DPDK KNI 与协议栈

Dpdk 提供三组数据交互方式，igb_uio，VFIO，KNI。

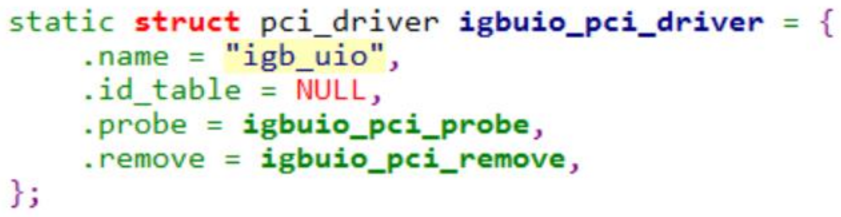
Igb_uio

Igb_uio 分三个部分，igb_uio 内核模块，内核 uio 框架，uio 用户接口

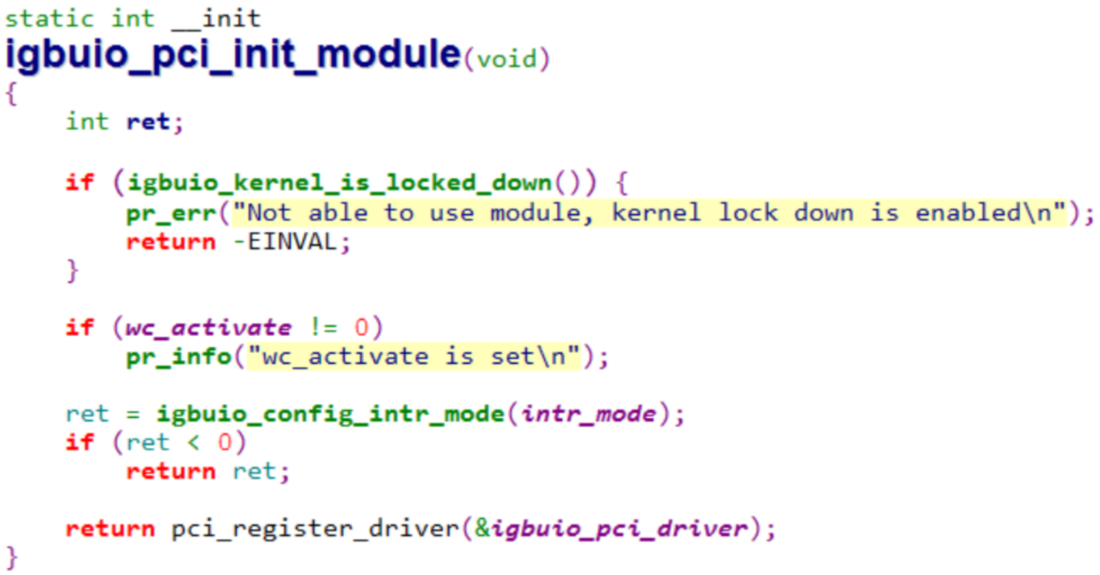
igb_uio 内核模块

igb_uio 驱动主要做的就是注册一个 pci 设备，在 DPDK 工具 dpdk_nic_bind.py 绑定 NIC 的时候这个驱动会 probe 到这个设备，进行相关配置。之后会注册一个 UIO 设备，probe 函数会将记录设备的资源比如 PCI 设备 BAR 空间的物理地址、大小等信息记录下来传给用户态。注册的 UIO 设备名为 igb_uio，内核态中断处理函数为 igbuio_pci_irqhandler，中断控制函数 igbuio_pci_irqcontrol。注册的主要工作如下：

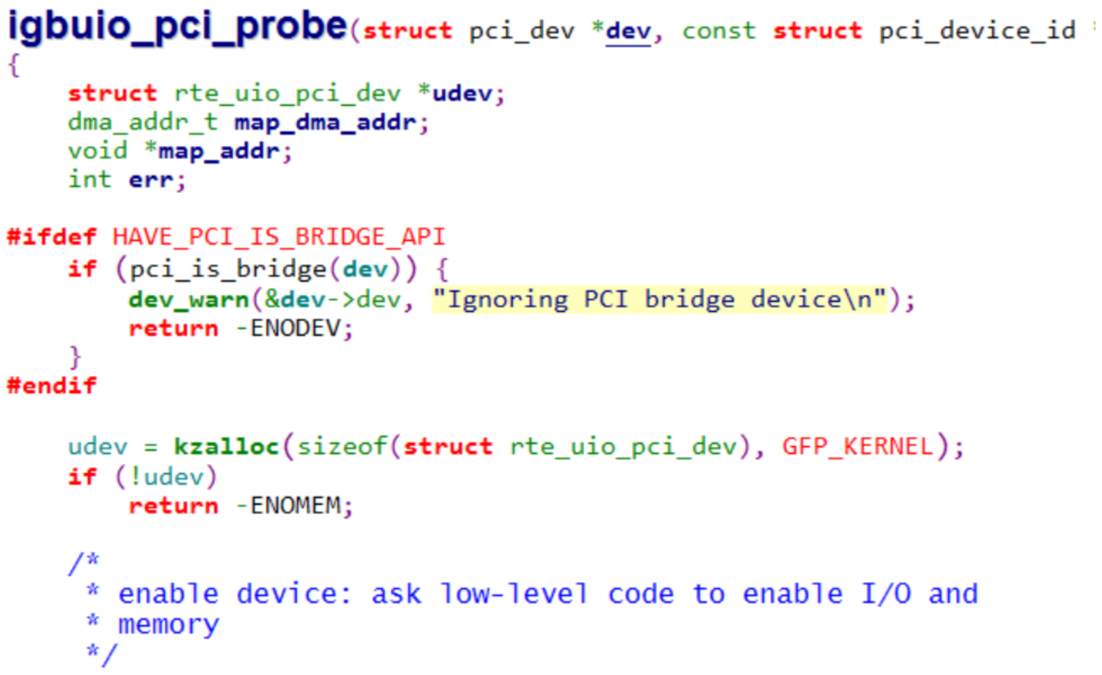
1. 初始化 uio_device 结构体指针，主要包括等待队列 wait、中断事件计数 event、次设备号 minor 等。
2. 在/dev 目录下创建了一个 uio 设备，设备名为 uioX，X 对应的就是次设备号 minor。
3. 在/sys/class/uio/uioX/目录下创建 maps 和 portio 接口。
4. 注册中断和中断处理函数 uio_interrupt



```
static struct pci_driver igbuio_pci_driver = {
    .name = "igb_uio",
    .id_table = NULL,
    .probe = igbuio_pci_probe,
    .remove = igbuio_pci_remove,
};
```



```
static int __init
igbuio_pci_init_module(void)
{
    int ret;

    if (igbuio_kernel_is_locked_down()) {
        pr_err("Not able to use module, kernel lock down is enabled\n");
        return -EINVAL;
    }

    if (wc_activate != 0)
        pr_info("wc_activate is set\n");

    ret = igbuio_config_intr_mode(intr_mode);
    if (ret < 0)
        return ret;

    return pci_register_driver(&igbuio_pci_driver);
}
```



```
igbuio_pci_probe(struct pci_dev *dev, const struct pci_device_id
{
    struct rte_uio_pci_dev *udev;
    dma_addr_t map_dma_addr;
    void *map_addr;
    int err;

#ifdef HAVE_PCI_IS_BRIDGE_API
    if (pci_is_bridge(dev)) {
        dev_warn(&dev->dev, "Ignoring PCI bridge device\n");
        return -ENODEV;
    }
#endif

    udev = kzalloc(sizeof(struct rte_uio_pci_dev), GFP_KERNEL);
    if (!udev)
        return -ENOMEM;

    /*
     * enable device: ask low-level code to enable I/O and
     * memory
     */
```

