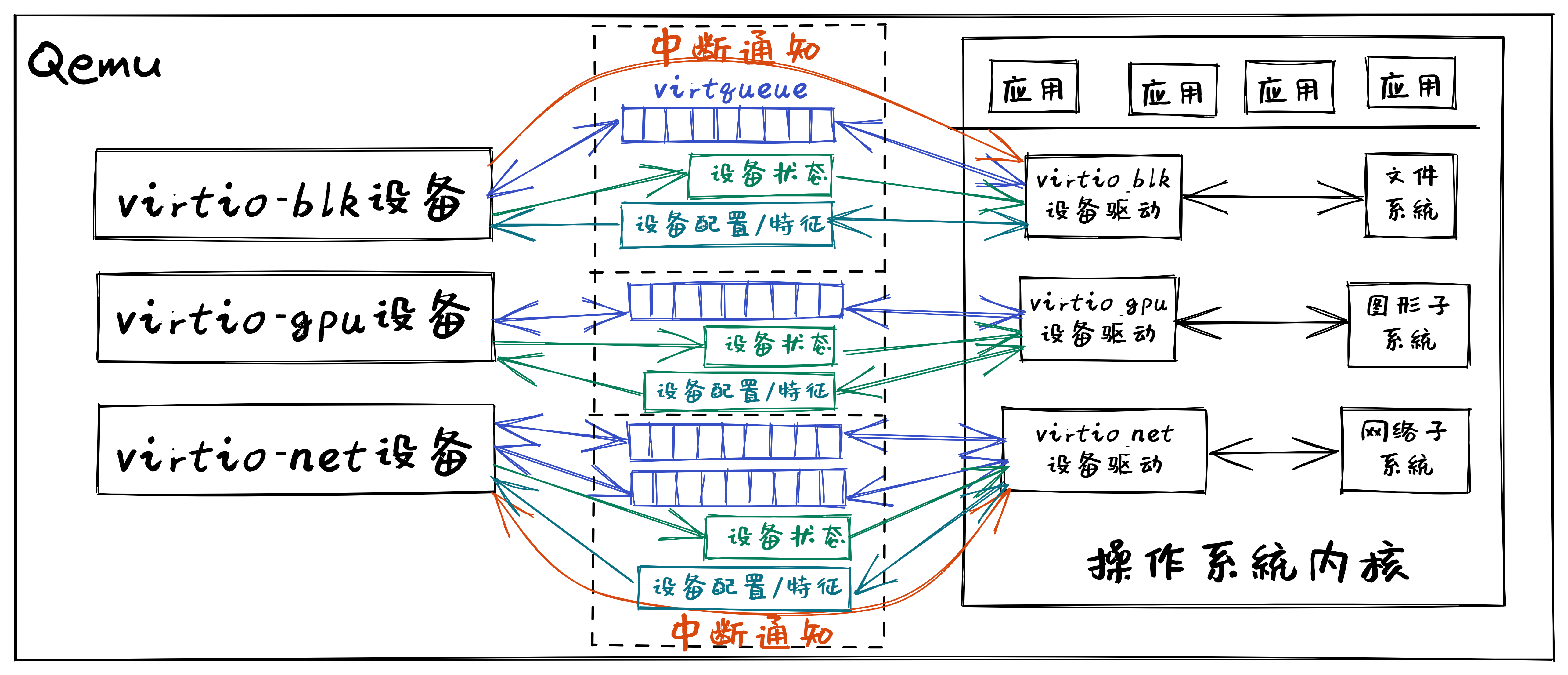
**VirtIO Sound**

**virtio标准**

**前后端**

http://rcore-os.cn/rCore-Tutorial-Book-v3/chapter9/2device-driver-2.html



对于guest os来说，需要一个virtio框架负责访问总线获取host支持的virtio device（后端），并通过接口支持virtio driver（前端）对相应virtio device的查找和绑定。

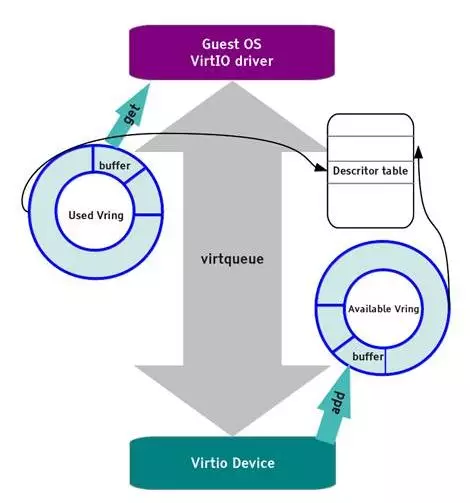
对于guest os中的各种驱动框架，则需要实现一个virtio driver，对于guest来说virtio device的具体实现是不可见的，不需要关心，只需要按照virtio1.2中的标准进行数据交换即可。

