LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 16 章　Linux 网络设备驱动

网络设备是完成用户数据包在网络媒介上发送和接收的设备，它将上层协议传递下来的数据包以特定的媒介访问控制方式进行发送，并将接收到的数据包传递给上层协议。

与字符设备和块设备不同，网络设备并不对应于/dev 目录下的文件，应用程序最终使用套接字（socket）完成与网络设备的接口。因而在网络设备身上并不能体现出“一切都是文件”的思想。

Linux 系统对网络设备驱动定义了 4 个层次，这 4 个层次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层和网络设备与媒介层。

在本章中，16.1 节讲解 Linux 网络设备驱动的层次结构，描述其 4 个层次各自的作用以及它们是如何协同合作以实现向下驱动网络设备硬件、向上提供数据包收发接口能力的。16.2～16.8 节主要讲解设备驱动功能层的各主要函数和数据结构，包括设备注册与注销、设备初始化、数据包收发函数、打开与释放函数等，在分析的基础上给出了抽象的设计模板。16.9 节是对前面各节讲解内容的例化，是对抽象设计模板的具体填充。

16.1 节与 16.2～16.8 节的内容是整体与部分的关系，而 16.1～16.8 节讲解的理论和设计模板与 16.9 节针对具体设备 CS8900 网卡进行的设计是抽象与具体的关系。

16.1 Linux 网络设备驱动的结构

Linux 网络设备驱动程序的体系结构如图 16.1 所示，从上到下可以划分为 4 层，依次为网络协议接口层、网络设备接口层、提供实际功能的设备驱动功能层以及网络设备与媒介层，这 4 层的作用如下所示。

- 网络协议接口层向网络层协议提供统一的数据包收发接口，不论上层协议为 ARP 还是 IP，都通过 dev_queue_xmit()函数发送数据，并通过 netif_rx()函数接收数据。这一层的存在使得上层协议独立于具体的设备。
- 网络设备接口层向协议接口层提供统一的用于描述具体网络设备属性和操作的结构体 net_device，该结构体是设备驱动功能层中各函数的容器。实际上，网络设备接口层从宏观上规划了具体操作硬件的设备驱动功能层的结构。
- 设备驱动功能层各函数是网络设备接口层 net_device 数据结构的具体成员，是驱使网络设备硬件完成相应动作的程序，它通过 hard_start_xmit()函数启动发送操作，并通过网络设备上的中断触发接收操作。
- 网络设备与媒介层是完成数据包发送和接收的物理实体，包括网络适配器和具体的传输媒介，网络适配器被设备驱动功能层中的函数物理上驱动。对于 Linux 系统而言，网络设备和媒介都可以是虚拟的。

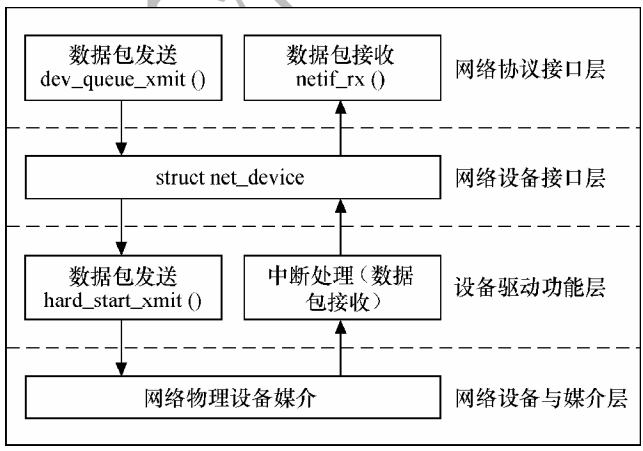


图 16.1　Linux 网络设备驱动程序的体系结构

在设计具体的网络设备驱动程序时，我们需要完成的主要工作是编写设备驱动功能层的相关函数以填充 net_device 数据结构的内容并将 net_device 注册入内核。

16.1.1 网络协议接口层

网络协议接口层最主要的功能是给上层协议提供了透明的数据包发送和接收接口。当上层 ARP 或 IP 协议需要发送数据包时，它将调用网络协议接口层的 dev_queue_xmit()函数发送该数据包，同时需传递给该函数一个指向 struct sk_buff 数据结构的指针。dev_queue_xmit()函数的原型为：

```
dev_queue_xmit (struct sk_buff * skb );
```

同样地，上层对数据包的接收也通过向 netif_rx()函数传递一个 struct sk_buff 数据结构的指针来完成。netif_rx()函数的原型为：

```
int netif_rx(struct sk_buff *skb);
```

sk_buff 结构体非常重要，它的含义为“套接字缓冲区”，用于在 Linux 网络子系统中的各层之间传递数据，是 Linux 网络子系统数据传递的“中枢神经”。

