish and Reset) and (optionally) Zone Append. The proposed standard extension aims to define this new functionality. This patch extends the virtio-blk section of virtio specification with the minimum set of requirements that are necessary to support ZBDs. The resulting device model is a subset of the models defined in ZAC/ZBC and ZNS standards documents. The included functionality mirrors the existing Linux kernel block layer ZBD support and should be sufficient to handle the host-managed and host-aware HDDs that are on the market today as well as ZNS SSDs that are entering the market at the time of submission of this patch. I would like to thank the following people for their useful feedback and suggestions while working on the initial iterations of this patch. Damien Le Moal <damien.lemoal@opensource.wdc.com> Matias Bjørling <Matias.Bjorling@wdc.com> Niklas Cassel <Niklas.Cassel@wdc.com> Hans Holmberg <Hans.Holmberg@wdc.com> Fixes: https://github.com/oasis-tcs/virtio-spec/issues/143 Signed-off-by: Dmitry Fomichev <dmitry.fomichev@wdc.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Reviewed-by: Damien Le Moal <damien.lemoal@opensource.wdc.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.2.3, 5.2.4, 5.2.5, and 5.2.6.

Revision	Date	Editor	Changes Made
985bbf397db4	07 Dec 2022	Xuan Zhuo	content: reserve device ID 44 for ISM device The virtio-ism device provides the ability to share memory between different guests on a host. A guest's memory got from ism device can be shared with multiple peers at the same time. This shared relationship can be dynamically created and released. The shared memory obtained from the device is divided into multiple ism regions for share. ISM device provides a mechanism to notify other ism region referrers of content update events. This patch is to reserve the next available device ID for virtio-ism. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/150 Signed-off-by: Xuan Zhuo <xuanzhuo@linux.alibaba.com> Signed-off-by: Jiang Liu <gerry@linux.alibaba.com> Signed-off-by: Dust Li <dust.li@linux.alibaba.com> Signed-off-by: Tony Lu <tonylu@linux.alibaba.com> Signed-off-by: Helin Guo <helinguo@linux.alibaba.com> Signed-off-by: Hans Zhang <hans@linux.alibaba.com> Signed-off-by: He Rongguang <herongguang@linux.alibaba.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.
f2b28698a28a	30 Jan 2023	Parav Pandit	virtio-net: Maintain network device spec in separate directory Move virtio network device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1, 7.3.4, and 7.2.4.
81694cddc4c1	30 Jan 2023	Parav Pandit	virtio-net: Fix spelling errors Fix two spelling errors in the virtio network device specification. Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.5, and 5.1.9.5.
335342f5cd88	30 Jan 2023	Parav Pandit	virtio-blk: Maintain block device spec in separate directory Move virtio block device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.2, 7.3.5, and 7.2.5.

Revision	Date	Editor	Changes Made
d3d06187eabb	30 Jan 2023	Parav Pandit	virtio-console: Maintain console device spec in separate directory Move virtio console device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.3, 7.3.6, and 7.2.6.
c71e88e86d35	30 Jan 2023	Parav Pandit	virtio-entropy: Maintain entropy device spec in separate directory Move virtio entropy device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.4, 7.3.7, and 7.2.7.
c06f3b670dd6	30 Jan 2023	Parav Pandit	virtio-balloon: Maintain mem balloon device spec in separate directory Move virtio memory balloon device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.5, 7.3.8, and 7.2.8.
d404f1c4e886	30 Jan 2023	Parav Pandit	virtio-scsi: Maintain scsi host device spec in separate directory Move virtio SCSI host device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.6, 7.3.9, and 7.2.9.

Revision	Date	Editor	Changes Made
442bb643a9ad	30 Jan 2023	Parav Pandit	virtio-gpu: Maintain gpu device spec in separate directory Move virtio gpu device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.10.
c9686f241819	30 Jan 2023	Parav Pandit	virtio-input: Maintain input device spec in separate directory Move virtio input device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.11, and 7.2.10.
8463bba27c79	30 Jan 2023	Parav Pandit	virtio-crypto: Maintain crypto device spec in separate directory Move virtio crypto device specification to its own file similar to recent virtio devices. While at it, place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.12, and 7.2.11.
828754b98e3b	30 Jan 2023	Parav Pandit	virtio-vsock: Maintain socket device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefano Garzarella <sgarzare@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.13, and 7.2.12.
8632f80e251f	30 Jan 2023	Parav Pandit	virtio-fs: Maintain file system device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.14, and 7.2.13.

