**VirtIO介绍**

**1 背景**

Virtio是用于半虚拟化的设备抽象层，和全虚拟化的设备模拟是相对的。但随着演进，VirtIO在多核，虚拟机，模拟器之间的通信架构中获得了广泛使用。整体应该会演化为系统间（包括核间）的事实上的通信标准。

Virtio于2008年提出，当时Linux中针对各个虚拟化系统（KVM，XEN，lguest等）有着各自的一套block, net, console等驱动，virtio希望能在Linux中实现能适用于各类hypervisor的统一驱动程序（前端），hypervisor实现设备的后端部分，这样Linux中的虚拟化驱动变为一套，新的hypervisor只需要实现后端。

为了实现上面的目的，virtio提出了两个机制：

1. 适用于所有driver的feature（特性）扩展机制：方便扩展virtio driver的feature，并支持feature的协商来保证向前/向后的兼容性；
2. Buffer传输机制vring/virtqueue：适用于所有driver，实现简单，零拷贝，无锁；

本文档将分为下面三部分来对VirtIO来进行介绍：

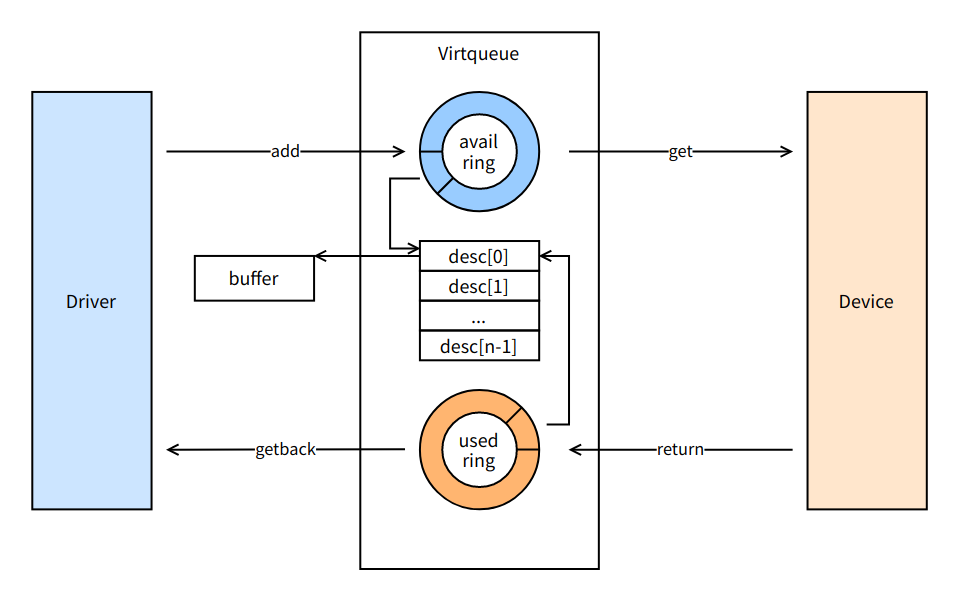
1. 第一部分：主要包含Virtqueue的数据接口，两端的数据发送流程，还有为性能优化的一些高级特性；
2. 第二部分：Virtio Device的一些基础概念说明；
3. 第三部分：Virtio的一种比较常用的传输层（MMIO）的介绍；

**2 Vring/Virtqueue**

一个 virtqueue 是一块由 guest 申请的共享内存区域，guest 和 host 可以在这块内存中读或者写，一端向这块共享内存中填充数据，然后被对端消费实现数据传递。

virtqueue有两种类型：

**Split Virtqueue**：初始的virtio queue的实现方式，每一个vring分为三部分：



1. **descriptor table**（描述符列表）：描述了Driver和Device交互的数据buffer，包含buffer的地址，长度和实现额外功能的标志位等数据；
2. **available ring**（可用环表）和**used ring**（已用环表）：
3. driver发送数据：driver将包含发送数据的descriptor table的索引放置在available ring中供device获取，device将数据收到后，将数据的索引重新放置在used ring中，表示归还给driver；
4. driver接收数据：guest将包含空白内存的descriptor table的索引放置在used ring中，后端驱动获取后，将要发送的数据填充到空白内存中，然后将索引再放置在used ring中，driver来接收；

