内容

1.pci 基本概念

2.virtio 设备初始化和使用的过程

pci 基本概念

0.缩写

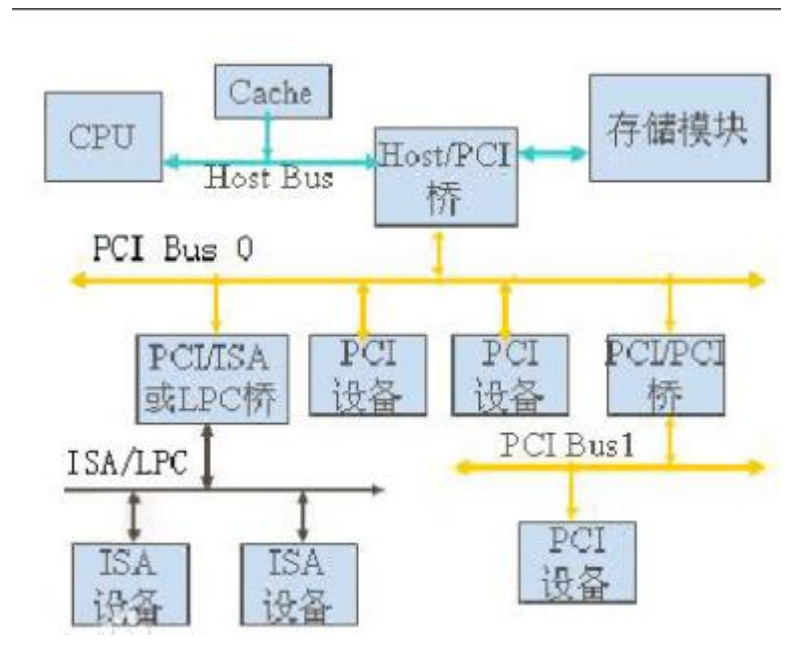
BDF : Bus Number, Device Number、Function Number
BAR : Base Address Registers

1.PCI 总线结构

PCI 总线是一个以 HOST 主桥为根的树型结构

PCI 总线是一种树型结构，并且独立于 CPU 总线，可以和 CPU 总线并行操作。

PCI 总线上可以挂接 PCI 设备和 PCI 桥片，PCI 总线上只允许有一个 PCI 主设备，其他的均为 PCI 从设备，而且读写操作只能在主从设备之间进行，从设备之间的数据交换需要通过主设备中转。



- PCI 总线由 HOST 主桥或者 PCI 桥管理，用来连接各类设备

- 在一个处理器系统中，可以通过 PCI 桥扩展 PCI 总线，并形成具有血缘关系的多级 PCI 总线，从而形成 PCI 总线树型结构。在处理器系统中有几个 HOST 主桥，就有几颗这样的 PCI 总线树，而每一颗 PCI 总线树都与一个 PCI 总线域对应。

- PCI 空间与处理器空间隔离:

深入理解 PCI 空间与处理器空间的不同是理解和使用 PCI 的基础。

PCI 设备具有独立的地址空间，即 PCI 总线地址空间，该空间与存储器地址空间通过 Host bridge 隔离。
— 处理器需要通过 Host bridge 才能访问 PCI 设备，而 PCI 设备需要通过 Host bridge 才能主存储器。

在 Host bridge 中含有许多缓冲，这些缓冲使得处理器总线与 PCI 总线工作在各自的时钟频率中，彼此互不干扰。

Host bridge 的存在也使得 PCI 设备和处理器可以方便地访问共享主存储器资源。
—处理器访问 PCI 设备时，必须通过 Host bridge 进行地址转换；而 PCI 设备访问主存储器时，也需要通过 Host bridge 进行地址转换。

2.基本概念

1）总线方式

•PCI 总线的地址总线与数据总线**是分时复用的**。这样做的好处是，一方面可以节省接插件的管脚数，另一方面便于实现突发数据传输。

2）数据传输过程

• 在做数据传输时，由一个 PCI 设备做发起者（主控，Initiator 或 Master），而另一个 PCI 设备做目标（从设备，Target 或 Slave）。

总线上的所有时序的产生与控制，都由 Master 来发起。PCI 总线在同一时刻只能供一对设备完成传输，这就要求有一个仲裁机构（Arbiter），来决定在谁有权力拿到总线的主控权。

• PCI 总线进行操作时，发起者（Master）先置 **REQ#**，当得到仲裁器（Arbiter）的许可时（**GNT#**），会将 **FRAME#**置低，并在 AD 总线上放置 **Slave 地址**，同时 **C/BE#**放置命令信号，说明接下来的传输类型。**所有 PCI 总线上设备都需对此地址译码**，被选中的设备要置 **DEVSEL#**以声明自己被选中。然后当 **IRDY#**与 **TRDY#**都置低时，可以传输数据。

当 Master 数据传输结束前，将 **FRAME#**置高以标明只剩最后一组数据要传输，并在传完数据后放开 **IRDY#**以释放总线控制权。

3）中断共享的实现

ISA 卡的一个重要局限在于中断是独占的，而我们知道计算机的中断号只有 16 个，系统又用掉了一些，这样当有多块 ISA 卡要用中断时就会有问题了。

硬件上，采用电平触发的办法：中断信号在系统一侧用电阻接高，而要产生中断的板卡上利用三极管的集电极将信号拉低。这样不管有几块板产生中断，中断信号都是低；而只有当所有板卡的中断都得到处理后，中断信号才会恢复高电平。

软件上，采用中断链的方法：假设系统启动时，发现板卡 A 用了中断 7，就会将中断 7 对应的内存区指向 A 卡对应的中断服务程序入口 ISR_A；然后系统发现板卡 B 也用中断 7，这时就会将中断 7 对应的内存区指向 ISR_B，同时将 ISR_B 的结束指向 ISR_A。以此类推，就会形成一个中断链。而当有中断发生时，系统跳转到中断 7 对应的内存，也就是 ISR_B。ISR_B 就要检查是不是 B 卡的中断，如果是，要处理，并将板卡上的拉低电路放开；如果不是，则呼叫 ISR_A。这样就完成了中断的共享。

3. 总线设备

在 PCI 总线中有三类设备，**PCI 主设备、PCI 从设备、桥设备**。

- 其中 PCI 从设备只能被动地接收来自 HOST 主桥，或者其他 PCI 设备的读写请求；
- 而 PCI 主设备可以通过总线仲裁获得 PCI 总线的使用权，主动地向其他 PCI 设备或者主存储器发起存储器读写请求。
- 而桥设备的主要作用是管理下游的 PCI 总线，并转发上下游总线之间的总线事务