Revision	Date	Editor	Changes Made
b067de47a506	30 Jan 2023	Parav Pandit	virtio-rpmb: Maintain rpmb device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.15, and 7.2.14.
b1cf73e96173	30 Jan 2023	Parav Pandit	virtio-iommu: Maintain iommu device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.16, and 7.2.15.
6813e3cc271e	30 Jan 2023	Parav Pandit	virtio-sound: Maintain sound device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.17, and 7.2.16.
5042a5031502	30 Jan 2023	Parav Pandit	virtio-mem: Maintain memory device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.18, and 7.2.17.
00b9935238bf	30 Jan 2023	Parav Pandit	virtio-i2c: Maintain i2c device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.19, and 7.2.18.

Revision	Date	Editor	Changes Made
674489b191ab	30 Jan 2023	Parav Pandit	virtio-scmi: Maintain scmi device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.20, and 7.2.19.
6c9c04d2bf5e	30 Jan 2023	Parav Pandit	virtio-gpio: Maintain gpio device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.21, and 7.2.20.
d04d253b1055	30 Jan 2023	Parav Pandit	virtio-pmem: Maintain pmem device spec in separate directory Place device specification, its driver and device conformance into its own directory to have self contained device specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/153 Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 7.3.22, and 7.2.21.
b1fb6b62495f	02 Feb 2023	Parav Pandit	virtio-net: Clarify VLAN filter table configuration The filtering behavior of the VLAN filter commands is not very clear as discussed in thread [1]. Hence, add the command description and device requirements for it. [1] https://lists.oasis-open.org/archives/virtio-dev/202301/msg00210.html Fixes: https://github.com/oasis-tcs/virtio-spec/issues/147 Suggested-by: Si-Wei Liu <si-wei.liu@oracle.com> Reviewed-by: Si-Wei Liu <si-wei.liu@oracle.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.9.5.3, 5.1.9.5, and 7.3.4.

Revision	Date	Editor	Changes Made
53b0cb13169c	02 Feb 2023	Parav Pandit	virtio-net: Avoid confusing device configuration text The added text in commit of Fixes tag was redundant and confusing in context of VLAN filtering description. Hence remove it as discussed in [1] and [2]. [1] https://lists.oasis-open.org/archives/virtio-dev/202301/msg00282.html [2] https://lists.oasis-open.org/archives/virtio-dev/202301/msg00286.html Fixes: 296303444f6b (”virtio-net: Clarify VLAN filter table configuration”) Suggested-by: Halil Pasic <pasic@linux.ibm.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> [CH: applied as editorial change] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.9.5.
3b9b6acb0936	09 Feb 2023	Michael S. Tsirkin	audio->sound Spec calls the device ”sound device”. Make the name in the ID section match. MST: applied as editorial change. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Cornelia Huck <cohuck@redhat.com> See 5.
0ce03bc6995a	14 Feb 2023	Parav Pandit	virtio-net: Avoid confusion between a card and a device Historically virtio network device is documented as an Ethernet card. A modern card in the industry has one to multiple ports, one to multiple PCI functions. However the virtio network device is usually just a single link/port network interface controller. Hence, avoid this confusing term ’card’ and align the specification to adhere to widely used specification term as ’device’ used for all virtio device types. Replaced ’card’ with ’network interface controller’. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/154 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.1.2, 5, 5.1, 5.1.3, and 5.1.5.
be2ce1ee17e0	15 Feb 2023	Parav Pandit	content.tex Fix Driver notifications label Driver notifications section is under ”Basic Facilities of a Virtio Device”. However, the label is placed under ”Virtqueues” section. Fix the label references. Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> [CH: pushed as an editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 2.9.