**Packed Virtqueue**：在virtio v1.1提出，将split virtqueue的desc table, avail ring和used ring合并到一起，对cache和硬件更友好；

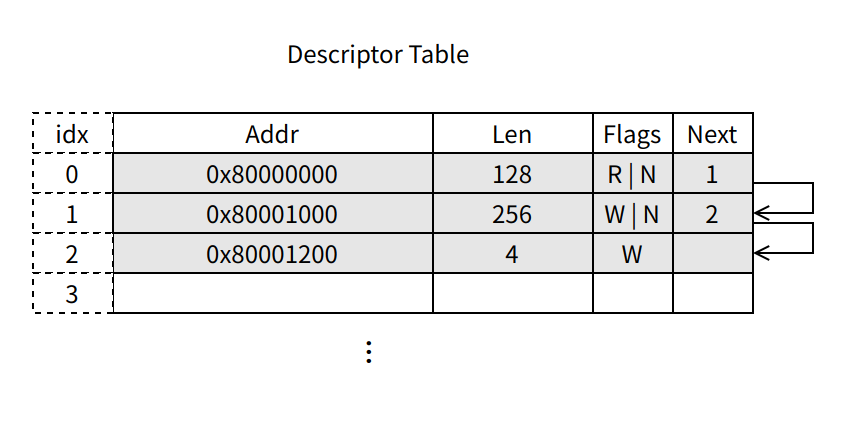
这里仅对Split Virtqueue作介绍，OpenAMP中目前仅实现了这一种。

**2.1 数据结构**

**2.1.1 Descriptor Table**

|  |
| --- |
| C /\* 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  struct virtq\_desc {  /\* Address (guest-physical).\*/  le64 addr;  /\* Length. \*/  le32 len;  /\* The flags as indicated above. \*/  le16 flags;  /\* Next field if (flags & NEXT) is active \*/  le16 next; };  struct indirect\_descriptor\_table {  /\* The actual descriptors (16 bytes each) \*/  struct virtq\_desc desc[len / 16]; }; |

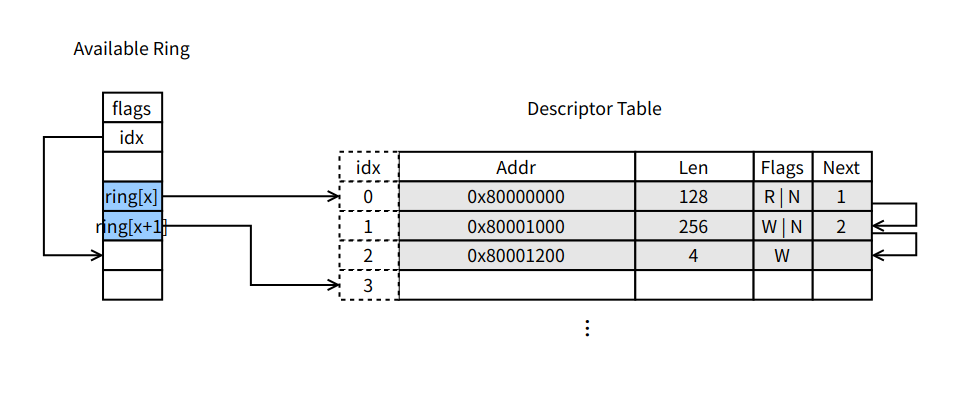
* addr：buffer的物理地址；
* len：buffer的长度；
* flags：VIRTQ\_DESC\_F\_NEXT，若置位，buffer为链表形式，next表示下个buffer在descriptor table的位置；VIRTQ\_DESC\_F\_WRITE ，若置位，该buffer device可写，若未置位，该buffer device只读；VIRTQ\_DESC\_F\_INDIRECT，使用间接的descriptor table来传输buffer（二级表）；
* next：VIRTQ\_DESC\_F\_WRITE，若置位，表示下个buffer在descriptor table的位置；



**2.1.2 Available Ring**

|  |
| --- |
| C #define VIRTQ\_AVAIL\_F\_NO\_INTERRUPT 1  struct virtq\_avail {  le16 flags;  le16 idx;  le16 ring[ /\* Queue Size \*/ ];  le16 used\_event; /\* Only if VIRTIO\_F\_EVENT\_IDX \*/ }; |

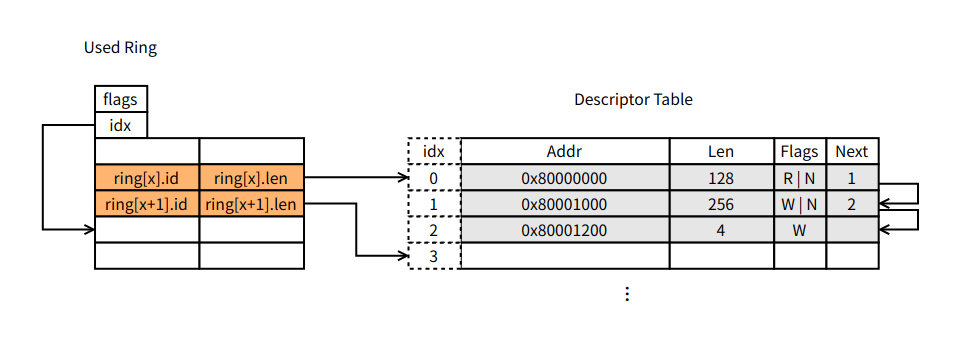
* flags：VIRTQ\_AVAIL\_F\_NO\_INTERRUPT，若置位，则device不能向driver端发送notification（中断）；
* idx：指向virtq\_avail.ring[]的有效边界；
* ring：descriptor table的索引数组，如果有效，通过desc\_table[ring[x]]获取到buffer信息；
* used\_event：使用特性VIRTIO\_F\_EVENT\_IDX时，flags中的VIRTQ\_AVAIL\_F\_NO\_INTERRUPT无效，device发送notification的时机由avail.used\_event来决定：device写used ring，刚好used\_ring.idx==used\_event时，device发送通知，否则不发，用于控制device通知的节奏；