Uio 框架

在 UIO 中，使用 read/mmap 在 user space 存取设备对应的内存区域；但是 UIO 还是有一小部分中断处理在内核中，这个个中断处理的主要职责是开关中断，并将中断计数值加一。户空间驱动要监测一个设备中断，它只需阻塞在对/dev/uioX 的 read()操作上， 当设备产生中断时，read()操作立即返回。

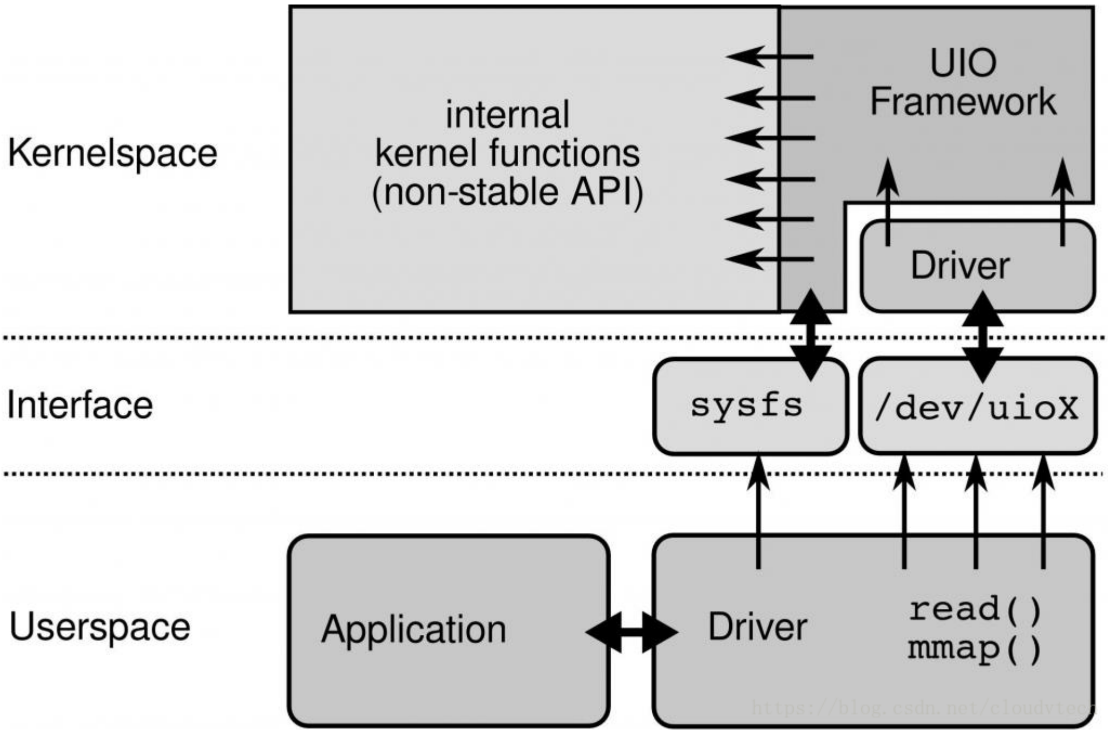
内核态的职责：

1. 分配和记录设备需要的资源和注册 uio 设备

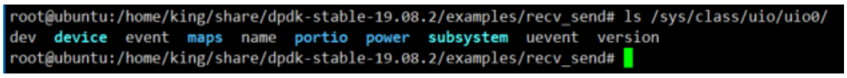
2. 使能设备
3. 申请资源
4. 读取并记录配置信息
5. 注册 uio 设备
6. 必须*在内核空间实现的小部分中断应答函数

用户态职责：

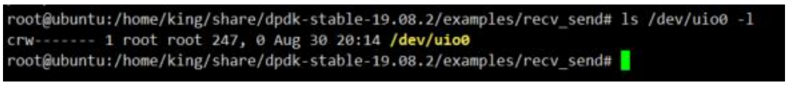
1. 获取中断事件(read/poll)
2. 处理中断(读写数据)



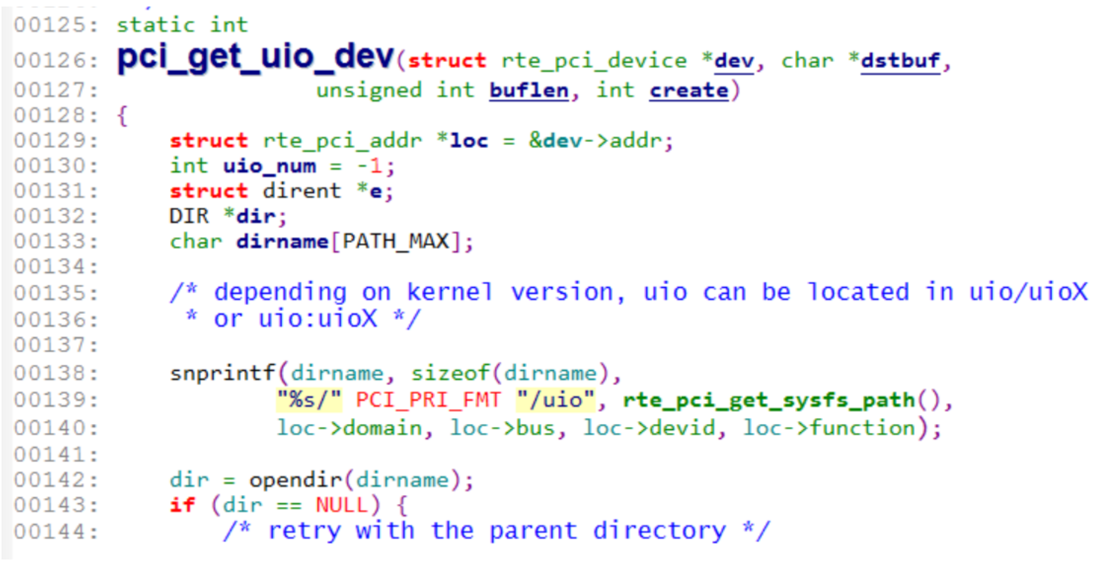
Uio 用户接口



```
root@ubuntu:/home/king/share/dpdk-stable-19.08.2/examples/recv_send# ls /sys/class/uio/uio0/
dev  device  event  maps  name  portio  power  subsystem  uevent  version
root@ubuntu:/home/king/share/dpdk-stable-19.08.2/examples/recv_send# 
```