Revision	Date	Editor	Changes Made
2ea4627093fb	20 Feb 2023	Alvaro Karsz	virtio-net: Mention VIRTIO_NET_F_HASH_REPORT dependency on VIRTIO_NET_F_CTRL_VQ If the VIRTIO_NET_F_HASH_REPORT feature is negotiated, the driver may send VIRTIO_NET_CTRL_MQ_HASH_CONFIG commands, thus, the control VQ feature should be negotiated. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/158 Signed-off-by: Alvaro Karsz <alvaro.karsz@solid-run.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.4.
73ce5bb02003	01 Mar 2023	Alvaro Karsz	virtio-net: Fix and update VIRTIO_NET_F_NOTF_COAL feature This patch makes several improvements to the notification coalescing feature, including: • Consolidating virtio_net_ctrl_coal_tx and virtio_net_ctrl_coal_rx into a single struct, virtio_net_ctrl_coal, as they are identical. • Emphasizing that the coalescing commands are best-effort. • Defining the behavior of coalescing with regards to delivering notifications when a change occur. • Stating that the commands should apply to all the receive/transmit virtqueues. • Stating that every receive/transmit virtqueue should count it's own packets. • A new intro explaining the entire coalescing operation. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/159 Signed-off-by: Alvaro Karsz <alvaro.karsz@solid-run.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.9.5.
3508347769af	01 Mar 2023	Parav Pandit	virtio-net: Improve introductory description The control VQ of the virtio network device is used beyond advance steering control. The control VQ dynamically changes multiple features of the initialized device. Hence, update this area of control VQ introductory description at few places and also place the link to its description. Also update the introduction section to better describe receive and transmit virtqueues. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/156 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1, 5.1.2, and 5.1.9.

Revision	Date	Editor	Changes Made
91a469991433	10 Mar 2023	Parav Pandit	transport-pci: Split PCI transport to its own file Place PCI transport specification in its own file to better maintain it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.1.
9e88ba9c47d0	10 Mar 2023	Parav Pandit	transport-mmio: Split MMIO transport to its own file Place MMIO transport specification in its own file to better maintain it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.2.
0af264f9d4ea	10 Mar 2023	Parav Pandit	transport-ccw: Split Channel IO transport to its own file Place Channel IO transport specification in its own file to better maintain it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.3.
deb0aa0c7faa	10 Mar 2023	Parav Pandit	transport-pci: Fix spellings and white spaces Now that we have individual files, fix reported spelling errors. While at it, remove trailing white spaces. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.1, 4.1.4.5, and 4.1.5.1.
ca97719ea35e	10 Mar 2023	Parav Pandit	transport-mmio: Fix spellings and white spaces Now that we have individual files, fix reported spelling errors. While at it, remove trailing white spaces. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.2.2, and 4.2.4.
8797f4d4e410	10 Mar 2023	Parav Pandit	transport-ccw: Fix spellings and white spaces Now that we have individual files, fix reported spelling errors. While at it, remove extra white spaces. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/157 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.3.1, 4.3.1.2, 4.3.2.6, and 4.3.3.2.

Revision	Date	Editor	Changes Made
d3f832b6605d	15 Mar 2023	Parav Pandit	virtio-net: Describe dev cfg fields read only Device configuration fields are read only. Avoid duplicating this description for multiple fields. Instead describe it one time and do it in the driver requirements section. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/161 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.4.
115ceb97f813	15 Mar 2023	Parav Pandit	virtio-net: Define cfg fields before description Currently some fields of the virtio_net_config structure are defined before introducing the structure and some are defined after. Better to define the configuration layout first followed by description of all the fields. Device configuration fields are described in the section. Change wording from 'listed' to 'described' as suggested in patch [1]. [1] https://lists.oasis-open.org/archives/virtio-dev/202302/msg00004.html Fixes: https://github.com/oasis-tcs/virtio-spec/issues/161 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.4.
2d1d8dfa3474	15 Mar 2023	Parav Pandit	virtio-net: Fix virtqueues spelling error Correct spelling from virtqueus to virtqueues. Signed-off-by: Parav Pandit <parav@nvidia.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Jiri Pirko <jiri@nvidia.com> [CH: pushed as editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.9.5.
2d5495083c12	15 Mar 2023	Parav Pandit	transport-pci: Remove duplicate word structure Remove duplicate word structure. Signed-off-by: Parav Pandit <parav@nvidia.com> Acked-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Reviewed-by: Jiri Pirko <jiri@nvidia.com> [CH: pushed as editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 4.1.4.10.