**2.1.3 Used Ring**

|  |
| --- |
| C #define VIRTQ\_USED\_F\_NO\_NOTIFY 1  /\* 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; };  struct virtq\_used {  le16 flags;  le16 idx;  struct virtq\_used\_elem ring[ /\* Queue Size \*/];  le16 avail\_event; /\* Only if VIRTIO\_F\_EVENT\_IDX \*/ }; |

* flags：VIRTQ\_USED\_F\_NO\_NOTIFY，若置位，则driver不能向device发送notification（中断）；
* idx：指向virtq\_used.ring[]的有效边界；
* ring：id - descriptor table的索引，desc\_table[ring[x].id]获取到buffer信息，len - device写到可写buffer（即VIRTQ\_DESC\_F\_WRITE）里面的长度，这里的buffer包含buffer链；
* avail\_event：使用特性VIRTIO\_F\_EVENT\_IDX时，VIRTQ\_USED\_F\_NO\_NOTIFY无效，driver发送notification的时机由avail\_event来决定：driver写avail ring，刚好avail\_ring.idx==avail\_event时，driver发送通知，否则不发，用于控制driver通知的节奏；

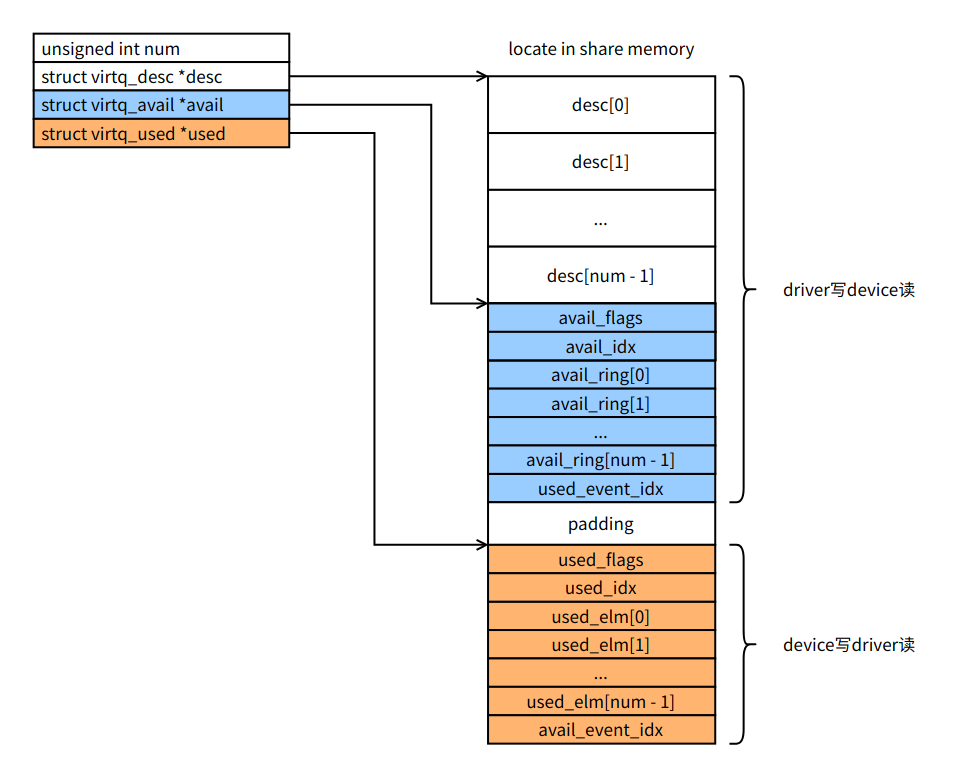


**2.1.4 Vring**

|  |
| --- |
| C struct virtq {  unsigned int num;  struct virtq\_desc \*desc;  struct virtq\_avail \*avail;  struct virtq\_used \*used; }; |

* num：virtqueue的长度，descriptor table，available ring中的ring，used ring中的ring的size和num相同；
* desc：descriptor table数组地址；
* avail：available ring地址；
* used：used ring地址；