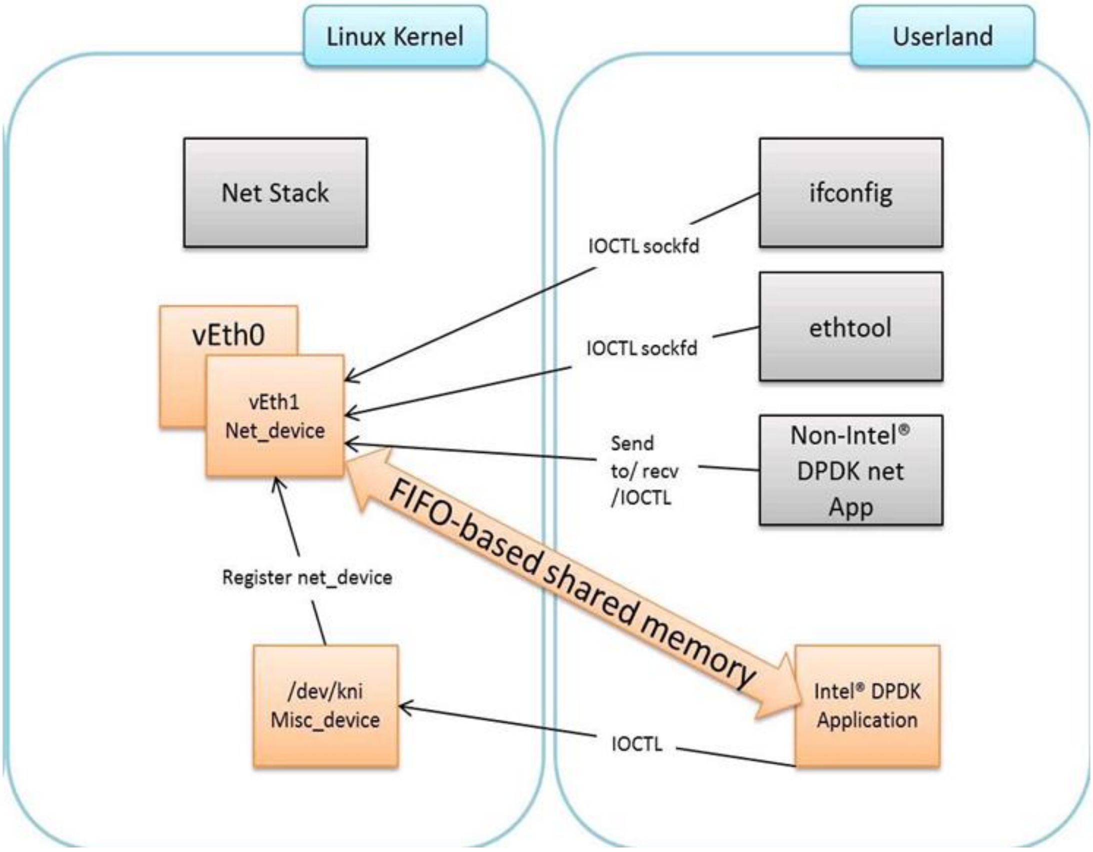


```
root@ubuntu:/home/king/share/dpdk-stable-19.08.2/examples/recv_send# ls /dev/uio0 -l
crw------- 1 root root 247, 0 Aug 30 20:14 /dev/uio0
root@ubuntu:/home/king/share/dpdk-stable-19.08.2/examples/recv_send# 
```



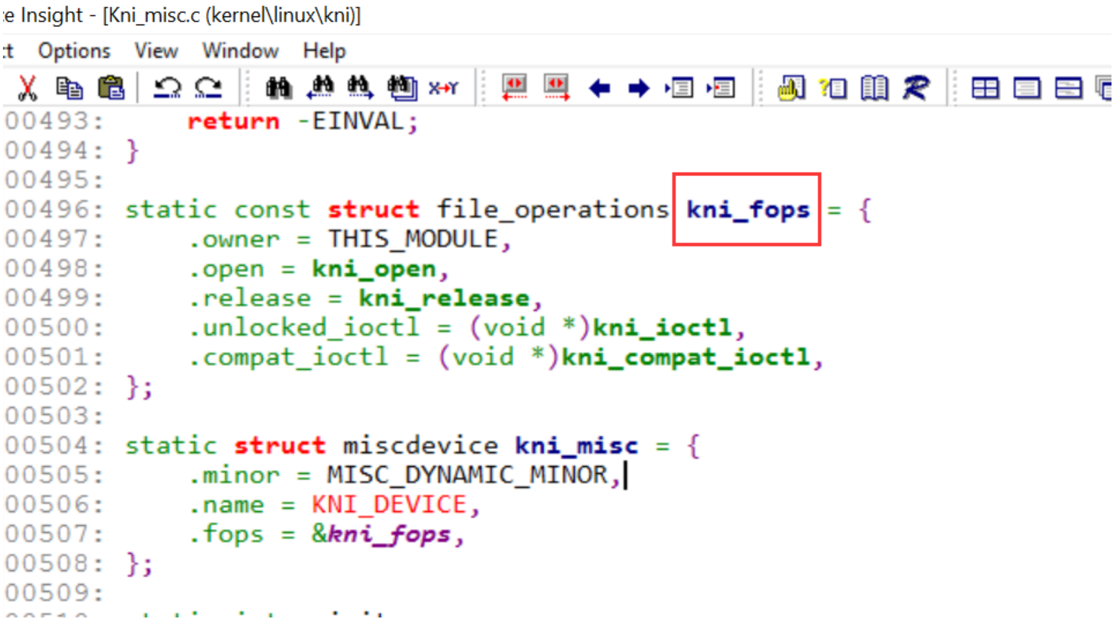
```
static int
pci_get_uio_dev(struct rte_pci_device *dev, char *dstbuf,
                unsigned int buflen, int create)
{
    struct rte_pci_addr *loc = &dev->addr;
    int uio_num = -1;
    struct dirent *e;
    DIR *dir;
    char dirname[PATH_MAX];

    /* depending on kernel version, uio can be located in uio/uioX
     * or uio:uioX */

    snprintf(dirname, sizeof(dirname),
            "%s/" PCI_PRI_FMT "/uio", rte_pci_get_sysfs_path(),
            loc->domain, loc->bus, loc->devid, loc->function);

    dir = opendir(dirname);
    if (dir == NULL) {
        /* retry with the parent directory */
```