**数据交换**

https://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf

前后端交互的数据结构是vring/vqueue，其中有三个主要部分：

* descriptor entry：这里每个entry都记录了一段gpa（Guest Physical Address）内存，available ring和used ring会用index引用这里的entry。
* available ring：待host处理的entry；由guest添加，host读取。
* used ring：host处理完的entry，会记录处理的长度；由host添加，guest读取。



virtqueue有下面几个标准ops：

* add\_buf：向available ring添加gpa buffer；
* kick：添加完buffer后提醒host可以开始进行处理；
* get\_buf：从used ring获取一个处理后的buffer；
* disable\_cb：关闭提醒，不再关心host是否处理了buffer；
* enable\_cb：打开提醒。

|  |
| --- |
| C struct virtqueue\_ops {  int (\*add\_buf)(struct virtqueue \*vq, struct scatterlist sg[],  unsigned int out\_num, unsigned int in\_num, void \*data);  void (\*kick)(struct virtqueue \*vq);  void \*(\*get\_buf)(struct virtqueue \*vq, unsigned int \*len);  void (\*disable\_cb)(struct virtqueue \*vq);  bool (\*enable\_cb)(struct virtqueue \*vq);  }; |

* *关于struct scatterlist：由于gva(Guest Virtual Address)对应的gpa通常是一堆零散的内存，所以Linux中使用“散列表”的数据结构把这些零散的内存串在一起使用。*
* *关于disable/enable cb：一般是在notify\_cb中，伴随着开关中断来使用的。*

**sound device**

https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.html

在virtio标准中，每种设备都定义了一套标准，规定了virtqueue的数量和功能，以及buffer传输的其他细节，这里我们只讨论sound device，下面简略介绍一下。

**4个virtqueue**

* controlq：用于guest向host发送控制信息。
* eventq：host向guest报告事件。
* tx：播放pcm，由guest提供pcm buffer。
* rx：录制pcm，由guest提供空白buffer，host填充。

**3种控制信息**

control message和event 都有公共的header，其成员是一个le32 code用于标识控制信息内容：

* jack：后端外接设备的查询和选择。
* pcm：start/stop/set\_param等播放控制。
* channel map：通道格式的配置。

|  |
| --- |
| C  /\* jack control request types \*/   VIRTIO\_SND\_R\_JACK\_INFO = 1,   VIRTIO\_SND\_R\_JACK\_REMAP,     /\* PCM control request types \*/   VIRTIO\_SND\_R\_PCM\_INFO = 0x0100,   VIRTIO\_SND\_R\_PCM\_SET\_PARAMS,   VIRTIO\_SND\_R\_PCM\_PREPARE,   VIRTIO\_SND\_R\_PCM\_RELEASE,   VIRTIO\_SND\_R\_PCM\_START,   VIRTIO\_SND\_R\_PCM\_STOP,     /\* channel map control request types \*/   VIRTIO\_SND\_R\_CHMAP\_INFO = 0x0200, |

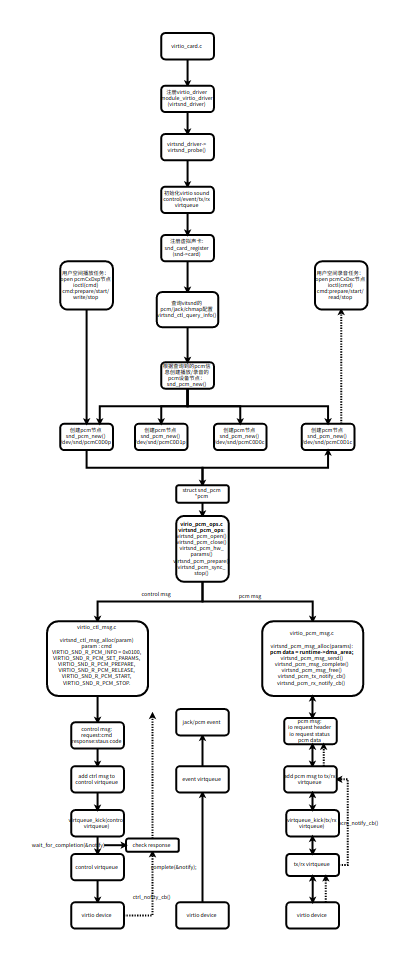
**2种事件**

* jack：连接和断开的通知。
* pcm：播放进度、播放下溢/录制上溢的通知。

|  |
| --- |
| C  /\* jack event types \*/   VIRTIO\_SND\_EVT\_JACK\_CONNECTED = 0x1000,   VIRTIO\_SND\_EVT\_JACK\_DISCONNECTED,     /\* PCM event types \*/   VIRTIO\_SND\_EVT\_PCM\_PERIOD\_ELAPSED = 0x1100,   VIRTIO\_SND\_EVT\_PCM\_XRUN, |