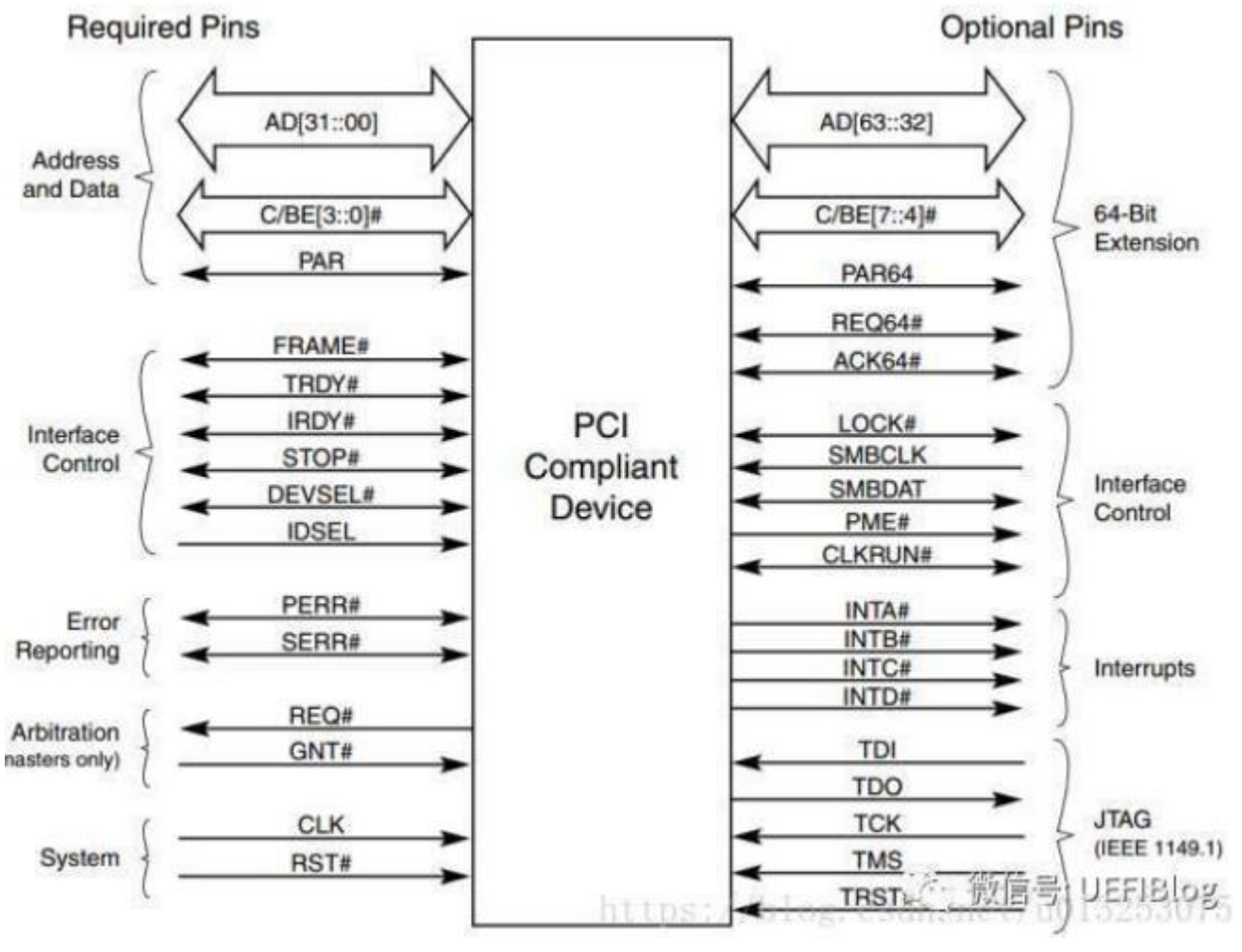
PCI Agent 设备 ： 一个 PCI 设备即是主设备也是从设备，但是在同一个时刻，这个 PCI 设备或者为主设备或者为从设备。

PCI 总线规范将 PCI 主从设备统称为 PCI Agent 设备。在处理器系统中常见的 PCI 网卡、显卡、声卡等设备都属于 PCI Agent 设备。

HOST 主桥 ： 在 PCI 总线中，HOST 主桥是一个特殊的 PCI 设备，该设备可以获取 PCI 总线的控制权访问 PCI 设备，也可以被 PCI 设备访问。但是 HOST 主桥并不是 PCI 设备。PCI 规范也没有规定如何设计 HOST 主桥。

桥设备 ：桥设备包括 PCI 桥、PCI-to-(E)ISA 桥和 PCI-to-Cardbus 桥。PCI 桥的存在使 PCI 总线极具扩展性，处理器系统可以使用 PCI 桥进一步扩展 PCI 总线。

4. 总线信号



1）**系统时钟信号 CLK IN：** 为所有 PCI 传输提供时序，对于所有的 PCI 设备都是输入信号。其频率最高可达 33MHz/66MHz，这一频率也称为 PCI 的工作频率。 RST# IN：复位信号。用来迫使所有 PCI 专用的寄存器、定序器和信号转为初始状态。

2）**地址和数据信号 AD**[31:: 00]T/S：地址、数据复用的信号。PCI 总线上地址和数据的传输，必需在 FRAME#有效期间进行。

当 **FRAME#有效时的第 1 个时钟**，AD[31:: 00]上的信号为**地址信号**，称**地址期**；

当 **IRDY#**和 **TRDY#**同时有效时，AD[31:: 00]上的信号为**数据信号**，称**数据期**。

一个 PCI 总线传输周期包含一个地址期和接着的一个或多个数据期。

C/BE[3:: 0]# T/S：**总线和字节命令**允许复用信号。

- 在地址期，这 4 条线上传输的是总线命令；
- 在数据期，它们传输的是字节允许信号，AD[31:: 00]线上 4 个数据字节中哪些字节为有效数据，以进行传输。

PAR T/S：奇偶校验信号。它通过 AD[31:：00]和 C/BE[3:：0]进行奇偶校验。主设备为地址周期和写数据周期驱动 PAR，从设备为读数据周期驱动 PAR。

3）接口控制信号

FRAME# S/T/S：**帧周期信号**，由主设备驱动。表示一次总线传输的开始和持续时间。

当 FRAME#有效时，预示总线传输的开始；在其有效期间，先传地址，后传数据；

当 FRAME#撤消时，预示总线传输结束，并在 IRDY#有效时进行最后一个数据期的数据传送。

IRDY# S/T/S：**主设备准备好信号**。

- IRDY#要与 TRDY#联合使用，当二者同时有效时，数据方能传输，否则，即为未准备好二进入等待周期。
- 在写周期，该信号有效时，表示数据已由主设备提交到 AD[31:：00]线上；
- 在读周期，该信号有效时，表示主设备已做好接收数据的准备。

TRDY# S/T/S：**从设备（被选中的设备）准备好信号**。

- 同样 TRDY#要与 IRDY#联合使用，只有二者同时有效，数据才能传输。

STOP# S/T/S：**从设备要求主设备停止当前的数据传送的信号。**

- 显然，该信号应由从设备发出。

LOCK# S/T/S：**锁定信号**。

- 当对一个设备进行可能需要多个总线传输周期才能完成的操作时，使用锁定信号 LOCK#，进行独占性访问。

例如，某一设备带有自己的存储器，那么它必需能进行锁定，以便实现对该存储器的完全独占性访问。也就是说，对此设备的操作是排它性的。

IDSEL IN：初始化设备选择信号。

- 在参数配置读/写传输期间，用作片选信号。

DEVSEL# S/T/S：设备选择信号。

- 该信号由从设备在识别处地址时发出，当它有效时，说明总线上有某处的某一设备已被选中，并作为当前访问的从设备。

4）仲裁信号（只用于总线主控器）

REQ# T/S：**总线占用请求信号**。

- 该信号有效表明驱动它的设备要求使用总线。它是一个点到点的信号线，任何主设备都有它自己的 REQ#信号。

GNT# T/S：**总线占用允许信号**。

- 该信号有效，表示申请占用总线的设备的请求已获得比准。

5）错误报告信号

PERR# S/T/S：**数据奇偶校验错误报告信号**。

- 一个设备只有在响应设备选择信号（DEVSEL#）和完成数据期之后，才能报告一个 PERR#。

SERR# O/D：**系统错误报告信号**。

- 用做报告地址奇偶错、特殊命令序列中的数据奇偶错，以及其他可能引起灾难性后果的系统错误。它可由任何设备发出。

6）中断信号 - 在 PCI 总线中，中断是可选项，不一定必须具有

INTA# O/D 、**INTB#** O/D、**INTC#** O/D、**INTD#** O/D：

- 用于请求中断，仅对多功能设备有意义。所谓的多功能设备是指：将几个相互独立的功能集中在一个设备中。各功能与中断线之间的连接是任意的，没有任何附加限制。

7.）其他可选信号

（1）高速缓存支持信号：SBO# IN/OUT、SDONE IN/OUT

（2）64 位总线扩展信号：REQ64# S/T/S、ACK65# S/T/S、AD[63:： 32]T/S、C/BE[7:： 4]#T/S、PAR64 T/S。

