Virtio-iommu specification v0.13

With page table extension

05 February 2021

Notices

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, http://publibfi.boulder.ibm.com/epubs/pdf/dz9zr009.pdf, and any future revisions
[S390 Common I/O]	ESA/390 Common I/O-Device and Self-Description, IBM Publication SA22-7204, http://publibfp.dhe.ibm.com/cgi-bin/bookmgr/BOOKS/dz9ar501/CCONTENTS, and any future revisions
[PCI]	Conventional PCI Specifications, http://www.pcisig.com/specifications/conventional/, PCI-SIG
[PCIe]	PCI Express Specifications http://www.pcisig.com/specifications/pciexpress/, PCI-SIG

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

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

[IEEE 802]	IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture, http://www.ieee802.org/, IEEE
[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
[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
[SMMUv3]	Arm System Memory Management Unit version 3 https://developer.arm.com/documentation/ihi0070/latest
[ARMv8-A]	Armv8-A Architecture Reference Manual https://developer.arm.com/documentation/ddi0487/latest

1.2 Non-Normative References

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

1.3 Terminology

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

1.3.1 Legacy Interface: Terminology

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)
```

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.

VIRTIO_IOMMU_F_BYPASS (3) When not attached to a domain, endpoints downstream of the IOMMU can access the guest-physical address space.

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_ATTACH_TABLE (6) The ATTACH_TABLE and INVALIDATE requests are available.

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 MUST offer at least one of the VIRTIO_IOMMU_F_MAP_UNMAP or VIRTIO_IOMMU_F_AT-TACH_TABLE features.

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;
};
```

5.13.4.1 Driver Requirements: Device configuration layout

The driver MUST NOT write to device configuration fields.

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*.

5.13.5 Device initialization

When the device is reset, endpoints are not attached to any domain.

If the VIRTIO_IOMMU_F_BYPASS feature is negotiated, all accesses from unattached endpoints are allowed and translated by the IOMMU using the identity function. If the feature is not negotiated, any memory access from an unattached endpoint fails. Upon attaching an endpoint in bypass mode to a new domain, any memory access from the endpoint fails, since the domain does not contain any mapping.

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.

5.13.5.2 Device Requirements: Device Initialization

If the driver does not accept the VIRTIO_IOMMU_F_BYPASS feature, the device SHOULD NOT let endpoints access the guest-physical address space.

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
#define VIRTIO_IOMMU_T_ATTACH_TABLE 6
#define VIRTIO_IOMMU_T_INVALIDATE   7
```

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`)

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.

5.13.6.3 ATTACH request

```
struct virtio_iommu_req_attach {
  struct virtio_iommu_req_head head;
  le32 domain;
  le32 endpoint;
  u8   reserved[8];
  struct virtio_iommu_req_tail tail;
};
```

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.

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.

The driver SHOULD NOT create the *domain* with an ATTACH request if the VIRTIO_IOMMU_F_MAP_-UNMAP feature was not negotiated. In this case the ATTACH_TABLE request is used.

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 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 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 ATTACH_TABLE request