Revision	Date	Editor	Changes Made
b0414098602f	15 Mar 2023	Parav Pandit	virtio-blk: Define dev cfg layout before its fields Define device configuration layout structure before describing its individual fields. This is an editorial change. Suggested-by: Cornelia Huck <cohuck@redhat.com> Reviewed-by: Max Gurtovoy <mgurtovoy@nvidia.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> See 5.1.4.
380ed02bdb88	04 Apr 2023	Parav Pandit	transport-pci: Remove empty line at end of file Remove empty line at end of file. Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: David Edmondson <david.edmondson@oracle.com> See 4.1.
1ed0754c6134	11 Apr 2023	Heng Qi	virtio-net: support the virtqueue coalescing moderation Currently, coalescing parameters are grouped for all transmit and receive virtqueues. This patch supports setting or getting the parameters for a specified virtqueue, and a typical application of this function is netdim[1]. When the traffic between virtqueues is unbalanced, for example, one virtqueue is busy and another virtqueue is idle, then it will be very useful to control coalescing parameters at the virtqueue granularity. [1] https://docs.kernel.org/networking/net_dim.html Fixes: https://github.com/oasis-tcs/virtio-spec/issues/166 Signed-off-by: Heng Qi <hengqi@linux.alibaba.com> Reviewed-by: Xuan Zhuo <xuanzhuo@linux.alibaba.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.3, 5.1.3.1, and 5.1.9.5.10.
362ebd007271	11 Apr 2023	Alvaro Karsz	virtio-net: define the VIRTIO_NET_F_CTRL_RX_EXTRA feature bit The VIRTIO_NET_F_CTRL_RX_EXTRA feature bit is mentioned in the spec since version 1.0, but it's not properly defined. This patch defines the feature bit and defines the dependency on VIRTIO_NET_F_CTRL_VQ. Since this dependency is missing in previous versions, we add it now as a "SHOULD". Fixes: https://github.com/oasis-tcs/virtio-spec/issues/162 Reviewed-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Alvaro Karsz <alvaro.karsz@solid-run.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.3, and 5.1.4.

Revision	Date	Editor	Changes Made
d3b2a19bc369	21 Apr 2023	Parav Pandit	device-types/multiple: replace queues with enqueues Queue is a verb and noun both. Replacing it with enqueue avoids ambiguity around plural queues noun vs verb; similar to virtio fs device description. Acked-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> [CH: pushed as editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.2.6, 5.18.6.7, 5.18.7, 5.16.6.1, 5.6.6.1, and 5.10.6.4.
aadefe688680	19 May 2023	Michael S. Tsirkin	virtio: document forward compatibility guarantees Feature negotiation forms the basis of forward compatibility guarantees of virtio but has never been properly documented. Do it now. Suggested-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 2.2.
f3ce853c8a91	19 May 2023	Michael S. Tsirkin	admin: introduce device group and related concepts Each device group has a type. For now, define one initial group type: SR-IOV type - PCI SR-IOV virtual functions (VFs) of a given PCI SR-IOV physical function (PF). This group may contain zero or more virtio devices according to NumVFs configured. Each device within a group has a unique identifier. This identifier is the group member identifier. Note: one can argue both ways whether the new device group handling functionality (this and following patches) is closer to a new device type or a new transport type. However, it's expected that we will add more features in the near future. To facilitate this as much as possible of the text is located in the new admin chapter. Effort was made to minimize transport-specific text. There's a bit of duplication with 0x1 repeated twice and no special section for group type identifiers. It seems ok to defer adding these until we have more group types. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> See 2.12.

Revision	Date	Editor	Changes Made
2cbaaa19b15a	19 May 2023	Michael S. Tsirkin	admin: introduce group administration commands This introduces a general structure for group administration commands, used to control device groups through their owner. Following patches will introduce specific commands and an interface for submitting these commands to the owner. Note that the commands are focused on controlling device groups: this is why group related fields are in the generic part of the structure. Without this the admin vq would become a ”whatever” vq which does not do anything specific at all, just a general transport like thing. I feel going this way opens the design space to the point where we no longer know what belongs in e.g. config space what in the control q and what in the admin q. As it is, whatever deals with groups is in the admin q; other things not in the admin q. There are specific exceptions such as query but that’s an exception that proves the rule ;) Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 2.12.1, and 1.1.
5f1a8ac61c15	19 May 2023	Michael S. Tsirkin	admin: introduce virtio admin virtqueues The admin virtqueues will be the first interface used to issue admin commands. Currently the virtio specification defines control virtqueue to manipulate features and configuration of the device it operates on: virtio-net, virtio-scsi, etc all have existing control virtqueues. However, control virtqueue commands are device type specific, which makes it very difficult to extend for device agnostic commands. Keeping the device-specific virtqueue separate from the admin virtqueue is simpler and has fewer potential problems. I don’t think creating common infrastructure for device-specific control virtqueues across device types worthwhile or within the scope of this patch series. To support this requirement in a more generic way, this patch introduces a new admin virtqueue interface. The admin virtqueue can be seen as the virtqueue analog to a transport. The admin queue thus does nothing device type-specific (net, scsi, etc) and instead focuses on transporting the admin commands. We also support more than one admin virtqueue, for QoS and scalability requirements. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> See 2.13, 2.2, and 6.