**Linux virtio sound调研**

**Linux virtio sound流程**



https://github.com/torvalds/linux/tree/master/sound/virtio

|  |
| --- |
| Bash // 向上对接alsa virtio\_card.c // 注册为alsa的一张声卡 virtio\_pcm\_ops.c // 注册alsa中的pcm\_ops  // 向下对接virtio virtio\_ctl\_msg.c // controlq的通用操作 virtio\_pcm\_msg.c // txq和rxq的通用操作  virtio\_chmap.c // 通道格式相关控制和事件处理 virtio\_jack.c // 设备选择相关控制和事件处理 virtio\_pcm.c // 数据相关控制和事件处理 |

**初始化virtqueue**

初始化virtio audio需要支持的control/event/tx/rx的virtqueue

|  |
| --- |
| C static vq\_callback\_t \*callbacks[VIRTIO\_SND\_VQ\_MAX] = {  [VIRTIO\_SND\_VQ\_CONTROL] = virtsnd\_ctl\_notify\_cb,  [VIRTIO\_SND\_VQ\_EVENT] = virtsnd\_event\_notify\_cb,  [VIRTIO\_SND\_VQ\_TX] = virtsnd\_pcm\_tx\_notify\_cb,  [VIRTIO\_SND\_VQ\_RX] = virtsnd\_pcm\_rx\_notify\_cb  }; static const char \*names[VIRTIO\_SND\_VQ\_MAX] = {  [VIRTIO\_SND\_VQ\_CONTROL] = "virtsnd-ctl",  [VIRTIO\_SND\_VQ\_EVENT] = "virtsnd-event",  [VIRTIO\_SND\_VQ\_TX] = "virtsnd-tx",  [VIRTIO\_SND\_VQ\_RX] = "virtsnd-rx"  }; rc = virtio\_find\_vqs(vdev, VIRTIO\_SND\_VQ\_MAX, vqs, callbacks, names,  NULL); for (i = 0; i < VIRTIO\_SND\_VQ\_MAX; ++i)  snd->queues[i].vqueue = vqs[i]; |

**get virtio device capability**

设备相关的信息需要从virtio\_device结构体中获取，这里需要virtio框架相应的接口支持。

主要是通过向virtio device发送VIRTIO\_SND\_R\_JACK\_INFO/VIRTIO\_SND\_R\_PCM\_INFO/VIRTIO\_SND\_R\_CHMAP\_INFO来查询virtio\_device所支持的jack/pcm/chmap信息:

|  |
| --- |
| C /\* LE (e.g. modern) Config space accessors. \*/ #define virtio\_cread\_le(vdev, structname, member, ptr)  int virtsnd\_jack\_parse\_cfg(struct virtio\_snd \*snd) {  struct virtio\_device \*vdev = snd->vdev;  struct virtio\_snd\_jack\_info \*info;  u32 i;  int rc;   virtio\_cread\_le(vdev, struct virtio\_snd\_config, jacks, &snd->njacks);  if (!snd->njacks)  return 0;  ...  rc = virtsnd\_ctl\_query\_info(snd, **VIRTIO\_SND\_R\_JACK\_INFO**, 0, snd->njacks,  sizeof(\*info), info); } |

查询pcm返回信息结构如下：

|  |
| --- |
| C struct virtio\_snd\_pcm\_info {   struct virtio\_snd\_info hdr;   le32 features; /\* 1 << VIRTIO\_SND\_PCM\_F\_XXX \*/   le64 formats; /\* 1 << VIRTIO\_SND\_PCM\_FMT\_XXX \*/   le64 rates; /\* 1 << VIRTIO\_SND\_PCM\_RATE\_XXX \*/   u8 direction;   u8 channels\_min;   u8 channels\_max;   u8 padding[5];   }; |

**Virtio audio ctl msg**

主要用于向virtio\_device发送以下jack/pcm/chmap control msg cmd：

|  |
| --- |
| C enum { // **ctrl msg cmd** /\* jack control request types \*/   **VIRTIO\_SND\_R\_JACK\_INFO = 1,**   **VIRTIO\_SND\_R\_JACK\_REMAP,**   /\* PCM control request types \*/   **VIRTIO\_SND\_R\_PCM\_INFO = 0x0100,**   **VIRTIO\_SND\_R\_PCM\_SET\_PARAMS,**   **VIRTIO\_SND\_R\_PCM\_PREPARE,**   **VIRTIO\_SND\_R\_PCM\_RELEASE,**   **VIRTIO\_SND\_R\_PCM\_START,**   **VIRTIO\_SND\_R\_PCM\_STOP,**    /\* channel map control request types \*/   **VIRTIO\_SND\_R\_CHMAP\_INFO = 0x0200,**   ......  **//ctrl msg response**  /\* common status codes \*/ **VIRTIO\_SND\_S\_OK = 0x8000,** **VIRTIO\_SND\_S\_BAD\_MSG,** **VIRTIO\_SND\_S\_NOT\_SUPP,** **VIRTIO\_SND\_S\_IO\_ERR**  }; |