```
struct virtio_iommu_req_attach_table {
  struct virtio_iommu_req_head head;
  le32 domain;
  le32 endpoint;
  le16 format;
  u8   descriptor[62];
  struct virtio_iommu_req_tail tail;
};
```

Attach an endpoint to a domain, in the same way as an ATTACH request. In addition, provide a pointer to a table describing the mappings. Instead of using MAP and UNMAP requests, the driver creates and removes mappings by writing into the tables. When a mapping that may have been cached in a TLB is removed, the driver sends an INVALIDATE request.

The driver discovers with a PROBE request the table formats supported by the device. The device can support multiple page tables and PASID table formats. A PASID table contains pointers to one or more page tables. The driver chooses a format it recognises, and sends the ATTACH_TABLE request.

Apart from *domain*, *endpoint* and *format*, each table format uses different parameters in field *descriptor*. Table formats and their associated parameters are described in section 5.13.7.

5.13.6.4.1 Driver Requirements: ATTACH_TABLE request

The driver SHOULD NOT send ATTACH_TABLE requests if the VIRTIO_IOMMU_F_ATTACH_TABLE feature was not negotiated.

The driver SHOULD NOT send ATTACH_TABLE requests with an existing *domain*. To attach additional endpoints to a domain created with ATTACH_DOMAIN, the driver SHOULD send an ATTACH request. All endpoints attached to a domain MUST support the same table formats.

5.13.6.5 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. If feature VIRTIO_IOMMU_F_BYPASS has been negotiated, then once this request completes all accesses from the endpoint are allowed and translated by the IOMMU using the identity function.

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.

5.13.6.5.1 Driver Requirements: DETACH request

The driver SHOULD set *reserved* to zero.

5.13.6.5.2 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.

5.13.6.6 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.

5.13.6.6.1 Driver Requirements: MAP request

The driver SHOULD set undefined *flags* bits to zero.

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.

domain SHOULD NOT have been created with an ATTACH_TABLE request.

5.13.6.6.2 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.

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.

5.13.6.7 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 5.13.6.7.1 and 5.13.6.7.2, 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.

5.13.6.7.1 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.

domain SHOULD NOT have been created with an ATTACH_TABLE request.

5.13.6.7.2 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 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.

5.13.6.8 INVALIDATE request

```
struct virtio_iommu_req_invalidate {
  struct virtio_iommu_req_head head;
  u8    scope;
  u8    caches;
  le16  flags;
  le32  domain;
  le32  pasid;
  le64  id;
  le64  virt_start;
  le64  nr_pages;
  u8    page_size;
  u8    reserved[19];
  struct virtio_iommu_req_tail tail;
};

#define VIRTIO_IOMMU_INVAL_S_DOMAIN   1
#define VIRTIO_IOMMU_INVAL_S_PASID    2
#define VIRTIO_IOMMU_INVAL_S_ADDRESS  3

#define VIRTIO_IOMMU_INVAL_C_PASID    (1 << 0)
#define VIRTIO_IOMMU_INVAL_C_TLB      (1 << 1)
#define VIRTIO_IOMMU_INVAL_C_EP_TLB   (1 << 2)
```

