我们的驱动程序一般三类：

1：字符设备：

file\_operation{

open

read

}

register\_chardev();

2.块设备

app read write

文件系统 vfat ext2 yaffs jffs2 //文件系统把对文件的操作转换为对flash等硬件操作

块设备驱动程序//一些读写不能直接像字符设备那样直接读写

硬件的操作：

一般的流程：

1.构造gendisk()//alloc\_disk

2.设置

.quene = blk\_init\_queue()//将读写放在队列中

设置属性，容量等

3.注册 add\_disk

3.网卡设备。

网卡驱动程序框架

网卡驱动程序“收发功能”：

只要把上层的数据发给网卡，从网卡来的数据构造成包给上层即可。网卡只需要“socket”编程，不需要打开某设备。

驱动程序都是以面向对象的思想写的，都有相关的结构体。

编程步骤

1，分配某结构体：net\_device

2，设置结构体。

1. 提供一个发包函数：hard\_start\_xmit() // dev->hard\_start\_xmit = net\_send\_packet;
2. ，提供收包的功能 ： net\_interrupt(int irq, void \*dev\_id)-->netif\_rx(skb);

收到数据后，网卡里面一般都有中断程序。在中断程序中有一个上报数据的函数。 netif\_rx(skb);//上报

3，注册结构体:register\_netdev(dev) 真实驱动中使用的是此注册函数。

4，硬件相关操作。

看内核中的“cs89x.c”这个真实的网卡驱动程序：

int \_\_init init\_module(void)

-->分配一个 net\_device 结构体

struct net\_device \*dev = alloc\_etherdev(sizeof(struct net\_local));

--->alloc\_netdev(sizeof\_priv, "eth%d", ether\_setup); 分配时用了eth%d这样的名字。

-->设置net\_device 结构：

MAC 地址，硬件相关操作

dev->dev\_addr[0] = 0x08;

dev->dev\_addr[1] = 0x89;

dev->dev\_addr[2] = 0x89;

dev->dev\_addr[3] = 0x89;

dev->dev\_addr[4] = 0x89;

dev->dev\_addr[5] = 0x89; //以上为MAC地址。

--> 硬件相关设置

-->ret = cs89x0\_probe1(dev, io, 1);

-->net\_device结构中有open,read等函数。

dev->open = net\_open;

dev->stop = net\_close;

dev->tx\_timeout = net\_timeout;

dev->watchdog\_timeo = HZ;

dev->hard\_start\_xmit = net\_send\_packet; //硬件启动传输。这是发包函数。

dev->get\_stats = net\_get\_stats;

dev->set\_multicast\_list = set\_multicast\_list;

dev->set\_mac\_address = set\_mac\_address;

-->retval = register\_netdev(dev);

接收到数据要做的事情：

irqreturn\_t net\_interrupt(int irq, void \*dev\_id) //这是一个中断

-->net\_rx(dev);

-->从硬件芯片里读出来

status = readword(ioaddr, RX\_FRAME\_PORT);

length = readword(ioaddr, RX\_FRAME\_PORT);

-->skb = dev\_alloc\_skb(length + 2); 分配一个skb缓冲。

-->netif\_rx(skb); //上报

-->netif\_wake\_queue(dev); 发送完数据后就唤醒队列

发包函数：

int net\_send\_packet(struct sk\_buff \*skb, struct net\_device \*dev)

接收到包后是上报了一个“sk\_buff”缓冲（netif\_rx(skb)）

sk\_buff 结构 是纽带，运用“hard\_start\_xmit()”和“netif\_rx()”：

应用层构造好一个包后，放到“skb\_buff”结构交给网卡驱动，调用“hard\_start\_xmit()”来发送。

网卡在中断程序中收到数据后，从芯片里把数据读出来构造一个“skb\_buff”结构数据， 调用“netif\_rx()”上报数据给应用层。

-->netif\_stop\_queue(dev); 先停止队列

-->将“skb\_buff”中数据写到网卡芯片：

writeword(dev->base\_addr, TX\_CMD\_PORT, lp->send\_cmd);

writeword(dev->base\_addr, TX\_LEN\_PORT, skb->len);

writewords(dev->base\_addr, TX\_FRAME\_PORT,skb->data,(skb->len+1) >>1);