当发送数据包时，Linux 内核的网络处理模块必须建立一个包含要传输的数据包的 sk_buff，然后将 sk_buff 递交给下层，各层在 sk_buff 中添加不同的协议头直至交给网络设备发送。同样地，当网络设备从网络媒介上接收到数据包后，它必须将接收到的数据转换为 sk_buff 数据结构并传递给上层，各层剥去相应的协议头直至交给用户。

1．套接字缓冲区成员

参看 linux/skbuff.h 中的源代码，sk_buff 结构体包含的主要成员如下

（1）各层协议头 h、nh 和 mac。

sk_buff 结构体中定义了 3 个协议头以对应于网络协议的不同层次，这 3 个协议头为传输层 TCP/UDP（及 ICMP 和 IGMP）协议头 h、网络层协议头 nh 和链路层协议头 mac。这 3 个协议头数据结构都被定义为联合体，如代码清单 16.1 所示。

代码清单 16.1 sk_buff 结构体协议头

```
1   union{
2     struct tcphdr *th;        /* TCP 头部 */
3     struct udphdr *uh;          /* UDP 头部*/
4     struct icmphdr *icmph;      /* ICMP 头部*/
5     struct igmphdr *igmph;      /* IGMP 头部*/
6     struct iphdr *ipiph;        /* IP 头部*/
7     struct ipv6hdr *ipv6h;    /* IPv6 头部*/
8     unsigned char *raw;         /* 数据链路层头部*/
9   }h;
10
11  union{
12    struct iphdr *iph;          /* IP 头部*/
13    struct ipv6hdr *ipv6h;      /* IPv6 头部*/
14    struct arphdr *arph;        /*ARP 头部*/
15    unsigned char *raw;       /* 数据链路层头部*/
16  }nh;
17
18  union{
19    unsigned char *raw;         /* 数据链路层头部*/
20  } mac;
```

（2）数据缓冲区指针 head、data、tail 和 end。

Linux 内核必须分配用于容纳数据包的缓冲区，sk_buff 结构体定义了 4 个指向这片缓冲区不同位置的指针 head、data、tail 和 end。

head 指针指向内存中已分配的用于承载网络数据的缓冲区的起始地址，sk_buff 和相关数据块在分配之后，该指针的值就被固定了。

data 指针则指向对应当前协议层有效数据的起始地址。每个协议层的有效数据含义并不相同，各层的有效数据信息包含的内容如下。

- 对于传输层而言，用户数据和传输层协议头属于有效数据。
- 对于网络层而言，用户数据、传输层协议头和网络层协议头是其有效数据。
- 对于数据链路层而言，用户数据、传输层协议头、网络层协议头和链路层头部都属于有效数据。

因此，data 指针的值需随着当前拥有 sk_buff 的协议层的变化进行相应的移动。

tail 指针则指向对应当前协议层有效数据负载的结尾地址，与 data 指针对应。

end 指针指向内存中分配的数据缓冲区的结尾，与 head 指针对应。和 head 指针一样，sk_buff 被分配之后，end 指针的值也就固定不变了。

很显然，有效数据必须位于分配的数据缓冲区内，即（data,tail）区间位于（head,end）区间内，如图 16.2 所示。因此，head、data、tail 和 end 这 4 个指针间存在如下关系：head <- data <- tail <- end。

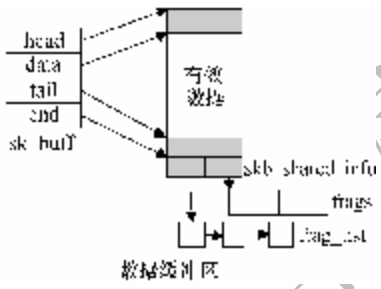


图 16.2　head、data、tail 和 end 指针

从图 16.2 中可以看出，end 指针所指地址数据缓冲区的末尾还包括一个 skb_shared_info 结构体的空间，这个结构体存放分隔存储的数据片段，意味着可以将数据包的有效数据分成几片存储在不同的内存空间中。图中的 frags 为分片数组，每一个分片的长度上限是一页。

（3）长度信息 len、data_len、truesize。

sk_buff 结构体中定义的 len 是指数据包有效数据的长度，包括协议头和负载（Payload）。

为了支持数据包的分片存放，sk_buff 中增加了 data_len 这个成员，它记录分片的数据长度。

truesize 表示缓存区的整体长度，置为 sizeof(struct sk_buff)加上传入 alloc_skb()函数或 dev_alloc_skb()函数（下文将要介绍）的长度，但不包括结构体 skb_shared_info 的长度。