```
#define VIRTIO_IOMMU_INVAL_F_LEAF     (1 << 0)
#define VIRTIO_IOMMU_INVAL_F_PASID    (1 << 1)
#define VIRTIO_IOMMU_INVAL_F_ID       (1 << 2)
```

Invalidate a mapping or configuration. When using PASID or page tables, the driver sends an INVALIDATE request to signal changes to table elements that could have been cached by the device, such as a mapping removal.

Field *scope* specifies which entries to invalidate:

VIRTIO_IOMMU_INVAL_S_DOMAIN invalidates all cached entries for the given *domain*.

VIRTIO_IOMMU_INVAL_S_PASID invalidates all cached entries for the given *domain* and *pasid*.

VIRTIO_IOMMU_INVAL_S_ADDRESS invalidates all cached entries for the given *domain*, *pasid* and virtual address range.

Field *caches* specifies which caches to invalidate:

VIRTIO_IOMMU_INVAL_C_PASID Invalidate entries cached from the PASID table.

VIRTIO_IOMMU_INVAL_C_TLB Invalidate entries from the Translation Lookaside Buffer (TLB).

VIRTIO_IOMMU_INVAL_C_EP_TLB Invalidate entries from the TLB of endpoints attached to the domain. For example when a PCIe endpoint has an Address Translation Cache (ATC), discovered and enabled through the PCIe ATS capability.

For example, when the driver removes a mapping from a table attached to the *domain*, it sends an INVALIDATE request with *scope* VIRTIO_IOMMU_INVAL_S_ADDRESS, *caches* VIRTIO_IOMMU_INVAL_C_TLB and, if necessary, VIRTIO_IOMMU_INVAL_C_EP_TLB. To invalidate the whole address space the driver sets the *scope* to VIRTIO_IOMMU_INVAL_S_DOMAIN, or to VIRTIO_IOMMU_INVAL_S_PASID when using a PASID table. When removing a page table directory from a PASID table entry, the driver sends an INVALIDATE request with *scope* VIRTIO_IOMMU_INVAL_S_PASID, and all *caches* flags set.

Field *flags* specifies additional information:

VIRTIO_IOMMU_INVAL_F_LEAF Only invalidate TLB entries cached from leaf table entries.

VIRTIO_IOMMU_INVAL_F_PASID The *pasid* field is valid.

VIRTIO_IOMMU_INVAL_F_ID The *id* field is valid.

Use of field *id* is specific to the table format. Some formats use only the PASID to identify address spaces within a domain, others use a separate ID for TLB entries.

For a *scope* of VIRTIO_IOMMU_INVAL_S_ADDRESS, the invalidation affects the range of virtual addresses of size $nr\_pages \times 2^{page\_size}$, starting at *virt_start*. Specifying the range size this way allows the device to efficiently remove TLB entries. For example a page table format could allow a 2MiB range to be mapped with either 512 4KiB pages, or a single 2MiB block. In the latter case a single TLB entry is used, so the driver specifies a *page_size* of 21 and the device does not need to iterate over all 4KiB multiples to invalidate the range.

The following table describes the restricted set of valid *caches*, *flags* and fields for each *scope*:

scope	DOMAIN	PASID	ADDRESS
caches	PASID, TLB	PASID, TLB, EP_TLB	TLB, EP_TLB
flags	ID	LEAF, PASID, ID	LEAF, PASID, ID
domain	Y	Y	Y
pasid	N	Y	Y
id	Y	Y	Y
virt_start	N	N	Y
nr_pages	N	N	Y
page_size	N	N	Y

5.13.6.8.1 Device Requirements: INVALIDATE request

After handling an INVALIDATE request, the device SHOULD NOT let endpoints attached to *domain* access virtual address in the invalidated range, unless the range is valid in the table attached to the *domain*. In other words, the device SHOULD access the virtual address through the attached tables instead of a cache.

For a *scope* of VIRTIO_IOMMU_INVAL_S_DOMAIN, the device SHOULD ignore fields *pasid*, *virt_start*, *nr_pages* and *page_size*.

For a *scope* of VIRTIO_IOMMU_INVAL_S_PASID, the device SHOULD ignore fields *virt_start*, *nr_pages*, *page_size*,

If no cache entry exist for the given parameters, the device SHOULD set the *status* field to VIRTIO_IOMMU_S_OK. In other words, spurious invalidations are allowed.

The device MAY ignore flag VIRTIO_IOMMU_INVAL_F_LEAF.

5.13.6.8.2 Driver Requirements: INVALIDATE request

domain MUST have been created with an ATTACH_TABLE request.

For a *scope* of VIRTIO_IOMMU_INVAL_S_DOMAIN, the driver SHOULD set fields *pasid*, *virt_start*, *nr_pages* and *page_size* to zero.

For a *scope* of VIRTIO_IOMMU_INVAL_S_PASID, the driver SHOULD set fields *virt_start*, *nr_pages*, *page_size* to zero.

5.13.6.9 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 $(length + 4)$ 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 5.13.6.10.

5.13.6.9.1 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.

5.13.6.9.2 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.

5.13.6.10 PROBE properties

```
#define VIRTIO_IOMMU_PROBE_T_RESV_MEM       1
#define VIRTIO_IOMMU_PROBE_T_PAGE_SIZE_MASK 2
#define VIRTIO_IOMMU_PROBE_T_INPUT_RANGE    3
#define VIRTIO_IOMMU_PROBE_T_OUTPUT_SIZE    4
#define VIRTIO_IOMMU_PROBE_T_PASID_SIZE     5
#define VIRTIO_IOMMU_PROBE_T_TABLE_FORMAT   6
```

5.13.6.10.1 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.

5.13.6.10.1.1 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.

5.13.6.10.1.2 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.

5.13.6.10.2 Property PAGE_SIZE_MASK