Ctrl msg主要分为request和response两部分：

Request:

|  |
| --- |
| C struct virtio\_snd\_pcm\_hdr {  /\* VIRTIO\_SND\_R\_PCM\_XXX \*/  /\* hdr.code = cmd \*/  struct virtio\_snd\_hdr hdr;  /\* 0 ... virtio\_snd\_config::streams - 1 \*/  \_\_le32 stream\_id; }; |

Response:

|  |
| --- |
| C struct virtio\_snd\_hdr {  /\* code对应以上的**ctrl msg response code VIRTIO\_SND\_S\_OK**   \_\_le32 code; }; |

主要通过以下接口来发送ctrl msg，向control virtqueue发送request，然后等待control virtqueue的nofity cb返回response并检验response status:

|  |
| --- |
| C int virtsnd\_ctl\_msg\_send(struct virtio\_snd \*snd, struct virtio\_snd\_msg \*msg,  struct scatterlist \*out\_sgs,  struct scatterlist \*in\_sgs, bool nowait)  {  ......  //发送request  spin\_lock\_irqsave(&queue->lock, flags);  rc = virtqueue\_add\_sgs(queue->vqueue, psgs, nouts, nins, msg,  GFP\_ATOMIC);  if (!rc) {  notify = virtqueue\_kick\_prepare(queue->vqueue);   list\_add\_tail(&msg->list, &snd->ctl\_msgs);  }  spin\_unlock\_irqrestore(&queue->lock, flags);  if (notify)  virtqueue\_notify(queue->vqueue);  ......  //等待response并检测code,wait completion在control virtqueue notify cb中被通知  rc = wait\_for\_completion\_interruptible\_timeout(&msg->notify, js);  if (rc <= 0) {  if (!rc) {  dev\_err(&vdev->dev,  "control message (0x%08x) timeout\n",  le32\_to\_cpu(request->code));  rc = -ETIMEDOUT;  }   goto on\_exit;  }   switch (le32\_to\_cpu(response->code)) {  case VIRTIO\_SND\_S\_OK:  rc = 0;  break;  case VIRTIO\_SND\_S\_NOT\_SUPP:  rc = -EOPNOTSUPP;  break;  case VIRTIO\_SND\_S\_IO\_ERR:  rc = -EIO;  break;  default:  rc = -EINVAL;  break;  }   ...... } |

**Virtio audio pcm msg**

Virtio audio pcm msg主要是通过tx/rx virtqueue向virtio\_device传输播放/录音的pcm数据，pcm msg的组成由I/O header，pcm data buffer，I/O status三部分组成：

I/O header：stream\_id对应的是当前使用的pcm节点的stream

|  |
| --- |
| C struct virtio\_snd\_pcm\_xfer {   le32 stream\_id;  }; |

Pcm data buffer：录音/播放的pcm数据

I/O status：

|  |
| --- |
| C struct virtio\_snd\_pcm\_status {  //status同ctrl msg的response的code   le32 status;   //对应传输的pcm data的长度，用于换算成时延，用于更新stream runtime audio data信息  le32 latency\_bytes; }; |

在pcm传输前要将传输的pcm数据信息通过**VIRTIO\_SND\_R\_PCM\_SET\_PARAMS**发送给virtio\_device：

|  |
| --- |
| C struct virtio\_snd\_pcm\_set\_params {  struct virtio\_snd\_pcm\_hdr hdr; /\* .code = VIRTIO\_SND\_R\_PCM\_SET\_PARAMS \*/  le32 buffer\_bytes;  le32 period\_bytes;  le32 features; /\* 1 << VIRTIO\_SND\_PCM\_F\_XXX \*/   u8 channels;   u8 format;   u8 rate;   u8 padding;   }; |

每个pcm msg的pcm数据大小为period\_bytes。

**pcm流传输流程**

参考https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html#x1-52900014

A PCM stream has the following command lifecycle:

1. SET PARAMETERS

The driver negotiates the stream parameters (format, transport, etc) with the device.

Possible valid transitions: set parameters, prepare.

1. PREPARE

The device prepares the stream (allocates resources, etc).

Possible valid transitions: set parameters, prepare, start, release.

Output only: the driver transfers data for pre-buffing.

1. START

The device starts the stream (unmute, putting into running state, etc).

Possible valid transitions: stop.

1. The driver transfers data to/from the stream.
2. STOP

The device stops the stream (mute, putting into non-running state, etc).

Possible valid transitions: start, release.