最终Layer out如下

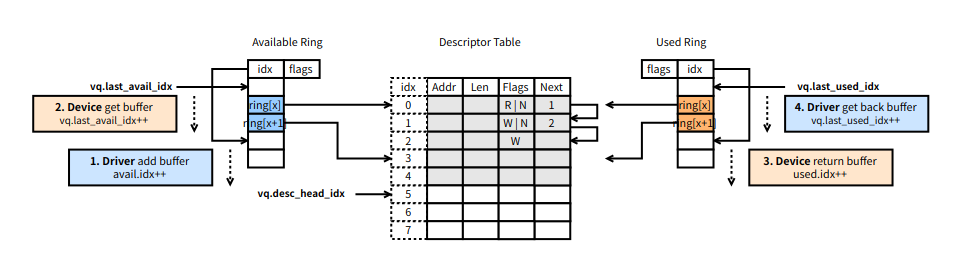


**2.1.5 Virtqueue**

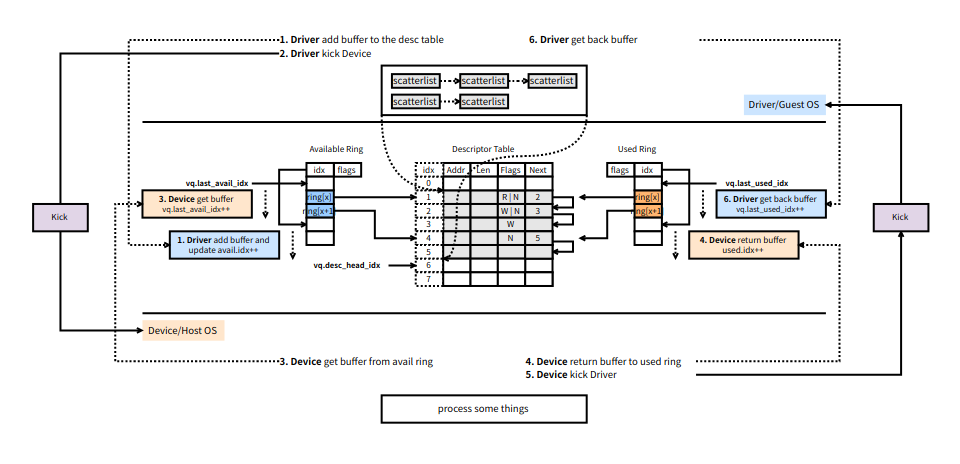
实际使用中，会在vring的基础上在创建一个真正的virtqueue的结构，用于维护vring，称之为virtqueue

|  |
| --- |
| C struct vq\_desc\_extra {  void \*cookie;  uint16\_t ndescs; };  struct virtqueue {  /\* Virtio ring \*/  struct virtq vring;  /\* Last consumed descriptor in the available table \*/  uint16\_t last\_avail\_idx;  /\* Last consumed descriptor in the used table \*/  uint16\_t last\_used\_idx;  /\* Head of the free chain in the descriptor table \*/  uint16\_t desc\_head\_idx;  /\* Store Cookie \*/  struct vq\_desc\_extra descx[0]; }; |

* vring：vring结构，见上一小节
* desc\_head\_idx：未使用的descriptor table的头的索引
* last\_used\_idx：最后被使用的used ring的索引
* last\_avail\_idx：最后被使用的avail ring的索引
* descx：存储driver端的cookie相关数据



**2.2 收发流程**



**2.2.1 Driver发送/Device接收**

1. Driver准备buffer后，添加到Descriptor Table中，然后更新对应的tx virtqueue的avail\_ring.idx表示有新的数据；
2. Driver通知Device发送了新的数据;
3. Device通过Available Ring找到了要接收的数据（desc[avail.ring[last\_avail\_idx + 1]]）；
4. Device处理数据：读取只读区域，填充可写区域，根据不同的device和不同的特性有不同的处理方式；
5. Device将处理好的数据通过Used Ring归还给Driver，更新rx virtqueue的used\_ring.idx表示有新的归还的数据；
6. Device通知Driver归还了新的数据；
7. Driver通过Used Ring找到了要收回的数据（desc[used.ring[last\_used\_idx + 1].id]）；
8. Driver根据收到的数据做对应的处理，不同的device和不同的特性有不同的处理方式；

**2.2.2 Device发送/Driver接收**

Device发送/Driver接收的过程和Driver发送/Device接收的过程基本一致，唯一的区别在于由于**buffer全部由Driver来进行管理**，所以Device发送的buffer来自于Driver预先填充到Available Ring中，之后Device的发送流程和2.2.1中的3 ~ 8完全相同；

**2.3 收发的无锁实现**

virtqueue为了性能，使用了无锁设计。为了实现无锁，virtio给driver给device划分的明确的访问权限，如下表，蓝色由Driver来维护，橙色由Device来维护

