**VirtIO Serial**

1. **初始化**

本节说明virtio-serial如何进行初始化，如何与对应的virtio device进行匹配

1. 驱动需要实现一个virtio\_register\_serial\_driver()函数，在virtio\_register\_drivers()函数中调用，如下：

|  |
| --- |
| C int virtio\_register\_drivers(void) {  int ret;  #ifdef CONFIG\_VIRTIO\_DRIVERS\_SERIAL  ret = virtio\_register\_serial\_driver();  if (ret < 0)  {  vrterr("virtio\_register\_serial\_driver failed, ret=%d\n", ret);  } #endif   return ret; } |

1. 驱动在virtio\_register\_serial\_driver()中调用virtio\_driver\_register()将自己的驱动注册到virtio总线中去；
2. Virtio Device在初始化时，会将设备注册到virtio总线中去，触发virtio\_serial\_probe函数执行驱动的初始化，初始化完后会注册/dev/ttyVx到Vela的VFS中去；

2. **数据结构**

每一个Virtio驱动中有一个自己维护的数据接口，其中包含了关联Vela驱动框架和VirtIO的全部信息，具体的描述如下：

|  |
| --- |
| C struct virtio\_serial\_priv\_s {  /\* Virtio设备的指针，每一个driver都必须包含一个 \*/  FAR struct virtio\_device \***vdev**;   /\* 对接NuttX Uart驱动框架需要的数据接口 \*/  FAR struct uart\_dev\_s udev;  /\* 设备的名称，串口会初始化成/dev/ttyXXX \*/  char name[NAME\_MAX]; }; |

3. **Functions**

**virtio\_register\_serial\_driver()**

直接调用virtio\_driver\_register()将virtio driver注册到virtio总线

|  |
| --- |
| C /\* serial driver全局结构 \*/ static struct virtio\_driver g\_virtio\_serial\_driver = {  .device = VIRTIO\_ID\_CONSOLE, /\* 定义了DeviceID，根据实际设备定义 \*/    /\* Virtio Driver的probe函数，在和对应的Virtio Device匹配上时，  \* Virtio总线会回调该接口进行驱动相关的初始化工作  \*/  .probe = virtio\_serial\_driver\_probe,    /\* Virtio Driver的remove函数，在对应的Virtio Device unregister或者  \* driver主动unregister时，会回调该函数来回收driver的资源  \*/  .remove = virtio\_serial\_driver\_remove, };  int virtio\_register\_serial\_driver(void) {  return virtio\_register\_driver(&g\_virtio\_serial\_driver); } |

**virtio\_serial\_driver\_probe()**

probe中主要就是对驱动进行初始化，大致可以分为三部分：

1. 申请所需要的内存；
2. virtio\_serial\_init()初始化virtio serial，其中virtio部分的初始化，需要按照virtio spec来实现，如virtqueue的个数，各自的作用，以及feature bit和configuration都在spec中比较详细的描述；
3. Vela提供的驱动注册接口uart\_register()注册驱动；