2．套接字缓冲区操作

下面我们来分析套接字缓冲区涉及到的操作函数，Linux 套接字缓冲区支持分配、释放、指针移动等功能函数。

（1）分配。

Linux 内核用于分配套接字缓冲区的函数有：

```
struct sk_buff *alloc_skb(unsigned int len,int priority);
```

```
struct sk_buff *dev_alloc_skb(unsigned int len);
```

alloc_skb()函数分配一个套接字缓冲区和一个数据缓冲区，参数 len 为数据缓冲区的空间大小，以 16 字节对齐，参数 priority 为内存分配的优先级。

dev_alloc_skb()函数只是以 GFP_ATOMIC 优先级（代表分配过程不能被中断）调用上面的 alloc_skb()函数，并保存 skb->head 和 skb->data 之间的 16 个字节。

分配成功之后，因为还没有存放具体的网络数据包，所以 sk_buff 的 data、tail 指针都指向存储空间的起始地址 head，而 len 的大小则为 0。

（2）释放。

Linux 内核用于释放套接字缓冲区的函数有：

```
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
```

上述函数用于释放被 alloc_skb()函数分配的套接字缓冲区和数据缓冲区。

Linux 内核内部使用 kree_skb()函数，而网络设备驱动程序中则必须用 dev_kfree_skb()、dev_kfree_skb_irq()或 dev_kfree_skb_any()函数进行套接字缓冲区的释放。其中，dev_kfree_skb()函数用于非中断上下文，dev_kfree_skb_irq()函数用于中断上下文，而 dev_kfree_skb_any()函数则在中断和非中断上下文中皆可采用。

（3）指针移动。

Linux 套接字缓冲区中的数据缓冲区指针移动操作包括 put（放置）、push（推）、pull（拉）、reserve（保留）等。

① put 操作

数据缓冲区指针 put 操作以下列函数完成：

```
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
unsigned char *__skb_put(struct sk_buff *skb, unsigned int len);
```

上述函数将 tail 指针下移，增加 sk_buff 的 len 值，并返回 skb->tail 的当前值。skb_put()和__ skb_put()的区别在于前者会检测放入缓冲区的数据，而后者不会检查。这两个函数主要用于在缓冲区尾部添加数据。

② push 操作

数据缓冲区指针 push 操作以下列函数完成：

```
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
unsigned char *__skb_push(struct sk_buff *skb, unsigned int len);
```

与 skb_put()和_ _skb_put()不同，skb_push()和_ _skb_push()会将 data 指针上移，因此也要增加 sk_buff 的 len 值。push 操作在存储空间的头部增加一段可以存储网络数据包的空间，而 put 操作则在存储空间的尾部增加一段可以存储网络数据包的空间，因此主要用于在数据包发送时添加头部。skb_push()与_ _skb_push()的区别和 skb_put()和_ _skb_put()的区别类似。

③ pull 操作

数据缓冲区指针 pull 操作以下列函数完成：

```
unsigned char * skb_pull(struct sk_buff *skb, unsigned int len);
```

skb_pull()函数将 data 指针下移，并减小 skb 的 len 值。这个操作一般用于下层协议向上层协议移交数据包，使 data 指针指向上一层协议的协议头。

④ reserve 操作

数据缓冲区指针 reserve 操作以下列函数完成：

```
void skb_reserve(struct sk_buff *skb, unsigned int len);
```

skb_reserve()函数将 data 指针和 tail 指针同时下移，这个操作主要用于在存储空间的头部预留 len 长度的空隙。

下面我们以一个具体的 UDP 数据包接收的 Linux 处理流程为例来说明 sk_buff 的操作过程，这一过程的绝大部分工作都由 Linux 内核完成，驱动工程师只需完成涉及的数据链路层部分。

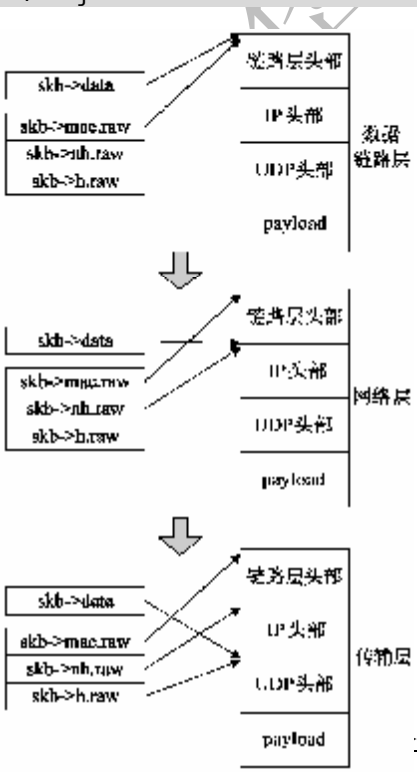
假设以太网适配器（以太网卡）收到了一个 UDP 数据包，Linux 从底层到应用层处理这一数据包的流程如下。