KNI 原理

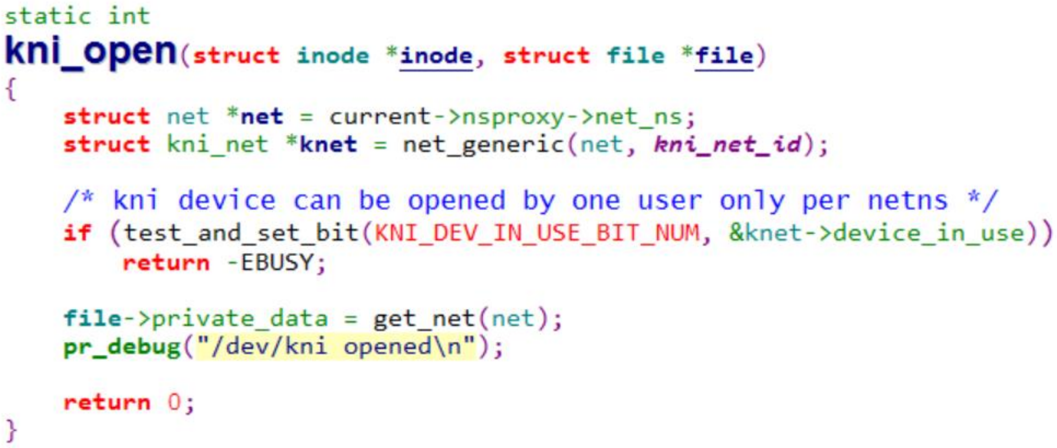


KNI 必然要也需要内核模块的支持，即 rte_kni.ko。其共有三个参数，分别是 lo_mode， kthread_mode 和 carrier。**lo_mode** 可配置为 lo_mode_none，lo_mode_fifo，和 lo_mode_fifo_skb，默认为 lo_mode_none。另外两个在实际产品中基本不会用到。**kthread_mode** 可配置为 single 和 multiple，默认为 single。**carrier** 可配置为 off 和 on，默认为 off。模块初始化函数 kni_init 也非常简单。除了解析上面的参数配置外，比较重要的就是注册 misc 设备和配置 lo_mode。

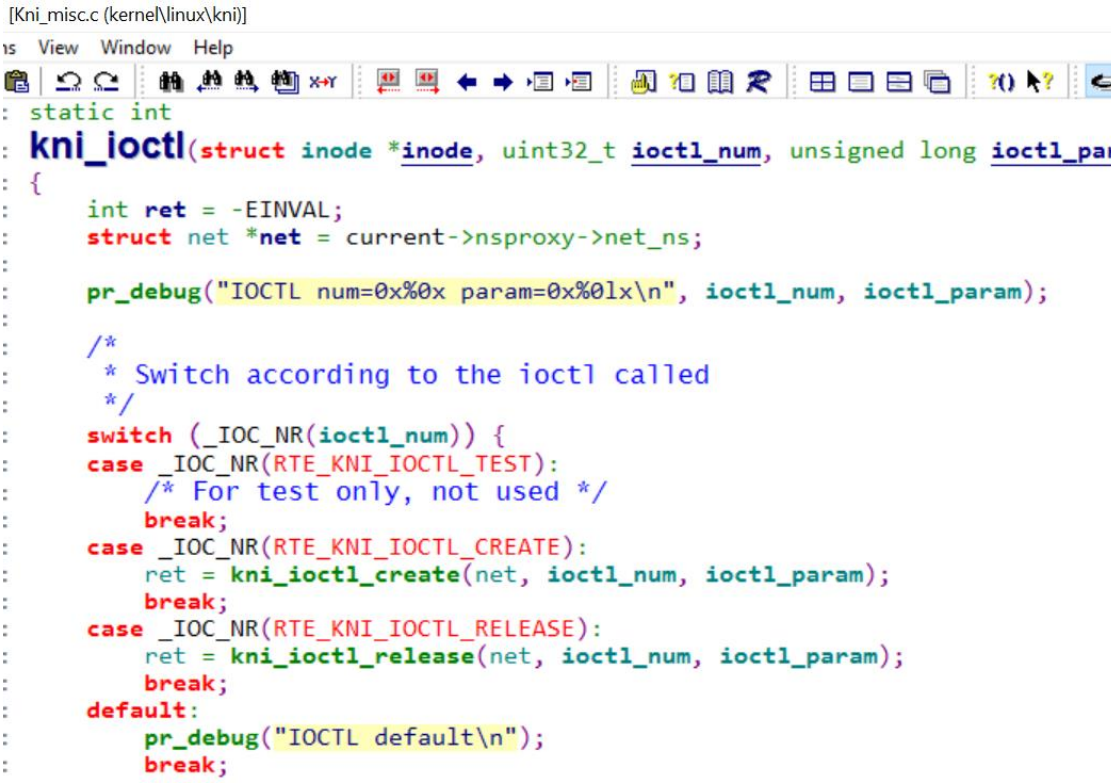
KNI 内核模块实现


e Insight - [Kni_misc.c (kernel\linux\kni)]
t Options View Window Help

```
        return -EINVAL;
}

static const struct file_operations kni_fops = {
        .owner = THIS_MODULE,
        .open = kni_open,
        .release = kni_release,
        .unlocked_ioctl = (void *)kni_ioctl,
        .compat_ioctl = (void *)kni_compat_ioctl,
};

static struct miscdevice kni_misc = {
        .minor = MISC_DYNAMIC_MINOR,
        .name = KNI_DEVICE,
        .fops = &kni_fops,
};
```