（3）测试访问端口/边界扫描信号：TCK IN、TDI IN、TDO OUT、TMS IN、TRST# IN。

5. 配置空间

1）PCI spec 规定了 PCI 设备必须提供的**单独地址空间**：

配置空间（configuration space）,前 64 个字节（其地址范围为 **0x00~0x3F**）是所有 PCI 设备必须支持的（有不少简单的设备也仅支持这些），此外 PCI/PCI-X 还扩展了 **0x40~0xFF** 这段配置空间，在这段空间主要存放一些与 MSI 或者 MSI-X 中断机制和电源管理相关的 **Capability** 结构

2）那么如何访问这段空间呢？

如下，在 CONFIG_ADDRESS 端口填入 BDF,即可以在 CONFIG_DATA 上写入或者读出 PCI 配置空间的内容：

- 通常我们是以三段编码来区分 PCI 设备，即 Bus Number, Device Number 和 Function Number,以后我们简称他们为 BDF。**有了 BDF 我们既可以唯一确定某一 PCI 设备**。

- 不同的芯片厂商访问配置空间的方法略有不同，我们以 Intel 的芯片组为例，其使用 **IO 空间**的 **CF8h / CFCh** 地址来访问 PCI 设备的配置寄存器:

CF8h: **CONFIG_ADDRESS**。PCI 配置空间地址端口。

CONFIG_ADDRESS 寄存器格式：

31 位：Enabled 位。

23:16 位：总线编号。 B

15:11 位：设备编号。 D

10: 8 位：功能编号。 F

7: 2 位：配置空间寄存器编号。

1: 0 位：恒为“00”。这是因为 CF8h、CFCh 端口是 32 位端口。

CFCh: CONFIG_DATA。PCI 配置空间数据端口。

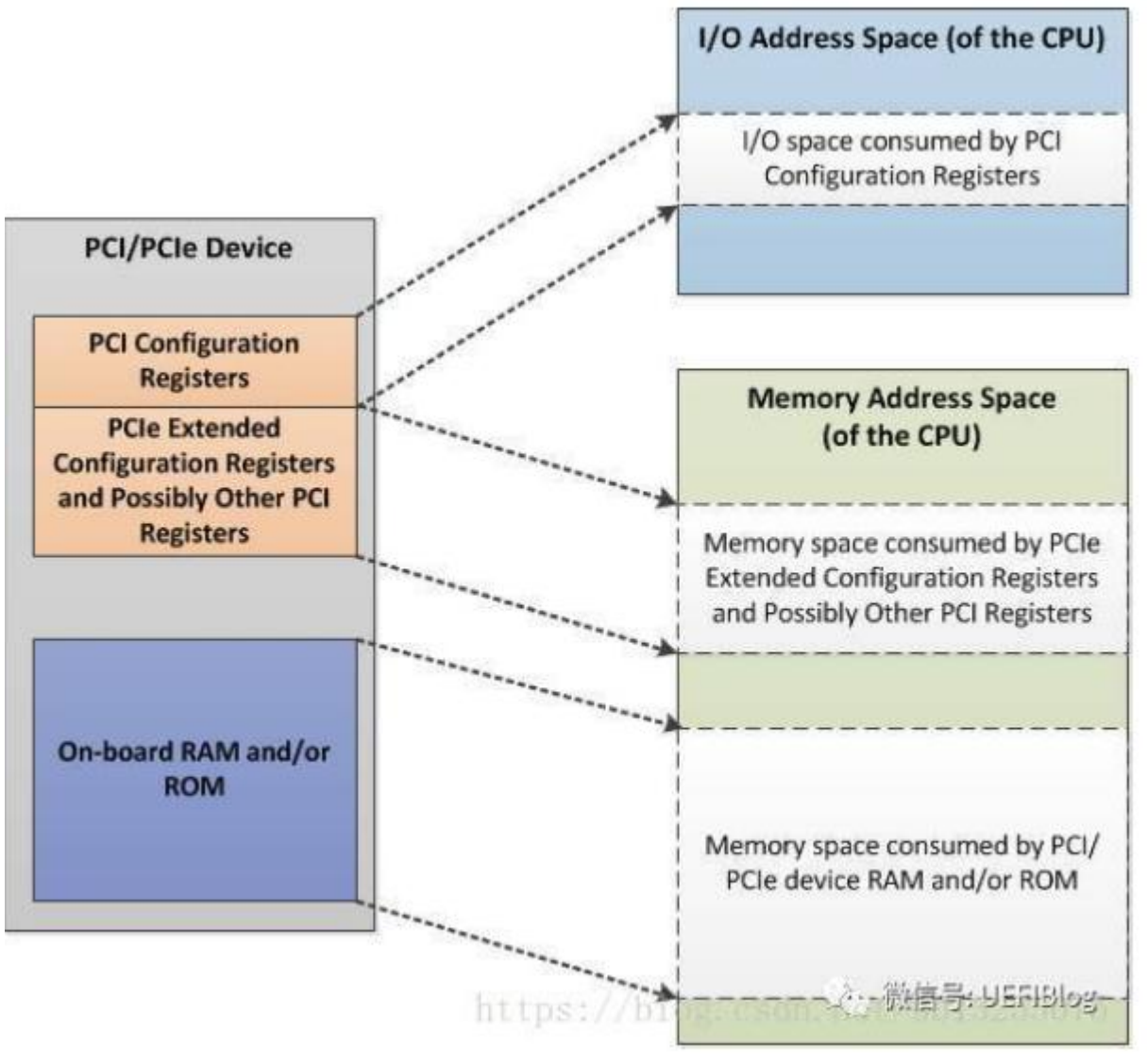
```
// arch\x86\pci\direct.c        pci 设备配置空间的读写操作 demo

#define PCI_CONF1_ADDRESS(bus, devfn, reg) \
        (0x80000000 | ((reg & 0xF00) << 16) | (bus << 16) \
        | (devfn << 8) | (reg & 0xFC))

static int pci_conf1_read(unsigned int seg, unsigned int bus,
                          unsigned int devfn, int reg, int len, u32 *value)
{
        ...
        outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);   // 将需要读取的 pci 设备地址写入 PCI 配置空间地址端口

        switch (len) {
        case 1:
                *value = inb(0xCFC + (reg & 3));            // 从 pci 配置空间数据端口 0xCFC 读出数据
                break;
        case 2:
                *value = inw(0xCFC + (reg & 2));
                break;
        case 4:
                *value = inl(0xCFC);
                break;
        }
...
        return 0;
}

static int pci_conf1_write(unsigned int seg, unsigned int bus,
                           unsigned int devfn, int reg, int len, u32 value)
{
...

        outl(PCI_CONF1_ADDRESS(bus, devfn, reg), 0xCF8);     // 将需要写入的 pci 设备地址写入 PCI 配置空间地址端口

        switch (len) {
        case 1:
                outb((u8)value, 0xCFC + (reg & 3));          // 将数据写入 pci 配置空间数据端口 0xCFC
                break;
        case 2:
                outw((u16)value, 0xCFC + (reg & 2));
                break;
        case 4:
                outl((u32)value, 0xCFC);
                break;
        }
...
}
```

