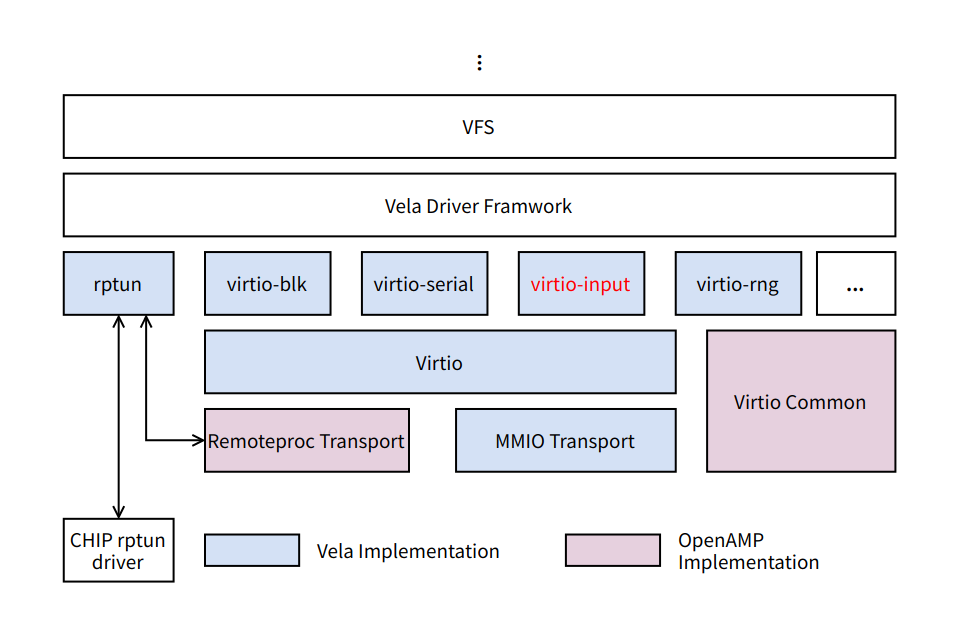
**VirtIO Input**

1. **背景概要**

在 Vela 上实现了基于 [VirtIO 1.2标准](https://docs.oasis-open.org/virtio/virtio/v1.2/cs01/virtio-v1.2-cs01.html#x1-3650007)的通过 MMIO 通信的 VirtIO-Input 驱动，实现将 input 设备的事件发送到 Vela 上的功能。

2. **框架结构**



上图为 Vela Virtio 框架图，大致可以分为三部分：

1. 驱动层：将 virtio 和 Vela 驱动框架进行对接，驱动层调用 virtio 提供的统一接口进行设备的初始化和数据的交互；
2. Virtio 层：提供统一的接口给 Driver，实现 Driver/Device的注册，卸载和匹配机制；
3. 传输层：提供不同传输层的支持；

3. **实现细节**

3.1 **工作流程**

3.1.1 **初始化与注册**

1. virtio\_register\_input\_driver - 注册g\_virtio\_input\_driver
2. virtio\_input\_probe和virtio\_input\_remove - 由 VirtIO 框架调用，实际的初始化/移除入口
3. 在 virtio\_input\_probe中注册事件回调函数 virtio\_input\_recv\_events，并且按照设备的类型(鼠标、键盘、touchpad)，注册 Vela input 驱动

3.1.2 **接收 virtio input 事件**

virtio\_input\_recv\_events 来自一个中断，在这里创建异步工作队列 virtio\_work\_queue 进行 input 事件处理。

3.1.3 **向 Vela 发送 input 事件**

在异步工作队列 virtio\_work\_queue 的 virtio\_input\_worker 函数中，使用 virtqueue\_get\_buffer 读取 input 事件，然后根据设备的类型(鼠标、键盘、touchpad)，使用 virtio\_send\_keyboard\_event virtio\_send\_mouse\_event virtio\_send\_touch\_event 发送到 Vela 中。

3.2 **数据结构**

|  |
| --- |
| C struct virtio\_input\_event {  uint16\_t type;  uint16\_t code;  uint32\_t value; }  typedef void (\*virtio\_send\_event\_handler)(struct virtio\_input\_priv \*,  struct virtio\_input\_event \*);  struct virtio\_input\_priv {  FAR struct virtio\_device \*vdev;  char name[NAME\_MAX]; /\* Device name \*/  struct virtio\_input\_event evt[VIRTIO\_INPUT\_EVT\_NUM];  size\_t evtnum; /\* Input event number \*/  struct work\_s work; /\* Supports the interrupt handling "bottom half" \*/  virtio\_send\_event\_handler eventhandler;   union  {  struct mouse\_lowerhalf\_s mouselower; /\* Mouse device lowerhalf instance \*/  struct keyboard\_lowerhalf\_s keyboardlower; /\* Keyboard device lowerhalf instance \*/  struct touch\_lowerhalf\_s touchlower; /\* Touchpad device lowerhalf instance \*/  };   union  {  struct mouse\_report\_s mousesample; /\* Mouse event \*/  struct keyboard\_event\_s keyboardsample; /\* Keyboard event \*/  struct touch\_sample\_s touchsample; /\* Touchpad event \*/  }; }; |