```
struct virtio_iommu_probe_page_size_mask {
  struct virtio_iommu_probe_property head;
  u8    reserved[4];
  le64  page_size_mask;
};
```

The PAGE_SIZE_MASK property overrides the global *page_size_mask* configuration for an endpoint.

The *page_size_mask* field behaves in the same way as the global *page_size_mask* field, described in 5.13.6. The least significant bit describes the mapping granularity, while additional bits are hints.

5.13.6.10.2.1 Driver Requirements: Property PAGE_SIZE_MASK

The driver MUST ignore *reserved*.

5.13.6.10.2.2 Device Requirements: Property PAGE_SIZE_MASK

The device SHOULD set *reserved* to zero.

The device MAY present multiple PAGE_SIZE_MASK property per endpoint.

5.13.6.10.3 Property INPUT_RANGE

```
struct virtio_iommu_probe_input_range {
  struct virtio_iommu_probe_property head;
  u8    reserved[4];
  le64  start;
  le64  end;
};
```

The INPUT_RANGE property overrides the global *input_range* configuration for an endpoint.

Fields *start* and *end* behave in the same way as the global *input_range.start* and *input_range.end* fields, described in 5.13.6.

5.13.6.10.3.1 Driver Requirements: Property INPUT_RANGE

The driver MUST ignore *reserved*.

5.13.6.10.3.2 Device Requirements: Property INPUT_RANGE

The device SHOULD set *reserved* to zero.

The device SHOULD NOT present more than one INPUT_RANGE property per endpoint.

The device MAY present an INPUT_RANGE property even if it does not offer the VIRTIO_IOMMU_F_-INPUT_RANGE feature.

5.13.6.10.4 Property OUTPUT_SIZE

```
struct virtio_iommu_probe_output_size {
  struct virtio_iommu_probe_property head;
  u8  bits_count;
  u8  reserved[3];
};
```

The OUTPUT_SIZE property describes the maximum guest-physical address that the device supports for this endpoint. *bits_count* is the number of bits that an address can use.

5.13.6.10.4.1 Driver Requirements: Property OUTPUT_SIZE

The driver MUST ignore *reserved*.

The driver SHOULD NOT use guest-physical addresses larger than the size defined by *bits_count* in a MAP request or in a page table.

5.13.6.10.4.2 Device Requirements: Property OUTPUT_SIZE

The device SHOULD set *reserved* to zero.

The device SHOULD NOT present more than one OUTPUT_SIZE property per endpoint.

5.13.6.10.5 Property PASID_SIZE

```
struct virtio_iommu_probe_pasid_size {
  struct virtio_iommu_probe_property head;
  u8  bits_count;
  u8  reserved[3];
};
```

The PASID_SIZE property describes the maximum PASID that the device supports for this endpoint. *bits_-count* is the number of PASID bits that the device can receive in a DMA transaction from this endpoint. This property is conflated with the endpoint PASID size, obtained through a platform specific mechanism such as a PCIe PASID capability.

5.13.6.10.5.1 Driver Requirements: Property PASID_SIZE

The driver MUST ignore *reserved*.

The driver SHOULD NOT use PASIDs larger than the size defined by *bits_count* when updating a PASID table, or in an INVALIDATE request.

5.13.6.10.5.2 Device Requirements: Property PASID_SIZE

The device SHOULD set *reserved* to zero.

The device SHOULD NOT present more than one PASID_SIZE property per endpoint.

5.13.6.10.6 Property TABLE_FORMAT

```
struct virtio_iommu_probe_table_format {
  struct virtio_iommu_probe_property head;
  le16  format;