Revision	Date	Editor	Changes Made
677aeaebf6a7	19 May 2023	Michael S. Tsirkin	pci: add admin vq registers to virtio over pci Add new registers to the PCI common configuration structure. These registers will be used for querying the indices of the admin virtqueues of the owner device. To configure, reset or enable the admin virtqueues, the driver should follow existing queue configuration/setup sequence. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 6, and 4.1.4.3.
a9a59f70be46	19 May 2023	Michael S. Tsirkin	mmio: document ADMIN_VQ as reserved Adding relevant registers needs more work and it's not clear what the use-case will be as currently only the PCI transport is supported. But let's keep the door open on this. We already say it's reserved in a central place, but it does not hurt to remind implementers to mask it. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> See 4.2.5.
325046c1460e	19 May 2023	Michael S. Tsirkin	ccw: document ADMIN_VQ as reserved Adding relevant registers needs more work and it's not clear what the use-case will be as currently only the PCI transport is supported. But let's keep the door open on this. We already say it's reserved in a central place, but it does not hurt to remind implementers to mask it. Note: there are more features to add to this list. Will be done later with a patch on top. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Reviewed-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 4.3.4.
3dc7196cba2d	19 May 2023	Michael S. Tsirkin	admin: command list discovery Add commands to find out which commands does each group support, as well as enable their use by driver. This will be especially useful once we have multiple group types. An alternative is per-type VQs. This is possible but will require more per-transport work. Discovery through the vq helps keep things contained. e.g. lack of support for some command can switch to a legacy mode note that commands are expected to be avolved by adding new fields to command specific data at the tail, so we generally do not need feature bits for compatibility. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 2.12.1.

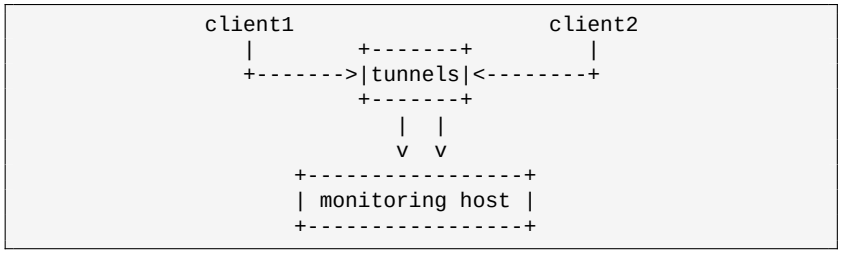
Revision	Date	Editor	Changes Made
bf1d6b0d24ae	19 May 2023	Michael S. Tsirkin	admin: conformance clauses Add conformance clauses for admin commands and admin virtqueues. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/171 Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> See 2.12.1, 2.13, and 4.1.4.3.
b04be31f0bf0	19 May 2023	Michael S. Tsirkin	ccw: document more reserved features vq reset and shared memory are unsupported, too. Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Fixes: https://github.com/oasis-tcs/virtio-spec/issues/160 Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Reviewed-by: Zhu Lingshan <lingshan.zhu@intel.com> See 4.3.4.
619f60ae4ccf	19 May 2023	Parav Pandit	admin: Fix reference and table formation This patch brings three fixes. 1. Opcode table has 3 columns, only two were enumerated. Due to this pdf generation script stops. Fix it and also have resizeable description column as it needs wrap. 2. Status description column content needs to wrap. Without it pdf does not read good. Fix it by having resizeable description column. 3. Fix the broken link to the Device groups. Fixes: 2cbaaa1 (”admin: introduce group administration commands”) Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: Cornelia Huck <cohuck@redhat.com> See 2.12.1.
c1cd68b97611	19 May 2023	Parav Pandit	transport-pci: Improve config msix vector description config_msix_vector is the register that holds the MSI-X vector number for receiving configuration change related interrupts. It is not ”for MSI-X”. Hence, replace the confusing text with appropriate one. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/169 Reviewed-by: Max Gurtovoy <mgurtovoy@nvidia.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.1.4.3.