3.3 **重点函数**

3.3.1 **virtio\_register\_input\_driver**

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

3.3.2 **virtio\_input\_probe**

|  |
| --- |
| C static int virtio\_input\_probe(FAR struct virtio\_device \*vdev) {  FAR struct virtio\_input\_priv \*priv;  FAR const char \*vqnames[VIRTIO\_INPUT\_NUM];  vq\_callback callbacks[VIRTIO\_INPUT\_NUM];  int ret;   priv = kmm\_zalloc(sizeof(\*priv));  if (priv == NULL)  {  vrterr("No enough memory\n");  return -ENOMEM;  }   priv->vdev = vdev;  vdev->priv = priv;   /\* Initialize the virtio device \*/   virtio\_set\_status(vdev, VIRTIO\_CONFIG\_STATUS\_DRIVER);  virtio\_set\_features(vdev, 0);  virtio\_set\_status(vdev, VIRTIO\_CONFIG\_FEATURES\_OK);   vqnames[VIRTIO\_INPUT\_EVENT] = "virtio\_input\_event";  callbacks[VIRTIO\_INPUT\_EVENT] = virtio\_input\_recv\_events;  ret = virtio\_create\_virtqueues(vdev, 0, VIRTIO\_INPUT\_NUM, vqnames,  callbacks);  if (ret < 0)  {  vrterr("virtio\_device\_create\_virtqueue failed, ret=%d\n", ret);  virtio\_reset\_device(vdev);  kmm\_free(priv);  return ret;  }   virtio\_set\_status(vdev, VIRTIO\_CONFIG\_STATUS\_DRIVER\_OK);  virtqueue\_enable\_cb(vdev->vrings\_info[VIRTIO\_INPUT\_EVENT].vq);  priv->evtnum = MIN(vdev->vrings\_info[VIRTIO\_INPUT\_EVENT].info.num\_descs,  VIRTIO\_INPUT\_EVT\_NUM);   /\* register lower half drivers \*/   virtio\_input\_register(priv);   virtio\_input\_fill\_event(priv);   return ret; } |

3.3.3 **virtio\_input\_remove**

|  |
| --- |
| C static void virtio\_input\_remove(FAR struct virtio\_device \*vdev) {  FAR struct virtio\_input\_priv \*priv = vdev->priv;   if (priv->eventhandler == virtio\_input\_send\_keyboard\_event)  {  keyboard\_unregister(&(priv->keyboardlower), priv->name);  }  else if (priv->eventhandler == virtio\_input\_send\_mouse\_event)  {  mouse\_unregister(&(priv->mouselower), priv->name);  }  else if (priv->eventhandler == virtio\_input\_send\_touch\_event)  {  touch\_unregister(&(priv->touchlower), priv->name);  }   virtio\_reset\_device(vdev);  virtio\_delete\_virtqueues(vdev);  kmm\_free(priv); } |

3.3.4 **virtio\_input\_register**

|  |
| --- |
| C static void virtio\_input\_register(FAR struct virtio\_input\_priv \*priv) {  if (virtio\_input\_select\_cfg(priv, VIRTIO\_INPUT\_CFG\_EV\_BITS, EV\_ABS))  {  priv->touchlower.maxpoint = 1;  snprintf(priv->name, NAME\_MAX, "/dev/virtinput%d",  g\_virtio\_touch\_idx++);  touch\_register(&(priv->touchlower),  priv->name,  priv->evtnum);  priv->eventhandler = virtio\_input\_send\_touch\_event;  }  else if (virtio\_input\_select\_cfg(priv, VIRTIO\_INPUT\_CFG\_EV\_BITS, EV\_REL))  {  snprintf(priv->name, NAME\_MAX, "/dev/virtmouse%d",  g\_virtio\_mouse\_idx++);  mouse\_register(&(priv->mouselower),  priv->name,  priv->evtnum);  priv->eventhandler = virtio\_input\_send\_mouse\_event;  }  else if (virtio\_input\_select\_cfg(priv, VIRTIO\_INPUT\_CFG\_EV\_BITS, EV\_KEY))  {  snprintf(priv->name, NAME\_MAX, "/dev/virtkbd%d",  g\_virtio\_keyboard\_idx++);  keyboard\_register(&(priv->keyboardlower),  priv->name,  priv->evtnum);  priv->eventhandler = virtio\_input\_send\_keyboard\_event;  } } |

3.3.5 **virtio\_input\_recv\_events**

|  |
| --- |
| C static void virtio\_input\_recv\_events(FAR struct virtqueue \*vq) {  FAR struct virtio\_input\_priv \*priv = vq->vq\_dev->priv;  int ret;   ret = work\_queue(HPWORK, &priv->work, virtio\_input\_worker, priv, 0);  if (ret != 0)  {  vrterr("ERROR: Failed to queue work: %d\n", ret);  }   return; } |