  /* Format-specific fields follow */
};
```

The TABLE_FORMAT property describes a page table or PASID table format that the device supports for this endpoint. The format is used in ATTACH_TABLE requests. Formats are described in section 5.13.7.

5.13.6.10.6.1 Device Requirements: Property TABLE_FORMAT

The device MAY present more than one TABLE_FORMAT property. If the device presents a PASID table format, it SHOULD also present a compatible page table format.

5.13.6.11 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.

5.13.6.11.1 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.

5.13.6.11.2 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[3], 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.

5.13.7 Table formats

Supported table formats in PROBE properties and ATTACH_TABLE requests are:

VIRTIO_IOMMU_PST_ARM_SMMU3 (1) Arm SMMUv3 Context Descriptor Tables.

VIRTIO_IOMMU_PGT_ARM64 (2) Arm VMSAv8-64 page tables.

5.13.7.1 Arm SMMUv3 Context Descriptor table

Attach Context Descriptor tables (PASID tables) in the format described in the Arm System Memory Management Unit v3 specification. The table contains context descriptors indexed by PASID, each pointing to a page directory.

5.13.7.1.1 PROBE properties for Arm SMMUv3 Context Descriptor tables

```
struct virtio_iommu_probe_pst_arm_smmu3 {
  struct virtio_iommu_probe_property head;
  le16  format;
  u8    reserved[2];
  le64  flags;
};

#define VIRTIO_IOMMU_PST_ARM_SMMU3_F_BTM    (1ULL << 0)
```

[3]This would happen for example if the device implements a more recent version of this specification, whose fault report contains additional fields.

Supported flags are:

VIRTIO_IOMMU_PST_ARM_SMMU3_F_BTM Broadcast TLB maintenance is supported. INVALIDATE requests for *caches* VIRTIO_IOMMU_INVAL_C_TLB can be replaced by broadcast TLBI instructions. INVALIDATE requests for *caches* VIRTIO_IOMMU_INVAL_C_EP_TLB, if the endpoint has PCIe ATS enabled, are still necessary.

5.13.7.1.2 ATTACH_TABLE request for Arm SMMUv3 Context Descriptor tables

```
struct virtio_iommu_req_attach_pst_arm_smmu3 {
  struct virtio_iommu_req_head head;
  le32  domain;
  le32  endpoint;
  le16  format;
  u8    s1fmt;
  u8    s1dss;
  le64  s1contextptr;
  u8    s1cdmax;
  u8    reserved[51];
  struct virtio_iommu_req_tail tail;
};

/* Stage-1 format */
#define VIRTIO_IOMMU_PST_ARM_SMMU3_LINEAR     0x0
#define VIRTIO_IOMMU_PST_ARM_SMMU3_4KL2       0x1
#define VIRTIO_IOMMU_PST_ARM_SMMU3_64KL2      0x2

/* Stage-1 default substream */
#define VIRTIO_IOMMU_PST_ARM_SMMU3_DSS_TERM   0x0
#define VIRTIO_IOMMU_PST_ARM_SMMU3_DSS_BYPASS 0x1
#define VIRTIO_IOMMU_PST_ARM_SMMU3_DSS_0      0x2
```

s1fmt The layout used for the context descriptor table:

LINEAR Single table level,

4KL2 Two-level tables with 4KB leaf tables,

64KL2 Two-level tables with 64KB leaf tables.

s1dss Default substream (PASID) behavior:

DSS_TERM Transactions without a substream are terminated.

DSS_BYPASS Transactions without a substream bypass translation.

DSS_0 Transactions without a substream use entry 0 of the table. Substream 0 is invalid.

s1contextptr Address of the context descriptor table, in guest-physical address space.

s1cdmax Size of the table. $2^{s1cdmax}$ is the number of context descriptors.

5.13.7.1.2.1 Driver Requirements: ATTACH_TABLE request for Arm SMMUv3 Context Descriptor tables

s1contextptr MUST be aligned on 64 bytes.

s1cdmax MUST be less than or equal to *bits_count* in the VIRTIO_IOMMU_PROBE_T_PASID_SIZE property.

5.13.7.2 Arm 64-bit page tables

Attach page tables in the Arm VMSAv8-64 format, as described by the Armv8-A Architecture Reference Manual. The descriptors are little-endian.

5.13.7.2.1 PROBE properties for Arm 64-bit page tables