|  |
| --- |
| C static int virtio\_serial\_probe(FAR struct virtio\_device \*vdev) {  FAR struct virtio\_serial\_priv\_s \*priv;  int ret;   /\* Alloc the virtio serial driver and uart buffer \*/   priv = kmm\_zalloc(sizeof(\*priv));  if (priv == NULL)  {  vrterr("No enough memory\n");  return -ENOMEM;  }   /\* 初始化串口相关结构，并创建virtio queue \*/   ret = virtio\_serial\_init(priv, vdev);  if (ret < 0)  {  vrterr("virtio\_serial\_init failed, ret=%d\n", ret);  goto err\_with\_priv;  }   /\* Uart driver register \*/   snprintf(priv->name, NAME\_MAX, "/dev/virttty%d", g\_virtio\_serial\_idx);  ret = uart\_register(priv->name, &priv->udev);  if (ret < 0)  {  vrterr("uart\_register failed, ret=%d\n", ret);  goto err\_with\_init;  }   g\_virtio\_serial\_idx++;  return ret;  err\_with\_init:  virtio\_serial\_uninit(priv); err\_with\_priv:  kmm\_free(priv);  return ret; }  static int virtio\_serial\_init(FAR struct virtio\_serial\_priv\_s \*priv,  FAR struct virtio\_device \*vdev) {  FAR const char \*vqnames[VIRTIO\_SERIAL\_NUM];  vq\_callback callbacks[VIRTIO\_SERIAL\_NUM];  FAR struct uart\_dev\_s \*udev;  int ret;   priv->vdev = vdev;  vdev->priv = priv;   /\* Uart device buffer and ops init   \* 申请串口的buffer  \*/   udev = &priv->udev;  udev->priv = priv;  udev->ops = &g\_virtio\_serial\_ops;  udev->recv.size = CONFIG\_DRIVERS\_VIRTIO\_SERIAL\_BUFSIZE;  udev->recv.buffer = virtio\_alloc\_buf(vdev, udev->recv.size, 16);  if (udev->recv.buffer == NULL)  {  vrterr("No enough memory\n");  return -ENOMEM;  }   udev->xmit.size = CONFIG\_DRIVERS\_VIRTIO\_SERIAL\_BUFSIZE;  udev->xmit.buffer = virtio\_alloc\_buf(vdev, udev->xmit.size, 16);  if (udev->xmit.buffer == NULL)  {  vrterr("No enough memory\n");  ret = -ENOMEM;  goto err\_with\_recv;  }   /\* Initialize the virtio device，调用virtio接口进行设备初始化，  \* 创建virtqueue  \*/   virtio\_set\_status(vdev, VIRTIO\_CONFIG\_STATUS\_DRIVER);  virtio\_set\_features(vdev, 0);  virtio\_set\_status(vdev, VIRTIO\_CONFIG\_FEATURES\_OK);   vqnames[VIRTIO\_SERIAL\_RX] = "virtio\_serial\_rx";  vqnames[VIRTIO\_SERIAL\_TX] = "virtio\_serial\_tx";  callbacks[VIRTIO\_SERIAL\_RX] = virtio\_serial\_rxready;  callbacks[VIRTIO\_SERIAL\_TX] = virtio\_serial\_txdone;  ret = virtio\_create\_virtqueues(vdev, 0, VIRTIO\_SERIAL\_NUM, vqnames,  callbacks);  if (ret < 0)  {  vrterr("virtio\_device\_create\_virtqueue failed, ret=%d\n", ret);  goto err\_with\_xmit;  }   virtio\_set\_status(vdev, VIRTIO\_CONFIG\_STATUS\_DRIVER\_OK);  return OK;  err\_with\_xmit:  virtio\_free\_buf(vdev, udev->xmit.buffer); err\_with\_recv:  virtio\_free\_buf(vdev, udev->recv.buffer);  virtio\_reset\_device(vdev);  return ret; } |

**g\_virt\_serial\_ops**

对于Vela的串口驱动框架在DMA模式下需要实现的operation：

|  |
| --- |
| C static const struct uart\_ops\_s g\_virtio\_serial\_ops = {  virtio\_serial\_setup, /\* setup \*/  virtio\_serial\_shutdown, /\* shutdown \*/  virtio\_serial\_attach, /\* attach \*/  virtio\_serial\_detach, /\* detach \*/  virtio\_serial\_ioctl, /\* ioctl \*/  NULL, /\* receive \*/  virtio\_serial\_rxint, /\* rxint \*/  virtio\_serial\_rxavailable, /\* rxavailable \*/ #ifdef CONFIG\_SERIAL\_IFLOWCONTROL  NULL, /\* rxflowcontrol \*/ #endif  virtio\_serial\_dmasend, /\* dmasend \*/  virtio\_serial\_dmareceive, /\* dmareceive \*/  virtio\_serial\_dmarxfree, /\* dmarxfree \*/  virtio\_serial\_dmatxavail, /\* dmatxavail \*/  virtio\_serial\_send, /\* send \*/  virtio\_serial\_txint, /\* txint \*/  virtio\_serial\_txready, /\* txready \*/  virtio\_serial\_txempty, /\* txempty \*/ }; |

核心的有两个：virtio\_serial\_dmasend()和virtio\_serial\_dmareceive()

以及关联的两个中断处理函数：virtio\_serial\_txdone()和virtio\_serial\_rxready()