3）pcie 的配置空间如何访问？ mmio 的引入

PCIe 规范在 PCI 规范的基础上，**将配置空间扩展到 4KB**。原来的 CF8/CFC 方法仍然可以访问所有 PCIe 设备配置空间的头 255B，但是该方法访问不了剩下的（4K-255）配置空间。怎么办呢？

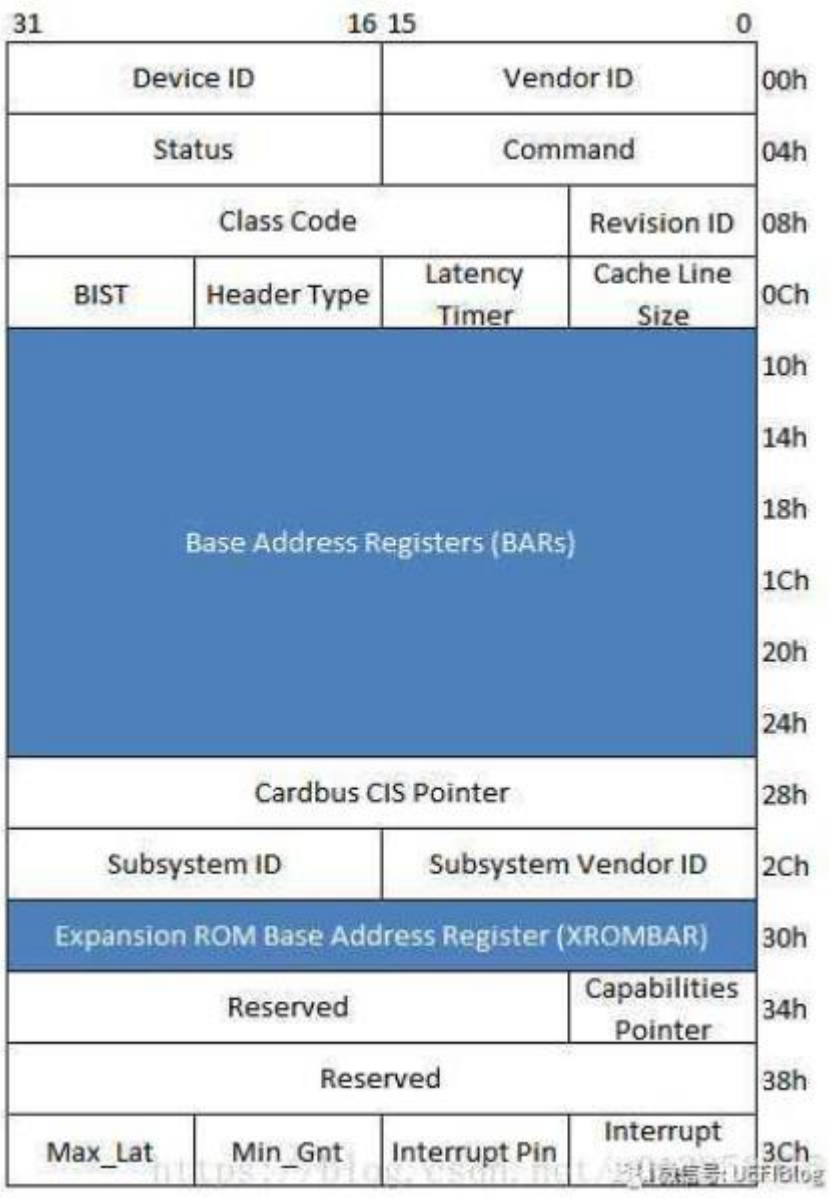
- Intel 提供了另外一种 PCIe 配置空间访问方法：通过**将配置空间映射到** Memory map IO（**MMIO**）空间，对 PCIe 配置空间可以像对内存一样进行读写访问了。
- 这样再加上 PCI 板子上的 RAM 或者 ROM，整个 PCIe Device 空间如下图

MMIO 这段空间有 256MB，因为按照 PCIe 规范，支持最多 256 个 buses，每个 Bus 支持最多 32 个 PCI devices，每个 device 支持最多 8 个 function，也就是说：占用内存的最大值为：256 * 32 * 8 * 4K = 256MB



4）pci 配置空间的内容

一般的 type 0（非 Bridge）设备



• 其中 Device ID 和 Vendor ID 是区分不同设备的关键，OS 和 UEFI 在很多时候就是通过匹配他们来找到不同的设备驱动（Class Code 有时也起一定作用）。

• BAR： Base Address Registers PCI 配置空间中从 0x10 到 0x24 的 **6 个 register，用来定义 PCI 需要的配置空间大小以及配置 PCI 设备占用的地址空间。**

-在 PCI 设备的配置空间中共有 6 个 BAR 寄存器，因此一个 PCI 设备最多可以使用 6 组 32 位的 PCI 总线地址空间，或者 3 组 64 位的 PCI 总线地址空间。

这些 BAR 空间可以保存 PCI 总线域的存储器地址空间或者 I/O 地址空间，目前多数 PCI 设备仅使用存储器地址空间。

- 每个 PCI 设备在 BAR 中描述自己需要占用多少地址空间，**UEFI 通过所有设备的这些信息构建一张完整的关系图，描述系统中资源的分配情况，然后在合理的将地址空间配置给每个 PCI 设备**。

- BAR 在 **bit0** 来表示该设备是映射到 **memory** 还是 **IO**，bar 的 bit0 是 readonly 的，也就是说，**设备寄存器是映射到 memory 还是 IO 是由设备制造商决定的，其他人无法修改。**

-访问 BAR 空间

在枚举过程中，PCI 驱动已经分配了 address 给各个 BAR，通过一些接口就可以访问到 BAR Resource。

位于 include/linux/pci.h，以宏的形式提供。

```
pci_resource_start(dev, bar)
pci_resource_end(dev, bar)
pci_resource_flags(dev, bar)
pci_resource_len(dev, bar)
```

通过 pci_resource_start 宏取得 bar 值之后，Linux 认为这个地址是 IO 地址，如果要访问的话可以通过 ioremap 映射到内核空间，然后通过 readl/writel 等 IO 接口进行操作。

demo

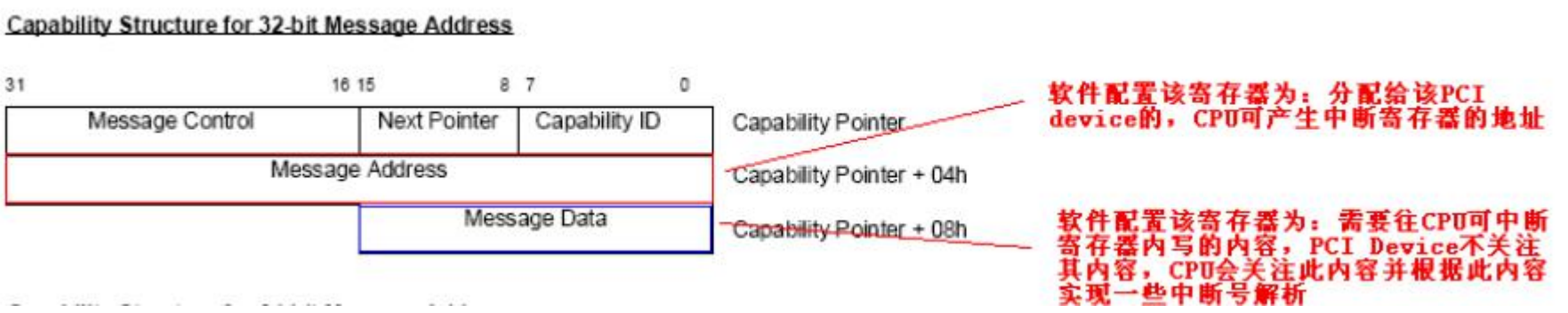
acrn-kernel\drivers\scsi\a100u2w.c

```
inia100_probe_one
{
	port = pci_resource_start(pdev, 0);
	host->base = port;
}
```

```
static u8 wait_chip_ready(struct orc_host * host)
{
	int i;

	for (i = 0; i < 10; i++) {  /* Wait 1 second for report timeout     */
		if (inb(host->base + ORC_HCTRL) & HOSTSTOP)  /* Wait HOSTSTOP set */
			return 1;
		msleep(100);
	}
	return 0;
}

static u8 wait_firmware_ready(struct orc_host * host)
{
	int i;

	for (i = 0; i < 10; i++) {  /* Wait 1 second for report timeout     */
		if (inb(host->base + ORC_HSTUS) & RREADY)          /* Wait READY set *
			return 1;
		msleep(100);  /* wait 100ms before try again  */
	}
	return 0;
```