```
struct virtio_iommu_probe_pgt_arm64 {
  struct virtio_iommu_probe_property head;
  le16  format;
  u8    reserved[2];
  le64  flags;
};

#define VIRTIO_IOMMU_PGT_ARM64_F_ASID16     (1ULL << 1)
#define VIRTIO_IOMMU_PGT_ARM64_F_HW_ACCESS  (1ULL << 2)
#define VIRTIO_IOMMU_PGT_ARM64_F_HW_DIRTY   (1ULL << 3)
```

VIRTIO_IOMMU_PGT_ARM64_F_ASID16 ASIDs can be up to 16 bits. When unset, ASID are only 8 bits.

VIRTIO_IOMMU_PGT_ARM64_F_HW_ACCESS Hardware management of the access bit. The device can write the access bit in page table entries.

VIRTIO_IOMMU_PGT_ARM64_F_HW_DIRTY Hardware management of the dirty bit. The device can set the dirty bit in page table entries.

5.13.7.2.2 ATTACH_TABLE request for Arm 64-bit page tables

Attach a single set of page tables to an endpoint.

```
struct virtio_iommu_req_attach_pgt_arm64 {
  struct virtio_iommu_req_head head;
  le32  domain;
  le32  endpoint;
  le16  format;
  u8    reserved[2]
  le64  tcr;
  le64  ttbr0;
  le64  ttbr1;
  le64  mair;
  u8    reserved1[32];
  struct virtio_iommu_req_tail tail;
};
```

tcr Translation Control Registers, corresponding to TCR_EL1 described in the Armv8-A Architecture Reference Manual.

ttbr0 Translation Table Base Register corresponding to TTBR0_EL1.

ttbr1 Translation Table Base Register corresponding to TTBR1_EL1.

mair Memory Attribute Index Register corresponding to MAIR_EL1.

5.13.7.2.2.1 Driver Requirements: ATTACH_TABLE request for Arm 64-bit page tables

The driver SHOULD set fields *reserved* and *reserved1* to zero.

If the endpoint supports a Context Descriptor table format, the driver SHOULD NOT send an ATTACH_-TABLE request with Arm 64-bit tables. It SHOULD instead attach a Context Descriptor table.

5.13.7.2.2.2 Device Requirements: ATTACH_TABLE request for Arm 64-bit page tables

The device MUST ignore fields *reserved* and *reserved1*.

5.13.7.2.3 INVALIDATE request for Arm 64-bit page tables

Bits [15:0] of *id* in an INVALIDATE request correspond to the address space ID (ASID) of the page directory.

6 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!

6.1 Conformance Targets

Conformance targets:

Driver A driver MUST conform to four conformance clauses:

- One of clauses 7.3.14,

Device A device MUST conform to four conformance clauses:

- One of clauses 7.2.14,

7.3.14 Clause 32: 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
- 5.13.6.5.1
- 5.13.6.6.1
- 5.13.6.7.1
- 5.13.6.9.1
- 5.13.6.10.1.1
- 5.13.6.11.1

7.2.14 Clause 15: IOMMU Device Conformance

An IOMMU device MUST conform to the following normative statements:

- 5.13.3.2
- 5.13.4.2
- 5.13.5.2
- 5.13.6.2
- 5.13.6.3.2
- 5.13.6.5.2
- 5.13.6.6.2

- 5.13.6.7.2
- 5.13.6.9.2
- 5.13.6.10.1.2
- 5.13.6.11.2