1. RELEASE

The device releases the stream (frees resources, etc).

Possible valid transitions: set parameters, prepare.

**pcm传输**

|  |
| --- |
| C int virtsnd\_pcm\_msg\_send(struct virtio\_pcm\_substream \*vss) {  struct snd\_pcm\_runtime \*runtime = vss->substream->runtime;  struct virtio\_snd \*snd = vss->snd;  struct virtio\_device \*vdev = snd->vdev;  struct virtqueue \*vqueue = virtsnd\_pcm\_queue(vss)->vqueue;  int i;  int n;  bool notify = false;   i = (vss->msg\_last\_enqueued + 1) % runtime->periods;  n = runtime->periods - vss->msg\_count;   // 得到剩余的buffer数量，并全部添加到txq/rxq的available ring上。  for (; n; --n, i = (i + 1) % runtime->periods) {  struct virtio\_pcm\_msg \*msg = vss->msgs[i];  struct scatterlist \*psgs[] = {  &msg->sgs[PCM\_MSG\_SG\_XFER],  &msg->sgs[PCM\_MSG\_SG\_DATA],  &msg->sgs[PCM\_MSG\_SG\_STATUS]  };  int rc;   msg->xfer.stream\_id = cpu\_to\_le32(vss->sid);  memset(&msg->status, 0, sizeof(msg->status));   if (vss->direction == SNDRV\_PCM\_STREAM\_PLAYBACK)  rc = virtqueue\_add\_sgs(vqueue, psgs, 2, 1, msg,  GFP\_ATOMIC);  else  rc = virtqueue\_add\_sgs(vqueue, psgs, 1, 2, msg,  GFP\_ATOMIC);   if (rc) {  dev\_err(&vdev->dev,  "SID %u: failed to send I/O message\n",  vss->sid);  return rc;  }   vss->msg\_last\_enqueued = i;  vss->msg\_count++;  }   // 通知host开始处理数据  if (!(vss->features & (1U << VIRTIO\_SND\_PCM\_F\_MSG\_POLLING)))  notify = virtqueue\_kick\_prepare(vqueue);   if (notify)  virtqueue\_notify(vqueue);   return 0; } |

**pcm传输完成**

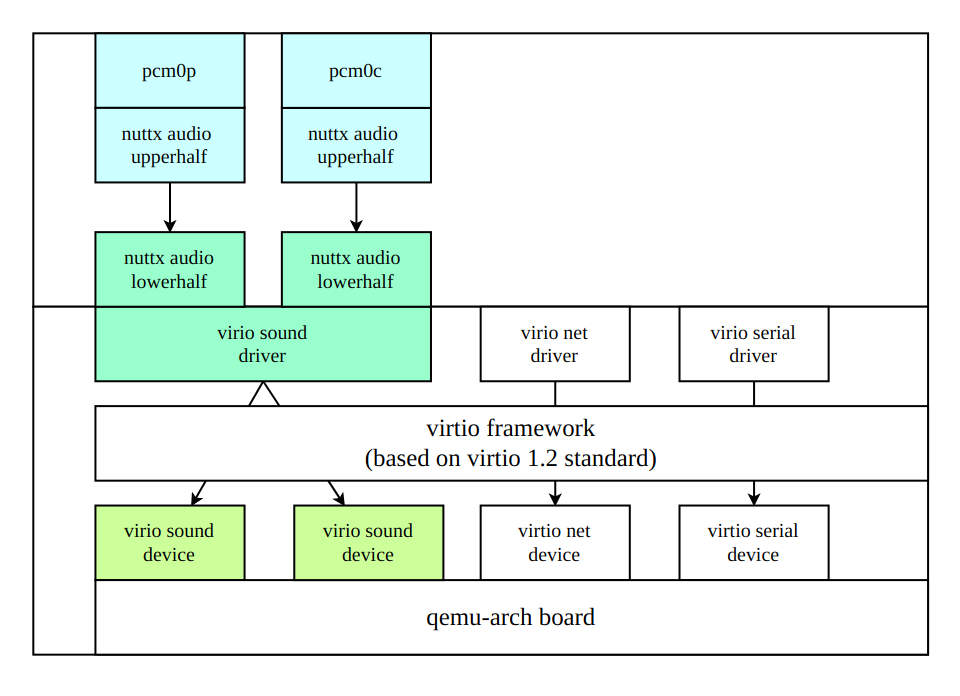
这里是txq和rxq使用的notify\_cb，关中断并处理所有已经播放/录制的buffer，这种写法也可以简单地扩展到eventq的事件处理函数上。

|  |
| --- |
| C static inline void virtsnd\_pcm\_notify\_cb(struct virtio\_snd\_queue \*queue) {  struct virtio\_pcm\_msg \*msg;  u32 written\_bytes;  unsigned long flags;   spin\_lock\_irqsave(&queue->lock, flags);  do {  virtqueue\_disable\_cb(queue->vqueue);  while ((msg = virtqueue\_get\_buf(queue->vqueue, &written\_bytes)))  virtsnd\_pcm\_msg\_complete(msg, written\_bytes);  if (unlikely(virtqueue\_is\_broken(queue->vqueue)))  break;  } while (!virtqueue\_enable\_cb(queue->vqueue));  spin\_unlock\_irqrestore(&queue->lock, flags); } |