**void virtio\_serial\_dmasend(struct uart\_dev\_s \*dev)**

描述：将串口驱动需要发送的数据（dev->dmatx中的buffer和nbuffer）填充到tx virtqueue中，传给device端

|  |
| --- |
| C static void virtio\_serial\_dmasend(FAR struct uart\_dev\_s \*dev) {  FAR struct virtio\_serial\_priv\_s \*priv = dev->priv;  FAR struct virtqueue \*vq = priv->vdev->vrings\_info[VIRTIO\_SERIAL\_TX].vq;  FAR struct uart\_dmaxfer\_s \*xfer = &dev->dmatx;  struct virtqueue\_buf vb[2];  uintptr\_t len;  int num = 0;   /\* Get the total send length \*/   len = xfer->length + xfer->nlength;  if (len == 0)  {  return;  }   /\* Set the virtqueue buffer \*/   if (xfer->length != 0)  {  vb[num].buf = xfer->buffer;  vb[num].len = xfer->length;  num++;  }   if (xfer->nlength != 0)  {  vb[num].buf = xfer->nbuffer;  vb[num].len = xfer->nlength;  num++;  }   /\* Add buffer to TX virtiqueue and notify the other size \*/   virtqueue\_add\_buffer(vq, vb, num, 0, (FAR void \*)len); } |

**bool virtio\_serial\_txdone(FAR struct virtqueue \*vq)**

描述：串口的发送完成中断函数，在tx virtqueue的buffer被device端取走后，device收到数据，会返回buffer到tx virtqueue中，然后通知driver端，driver端会回调该函数，表示发送完成。该函数中，会获取发送的长度，然后调用uart\_xmitchars\_done()对已发送的buffer进行标记，然后调用uart\_xmitchars\_dma()准备下一次的发送。

|  |
| --- |
| C static void virtio\_serial\_txdone(FAR struct virtqueue \*vq) {  FAR struct virtio\_serial\_priv\_s \*priv = vq->vq\_dev->priv;  uintptr\_t len;   /\* Cookie not NULL indicate the tx completed \*/   len = (uintptr\_t)virtqueue\_get\_buffer(vq, NULL, NULL);  if (len == 0)  {  return;  }   /\* Call uart\_xmitchars\_done to notify the upperhalf \*/   priv->udev.dmatx.nbytes = len;  uart\_xmitchars\_done(&priv->udev);  uart\_xmitchars\_dma(&priv->udev); } |

**int virtio\_serial\_dmareceive(struct uart\_dev\_s \*dev)**

描述：将空闲的buffer填充到rx virtqueue中，发送给device端，device端在有数据发送时，会将数据填充好后返回给driver

|  |
| --- |
| C static void virtio\_serial\_dmareceive(FAR struct uart\_dev\_s \*dev) {  FAR struct virtio\_serial\_priv\_s \*priv = dev->priv;  FAR struct virtqueue \*vq = priv->vdev->vrings\_info[VIRTIO\_SERIAL\_RX].vq;  FAR struct uart\_dmaxfer\_s \*xfer = &dev->dmarx;  struct virtqueue\_buf vb[2];  int num = 0;   if (xfer->length != 0)  {  vb[num].buf = xfer->buffer;  vb[num].len = xfer->length;  num++;  }   if (xfer->nlength != 0)  {  vb[num].buf = xfer->nbuffer;  vb[num].len = xfer->nlength;  num++;  }   /\* Add buffer to the RX virtqueue and notify the device side \*/   virtqueue\_add\_buffer(vq, vb, 0, num, xfer); } |

**bool virtio\_serial\_rxready(FAR struct virtqueue \*vq)**

描述：串口的接收ready中断，在device填充好buffer返回给driver后，该函数会被回调，表示收到了数据。driver会获取接收到的buffer的长度，调用uart\_recvchars\_done()将数据真正的接收，然后调用uart\_recvchars\_dma()准备下一次的接收；

|  |
| --- |
| C static void virtio\_serial\_rxready(FAR struct virtqueue \*vq) {  FAR struct virtio\_serial\_priv\_s \*priv = vq->vq\_dev->priv;  FAR struct uart\_dmaxfer\_s \*xfer;  uint32\_t len;   /\* Received some data, call uart\_recvchars\_done() \*/   xfer = virtqueue\_get\_buffer(vq, &len, NULL);  if (xfer == NULL)  {  return;  }   xfer->nbytes = len;  uart\_recvchars\_done(&priv->udev);  uart\_recvchars\_dma(&priv->udev); } |

具体使用方法参考[VirtIO-Serial](https://xiaomi.f.mioffice.cn/wiki/wikk4L81JT75lDcNveSsFApThjd)，rpmsg类似实现参考[rpmsg uart](https://xiaomi.f.mioffice.cn/wiki/wikk4X6PPxXy05jjjDE16diGROd)，serial驱动参考[Serial](https://xiaomi.f.mioffice.cn/wiki/wikk4B1t1XveFfA9EoYrPfNvkxc)。