**概念**

Octet：8位数据，网络设备和协议所使用的最小单位，相当于字节

协议头（header）：指TCP头，IP头，MAC头等，我们驱动程序所关心的是MAC头，所后面讲的协议头我们指的是MAC头

**设备注册**

**分配设备**

内核使用net\_device结构（头文件<linux/netdevice.h>）表示一个网络接口，在使用网络设备接口时，我们需要先注册它

struct net\_device \*alloc\_netdev(int sizeof\_priv,

                                const char \*name,

                                void (\*setup)(struct net\_device \*));

Sizeof\_priv：驱动程序可以在net\_device中保存自定义数据，sizeof\_priv为该数据的大小

Name：接口名称

Setup：alloc\_netdev函数并没有初始化net\_device结构，setup完成其后的初始化

释放设备结构

free\_netdev(struct net\_device \*snull\_devs);

**初始化设备（setup函数）**

我们在setup函数中初始化设备

// net\_device结构很复杂，我们使用内核提供的 ether\_setup 初始化一些不重要的字段

ether\_setup(dev);

dev->open = snull\_open;                     // 打开设备的指针

dev->stop = snull\_release;                  // 关闭设备的指针

dev->set\_config = snull\_config;

dev->hard\_start\_xmit = snull\_tx;            //

dev->do\_ioctl = snull\_ioctl;                // ioctl 文件控制函数

dev->get\_stats = snull\_stats;

dev->rebuild\_header = snull\_rebuild\_header;

dev->hard\_header = snull\_header;

// 处理传输超时

dev->tx\_timeout = snull\_tx\_timeout;

dev->watchdog\_timeo = timeout;

/\* 添加IFF\_NOARP标志（不使用ARP地址解析协议） \*/

dev->flags |= IFF\_NOARP;

dev->features |= NETIF\_F\_NO\_CSUM;

dev->hard\_header\_cache = NULL;              /\* Disable caching \*/

**注册设备**

初始化后，我们便可以向内核注册我们的设备了

register\_netdev(struct net\_device \*snull\_devs);

注销设备

unregister\_netdev(struct net\_device \*snull\_devs);

**Net\_device结构**

**设备方法**

int (\*open)(struct net\_device \*dev)：

打开接口

int (\*stop)(struct net\_device \*dev)：

停止接口

int (\*hard\_start\_xmit) (struct sk\_buff \*skb, struct net\_device \*dev);

起始报文的发送的方法

int (\*hard\_header) (struct sk\_buff \*skb, struct net\_device \*dev, unsigned short type, void \*daddr, void \*saddr, unsigned len)：

根据源与目标硬件地址构建MAC头

void (\*tx\_timeout)(struct net\_device \*dev)：

发送方法超时将调用该方法

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

任何时候当一个应用程序需要获取接口的统计信息, 调用这个方法

int weight：

int (\*poll)(struct net\_device \*dev; int \*quota)：

NAPI（轮询方式）接收数据的驱动需提供该方法，NAPI ( 以及 weight 成员) 在"接收中断缓解"一节中涉及.

**设备属性**

unsigned long trans\_start：

保存一个 jiffy 值. 在开始发送一个报文时驱动负责分别更新这些值

int watchdog\_timeo：

发送超时值（按 jiffy 计算），超时将调用驱动的 tx\_timeout 函数.

void \*priv;

保存驱动程序自定义数据，该字段不能直接访问，应使用alloc\_netdev 设置该成员， 使用 netdev\_priv 获取该成员

spinlock\_t xmit\_lock;

hard\_start\_xmit可能被内核同时多个调用，xmit\_lock可用于对其中的一些代码段进行上锁

**snull\_priv结构**

Snull\_priv是本章节驱动程序使用的自定义数据结构，其结构如下：

struct snull\_priv

{

    struct net\_device\_stats stats;

    int status;

    struct snull\_packet \*ppool;

    struct snull\_packet \*rx\_queue; /\* List of incoming packets \*/

    int rx\_int\_enabled;

    int tx\_packetlen;

    u8 \*tx\_packetdata;

    struct sk\_buff \*skb;

    spinlock\_t lock;

};

**打开与关闭设备**

使用ifconfig向接口赋予地址时，会调用设备的open方法打开设备，关闭接口则会调用stop方法，两个函数在成功时均返回0

**open方法**

Open的任务：

从硬件设备复制硬件地址到net\_device的dev\_addr中

启用接口传输队列

示例：

int snull\_open(struct net\_device \*dev)