**Vela virtio sound**

**架构图**

由guest上的virtio framework访问总线注册virtio设备，并向上提供接口支持virtio驱动与virtio设备绑定在一起；对于guest的各驱动框架来说，则需要基于virtio driver标准实现一个lowerhalf。



**Vela virtio sound driver主要流程**

**1）构建并注册一个virtio driver:**

|  |
| --- |
| C static struct virtio\_driver g\_virtio\_audio\_driver = {  LIST\_INITIAL\_VALUE(g\_virtio\_audio\_driver .node), /\* node \*/  VIRTIO\_ID\_SOUND, /\* device id \*/  virtio\_audio\_probe, /\* probe \*/  virtio\_audio\_remove, /\* remove \*/ }; virtio\_register\_driver(&g\_virtio\_audio\_driver); |

注册完之后会进入到virtio\_audio\_probe()

**2）初始化virtio device**

**初始化control/event/tx/rx virtqueue**

|  |
| --- |
| C static vq\_callback\_t \*callbacks[VIRTIO\_SND\_VQ\_MAX] = {  [VIRTIO\_SND\_VQ\_CONTROL] = virtsnd\_ctl\_notify\_cb,  [VIRTIO\_SND\_VQ\_EVENT] = virtsnd\_event\_notify\_cb,  [VIRTIO\_SND\_VQ\_TX] = virtsnd\_pcm\_tx\_notify\_cb,  [VIRTIO\_SND\_VQ\_RX] = virtsnd\_pcm\_rx\_notify\_cb  }; static const char \*audio\_virqueue\_names[VIRTIO\_SND\_VQ\_MAX] = {  [VIRTIO\_SND\_VQ\_CONTROL] = "virtsnd-ctl",  [VIRTIO\_SND\_VQ\_EVENT] = "virtsnd-event",  [VIRTIO\_SND\_VQ\_TX] = "virtsnd-tx",  [VIRTIO\_SND\_VQ\_RX] = "virtsnd-rx"  }; virtio\_create\_virtqueues(vdev, 0, VIRTIO\_SND\_VQ\_RX, audio\_virqueue\_names,  callbacks); |

**设置virtio device status**

|  |
| --- |
| C virtio\_set\_status(vdev, VIRTIO\_CONFIG\_STATUS\_DRIVER\_OK); |

**填充空buffer到event virtqueue**

参考https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html#x1-52900014

**查询pcm/chmap/jack信息**

|  |
| --- |
| C rc = virtsnd\_ctl\_query\_info(snd, VIRTIO\_SND\_R\_PCM\_INFO, 0,  snd->nsubstreams, sizeof(\*info), info); |

主要是获取virtio\_device支持substream的feature/format/rate/channel/direction等信息，其中direction为VIRTIO\_SND\_D\_OUTPUT就是播放流，为VIRTIO\_SND\_D\_INPUT则是录音流。

**注册Vela声卡**

根据查询到的pcm/chmap substream信息来注册Vela声卡，每个substream注册1个Vela pcm设备，生成播放的/dev/audio/pcmxp或者录音的/dev/audio/pcmxc设备节点，初始化lower half driver的ops并关联uper half driver。

|  |
| --- |
| C #根据substream的direction来注册pcmxp/pcmxc节点 #初始化lower half ops接口集合。 virtio\_audio->ops = virtio\_audio\_ops; audio\_register(devname, virtio\_audio); |

**3）构建并初始化lower half driver的ops virtio\_audio\_ops**

|  |
| --- |
| C static const struct audio\_ops\_s virtio\_audio\_ops= {  virtio\_aud\_getcaps, /\* getcaps \*/  virtio\_aud\_configure, /\* configure \*/  virtio\_aud\_shutdown, /\* shutdown \*/  virtio\_aud\_start, /\* start \*/ #ifndef CONFIG\_AUDIO\_EXCLUDE\_STOP  virtio\_aud\_stop, /\* stop \*/ #endif #ifndef CONFIG\_AUDIO\_EXCLUDE\_PAUSE\_RESUME  virtio\_aud\_pause, /\* pause \*/  virtio\_aud\_resume, /\* resume \*/ #endif  NULL, /\* allocbuffer \*/  NULL, /\* freebuffer \*/  virtio\_aud\_enqueuebuffer, /\* enqueue\_buffer \*/  virtio\_aud\_cancelbuffer, /\* cancel\_buffer \*/  virtio\_aud\_ioctl, /\* ioctl \*/  virtio\_aud\_read, /\* read \*/  virtio\_aud\_write, /\* write \*/  virtio\_aud\_reserve, /\* reserve \*/  virtio\_aud\_release /\* release \*/ }; |