Revision	Date	Editor	Changes Made
0f433d62e81d	19 May 2023	Parav Pandit	transport-pci: Improve queue msix vector register desc queue_msix_vector register is for receiving virtqueue notification interrupts from the device for the virtqueue. ”for MSI-X” is confusing term. Also it is the register that driver ”writes” to, similar to many other registers such as queue_desc, queue_driver etc. Hence, replace the verb from use to write. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/169 Signed-off-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Max Gurtovoy <mgurtovoy@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.1.4.3.
b0fbccd4062f	19 May 2023	Parav Pandit	content: Add vq index text Introduce vq index and its range so that subsequent patches can refer to it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 2.6.
362f1cac2516	19 May 2023	Parav Pandit	content.tex Replace virtqueue number with index Replace virtqueue number with index to align to rest of the specification. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 2.9.

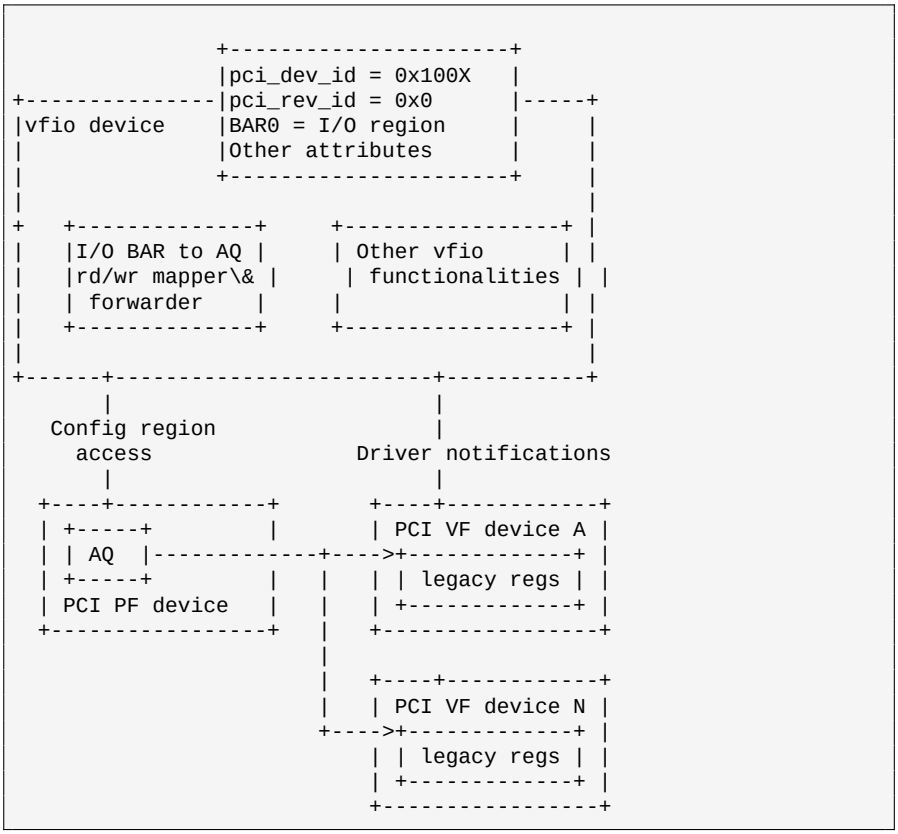
Revision	Date	Editor	Changes Made
cc4a5604b259	19 May 2023	Parav Pandit	content: Rename confusing queue_notify_data and vqn names Currently queue_notify_data register indicates the device internal queue notification content. This register is meaningful only when feature bit VIRTIO_F_NOTIF_CONFIG_DATA is negotiated. However, above register name often get confusing association with very similar feature bit VIRTIO_F_NOTIFICATION_DATA. When VIRTIO_F_NOTIFICATION_DATA feature bit is negotiated, notification really involves sending additional queue progress related information (not queue identifier or index). Hence 1. to avoid any misunderstanding and association of queue_notify_data with similar name VIRTIO_F_NOTIFICATION_DATA, and 2. to reflect that queue_notify_data is the actual device internal virtqueue identifier/index/data/cookie, a. rename queue_notify_data to queue_notif_config_data. b. rename ambiguous vqn to a union of vq_index and vq_config_data c. The driver notification section assumes that queue notification contains vq index always. CONFIG_DATA feature bit introduction missed to update the driver notification section. Hence, correct it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Acked-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Reviewed-by: David Edmondson <david.edmondson@oracle.com> See 2.9, 4.1.4.3, and 4.1.5.2.
fbb119dad56d	19 May 2023	Parav Pandit	transport-pci: Avoid first vq index reference Drop reference to first virtqueue as it is already covered now by the generic section in first patch. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Acked-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.1.5.1.
a7a21e451987	19 May 2023	Parav Pandit	transport-mmio: Rename QueueNum register These are further named differently between pci and mmio transport. PCI transport indicates queue size as queue_size. To bring consistency between pci and mmio transport, rename the QueueNumMax and QueueNum registers to QueueSizeMax and QueueSize respectively. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: Cornelia Huck <cohuck@redhat.com> Reviewed-by: Jiri Pirko <jiri@nvidia.com> Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.2.2, and 4.2.4.