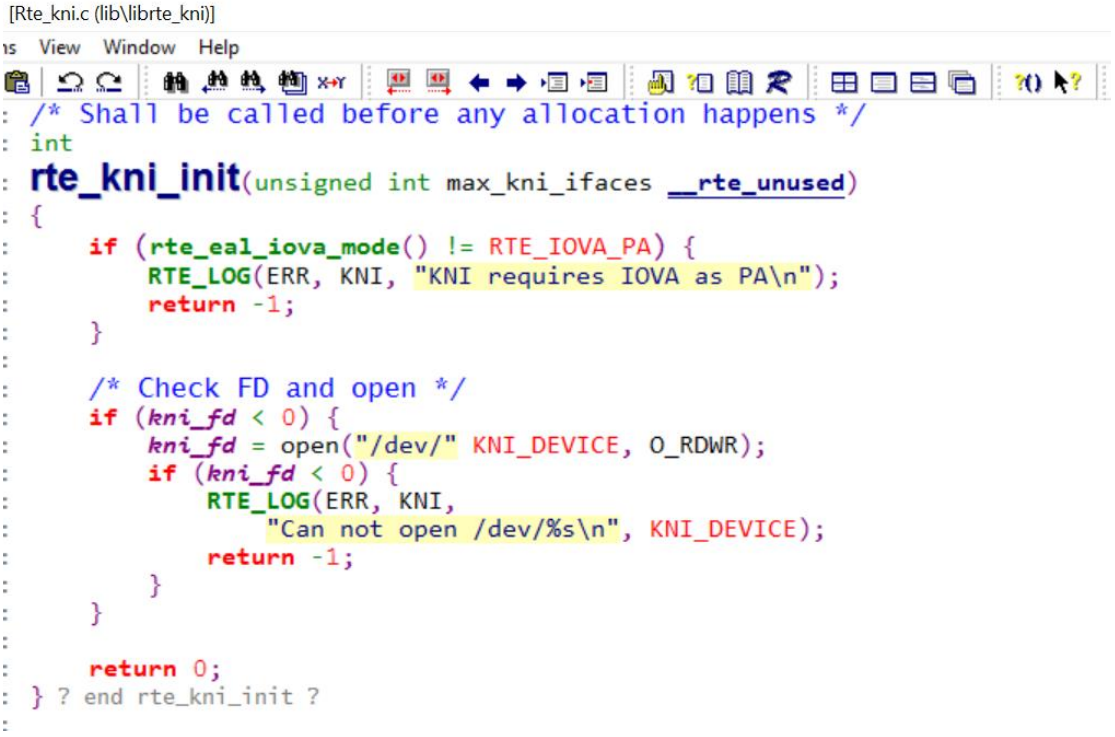
```
static int
kni_open(struct inode *inode, struct file *file)
{
    struct net *net = current->nsproxy->net_ns;
    struct kni_net *knet = net_generic(net, kni_net_id);

    /* kni device can be opened by one user only per netns */
    if (test_and_set_bit(KNI_DEV_IN_USE_BIT_NUM, &knet->device_in_use))
        return -EBUSY;

    file->private_data = get_net(net);
    pr_debug("/dev/kni opened\n");

    return 0;
}
```



[Kni_misc.c (kernel\linux\kni)]

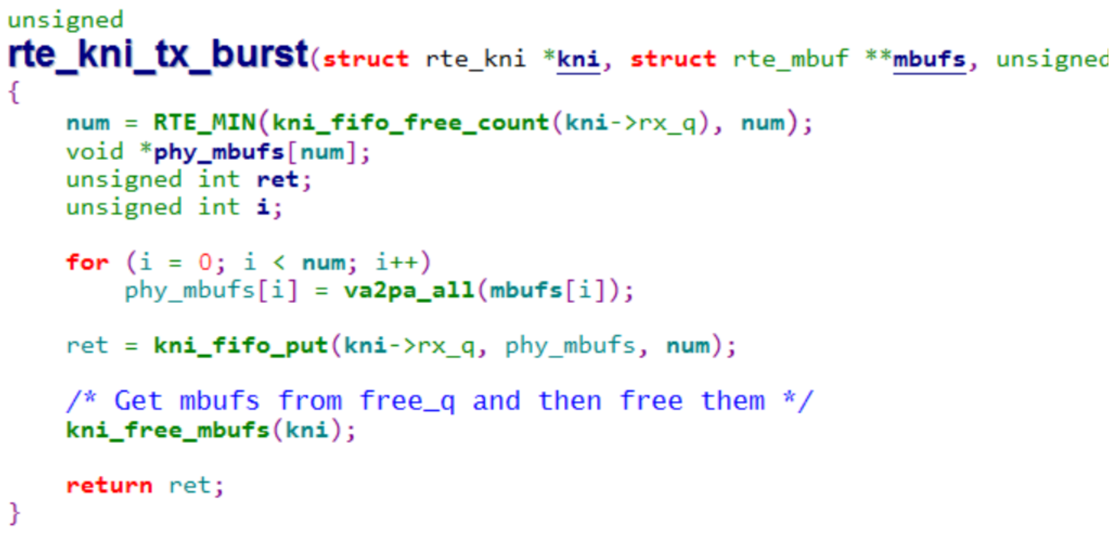
```
static int
kni_ioctl(struct inode *inode, uint32_t ioctl_num, unsigned long ioctl_pa
{
    int ret = -EINVAL;
    struct net *net = current->nsproxy->net_ns;

    pr_debug("IOCTL num=0x%0x param=0x%0lx\n", ioctl_num, ioctl_param);

    /*
     * Switch according to the ioctl called
     */
    switch (_IOC_NR(ioctl_num)) {
    case _IOC_NR(RTE_KNI_IOCTL_TEST):
        /* For test only, not used */
        break;
    case _IOC_NR(RTE_KNI_IOCTL_CREATE):
        ret = kni_ioctl_create(net, ioctl_num, ioctl_param);
        break;
    case _IOC_NR(RTE_KNI_IOCTL_RELEASE):
        ret = kni_ioctl_release(net, ioctl_num, ioctl_param);
        break;
    default:
        pr_debug("IOCTL default\n");
        break;
```