-->dev\_kfree\_skb (skb); 然后释放skb\_buff。

写一个虚拟网卡驱动：

我们参考cs89x0:

1，分配net\_device 结构：

net\_device \*alloc\_netdev(int sizeof\_priv, const char \*name,void (\*setup)(struct net\_device \*))

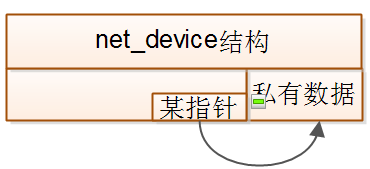
//1,分配 net\_device 结构体

vnet\_dev = alloc\_netdev(0, "vnet%d", ether\_setup);

直接用“alloc\_netdev()”自忆定义网卡的名字为“vent%d”.其中“ether\_setup”是默认的设置函数。

sizeof\_priv 是私有数据，我们的这里定义私有数据为0.

内核中经常会在只分配一个结构体的时候多分配一个内存。这块内存就是用来存放自已的“私有数据”



这里不需要私有信息，所以直接分配"net\_device"结构中没有分配“私有数据空间”，上面指为了“0”



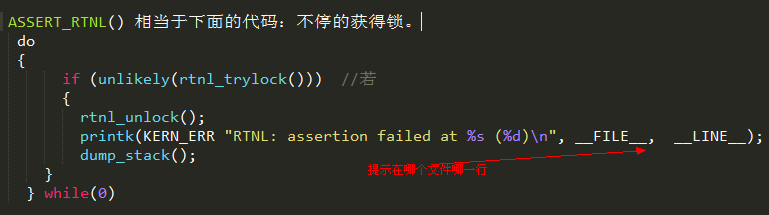
这就是最简单的网卡驱动程序。//具体的可以查看我们的代码

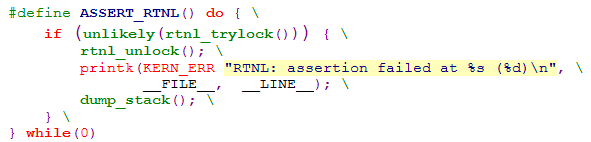


要用上面的“register\_netdev()”来注册，里面获得锁后用“register\_netdevice（）” 来注册。



想获得一把锁，但没有成功。而真实的注册函数“register\_netdev()”中有获得锁和解锁的过程。





网卡驱动程序框架:

app: socket 编程

--------------------------------------------------

---------------

--------------- 若干层网络协议--纯软件（网络七层模型）

---------------

---------------

hard\_start\_xmit|| /\ //发送数据

\/ || netif\_rx sk\_buff //收到数据

---------------

硬件相关的驱动程序(要提供hard\_start\_xmit, 有数据时要用netif\_rx 上报)



怎么写网卡驱动程序？

1. 分配一个net\_device 结构体

2. 设置:

2.1 发包函数: hard\_start\_xmit

2.2 收到数据时(在中断处理函数里)用netif\_rx 上报数据

2.3 其他设置

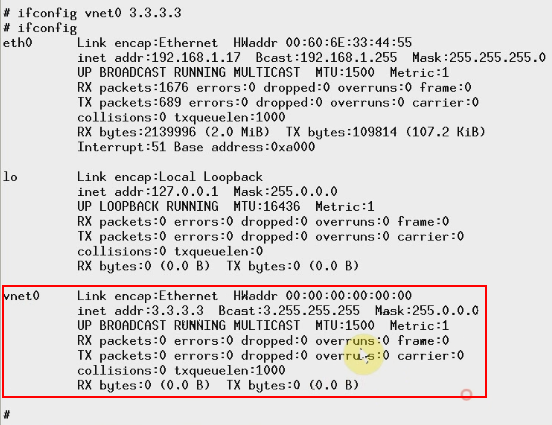
3. 注册: register\_netdev

写好的驱动测试：

1.insmod

2. ifconfig vnet0 3.3.3.3 //设置 ip vnet0 参数是我们写代码中的名字

ifconfig // 查看



3. ping 3.3.3.3 // 成功



ping 应用程序可以ping 通自已，说明ping 从应用程序中进来，没有经过网卡驱动层直接从应用程序返回了。

这说明“IP” 是纯软件的概念。

ping 3.3.3.4 // 死机



ping 时会调用“dev\_hard\_start\_xmit”函数。但这个函数没有提供，故死机了。

加上发包函数后，会发现可以ping 了：//具体的代码可以看我们代码

//2,设置

vnet\_dev->hard\_start\_xmit = virt\_net\_sendpacket;

static int virt\_net\_sendpacket (struct sk\_buff \*skb,struct net\_device \*dev)

{//里面什么也没做，只是打印出此函数被调用的次数。

static int cnt = 0;

printk("virt\_net\_sendpacket = %d\n",++cnt);

return 0;

}

重新编译驱动后再ping：

1. ifconfig vnet0 3.3.3.3
2. 