• Capabilities 结构

PCI-X 和 PCIe 总线规范要求其设备必须支持 Capabilities 结构。在 PCI 总线的基本配置空间中，包含一个 Capabilities Pointer 寄存器，该寄存器存放 Capabilities 结构链表的头指针。在一个 PCIe 设备中，可能含有　多个 Capability 结构，这些寄存器组成一个链表



MSI：PCI 提供了一组 Capability Struct，其 Capability ID = 5。软件需要根据 Capabilities List 在 40-FF pci 配置空间中，找到 MSI 的 Capability Struct。

需要重点关注就两个寄存器 Message Address 和 Message Data。

寄存器 Message Address： CPU 可产生中断的寄存器的地址。

寄存器 Message Data ： 系统软件配置此寄存器为符合 CPU 中断号解析规则的内容，内容由 CPU 关注，pci device 不关注其内容，可以配置对低几位可自动修改，从而可以产生多个消息。

MSI 和 INTx 的区别：

MSI 和 INTx 中断是 pci/pcie 总线的两种中断方式。

.在 PCI 总线里面 INTx 中断是由四条可选的中断线决定的，这种中断方式是共享式的，所有的 pci 设备把中断信号在一条中短线上相与，再上报给 cpu，cpu 收到中断以后再查询具体是哪个设备产生了中断。

• 在 PCIE 总线里面已经没有了实体的 INTx 物理中断线了，而是通过往配置的 CPU 中断寄存器里进行 memory 写操作，来产生中断。

virtio pci

0.概念

virtio 设备本身是基于 PCI 总线的，因此本质上就是一个 **PCI 设备**，和所有其他 PCI 设备一样，virtio 也有自己的 vendor ID 0x1AF4，device ID 从 0x1000 - 0x103F，**subsystem device** ID 如下：

Subsystem Device ID	Virtio Device
1	Network card
2	Block device
3	Console
4	Entropy source
5	Memory ballooning
6	IoMemory
7	Rpmsg
8	SCSI host
9	9P transport
10	Mac80211 wlan

1. virtio 的配置空间

virtio 设备的第一块 IO region(BAR0 指向的空间?)用来存放 virtio 设备的配置空间，如下所示：

Bits	32	32	32	16	16	16	8	8
R/W	R	R+W	R+W	R	R+W	R+W	R+W	R
Purpose	Device Features	Guest Features	Queue Address	Queue Size	Queue Select	Queue Notify	Device Status	ISR Status

如果配置空间包含了后面两个域，即 CONFIG_VECTOR 以及 QUEUE_VECTOR，表明这个 PCI 设备开启了 MSI-X 中断，否则后面两个域不会在配置空间中

Bits	16	16
R/W	R+W	R+W
Purpose(MSI-X)	Configuration Vector	Queue Vector

可以从内核 include/linux/virtio_pci.h 中找到 virtio 配置空间的定义代码

```
/* A 32-bit r/o bitmask of the features supported by th
#define VIRTIO_PCI_HOST_FEATURES  0

/* A 32-bit r/w bitmask of features activated by the gu
#define VIRTIO_PCI_GUEST_FEATURES    4

/* A 32-bit r/w PFN for the currently selected queue */
#define VIRTIO_PCI_QUEUE_PFN     8

/* A 16-bit r/o queue size for the currently selected q
#define VIRTIO_PCI_QUEUE_NUM         12

/* A 16-bit r/w queue selector */
#define VIRTIO_PCI_QUEUE_SEL     14

/* A 16-bit r/w queue notifier */
#define VIRTIO_PCI_QUEUE_NOTIFY      16

/* An 8-bit device status register.  */
#define VIRTIO_PCI_STATUS            18

/* An 8-bit r/o interrupt status register.  Reading the
 * current contents of the ISR and will also clear it.
 * a read-and-acknowledge. */
#define VIRTIO_PCI_ISR               19

/* MSI-X registers: only enabled if MSI-X is enabled. *
/* A 16-bit vector for configuration changes. */
#define VIRTIO_MSI_CONFIG_VECTOR     20
/* A 16-bit vector for selected queue notifications. */
#define VIRTIO_MSI_QUEUE_VECTOR      22
```

2. 设备的操作

// drivers\virtio\Virtio_pci_legacy.c
对于设备的操作都在 virtio_config_ops 里面

```
static const struct virtio_config_ops virtio_pci_config_ops = {
        .get            = vp_get,
        .set            = vp_set,
        .get_status     = vp_get_status,
        .set_status     = vp_set_status,
        .reset          = vp_reset,
        .find_vqs       = vp_find_vqs,
        .del_vqs        = vp_del_vqs,
        .get_features   = vp_get_features,
        .finalize_features = vp_finalize_features,
        .bus_name       = vp_bus_name,
        .set_vq_affinity = vp_set_vq_affinity,
        .get_vq_affinity = vp_get_vq_affinity,
};
```

- vp_get, vp_set 最终都是通过 ioread/iowrite 操作来读取 pci 总线地址，这两个函数目前都是对于设备自己的配置做一些读写操作
- vp_get_status, vp_set_status 用于读写设备状态，由于 device status 总共只有 1 个字节，因此只需要一次 ioread8/iowrite8 即可。而 vp_reset 相当于把 VIRTIO_PCI_STATUS 写入 0
- vp_get_features, vp_finalize_features 也类似，由于 features 是 32bit 的，因此调用 ioread32/iowrite32 来实现，
  vp_get_features 用于获取 host feature，因此会读取 VIRTIO_PCI_HOST_FEATURES，vp_finalize_features 用于配置 guest features

3. uos 中 virtio 设备的初始化

```
//PCI Init: PCI initialization scans PCI bus/slot/function to identify each configured PCI device on the acrn-dm command line
            and initializes their configuration space by calling their dedicated vdev_init() function.
             For more detail of DM PCI emulation please refer to section 4.6.
```

```
// devicemodel\core\main.c

dm_run
      vm_init_vdevs
            init_pci
            {
                  for (bus = 0; bus < MAXBUSES; bus++) {
                        for (slot = 0; slot < MAXSLOTS; slot++) {
                              for (func = 0; func < MAXFUNCS; func++) {
                                          ops = pci_emul_finddev(fi->fi_name);  // 根据 class_name 找到对应的后端驱动，返回 ops 结构
                                          error = pci_emul_init(ctx, ops, bus, slot,
                                          {
                                          ...
                                                (*ops->vdev_init)(ctx, pdi, fi->fi_param);
                                          ...
                                          }
                  }
            }
```

```
acrn-dm -A -m 2048M -c 3 \
  -s 0:0,hostbridge \
  -s 1:0,lpc -l com1,stdio \
  -s 5,virtio-console,@pty:pty_port \
  -s 3,virtio-blk,b,/data/clearlinux/clearlinux.img \
  -s 4,virtio-net,tap_LaaG --vsbl /usr/share/acrn/bios/VSBL.bin \
  --intr_monitor 10000,10,1,100 \
  -B "root=/dev/vda2 rw rootwait maxcpus=3 nohpet console=hvc0 \
  console=ttyS0 no_timer_check ignore_loglevel log_buf_len=16M \
  consoleblank=0 tsc=reliable i915.avail_planes_per_pipe=0x070F00 \
  i915.enable_guc_loading=0 \
  i915.enable_hangcheck=0 i915.nuclear_pageflip=1 \
  i915.enable_guc_submission=0 i915.enable_guc=0" vm1
```