KNI 用户接口

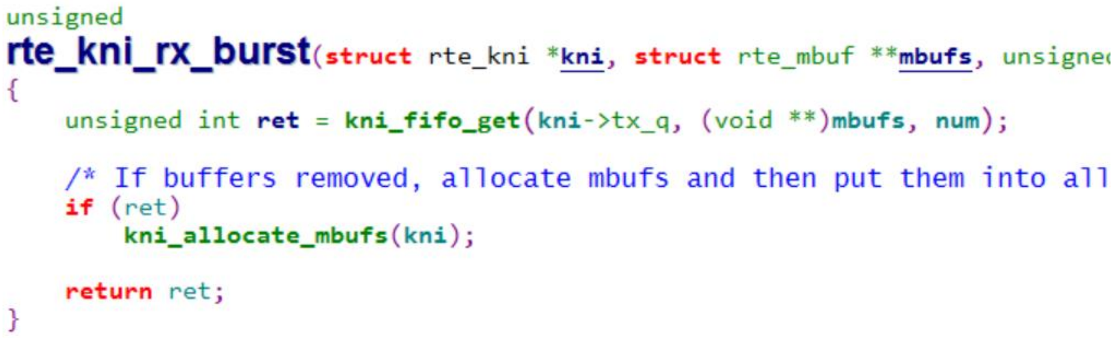


[Rte_kni.c (lib\librte_kni)]

```
/* Shall be called before any allocation happens */
int
rte_kni_init(unsigned int max_kni_ifaces __rte_unused)
{
    if (rte_eal_iova_mode() != RTE_IOVA_PA) {
        RTE_LOG(ERR, KNI, "KNI requires IOVA as PA\n");
        return -1;
    }

    /* Check FD and open */
    if (kni_fd < 0) {
        kni_fd = open("/dev/" KNI_DEVICE, O_RDWR);
        if (kni_fd < 0) {
            RTE_LOG(ERR, KNI,
                "Can not open /dev/%s\n", KNI_DEVICE);
            return -1;
        }
    }

    return 0;
} ? end rte_kni_init ?
```



```
unsigned
rte_kni_tx_burst(struct rte_kni *kni, struct rte_mbuf **mbufs, unsigned
{
    num = RTE_MIN(kni_fifo_free_count(kni->rx_q), num);
    void *phy_mbufs[num];
    unsigned int ret;
    unsigned int i;

    for (i = 0; i < num; i++)
        phy_mbufs[i] = va2pa_all(mbufs[i]);

    ret = kni_fifo_put(kni->rx_q, phy_mbufs, num);

    /* Get mbufs from free_q and then free them */
    kni_free_mbufs(kni);

    return ret;
}
```



```
unsigned
rte_kni_rx_burst(struct rte_kni *kni, struct rte_mbuf **mbufs, unsigne
{
    unsigned int ret = kni_fifo_get(kni->tx_q, (void **)mbufs, num);

    /* If buffers removed, allocate mbufs and then put them into all
    if (ret)
        kni_allocate_mbufs(kni);

    return ret;
}
```

VFIO

VFIO 原理

VFIO就是内核针对IOMMU提供的软件框架,支持DMA Remapping和Interrupt Remapping,这里只讲 DMA Remapping。VFIO 利用 IOMMU 这个特性，可以屏蔽物理地址对上层的可见性，可以用来开发用户态驱动，也可以实现设备透传。

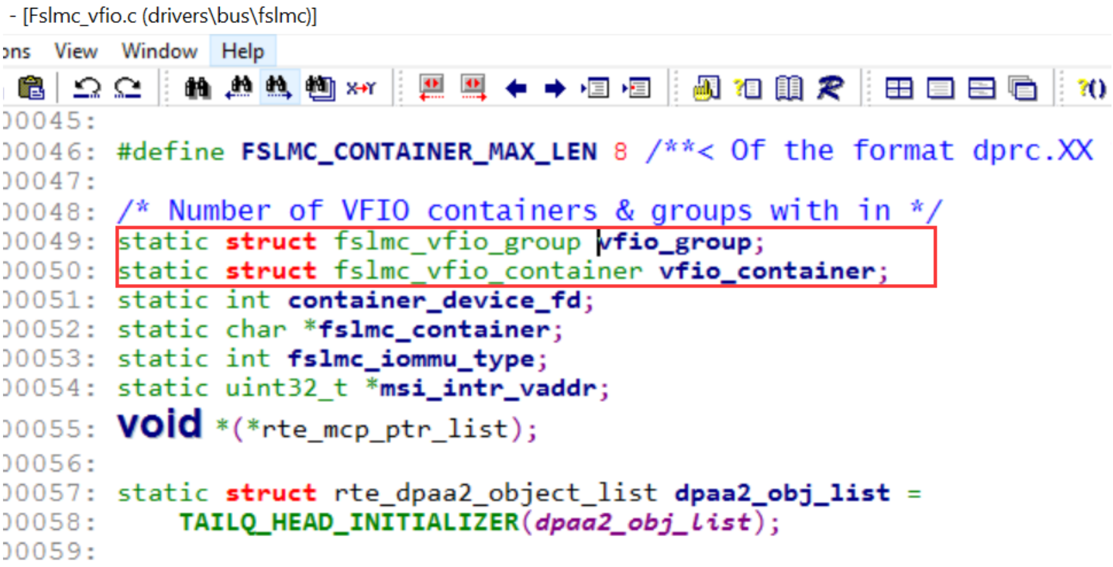
概念介绍

先介绍 VFIO 中的几个重要概念，主要包括 Group 和 Container。

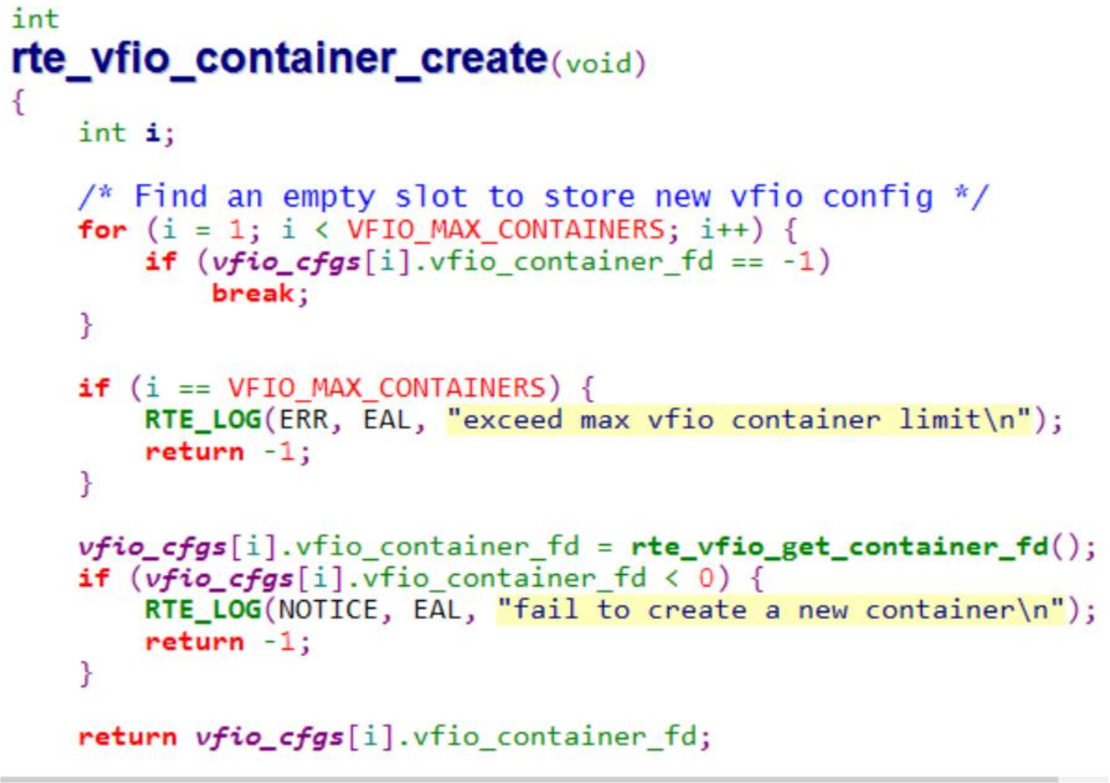
1) Group：group 是 IOMMU 能够进行 DMA 隔离的最小硬件单元，一个 group 内可能只有一个 device，也可能有多个 device，这取决于物理平台上硬件的 IOMMU 拓扑结构。 设备直通的时候一个 group 里面的设备必须都直通给一个虚拟机。 不能够让一个 group 里的多个 device 分别从属于 2 个不同的 VM，也不允许部分 device 在 host 上而另一部分被分配到 guest 里， 因为就这样一个 guest 中的 device 可以利用 DMA 攻击获取另外一个 guest 里的数据，就无法做到物理上的 DMA 隔离。