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Descriptor Table | Available Ring | Used Ring | desc\_head\_idx | last\_avail\_idx | last\_used\_idx |
| Drvier | RW | RW | R | RW | × | RW |
| Device | R | R | RW | × | RW | × |

**2.3.1 Driver发送/Device接收**

Driver写入buffer到Descriptor Table，更新Available ring

|  |
| --- |
| C void add\_buffer(struct virtqueue \*vq, void \*buffer, size\_t buffer\_len) {  /\* 1. 填充buffer到Descriptor Table \*/  desc\_idx = desc\_head\_idx;  desc[desc\_idx].addr = buffer;  desc[desc\_idx].len = buffer\_len;    /\* cache flush保证device端能读到正确的descriptor \*/  **CACHE\_FLUSH(desc[desc\_idx]);**    /\* 2. 更新Available Ring \*/  idx = avail.idx & (ring.num - 1);  avail.ring[idx] = desc\_idx;    /\* cache flush保证device端能读到正确的avail ring \*/  **CACHE\_FLUSH(avail.ring[idx]);**    /\* 保证前面的内存已经写入memory, 因为device会根据avail.idx  \* 和last\_avail\_idx来寻找可用的descriptor，要保证avail.idx的更新  \* 在ring[idx]的更新之后  \*/  **atomic\_thread\_fence(memory\_order\_seq\_cst);**    /\* 3. 最后更新avail.idx，这时desc中的buffer访问权限给到device  \* driver不应该再访问该buffer  \*/  avail.idx++;    /\* cache flush保证device端能读到正确的avail.idx \*/  **CACHE\_FLUSH(avail.idx);** } |

Device根据avail.idx和last\_avail\_idx找到Descriptor Table然后得到buffer

|  |
| --- |
| C void \*get\_avail\_buffer(uint32\_t \*buffer\_len) {  /\* Avail.idx is updated by driver, invalidate it \*/  **CACHE\_INVALIDATE(avail.idx);**  /\* 1. 判断是否有新的buffer \*/  if (last\_avail\_idx == avail.idx)  return NULL;   /\* 2. 获取avail ring中的的idx \*/  idx = last\_avail\_idx++ & (vring.num - 1);    /\* Avail.ring is updated by driver, invalidate it \*/  **CACHE\_INVALIDATE(avail->ring[idx]);**    /\* 3. 拿到descriptor table的idx \*/  avail\_idx = avail->ring[idx];    /\* Invalidate the desc entry written by driver before accessing it \*/  **CACHE\_INVALIDATE(desc[avail\_idx]);**    /\* 4. 获取buffer后返回 \*/  buffer = PHY2VIRT(desc[avail\_idx].addr);  \*buffer\_len = desc[avail\_idx].len;  return buffer; } |

**2.3.2 Device发送/Driver接收**

Device将Driver传过来的buffer填充好后，通过Used Ring返回到Driver

|  |
| --- |
| C void add\_used\_buffer(uint16\_t head\_idx, uint32\_t len) {  /\* CACHE: used is never written by driver, so it's safe to directly access it \*/  /\* 1. Device将descriptor table idx填充到used中 \*/  used\_idx = used.idx & (ring.num - 1);  used\_desc = &used.ring[used\_idx];  used\_desc->id = head\_idx; /\* Descriptor table idx \*/  used\_desc->len = len; /\* Used buffer length \*/    /\* cache flush used.ring[used\_idx]，该部分会被driver读 \*/  **CACHE\_FLUSH(used.ring[used\_idx]);**    /\* 保证used.idx一定在used.ring[used\_idx]更新后才更新 \*/  **atomic\_thread\_fence(memory\_order\_seq\_cst);**    /\* 2. 更新used.idx，这样Driver能识别到used.ring中有新buffer，  \* 此时buffer的访问权限已归还给Driver，Device不能再访问该buffer  \*/  used.idx++;    /\* cache flush used.idx.该部分会被driver读 \*/  **CACHE\_FLUSH(used.idx);** } |

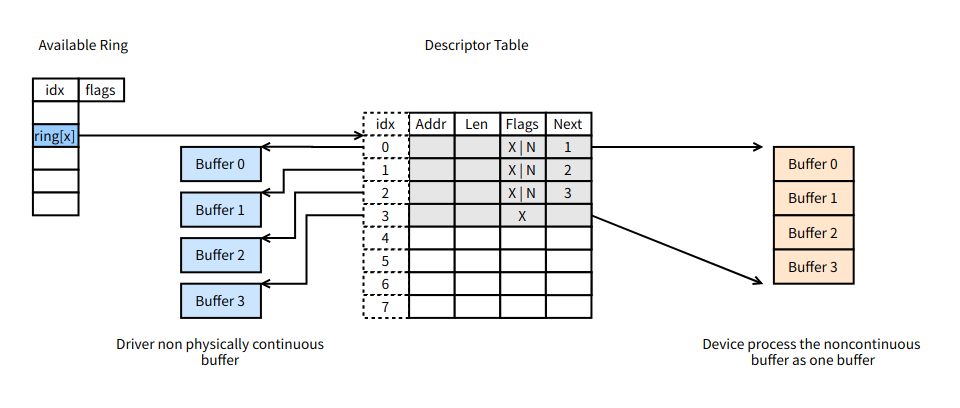
Driver从将Device规范的buffer，从Used Ring中获取，返回cookie