根据 **acrn-dm command line** 传入的名字匹配虚拟 pci 设备的 class_name 找到对应的驱动

```
Virtio_input.c (devicemodel\hw\pci\virtio):     .class_name = "virtio-input",
Virtio_ipu.c (devicemodel\hw\pci\virtio):       .class_name = "virtio-ipu",
Virtio_mei.c (devicemodel\hw\pci\virtio):       .class_name     = "virtio-heci",
Virtio_net.c (devicemodel\hw\pci\virtio):        .class_name      = "virtio-net",
Virtio_rnd.c (devicemodel\hw\pci\virtio):        .class_name      = "virtio-rnd",
Virtio_rpmb.c (devicemodel\hw\pci\virtio):      .class_name = "virtio-rpmb",
Wdt_i6300esb.c (devicemodel\hw\pci):            .class_name = "wdt-i6300esb",
```

4.virtqueue - 前后端数据的交互

0）结构关系

- 主要有 virtqueue 、virtio_pci_vq_info 、vring_virtqueue 、vring 这几个结构
- 在 virtio 设备初始化的过程中，会通过 setup_vp 创建 virtqueue，目前的 virtqueue 队列都是通过 vring 来实际工作的
  - 我们可以把 **virtqueue** 当做一个接口类，而把 **vring_virtqueue** 当做这个接口的一个实现

```
struct virtqueue {
    struct list_head list;
    void (*callback)(struct virtqueue *vq);
    const char *name;
    struct virtio_device *vdev;
    void *priv;
}
```

```
struct virtio_pci_vq_info
{
    struct virtqueue *vq;
    /* the number of entries in the queue */
    int num;.
    /* the index of the queue */
    int queue_index;
    /* the virtual address of the ring queue */
    void *queue;
    /* the list node for the virtqueues list */
    struct list_head node;
    /* MSI-X vector (or none) */
    unsigned msix_vector;
};
```

```
struct vring_virtqueue
{
    struct virtqueue vq;  /* virtqueue的接口类定义 */

    /* Actual memory layout for this queue */
    struct vring vring;  /* vring memory layout指针 */

    /* Other side has made a mess, don't try any more. */
    bool broken;

    /* Host supports indirect buffers */
    bool indirect;  /* transport feature位，是否支持indirect buffer */

    /* Host publishes avail event idx */
    bool event;  /* transport feature位，是否支持event idx interrupt/notify */

    /* Number of free buffers */
    unsigned int num_free;  /* vring desc里面还剩余的free buffer个数，free buffer是free_head开头的一个list */

    /* Head of free buffer list. */
    unsigned int free_head;  /* vring desc的free buffer head index */

    /* Number we've added since last sync. */
    unsigned int num_added;  /*
                                表示 virtqueue_add_buf 被调用的次数，*/

    /* Last used index we've seen. */
    u16 last_used_idx;
```

```
struct vring {
unsigned int num;

struct vring_desc *desc;

struct vring_avail *avail;

struct vring_used *used;
};
```

```
struct vring_desc {
/* Address (guest-physical). */
__virtio64 addr;
/* Length. */
__virtio32 len;
/* The flags as indicated above. */
__virtio16 flags;
/* We chain unused descriptors via this, too */
__virtio16 next;
};


struct vring_avail {
__virtio16 flags;
__virtio16 idx;
__virtio16 ring[];    //保存了 vring_desc 结构地址
};


struct vring_used {
__virtio16 flags;
__virtio16 idx;
struct vring_used_elem ring[];
};
```

https://www.draw.io/

virtqueue striuct.xml

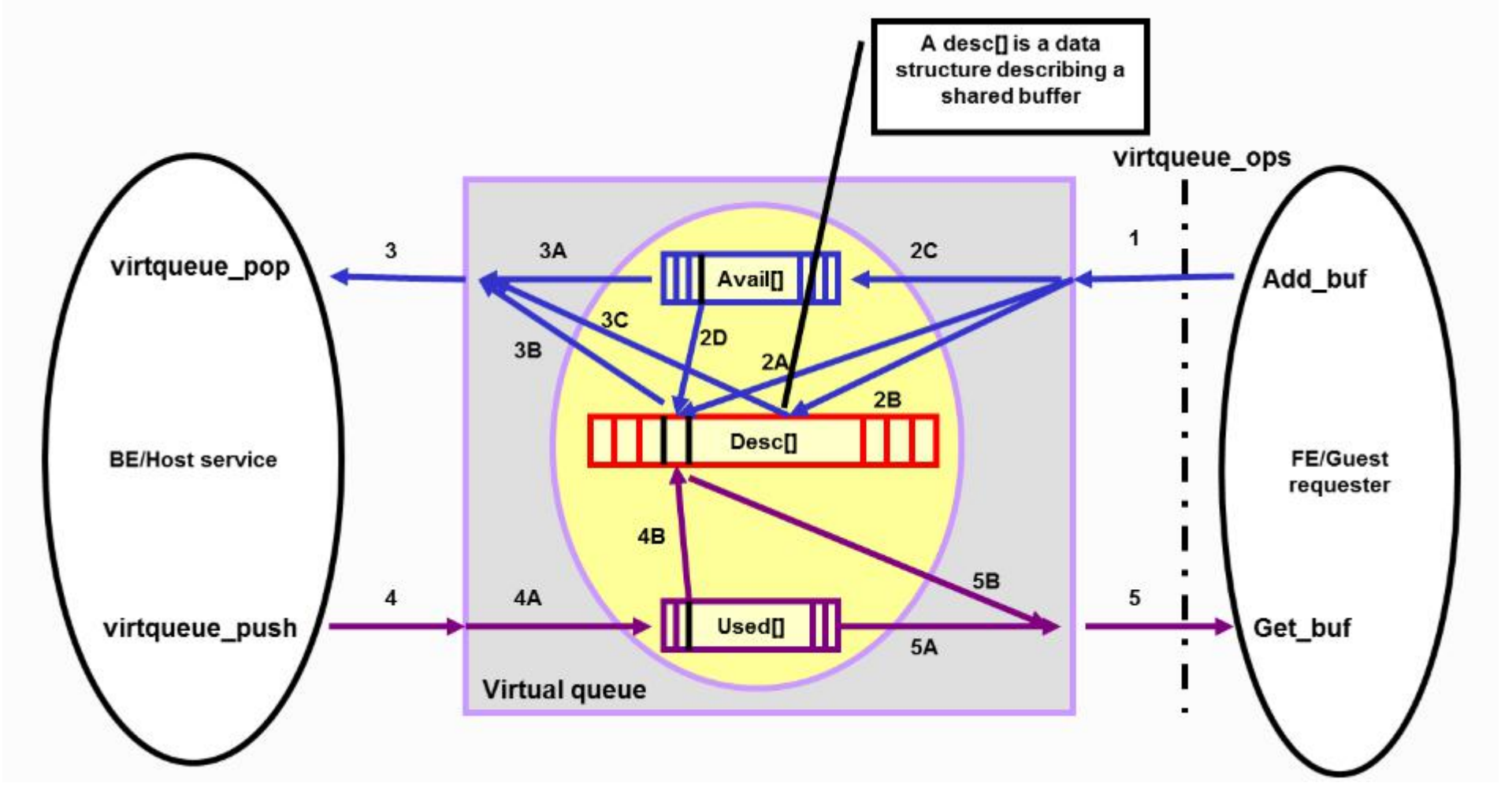
a，前端填充好 desc（addr/len），并更新 vring->avail（**ring**[0]）

b，后端读取 avail **ring 索引**

-找到 desc(if ring[0]=2,then desctable[2] 记录的就是一个逻辑 buffer 的首个物理块的信息

-填充 buffer 数据；将 buffer 索引存在 desc，将 desc 索引存放在 used ring 中

c，前端读取 used ring 索引，找到 desc，获取 buffer 数据



1 ） virtqueue 相关的操作

（1） vring_create_virtqueue vring_new_virtqueue

创建一个 vring_virtqueue，其中内存分配大小是 sizeof(struct vring_virtqueue) + num * sizeof(void *)，传入的 pages 内存是 vring 的 layout memory