{

    // 复制硬件地址（MAC地址）到dev\_addr中，这里是随便写的，实际要从硬件中复制

    memcpy(dev->dev\_addr, "\0SNUL0", ETH\_ALEN);

    // 启用传输队列

    netif\_start\_queue(dev);

    return 0;

}

**stop方法**

int snull\_release(struct net\_device \*dev)

{

    // 停止传输队列

    netif\_stop\_queue(dev);

    return 0;

}

**数据包的传输（发送）**

内核要发送一个数据包，会调用设备的hard\_start\_stransmit函数将设备放入外发队列中，每个内核处理的报文都包含在一个 socket 缓存结构（sk\_buff结构）里，定义见<linux/skbuff.h>.

后面的代码中，我们将skb变量名作为指向sk\_buff的指针

Skb->data：要传输的数据包

Skb->len：数据的octet的长度

示例：

int snull\_tx(struct sk\_buff \*skb, struct net\_device \*dev)

{

    int len;

    char \*data, shortpkt[ETH\_ZLEN];

    // 获取驱动程序的自定义数据

    struct snull\_priv \*priv = netdev\_priv(dev);

    data = skb->data;

len = skb->len;

dev->trans\_start = jiffies; // 开始发送时更新trans\_start的值

priv->skb = skb;

    // 自定义函数，将数据传送到设备

snull\_hw\_tx(data, len, dev);

    return 0;   // 传输成功，非0表示失败

}

**控制并发传输**

在实际的硬件中，驱动程序一直向硬件的发送缓存写入数据，硬件的发送缓存可能会被耗光，这时需要暂定传输队列

void netif\_stop\_queue(struct net\_device \*dev);  // 暂停传输队列

void netif\_wake\_queue(struct net\_device \*dev);  // 启用传输队列

**传输超时**

如果传输在trans\_start+watchdog\_timeo时间之前未完成，则tx\_timeout方法被调用

**数据包接收**

驱动程序有两种接收数据的方式：中断方式和轮询方式

**中断方式**

// 中断处理例程

static void snull\_regular\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    int statusword;

    struct snull\_priv \*priv;

    struct snull\_packet \*pkt = NULL;

struct net\_device \*dev = (struct net\_device \*)dev\_id;

    if (!dev)

        return;

    // 获取驱动自定义数据

    priv = netdev\_priv(dev);

    spin\_lock(&priv->lock);

    // 假装获取设备状态，实际设备中我们应该从设备中获取

    statusword = priv->status;

    // 假装重置设备状态

    priv->status = 0;

    // 接收中断

    if (statusword & SNULL\_RX\_INTR)

    {

        // 假装从设备中获取接收到的数据

        pkt = priv->rx\_queue;

        if (pkt)

        {

            priv->rx\_queue = pkt->next;

            // 自定义函数，处理接收的数据，下面会讲到

            snull\_rx(dev, pkt);

        }

}

    // 发送中断

    if (statusword & SNULL\_TX\_INTR)

    {

        // 发送完成

        priv->stats.tx\_packets++;

        priv->stats.tx\_bytes += priv->tx\_packetlen;

        // 释放skb

        dev\_kfree\_skb(priv->skb);

}

    spin\_unlock(&priv->lock);

    if (pkt)

        snull\_release\_buffer(pkt); /\* Do this outside the lock! \*/

    return;

}

void snull\_rx(struct net\_device \*dev, struct snull\_packet \*pkt)

{

    struct sk\_buff \*skb;

    // 获取自定义的数据

    struct snull\_priv \*priv = netdev\_priv(dev);

    // 从内核申请skb

    skb = dev\_alloc\_skb(pkt->datalen + 2);

    // 如果申请失败

    if (!skb)

    {

        if (printk\_ratelimit())

            printk(KERN\_NOTICE "snull rx: low on mem - packet dropped\n");

        priv->stats.rx\_dropped++;

        goto out;

}

    // 向skb填充数据

    memcpy(skb\_put(skb, pkt->datalen), pkt->data, pkt->datalen);

    skb->dev = dev;

    skb->protocol = eth\_type\_trans(skb, dev);

    skb->ip\_summed = CHECKSUM\_UNNECESSARY;  // 不进行校检

    priv->stats.rx\_packets++;

priv->stats.rx\_bytes += pkt->datalen;