|  |
| --- |
| C void \*get\_buffer(uint32\_t \*len, uint16\_t \*idx) {  /\* Used.idx is updated by the Device, so invalidate it \*/  **CACHE\_INVALIDATE(used.idx);**  /\* 判断是否有Device归还的buffer \*/  if (last\_used\_idx == used.idx)  return NULL;   used\_idx = last\_used\_idx++ & (vring.num - 1);  uep = &used.ring[used\_idx];    /\* 保证cache invalidate在获取used\_idx后 \*/  **atomic\_thread\_fence(memory\_order\_seq\_cst);**   /\* Used.ring is written by Device, invalidate it \*/  **CACHE\_INVALIDATE(used.ring[used\_idx]);**    /\* 返回descriptor idx和Device返回的长度 \*/  \*idx = uep->id;  \*len = uep->len;    /\* 拿到Cookie，该Cookie为在Driver Add buffer时设置 \*/  cookie = vq\_descx[uep->id].cookie;  vq\_descx[uep->id].cookie = NULL;  return cookie; } |

**2.4 不连续内存支持**

virtio支持将不连续的内存作为一个整体进行发送，无需将分散的数据重组为一个大整体后发送

最典型的场景就是virtio-net驱动中，协议栈使用的IOB\_BUFFERSIZE < 1512时，virtio可以直接将非连续的多个buffer发送出去，无需在driver内部合成一整个buffer后发送。



**2.5 通知抑制和Batch提交**

[2.1.2 Available Ring](https://xiaomi.f.mioffice.cn/wiki/wikk49h3wuek9HIAixufJC85yMf#doxk4YUWKKyeu4uqqYvooKQOC7g) 和 [2.1.3 Used Ring](https://xiaomi.f.mioffice.cn/wiki/wikk49h3wuek9HIAixufJC85yMf#doxk4s4EWOA4UEEoGu8LkJIdBbd)中描述了avail.flags，avail.used\_event和used.flags，used.avail\_event的作用，这里做详细说明

**2.5.1 通知抑制**

未使能VIRTIO\_RING\_F\_EVENT\_IDX 情况下

* Driver：如果Driver端不希望接收RX中断，可以设置

rxvq.ring.avail.flags |= VIRTQ\_AVAIL\_F\_NO\_INTERRUPT

Device在发送中断前，会检查txvq.ring.avail.flags，如果VIRTQ\_AVAIL\_F\_NO\_INTERRUPT置位，则不发送中断；

* Device：同理，Device端不希望收到数据时马上就被通知，可以设置

rxvq.ring.used.flags |= VIRTQ\_USED\_F\_NO\_NOTIFY

Driver在发送通知前，会检查txvq.ring.used.flags，如果VIRTQ\_USED\_F\_NO\_NOTIFY置位，则不发送通知；

OpenAMP有实现现成的接口：virtqueue\_enable\_cb(VQ), virtqueue\_disable\_cb(VQ)

**2.5.2 Batch提交**

一般driver在给device发送数据后，会调用virtqueue\_kick()通知device有数据需要处理，但每次都通知效率比较低下，理想的情况应该是在virtqueue已满，或者是无更多buffer需要发送时再调用virtuqueue\_kick()

|  |
| --- |
| C /\* Notify other side every time add buffer to the virtqueue \*/ void xxx\_transimit() {  ...  virtqueue\_add\_buffer(...);  virtqueue\_kick(); }  /\* Notify other size at a suitable moment \*/ void xxx\_transimit() {  ...  virtqueue\_add\_buffer(...);  if (virtqueue full or no more buffer need transimit)  virtqueue\_kick(); } |