（1）网卡收到一个 UDP 数据包后，驱动程序需要创建一个 sk_buff 结构体和数据缓冲区，将收到的数据全部复制到 data 指向的空间，并将 skb->mac.raw 指向 data。此时，有效数据的开始位置是一个以太网头，skb->mac.raw 指向链路层的以太网头部。

代码清单 16.2 演示了驱动程序在接收到数据包后分配 sk_buff 和数据缓冲区并将数据包复制到缓冲区的过程。

代码清单 16.2　分配 sk_buff 和数据缓冲区

```
1   /* 分配新的套接字缓冲区和数据缓冲区 */
2   skb = dev_alloc_skb(length + 2);
3   if (skb == NULL)
4   {
5     .../* 分配失败 */
6     return ;
7   }
8
9   skb_reserve(skb, 2); /* 预留空间以使网络层协议头对齐 */
10
11  /* 将硬件上接收到的数据复制到数据缓冲区 */
12  readwords(ioaddr, RX_FRAME_PORT, skb_put(skb, length), length >> 1);
13  if (length &1)
14    skb->data[length - 1] = readword(ioaddr, RX_FRAME_PORT);
```



（2）数据链路层通过调用 skb_pull()函数剥掉以太网协议头，向网络层 IP 传递数据包。在剥离过程中，data 指针会下移一个以太网头部的长度 sizeof(struct ethhdr)，而 len 则减去 sizeof(struct ethhdr)。此时，有效数据的开始位置是一个 IP 头部，skb->nh.raw

指向 data，即 IP 头部，而 skb->mac.raw 仍指向以太网头部。

（3）网络层通过 skb_pull()函数剥掉 IP 头，向传输层 UDP 传递数据包。在剥离过程中，data 指针会下移一个 IP 头部的长度 sizeof(struct iphdr)，而 len 则减去 sizeof(struct iphdr)。此时，有效数据的开始位置是一个 UDP 头部，skb->h.raw 指向 data，即 UDP 首部，而 skb->nh.raw 指向 IP 头部，skb->mac.raw 仍指向以太网头部。

（4）应用程序在调用 recv()接收数据时，从 skb->data+sizeof(struct udphdr)的位置开始复制到应用层缓冲区。由此可见，UDP 头部到最后也没有被剥离。

如图 16.3 所示为上述过程中各指针的指向和移动过程。

16.1.2 网络设备接口层

网络设备接口层的主要功能是为千变万化的网络设备定义了统一、抽象的数据结构 net_device 结构体，以不变应万变，实现多种硬件在软件层次上的统一。

net_device 结构体在内核中指代一个网络设备，网络设备驱动程序只需通过填充 net_device 的具体成员并注册 net_device 即可实现硬件操作函数与内核的挂接。

net_device 本身是一个巨型结构体，包含网络设备的属性描述和操作接口。当我们编写网络设备驱动程序时，只需要了解其中的一部分。

（1）全局信息。

```
char name[IFNAMESIZ];
```

name 是网络设备的名称。

```
int (*init)(struct net_device *dev);
```

init 为设备初始化函数指针，如果这个指针被设置了，则网络设备被注册时将调用该函数完成对 net_device 结构体的初始化。但是，设备驱动程序可以不实现这个函数并将其赋值为 NULL。

（2）硬件信息。

```
unsigned long mem_end;
unsigned long mem_start;
```

mem_start 和 mem_end 分别定义了设备所使用的共享内存的起始和结束地址。

```
unsigned long base_addr;
unsigned char irq;
unsigned char if_port;
unsigned char dma;
```

base_addr 为网络设备 I/O 基地址。

irq 为设备使用的中断号。

if_port 指定多端口设备使用哪一个端口，该字段仅针对多端口设备。例如，如果设备同时支持 IF_PORT_10BASE2（同轴电缆）和 IF_PORT_10BASET（双绞线），则可使用该字段。