Revision	Date	Editor	Changes Made
9ddc59553984	19 May 2023	Parav Pandit	transport-mmio: Avoid referring to zero based index VQ range is already described in the first patch in basic virtqueue section. Hence remove the duplicate reference to it. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Acked-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.2.2, and 4.2.4.
e7a764f66598	19 May 2023	Parav Pandit	transport-ccw: Rename queue depth/size to other transports max_num field reflects the maximum queue size/depth. Hence align name of this field with similar field in PCI and MMIO transport to max_queue_size. Similarly rename 'num' to 'size'. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.3.2.2.
c3092410ac51	19 May 2023	Parav Pandit	transport-ccw: Refer to the vq by its index Currently specification uses virtqueue index and number interchangeably to refer to the virtqueue. Instead refer to it by its index. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: Halil Pasic <pasic@linux.ibm.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.3.3.2.
d6f310dbb3bf	19 May 2023	Parav Pandit	virtio-net: Avoid duplicate receive queue example Receive queue number/index example is duplicate which is already defined in the Setting RSS parameters section. Hence, avoid such duplicate example and prepare it for the subsequent patch to describe using receive queue handle. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: Cornelia Huck <cohuck@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 5.1.9.5.

Revision	Date	Editor	Changes Made
da0e16928d0b	19 May 2023	Parav Pandit	virtio-net: Describe RSS using rss rq id The content of the indirection table and unclassified_queue were originally described based on mathematical operations. In order to make it easier to understand and to avoid intermixing the array index with the vq index, introduce a structure rss_rq_id (RSS receive queue ID) and use it to describe the unclassified_queue and indirection_table fields. As part of it, have the example that uses non-zero virtqueue index which helps to have better mapping between receiveX object with virtqueue index and the actual value in the indirection table. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 5.1.9.5.
f9ff777fba59	19 May 2023	Parav Pandit	virtio-net: Update vqn to vq_index for cvq cmds Replace field name vqn to vq_index for recent virtqueue level commands. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/163 Reviewed-by: David Edmondson <david.edmondson@oracle.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 5.1.9.5.
74460ef69d5f	19 May 2023	Parav Pandit	transport-mmio: Replace virtual queue with virtqueue Basic facilities define the virtqueue construct for device <-> driver communication. PCI transport and individual devices description also refers to it as virtqueue. MMIO refers to it as 'virtual queue'. Align MMIO transport description to call such object a virtqueue. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/168 Reviewed-by: Stefan Hajnoczi <stefanha@redhat.com> Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> See 4.2.2, 4.2.3.2, and 4.2.4.
6724756eaf0a	07 Jul 2023	Parav Pandit	admin: Split opcode table rows with a line Currently all opcode appears to be in a single row. Separate them with a line similar to other tables. Signed-off-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Cornelia Huck <cohuck@redhat.com> [CH: pushed as editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 2.12.1.