2) Container：对于虚机，Container 这里可以简单理解为一个 VM Domain 的物理内存空间。对于用户态驱动，Container 可以是多个 Group 的集合。

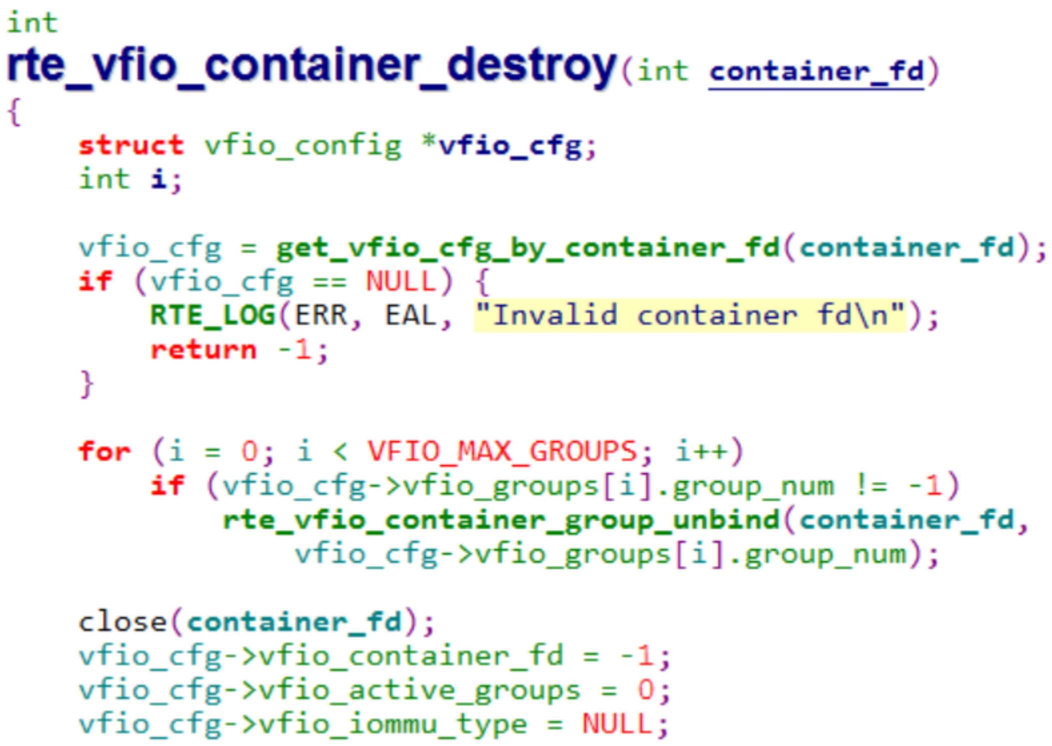
VFIO 模块实现



```c

#define FSLMC_CONTAINER_MAX_LEN 8 /**< Of the format dprc.XX

/* Number of VFIO containers & groups with in */
static struct fslmc_vfio_group vfio_group;
static struct fslmc_vfio_container vfio_container;
static int container_device_fd;
static char *fslmc_container;
static int fslmc_iommu_type;
static uint32_t *msi_intr_vaddr;
void *(*rte_mcp_ptr_list);

static struct rte_dpaa2_object_list dpaa2_obj_list =
    TAILQ_HEAD_INITIALIZER(dpaa2_obj_list);

```

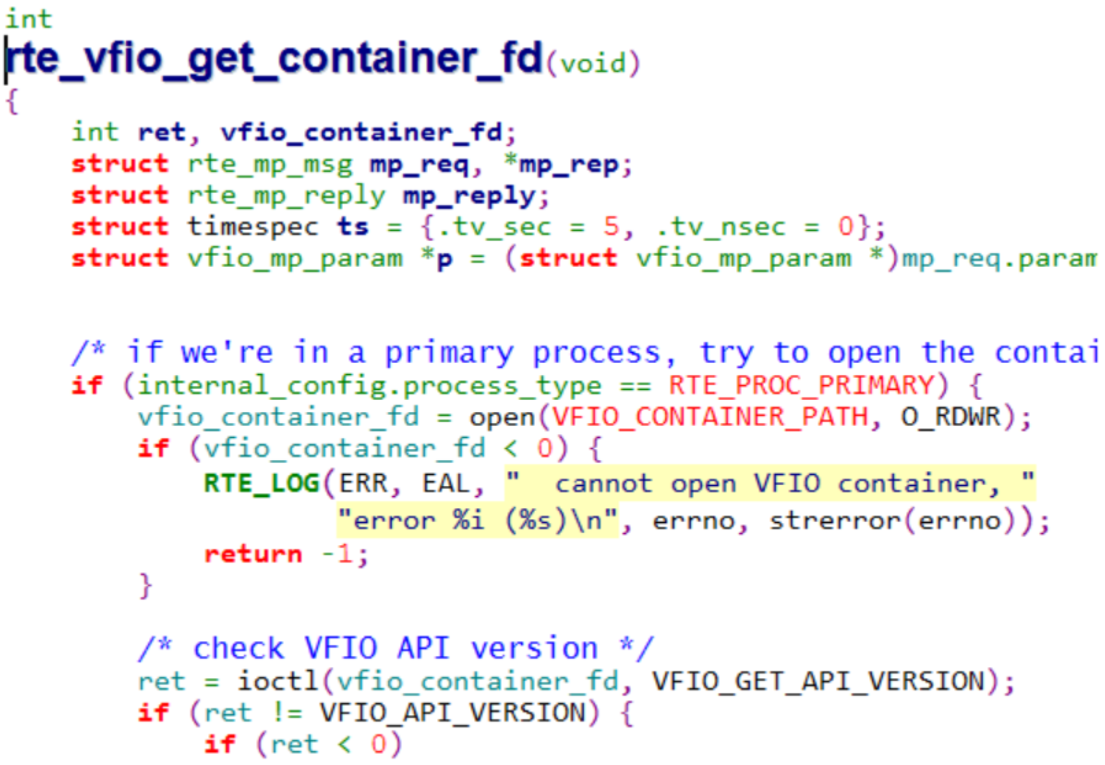


```c
int
rte_vfio_container_create(void)
{
    int i;

    /* Find an empty slot to store new vfio config */
    for (i = 1; i < VFIO_MAX_CONTAINERS; i++) {
        if (vfio_cfgs[i].vfio_container_fd == -1)
            break;
    }

    if (i == VFIO_MAX_CONTAINERS) {
        RTE_LOG(ERR, EAL, "exceed max vfio container limit\n");
        return -1;
    }

    vfio_cfgs[i].vfio_container_fd = rte_vfio_get_container_fd();
    if (vfio_cfgs[i].vfio_container_fd < 0) {
        RTE_LOG(NOTICE, EAL, "fail to create a new container\n");
        return -1;
    }

    return vfio_cfgs[i].vfio_container_fd;
```