上面发送了6 个数据包，但统计信息里显示还是“0”个。

在“net\_dev”结构体中有统计信息成员：

有发送的包数字节数等等….

struct net\_device ｛

… …

struct net\_device\_stats\* (\*get\_stats)(struct net\_device \*dev);

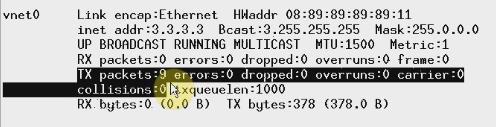
struct net\_device\_stats stats;

… …

｝



再编译后测试：



假如我们想要ping 3.3.3.4的 时候 ：



上层通过xmit发数据，到另一台机器的ip为3.3.3.4 然后他返回数据给 我们的虚拟网卡

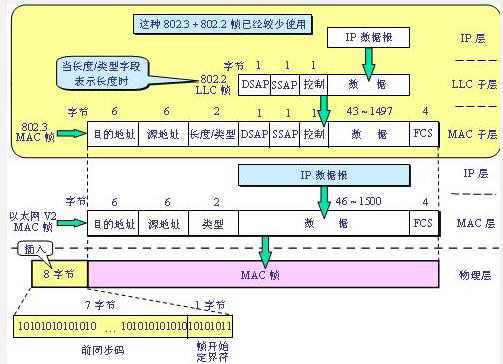
通过rxif\_net()收到数据上报给应用层。这样才能ping的通。所以我们作假在xmit 发包的驱动数据中上报作假的收报数据就能ping 通。

把数据写入网卡后，网卡并不是立即全部发送出去。网卡发送完数据后会产生一个中断。

一般的网络包有如下的格式组成：

mac ip (里面还有check) 类型 数据

目的mac源mac 目的ip源ip





一开始是MAC 头：收发数据包的MAC 头要对调。



/ 从硬件读出/保存数据

/\* 对调"源/目的"的mac 地址 \*/

ethhdr = (struct ethhdr \*)skb->data; //数据放在skb的data中。

memcpy(tmp\_dev\_addr, ethhdr->h\_dest, ETH\_ALEN); //将目的MAC拷贝到临时数组中.

memcpy(ethhdr->h\_dest, ethhdr->h\_source, ETH\_ALEN);//将源、目的MAC对调。

memcpy(ethhdr->h\_source, tmp\_dev\_addr, ETH\_ALEN); //对调 源、目的MAC。

接着IP 头也要对调，将ping 包的类型修改为回复：



/\* 对调"源/目的"的ip 地址 \*/

ih = (struct iphdr \*)(skb->data + sizeof(struct ethhdr));

saddr = &ih->saddr;

daddr = &ih->daddr;

tmp = \*saddr; //源IP放到tmp

\*saddr = \*daddr; //指针所指内容交换

\*daddr = tmp;

//((u8 \*)saddr)[2] ^= 1; /\* change the third octet (class C) \*/

//((u8 \*)daddr)[2] ^= 1;

type = skb->data + sizeof(struct ethhdr) + sizeof(struct iphdr);

//printk("tx package type = %02x\n", \*type);

// 修改类型, 原来0x8表示ping

\*type = 0; /\* 0表示reply \*/

重新计算校验码：

ih->check = 0; /\* and rebuild the checksum (ip needs it) \*/

ih->check = ip\_fast\_csum((unsigned char \*)ih,ih->ihl); //校验码要重新计算

重新编译并测试：



就可以ping通了因为我们在发包的时候构造了收包过程。可以看到收到的包字节数等都是相同的。

一般的网卡驱动芯片，厂家会提供我们要根据手册等进行配置一些寄存器就行了。

可以参考我们的dm9000网卡的。