**3 Virtio Device**

**3.1 Device ID**

Device端提供设备的ID，Driver端读取

|  |
| --- |
| C #define VIRTIO\_ID\_NETWORK 1UL #define VIRTIO\_ID\_BLOCK 2UL #define VIRTIO\_ID\_CONSOLE 3UL #define VIRTIO\_ID\_ENTROPY 4UL #define VIRTIO\_ID\_BALLOON 5UL #define VIRTIO\_ID\_IOMEMORY 6UL #define VIRTIO\_ID\_RPMSG 7UL #define VIRTIO\_ID\_SCSI 8UL #define VIRTIO\_ID\_9P 9UL #define VIRTIO\_ID\_RPROC\_SERIAL 11UL #define VIRTIO\_ID\_GPU 16UL #define VIRTIO\_ID\_INPUT 18UL #define VIRTIO\_ID\_VSOCK 19UL #define VIRTIO\_ID\_CRYPTO 20UL #define VIRTIO\_ID\_IOMMU 23UL #define VIRTIO\_ID\_MEM 24UL #define VIRTIO\_ID\_SOUND 25UL #define VIRTIO\_ID\_FS 26UL #define VIRTIO\_ID\_PMEM 27UL #define VIRTIO\_ID\_RPMB 28UL #define VIRTIO\_ID\_SCMI 32UL #define VIRTIO\_ID\_I2C\_ADAPTER 34UL #define VIRTIO\_ID\_BT 40UL #define VIRTIO\_ID\_GPIO 41UL |

**3.2 Device Status Field**

一块共享内存区域，存储设备状态，Guest OS和Host OS均可以访问，通过这个状态来进行设备的初始化

* **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.
* **FAILED (128)**：Indicates that something went wrong in the guest, and it has given up on the device. This could be an internal error, or the driver didn’t like the device for some reason, or even a fatal error during device operation.
* **FEATURES\_OK (8)**：Indicates that the driver has acknowledged all the features it understands, and feature negotiation is complete.
* **DRIVER\_OK (4)**：Indicates that the driver is set up and ready to drive the device.
* **DEVICE\_NEEDS\_RESET (64)**：Indicates that the device has experienced an error from which it can’t recover.

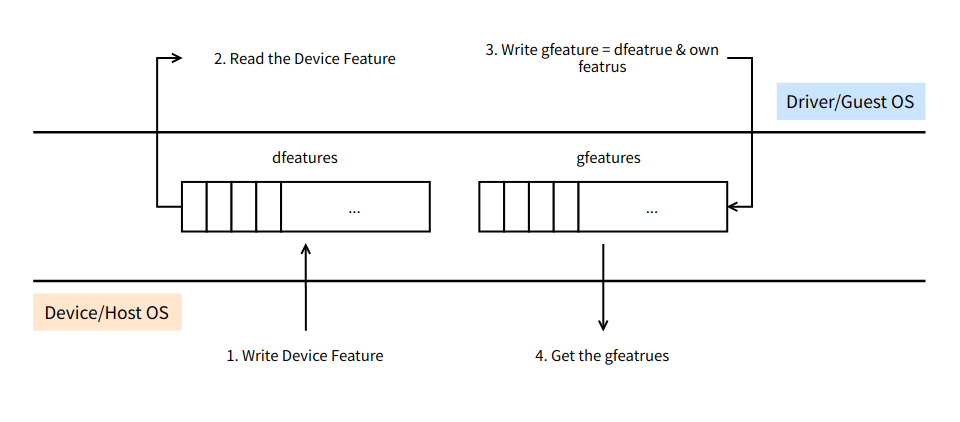
The device status field starts out as 0, and is reinitialized to 0 by the device during reset.

**3.3 Feature协商**

一块共享内存区域，存储了Device端支持的**dfeature**，以及最终确定的**gfeatrue**

每一个Bit位表示一种Feature，以block设备为例子

* **1 << VIRTIO\_BLK\_F\_RO (5)**：表示设备是只读的；
* **1 << VIRTIO\_BLK\_F\_FLUSH (9)**：表示block设备是否支持flush命令；
* **...**



1. Device将自己的Featrue写入device feature处；
2. Driver读取后，和自己支持的feature相与，得到最终的Featrue Bit然后写入到gfeature；
3. Device读取gfeature后，两边完成了feature的协商，Device和Driver都只使用两边均支持的特性来进行（gfeature）交互；

通过如上机制，实现了feature的协商、扩展和前后兼容。

**3.4 Config读取和写入**

某些复杂的virtio设备，可能包含一些配置信息（和支持的featrue有关），获取该配置信息的方式和传输层有关，Driver获取配置信息后，用于初始化、运行过程中。

|  |
| --- |
| C 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; }; |

**4 MMIO Transport Layer**

对于不支持PCI的虚拟环境（比如嵌入式环境），Virtio提供了MMIO（Simple memory mapped IO）传输层的支持。简单来说就是Device定义一块和Driver共享的memory，Driver和Device通过读取和写入这块memory来进行交互。

**4.1 MMIO Device Discovery**

MMIO没有通用的设备发现机制，一般都是在初始化过程中直接指定MMIO Register的地址大小和中断号，在Linux中，MMIO设备在设备树中定义：

|  |
| --- |
| C virtio\_block@1e000 {  compatible = "virtio,mmio";   reg = <0x1e000 0x200>; /\* MMIO Register Address and Register Size \*/  interrupts = <42>; /\* Interrupt number \*/ } |