（2） virtqueue_add_outbuf(struct virtqueue *vq,struct scatterlist *sg, unsigned int num,void *data,gfp_t gfp)

virtqueue_add_buf 把传入的 scatterlist 填入 vring_desc 的 free buffer 链表中，并更新 avail ring 的 idx 的 entry，指向新加入的 free buffer 链表头。

由于在前后端 idx 同步之前，有可能会有多次的 virtqueue_add_buf 调用，因此 vring_virtqueue 用了一个 num_added 来表示 virtqueue_add_buf 被调用的次数，

(3) void *virtqueue_get_buf(struct virtqueue *_vq, unsigned int *len) -- get the next used buffer
guest 端从 used 端获取数据
virtqueue_get_buf 用于回收 last_used_idx 指向的一个 used ring 的 entry

（4） virtqueue_kick

virtqueue_kick 用于通知 qemu/vhost 端 avail ring 有更新，其中 virtqueue_kick_prepare 用于计算是否需要 kick，而 virtqueue_notify 通过写入 virtio bar0 配置空间的 QUEUE_NOTIFY 字段产生 VMEXIT 从而被 qemu/vhost 捕获

（5）virtqueue_disable_cb virtqueue_enable_cb
virtqueue_disable_cb 用于关闭中断，virtqueue_enable_cb 用于打开中断

2）前端初始化 ：virtqueue 的创建过程以及前端接收数据时中断处理函数的注册

// drivers\virtio\virtio_pci_common.c
virtqueue 的初始化通过 virtio_config_ops->find_vqs = vp_find_vqs 来进行,所有的前端驱动或直接或间接的都会调用到这个函数来初始化 virtqueue
vp_find_vqs

```
int vp_find_vqs(struct virtio_device *vdev, unsigned nvqs,
                struct virtqueue *vqs[], vq_callback_t *callbacks[],
                const char * const names[], const bool *ctx,
                struct irq_affinity *desc)
{
..
        // ①  Virtio_pci_common.c (drivers\virtio)
```

```
vp_find_vqs_intx(vdev, nvqs, vqs, callbacks, names, ctx);
{
...
        request_irq(vp_dev->pci_dev->irq, vp_interrupt, IRQF_SHARED,dev_name(&vdev->dev), vp_dev);
                {
                    ...
                    vp_vring_interrupt(irq, opaque);
                    {
                        list_for_each_entry(info, &vp_dev->virtqueues, node) {
                                vring_interrupt(irq, info->vq
                                {
                                    ...
                                        vq->vq.callback(&vq->vq);        //后端数据处理完成，通过发送中断的方式通知前端，最终中断发生时调用到 vp_find_vqs 函数注册的 callbacks
                                    ...
                                }
                    }
                    ...
                }
...
// ②
    for (i = 0; i < nvqs; ++i)
      vqs[i] = vp_setup_vq(vdev, i, callbacks[i], names[i],ctx ? ctx[i] : false,VIRTIO_MSI_NO_VECTOR);
        {
            vp_dev->setup_vq(vp_dev, info, index, callback, name, ctx,msix_vec);
                {
                // Virtio_pci_legacy.c (drivers\virtio)
                    setup_vq
                    {
                     vring_create_virtqueue(index, num,VIRTIO_PCI_VRING_ALIGN, &vp_dev->vdev, true, false, ctx,vp_notify, callback, name);
                    }
                }
        }
    ...
    }
...
}
```

5. virtual net uos 前端发送数据到后端的过程

主要流程：前端通过 virtqueue_add_outbuf 函数往 virtqueue->vring-> avalible 里面填充数据后，通过 kick 通知后端接收数据，后端通过 notifer 回调函数接收处理前端传过来的数据。

virtqueue_add_outbuf

```
Virtio_balloon.c (drivers\virtio):      virtqueue_add_outbuf(vq, &sg, 1, vb, GFP_KERNEL);
Virtio_balloon.c (drivers\virtio):      virtqueue_add_outbuf(vq, &sg, 1, vb, GFP_KERNEL);
Virtio_balloon.c (drivers\virtio):          err = virtqueue_add_outbuf(vb->stats_vq, &sg, 1, vb,
Virtio_console.c (drivers\char):if (virtqueue_add_outbuf(vq, sg, 1, &portdev->cpkt, GFP_ATOMIC) == 0) {
Virtio_console.c (drivers\char):err = virtqueue_add_outbuf(out_vq, sg, nents, data, GFP_ATOMIC);
Virtio_input.c (drivers\virtio):        rc = virtqueue_add_outbuf(vi->sts, sg, 1, stsbuf, GFP_ATOMIC);
Virtio_net.c (drivers\net):err = virtqueue_add_outbuf(sq->vq, sq->sg, 1, xdpf, GFP_ATOMIC);
Virtio_net.c (drivers\net):return virtqueue_add_outbuf(sq->vq, sq->sg, num_sg, skb, GFP_ATOMIC);
Virtio_ring.c (drivers\virtio):int virtqueue_add_outbuf(struct virtqueue *vq,
Virtio_ring.c (drivers\virtio):EXPORT_SYMBOL_GPL(virtqueue_add_outbuf);
Virtio_rpmsg_bus.c (drivers\rpmsg):err = virtqueue_add_outbuf(vrp->svq, &sg, 1, msg, GFP_KERNEL);
Virtio_rpmsg_bus.c (drivers\rpmsg):     dev_err(dev, "virtqueue_add_outbuf failed: %d\n", err);
```

virtio-net Frontend Driver

```
start_xmit -->                          // virtual NIC driver xmit in virtio_net
    xmit_skb -->
        virtqueue_add_outbuf --> // add out buffer to shared virtqueue
            virtqueue_add -->

    virtqueue_kick -->                  // notify the backend
        virtqueue_notify -->
            vp_notify -->
                iowrite16 -->       // trap here, HV will first get notified   写的是哪个 io 地址？
```

```
// acrn-kernel\drivers\virtio\virtio_pci_legacy.c
static struct virtqueue *setup_vq(...)
{
...
        vq = vring_create_virtqueue(index, num,
                VIRTIO_PCI_VRING_ALIGN, &vp_dev->vdev,
                true, false, ctx,
                vp_notify, callback, name);    -> iowrite16(vq->index, (void __iomem *)vq->priv);
        vq->priv = (void __force *)vp_dev->ioaddr + VIRTIO_PCI_QUEUE_NOTIFY;    // 16 virtio_pci.h
...
}
```

ACRN Hypervisor

```
vmexit_handler -->                          // vmexit because    VMX_EXIT_REASON_IO_INSTRUCTION
    pio_instr_vmexit_handler -->
        emulate_io -->                      // ioreq cant be processed in HV, forward it to VHM
            acrn_insert_request_wait -->
                fire_vhm_interrupt -->    // interrupt SOS, VHM will get notified
```