Revision	Date	Editor	Changes Made
1518c9ce2cde	07 Jul 2023	Parav Pandit	admin: Fix section numbering Requirements are put one additional level down. Fix it. Signed-off-by: Parav Pandit <parav@nvidia.com> Reviewed-by: Cornelia Huck <cohuck@redhat.com> [CH: pushed as editorial update] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 2.12.1.
9c3ba1ec6acb	14 Jul 2023	Heng Qi	virtio-net: support inner header hash 1. Currently, a received encapsulated packet has an outer and an inner header, but the virtio device is unable to calculate the hash for the inner header. The same flow can traverse through different tunnels, resulting in the encapsulated packets being spread across multiple receive queues (refer to the figure below). However, in certain scenarios, we may need to direct these encapsulated packets of the same flow to a single receive queue. This facilitates the processing of the flow by the same CPU to improve performance (warm caches, less locking, etc.).  To achieve this, the device can calculate a symmetric hash based on the inner headers of the same flow. 2. For legacy systems, they may lack entropy fields which modern protocols have in the outer header, resulting in multiple flows with the same outer header but different inner headers being directed to the same receive queue. This results in poor receive performance. To address this limitation, inner header hash can be used to enable the device to advertise the capability to calculate the hash for the inner packet, regaining better receive performance. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/173 Signed-off-by: Heng Qi <hengqi@linux.alibaba.com> Reviewed-by: Xuan Zhuo <xuanzhuo@linux.alibaba.com> Reviewed-by: Parav Pandit <parav@nvidia.com> [CH: added missing lstlisting and hyperref escapes, fixed references] Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.1.3, 5.1.3.1, 5.1.4, 5.1.9.4, 5.1.9.4.4, 7.3.4, 7.2.4, and 1.1.
73c2fd96af96	17 Jul 2023	Haixu Cui	virtio-spi: define the DEVICE ID for virtio SPI master Define the DEVICE ID of virtio SPI master device as 45. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/174 Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 5.

Revision	Date	Editor	Changes Made
03c2d32e5093	21 Jul 2023	Parav Pandit	admin: Add group member legacy register access commands Introduce group member legacy common configuration and legacy device configuration access read/write commands. Group member legacy registers access commands enable group owner driver software to access legacy registers on behalf of the guest virtual machine. Usecase: ======== 1. A hypervisor/system needs to provide transitional virtio devices to the guest VM at scale of thousands, typically, one to eight devices per VM. 2. A hypervisor/system needs to provide such devices using a vendor agnostic driver in the hypervisor system. 3. A hypervisor system prefers to have single stack regardless of virtio device type (net/blk) and be future compatible with a single vfio stack using SR-IOV or other scalable device virtualization technology to map PCI devices to the guest VM. (as transitional or otherwise) Motivation/Background: ==================== The existing virtio transitional PCI device is missing support for PCI SR-IOV based devices. Currently it does not work beyond PCI PF, or as software emulated device in reality. Currently it has below cited system level limitations: [a] PCIe spec citation: VFs do not support I/O Space and thus VF BARs shall not indicate I/O Space. [b] cpu arch citiation: Intel 64 and IA-32 Architectures Software Developer's Manual: The processor's I/O address space is separate and distinct from the physical-memory address space. The I/O address space consists of 64K individually addressable 8-bit I/O ports, numbered 0 through FFFFH. [c] PCIe spec citation: If a bridge implements an I/O address range,...I/O address range will be aligned to a 4 KB boundary. Overview: ========= Above usecase requirements is solved by PCI PF group owner accessing its group member PCI VFs legacy registers using the administration commands of the group owner PCI PF. Two types of administration commands are added which read/write PCI VF registers. Software usage example: ====================== 1. One way to use and map to the guest VM is by using vfio driver framework in Linux kernel. ...

Revision	Date	Editor	Changes Made
			...  2. Continue to use the virtio pci driver to bind to the listed device id and use it as in the host. 3. Use it in a light weight hypervisor to run bare-metal OS. Fixes: https://github.com/oasis-tcs/virtio-spec/issues/167 Signed-off-by: Parav Pandit <parav@nvidia.com> Signed-off-by: Michael S. Tsirkin <mst@redhat.com> Signed-off-by: Cornelia Huck <cohuck@redhat.com> See 2.12.1.1, 2.12.1, and 7.1.