    // 向内核发送skb，通知有数据到达

    netif\_rx(skb);

out:

    return;

}

**轮询方式（NAPI）**

接口每秒会接受上千的数据包，使用中断方式会使系统性能下降

使用轮询方式需要实现poll函数

轮询方式基于中断方式

中断例程：

// 中断处理例程

static void snull\_regular\_interrupt(int irq, void \*dev\_id, struct pt\_regs \*regs)

{

    int statusword;

    struct snull\_priv \*priv;

    struct snull\_packet \*pkt = NULL;

struct net\_device \*dev = (struct net\_device \*)dev\_id;

    if (!dev)

        return;

    // 获取驱动自定义数据

    priv = netdev\_priv(dev);

spin\_lock(&priv->lock);

    // 假装获取设备状态，实际设备中我们应该从设备中获取

    statusword = priv->status;

    // 假装重置设备状态

priv->status = 0;

    // 接收中断

    if (statusword & SNULL\_RX\_INTR)

    {

        // 自定义函数，禁用之后的中断

        snull\_rx\_ints(dev, 0);

        // 调度接口任务，调用此方法使得poll函数在之后的某个时刻被调用

        netif\_rx\_schedule(dev);

}

    // 发送中断

    if (statusword & SNULL\_TX\_INTR)

    {

        // 发送完成

        priv->stats.tx\_packets++;

        priv->stats.tx\_bytes += priv->tx\_packetlen;

        // 释放skb

        dev\_kfree\_skb(priv->skb);

}

    spin\_unlock(&priv->lock);

    if (pkt)

        snull\_release\_buffer(pkt); /\* Do this outside the lock! \*/

    return;

}

Poll函数：

static int snull\_poll(struct net\_device \*dev, int \*budget)

{

    // budget是内核所能接收的数据包的个数

    // dev->quota是设备初始化阶段分配给设备的weight的值

    int npackets = 0, quota = min(dev->quota, \*budget);

    struct sk\_buff \*skb;

    // 获取自定义的数据

    struct snull\_priv \*priv = netdev\_priv(dev);

struct snull\_packet \*pkt;

    while (npackets < quota && priv->rx\_queue)

    {

        // 假装从设备获取数据

        pkt = snull\_dequeue\_buf(dev);

        // 从内核申请分配skb

        skb = dev\_alloc\_skb(pkt->datalen + 2);

        // 将数据拷贝到skb

        memcpy(skb\_put(skb, pkt->datalen), pkt->data, pkt->datalen);

        skb->dev = dev;

        skb->protocol = eth\_type\_trans(skb, dev);

        skb->ip\_summed = CHECKSUM\_UNNECESSARY;

        npackets++;

        // 将数据包传给内核

        netif\_receive\_skb(skb);

}

    \*budget -= npackets;

    dev->quota -= npackets;

    // 假装判断设备是否还有数据

    if (!priv->rx\_queue)

    {

        // 告诉内核接收完成，关闭轮询方式

        netif\_rx\_complete(dev);

        // 自定义函数，开启中断方式

        snull\_rx\_ints(dev, 1);

        return 0;   // 接收完成

    }

    // 接收未完成

    return 1;

}

**链路状态的改变**

驱动程序在检测链路状态发生改变时，可以通知系统

void netif\_carrier\_off(struct net\_device \*dev);     // 链路不可用

void netif\_carrier\_on(struct net\_device \*dev);      // 链路可用

**套接字缓存区（sk\_buff）**

**Sk\_buff结构**

struct net\_device \*dev：

接收或发送这个缓存的设备

unsigned char \*head：

指向分配内存的开始

unsigned char \*data：

有效数据的开始

unsigned char \*tail：

有效数据的结尾

unsigned char \*end：

分配内存的结尾

unsigned int len：

数据包中全部数据的长度

**sk\_buff操作**

struct sk\_buff \*alloc\_skb(unsigned int len, int priority)：

分配一个skb

void kfree\_skb(struct sk\_buff \*skb)：

释放skb

unsigned char \*skb\_put(struct sk\_buff \*skb, int len);

将tail的位置后移len的字节，更新tail位置和len的值，返回值为之前的tail的位置