virtio\_audio\_ops接口集合是重点，对标Linux virtio sound driver当中的virtsnd\_pcm\_ops，

比如virtio\_aud\_getcaps可返回查询到的pcm/chmap/jack信息。

virtio\_aud\_configure实现stream流的param配置，参考virtsnd\_pcm\_ops的hw\_params()接口。

virtio\_aud\_start用于通过control virtqueue向virtio\_device发送VIRTIO\_SND\_R\_PCM\_START cmd。

**4）实现virtio audio ctl msg接口**

用于查询/播放/录音流程中通过control virtqueue向virtio\_device发送对应的control request cmd。

其实现可参考linux virtio driver virtio\_ctl\_msg.c

**5）实现virtio audio pcm msg接口**

用于播放/录音流程中pcm数据的交互，比如播放时从将enqueuebuffer的pcm数据通过tx virtqueue发送给virtio device，录音时从rx virtqueue中获取pcm数据填充到enqueuebuffer中。

其实现可参考linux virtio driver virtio\_pcm\_msg.c

思考

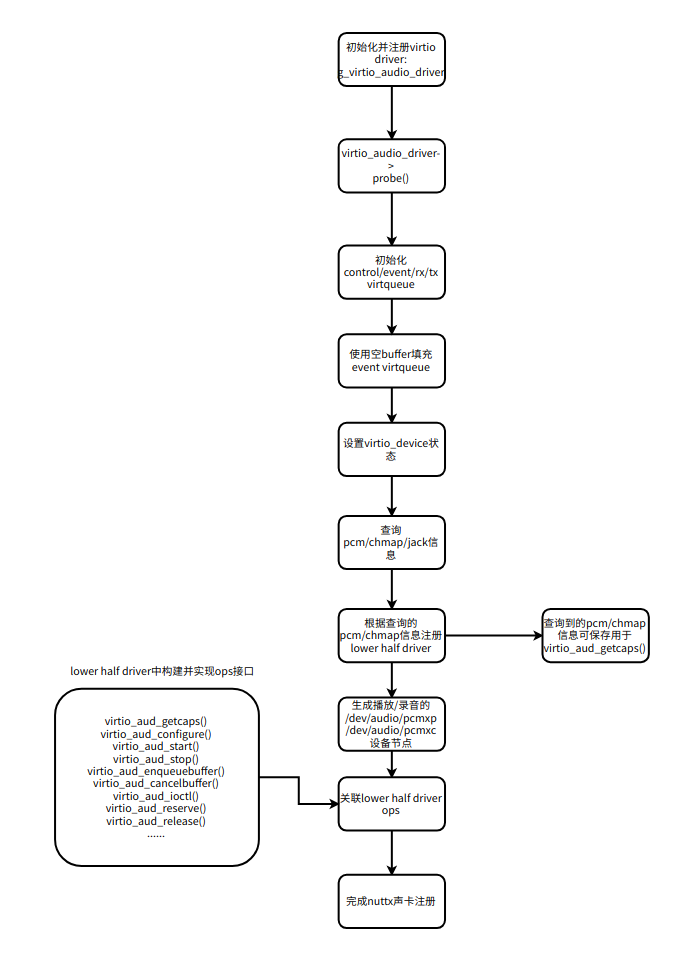
是否可以考虑实现一个组合节点的lower half driver：

virtio audio ctl msg接口和virtio audio pcm msg接口关联到lower dev，与lower half driver组合起来：