在Vela中（或者其他无设备树支持的RTOS中），一般是提供一个API，指定MMIO Register地址和IRQ来显式初始化，Vela提供的API如下：

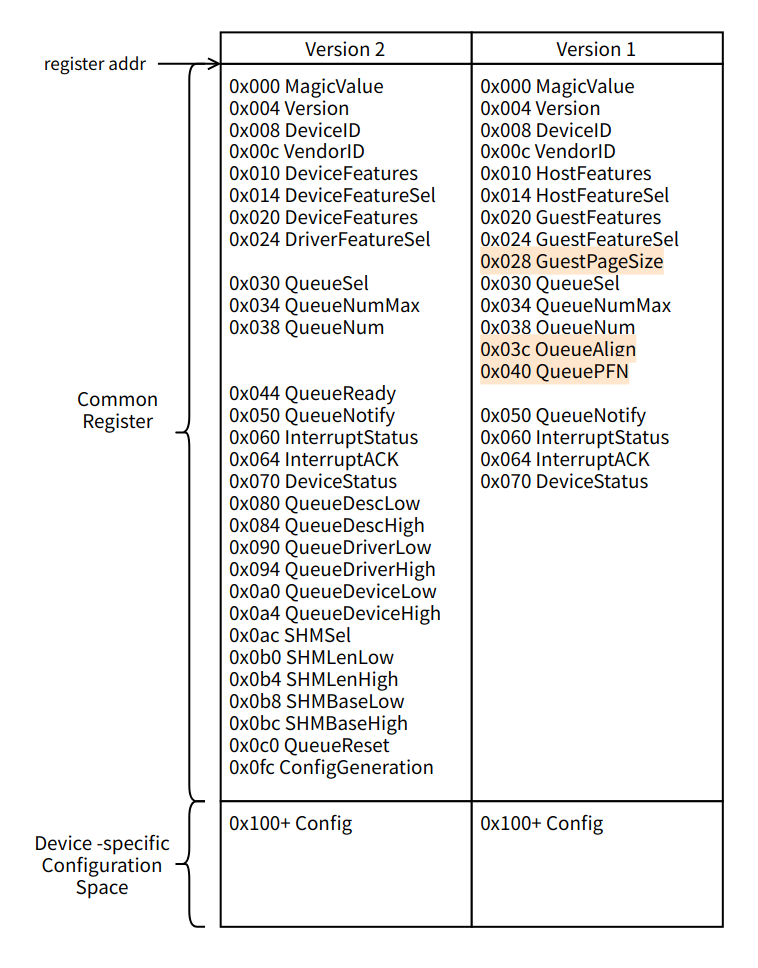
|  |
| --- |
| C int virtio\_register\_mmio\_device(FAR void \*regs, int irq); |

**4.2 MMIO Register**

Device会提供一块内存（3.1）作为MMIO的配置区域，称之为MMIO Register，同时针对不同的设备，提供了一块专门的配置区域，Driver和Device可以通过该区域来进行配置项的协商和传，比如virtio spec对net设备的配置信息描述如下：

|  |
| --- |
| C 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; }; |

详细的说明可以参考：https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.pdf



目前MMIO Version有两个版本，其Register Layout有些许不同，可以通过0x004 Version处的值来进行识别，Version 1主要是一些老软件还在使用，比如Qemu 2.12.0版本使用的是Version 1，Vela目前的MMIO实现两个版本都兼容。

部分重要的Register：

* 0x008 DeviceID：Driver侧读取该寄存器可以得到设备的ID号，见[3.1 Device ID](https://xiaomi.f.mioffice.cn/wiki/wikk49h3wuek9HIAixufJC85yMf#doxk4OWAyCucqAiWU41Y9qi5qYg)；
* 0x050 QueueNotify：Driver通过写入该寄存器可以通知Device端（配合0x030 QueueSel使用）；
* 0x060 InterruptStatus和0x064 InterruptACK：中断处理的寄存器；
* 0x070 DeviceStatus：设备的状态，见[3.2 Device Status Field](https://xiaomi.f.mioffice.cn/wiki/wikk49h3wuek9HIAixufJC85yMf#doxk4Sqsgo2GceSSWGYtOXuBOub)；
* 0x030 QueueSel, 0x038 QueueNum和0x080 ~ 0x0a4 Queue地址：Driver通过这些寄存器告知Virtqueue的数量，以及其中Descriptor Table、Available Ring和Used Ring的地址；

**参考文档**

1. https://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
2. https://docs.oasis-open.org/virtio/virtio/v1.2/csd01/virtio-v1.2-csd01.pdf
3. [OpenAMP.pptx](https://xiaomi.f.mioffice.cn/wiki/wikk4qeQOlrTjm9OiGwXx3b7xG5)
4. [VirtIO/QEMU介绍](https://xiaomi.f.mioffice.cn/wiki/wikk4r8ZUa03fYh4zPMGYDCf1ep)