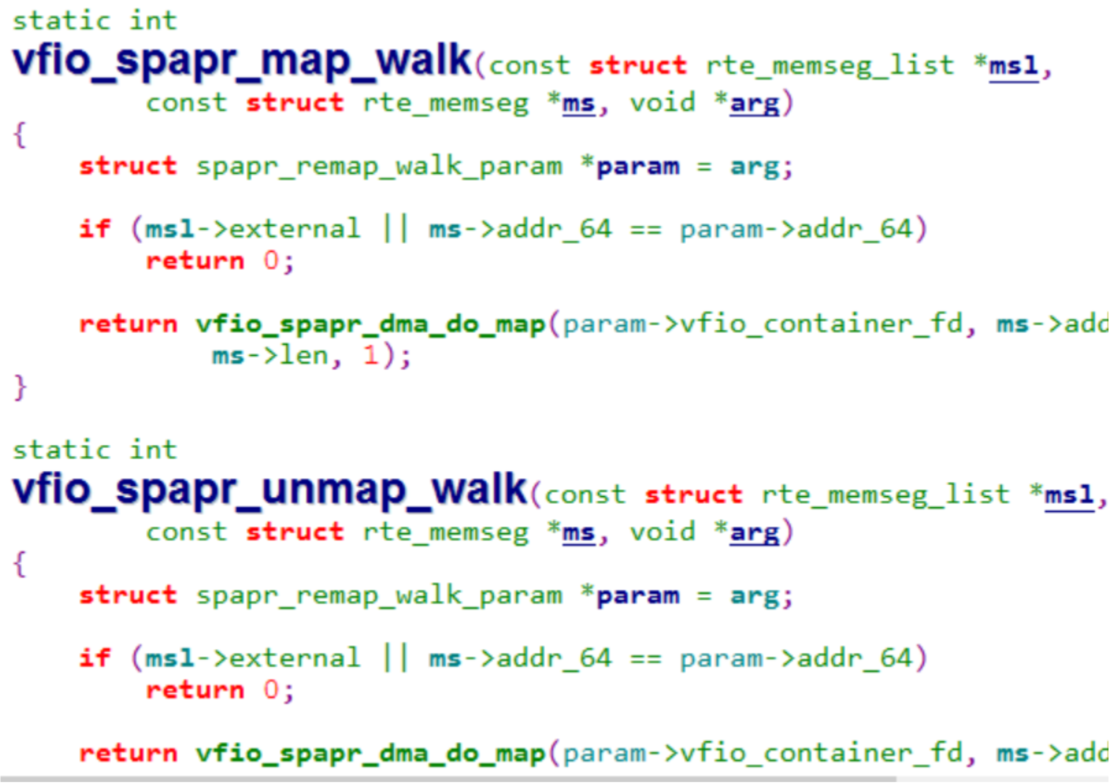


```c
int
rte_vfio_container_destroy(int container_fd)
{
	struct vfio_config *vfio_cfg;
	int i;

	vfio_cfg = get_vfio_cfg_by_container_fd(container_fd);
	if (vfio_cfg == NULL) {
		RTE_LOG(ERR, EAL, "Invalid container fd\n");
		return -1;
	}

	for (i = 0; i < VFIO_MAX_GROUPS; i++)
		if (vfio_cfg->vfio_groups[i].group_num != -1)
			rte_vfio_container_group_unbind(container_fd,
				vfio_cfg->vfio_groups[i].group_num);

	close(container_fd);
	vfio_cfg->vfio_container_fd = -1;
	vfio_cfg->vfio_active_groups = 0;
	vfio_cfg->vfio_iommu_type = NULL;
```

VFIO 应用程序接口



```c
int
rte_vfio_get_container_fd(void)
{
	int ret, vfio_container_fd;
	struct rte_mp_msg mp_req, *mp_rep;
	struct rte_mp_reply mp_reply;
	struct timespec ts = {.tv_sec = 5, .tv_nsec = 0};
	struct vfio_mp_param *p = (struct vfio_mp_param *)mp_req.param


	/* if we're in a primary process, try to open the contai
	if (internal_config.process_type == RTE_PROC_PRIMARY) {
		vfio_container_fd = open(VFIO_CONTAINER_PATH, O_RDWR);
		if (vfio_container_fd < 0) {
			RTE_LOG(ERR, EAL, "  cannot open VFIO container, "
					"error %i (%s)\n", errno, strerror(errno));
			return -1;
		}

		/* check VFIO API version */
		ret = ioctl(vfio_container_fd, VFIO_GET_API_VERSION);
		if (ret != VFIO_API_VERSION) {
			if (ret < 0)
```



```
static int
vfio_spapr_map_walk(const struct rte_memseg_list *msl,
        const struct rte_memseg *ms, void *arg)
{
    struct spapr_remap_walk_param *param = arg;

    if (msl->external || ms->addr_64 == param->addr_64)
        return 0;

    return vfio_spapr_dma_do_map(param->vfio_container_fd, ms->add
            ms->len, 1);
}

static int
vfio_spapr_unmap_walk(const struct rte_memseg_list *msl,
        const struct rte_memseg *ms, void *arg)
{
    struct spapr_remap_walk_param *param = arg;

    if (msl->external || ms->addr_64 == param->addr_64)
        return 0;

    return vfio_spapr_dma_do_map(param->vfio_container_fd, ms->add
```