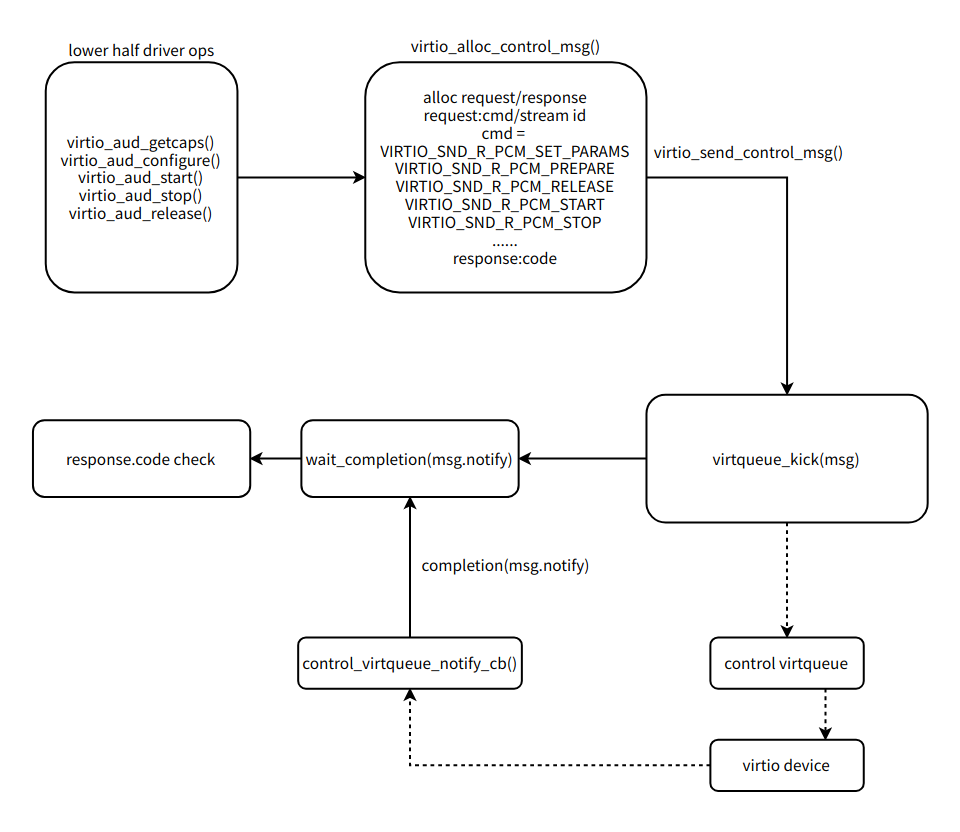
|  |
| --- |
| C //Init audio ctrl msg //Init audio pcm msg // 关联 audio ctrl msg、audio pcm msg、lower ops 到lower dev struct audio\_lowerhalf\_s \*virtio\_ctrl\_and\_pcm = virtio\_ctrl\_io\_initialize(virtio\_audio\_ctrl, virtio\_audio\_pcm, &g\_virtio\_audio\_ctrl\_pcm.lower); // wrapper a new lower interface as a PCM devices  struct audio\_lowerhalf\_s \*vitio\_audio\_pcm = pcm\_decode\_initialize(virtio\_ctrl\_and\_pcm); // 注册声卡 audio\_register(devname, vitio\_audio\_pcm); |

**流程图**

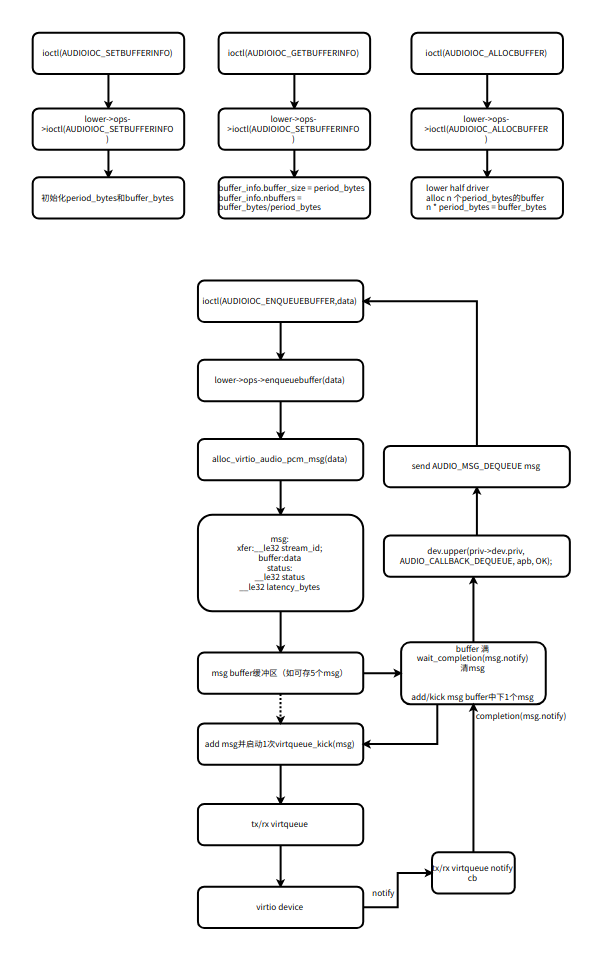
系统启动过程中注册声卡信息



实现audio ctrl msg接口用于控制音频流



实现audio pcm msg传输接口用于录音/播放的pcm数据传输。



QEMU上的配置和使用参考[Audio](https://xiaomi.f.mioffice.cn/wiki/wikk4fA5TJrYWMcQqbe7qYMEJ6b)，驱动架构参考[Audio](https://xiaomi.f.mioffice.cn/wiki/wikk4Hikak0x5SnQNPNmUkaEx8f)。