VHM Module

```
vhm_intr_handler -->                        // VHM interrupt handler
    tasklet_schedule -->
        io_req_tasklet -->
            acrn_ioreq_distribute_request --> // ioreq can't be processed in VHM, forward it to device DM
                acrn_ioreq_notify_client -->
                    wake_up_interruptible --> // wake up DM to handle ioreq
```

ACRN Device Model / virtio-net Backend Driver

```
handle_vmexit -->
    vmexit_inout -->
        emulate_inout -->                   ①
            pci_emul_io_handler
          pci_vdev->ops->vdev_barread    -->
            // acrn-hypervisor\devicemodel\hw\pci\virtio\virtio_net.c
                virtio_pci_write -->
                    virtio_pci_legacy_write -->          // case VIRTIO_CR_QNOTIFY: 分支，调用到设备的 notify 函数
             net->queues[VIRTIO_NET_TXQ].notify = virtio_net_ping_txq;
                        virtio_net_ping_txq -->         // start TX thread to process, notify thread return
                          virtio_net_tx_thread -->    // this is TX thread
                              virtio_net_proctx --> // call corresponding backend (tap) to process
                                  virtio_net_tap_tx -->
                                      writev -->       // write data to tap device
```

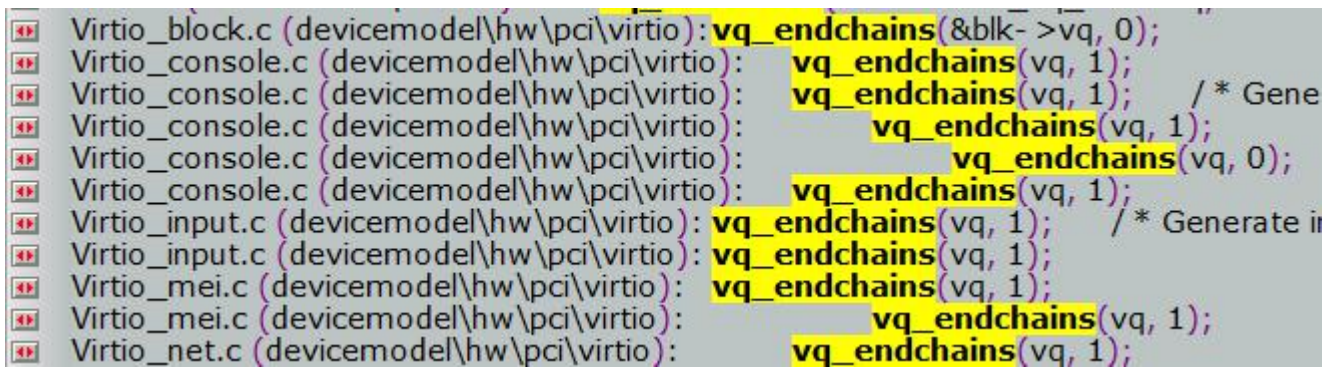
```
// ① devicemodel\core\inout.c
emulate_inout(struct vmctx *ctx, int *pvcpu, struct pio_request *pio_request)
{
...
            handler = inout_handlers[port].handler;
            retval = handler(ctx, *pvcpu, in, port, bytes,(uint32_t *)&(pio_request->value), arg);
...
        }

/ /acrn-hypervisor\devicemodel\hw\pci\core.c
modify_bar_registration(struct pci_vdev *dev, int idx, int registration)
{
...
        switch (dev->bar[idx].type) {
        case PCIBAR_IO:                 // pci
            bzero(&iop, sizeof(struct inout_port));
            iop.name = dev->name;
            iop.port = dev->bar[idx].addr;
            iop.size = dev->bar[idx].size;
            if (registration) {
                iop.flags = IOPORT_F_INOUT;
                iop.handler = pci_emul_io_handler;
                iop.arg = dev;
                error = register_inout(&iop);
```

```
                case PCIBAR_MEM32:

        case PCIBAR_MEM64:          // pcie
                bzero(&mr, sizeof(struct mem_range));
                mr.name = dev->name;
                mr.base = dev->bar[idx].addr;
                mr.size = dev->bar[idx].size;
                if (registration) {
                        mr.flags = MEM_F_RW;
                        mr.handler = pci_emul_mem_handler;
                        mr.arg1 = dev;
                        mr.arg2 = idx;
                        error = register_mem(&mr);
...
}
```

6. 后端数据传输到前端的过程

后端通过 vq_endchains 函数以 ioctl 的方式控制 sos 产生一个 msi 中断通知前端的 uos 接收数据，最终调用到前端通过 virtio_find_vqs 函数注册的中断处理回调函数，处理后端发送过程的数据。

vq_endchains




```
// 1. acrn DM
virtio_input_init(struct vmctx *ctx, struct pci_vdev *dev, char *opts)
{

...
      mevent_add(vi->fd, EVF_READ, virtio_input_read_event, vi);
         {
            ...
             read(vi->fd, &host_event, sizeof(host_event));
             vent.type = host_event.type;
           event.code = host_event.code;
           event.value = host_event.value;
           virtio_input_send_event(vi, &event);
             {
                ...
                        vq_endchains(vq, 1);
                {
                    ...
                    vq_interrupt(base, vq);
                    {
                        ...
                        pci_generate_msix
                          or
                        pci_generate_msi
                        {
                            ...                                          vm_lapic_msi
                                {
                                    ...
                                    ioctl(ctx->fd, IC_INJECT_MSI, &msi);
                                    ...
                                }
                            ...
                        }
                        ...
                    }
                ...
                }

        ...
    }
...
```

```
        }
...

}


//  sos vhm  Vhm_dev.c (drivers\char\vhm)
vhm_dev_ioctl
{
...
    case IC_INJECT_MSI: {
            struct acrn_msi_entry msi;

            if (copy_from_user(&msi, (void *)ioctl_param, sizeof(msi)))
                  return -EFAULT;

            ret = hcall_inject_msi(vm->vmid, virt_to_phys(&msi));
...

}
```

// 2. acrn DM : 中断导致虚拟机退出

```
  vmexit_handler -->                // vmexit because VMX_EXIT_REASON_VMCALL
    vmcall_vmexit_handler -->
```

// 3.sos 向 cpu 写入要产生 msi 中断的信息产生一个中断到 uos

```
        hcall_inject_msi -->     // insert interrupt into UOS
            vlapic_intr_msi -->
                vlapic_deliver_intr -->
                    vlapic_set_intr -->
                        vcpu_make_request  -->
                            send_single_ipi
                                msr_write
                                       cpu_msr_write
                                        {
                                          uint32_t msrl, msrh;

                                          msrl = (uint32_t)msr_val;
                                          msrh = (uint32_t)(msr_val >> 32U);
                                          asm volatile (" wrmsr " : : "c" (reg), "a" (msrl), "d" (msrh));
                                        }
```

// 4. uos 接收和处理中断 :调用 vq 的 cb 函数

```
vring_interrupt -->               // virtio-net frontend driver interrupt handler
     vq->vq.callback(&vq->vq);
```

// 5. virtio_input.c 中初始化 virtqueue 时给 cb 赋值

```
 // virtio_input.c (acrn-kernel\drivers\virtio)
virtinput_init_vqs(struct virtio_input * vi)
{
...
      vq_callback_t *cbs[] = { virtinput_recv_events,
                               virtinput_recv_status };
      static const char * const names[] = { "events", "status" };
      int err;

      err = virtio_find_vqs(vi->vdev, 2, vqs, cbs, names, NULL);

...

}
```