3.3.6 **virtio\_input\_worker**

|  |
| --- |
| C static void virtio\_input\_worker(FAR void \*arg) {  FAR struct virtio\_input\_priv \*priv = (FAR struct virtio\_input\_priv \*)arg;  FAR struct virtqueue \*vq = priv->vdev->vrings\_info[VIRTIO\_INPUT\_EVENT].vq;  FAR struct virtio\_input\_event \*evt;  FAR struct virtqueue\_buf vb;  uint32\_t len;   while ((evt = (FAR struct virtio\_input\_event \*)  virtqueue\_get\_buffer(vq, &len, NULL)) != NULL)  {  vrtinfo("virtio\_input\_worker (type,code,value) - (%d,%d,%d).\n",  evt->type, evt->code, evt->value);   priv->eventhandler(priv, evt);   vb.buf = evt;  vb.len = len;  virtqueue\_add\_buffer(vq, &vb, 0, 1, vb.buf);  }   virtqueue\_kick(vq); } |

3.3.7 **virtio\_input\_send\_keyboard\_event**

|  |
| --- |
| C static void virtio\_input\_send\_keyboard\_event(FAR struct virtio\_input\_priv \*priv,  FAR struct virtio\_input\_event \*event) {  if (event->type == EV\_KEY)  {  priv->keyboardsample.code =  virtio\_input\_translate\_keycode(event->code);  priv->keyboardsample.type = event->value;  }  else if (event->type == EV\_SYN && event->code == SYN\_REPORT)  {  keyboard\_event(&(priv->keyboardlower),  priv->keyboardsample.code,  priv->keyboardsample.type);  memset(&priv->keyboardsample, 0, sizeof(priv->keyboardsample));  } } |

3.3.8 **virtio\_input\_send\_mouse\_event**

|  |
| --- |
| C static void virtio\_input\_send\_mouse\_event(FAR struct virtio\_input\_priv \*priv,  FAR struct virtio\_input\_event \*event) {  if (event->type == EV\_REL)  {  switch (event->code)  {  case REL\_X:  priv->mousesample.x = event->value;  break;   case REL\_Y:  priv->mousesample.y = event->value;  break;   #ifdef CONFIG\_INPUT\_MOUSE\_WHEEL  case REL\_WHEEL:  priv->mousesample.wheel = event->value;  break;  #endif  }  }  else if (event->type == EV\_KEY)  {  switch (event->code)  {  case BTN\_LEFT:  if (event->value)  {  priv->mousesample.buttons |= MOUSE\_BUTTON\_1;  }  break;   case BTN\_RIGHT:  if (event->value)  {  priv->mousesample.buttons |= MOUSE\_BUTTON\_2;  }  break;   case BTN\_MIDDLE:  if (event->value)  {  priv->mousesample.buttons |= MOUSE\_BUTTON\_3;  }  break;  }  }  else if (event->type == EV\_SYN && event->code == SYN\_REPORT)  {  mouse\_event(priv->mouselower.priv, &priv->mousesample);  memset(&priv->mousesample, 0, sizeof(priv->mousesample));  } } |

3.3.9 **virtio\_input\_send\_touch\_event**

|  |
| --- |
| C static void virtio\_input\_send\_touch\_event(FAR struct virtio\_input\_priv \*priv,  FAR struct virtio\_input\_event \*event) {  if (event->type == EV\_ABS)  {  switch (event->code)  {  case ABS\_PRESSURE:  priv->touchsample.point[0].flags |= TOUCH\_PRESSURE\_VALID;  priv->touchsample.point[0].pressure = event->value;  break;   case ABS\_X:  priv->touchsample.point[0].flags |= TOUCH\_POS\_VALID;  priv->touchsample.point[0].x = event->value;  break;   case ABS\_Y:  priv->touchsample.point[0].flags |= TOUCH\_POS\_VALID;  priv->touchsample.point[0].y = event->value;  break;  }  }  else if (event->type == EV\_SYN && event->code == SYN\_REPORT)  {  priv->touchsample.npoints = 1;  priv->touchsample.point[0].timestamp = touch\_get\_time();   touch\_event(priv->touchlower.priv, &priv->touchsample);  memset(&priv->touchsample, 0, sizeof(priv->touchsample));  } } |

在QEMU上的配置和使用参考[Input](https://xiaomi.f.mioffice.cn/wiki/wikk4azzFG85ez1QjawKdgOQTsc)，rpmsg类似实现参考[Uinput](https://xiaomi.f.mioffice.cn/wiki/wikk4xILnAysSDLMya58pLdrtKg)，驱动架构参考[Input驱动](https://xiaomi.f.mioffice.cn/wiki/wikk401eCI9HXHJxQfmYhLRpmrf) 。