dma 指定分配给设备的 DMA 通道。

（3）接口信息。

```
unsigned short hard_header_len;
```

hard_header_len 是网络设备的硬件头长度，在以太网设备的初始化函数中，该成员被赋为 ETH_HLEN，即 14。

```
unsigned short type;
```

type 是接口的硬件类型。

```
unsigned mtu;
```

mtu 指最大传输单元（MTU）。

```
unsigned char dev_addr[MAX_ADDR_LEN];
unsigned char broadcast[MAX_ADDR_LEN];
```

dev_addr[ ]、broadcast[ ]无符号字符数组，分别用于存放设备的硬件地址和广播地址。对于以太网而言，这两个地址的长度都为 6 个字节。以太网设备的广播地址为 6 个 0xFF，而 MAC 地址需由驱动程序从硬件上读出并填充到 dev_addr[ ]中。

```
unsigned short flags;
```

flags 指网络接口标志，以 IFF_（interface flags）开头，部分标志由内核来管理，其他的在接口初始化时被设置以说明设备接口的能力和特性。接口标志包括 IFF_UP（当设备被激活并可以开始发送数据包时，内核设置该标志）、IFF_AUTOMEDIA（设备可在多种媒介间切换）、IFF_BROADCAST（允许广播）、IFF_DEBUG（调试模式，可用于控制 printk 调用的详细程度）、IFF_LOOPBACK（回环）、IFF_MULTICAST（允许组播）、IFF_NOARP（接口不能执行 ARP）、IFF_POINTOPOINT（接口连接到点到点链路）等。

（4）设备操作函数。

```
int (*open)(struct net_device *dev);
int (*stop)(struct net_device *dev);
```

open()函数的作用是打开网络接口设备，获得设备需要的 I/O 地址、IRQ、DMA 通道等。stop()函数的作用是停止网络接口设备，与 open()函数的作用相反。

```
int  (*hard_start_xmit) (struct sk_buff *skb,struct net_device *dev);
```

hard_start_xmit()函数会启动数据包的发送，当系统调用驱动程序的 hard_start_xmit()函数时，需要向其传入一个 sk_buff 结构体指针，以使得驱动程序能获取从上层传递下来的数据包。

```
void (*tx_timeout)(struct net_device *dev);
```

当数据包的发送超时时，tx_timeout ()函数会被调用，该函数需采取重新启动数据包发送过程或重新启动硬件等策略来恢复网络设备到正常状态。

```
int  (*hard_header) (struct sk_buff *skb,
                     struct net_device *dev,
                     unsigned short type,
                     void *daddr,
                     void *saddr,
                     unsigned len);
```

hard_header()函数完成硬件帧头填充，返回填充的字节数。传入该函数的参数包括 sk_buff 指针、设备指针、协议类型、目的地址、源地址以及数据长度。对于以太网设备而言，将内核提供的 eth_header()函数赋值给 hard_header 指针即可。

```
struct net_device_stats* (*get_stats)(struct net_device *dev);
```

get_stats()函数用于获得网络设备的状态信息，它返回一个 net_device_stats 结构体。net_device_stats 结构体保存了网络设备详细的流量统计信息，如发送和接收到的

数据包数、字节数等，详见 16.8 节。

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
int (*set_config)(struct net_device *dev, struct ifmap *map);
int (*set_mac_address)(struct net_device *dev, void *addr);
```

do_ioctl()函数用于进行设备特定的 I/O 控制。

set_config()函数用于配置接口，可用于改变设备的 I/O 地址和中断号。

set_mac_address()函数用于设置设备的 MAC 地址。

```
int (*poll)( struct net_device *dev,int quota);
```

对于 NAPI（网络中断缓和）兼容的设备驱动，将以轮询方式操作接口，接收数据包。NAPI 是 Linux 系统上采用的一种提高网络处理效率的技术，它的核心概念就是不采用中断的方式读取数据包，而是采用首先借助中断唤醒数据包接收的服务程序，然后以轮询方式获取数据包。

net_device 结构体的上述成员需要在设备初始化时被填充，详见 16.3 节。

（5）辅助成员。

```
unsigned long trans_start;
unsigned long last_rx;
```

trans_start 记录最后的数据包开始发送时的时间戳，last_rx 记录最后一次接收到数据包时的时间戳，这两个时间戳记录的都是 jiffies，驱动程序应维护这两个成员。

```
void *priv;
```

priv 为设备的私有信息指针，与 filp->private_data 的地位相当。设备驱动程序中应该以 netdev_priv()函数获得该指针。

```
spinlock_t xmit_lock;
int xmit_lock_owner;
```

xmit_lock 是避免 hard_start_xmit()函数被同时多次调用的自旋锁。xmit_lock_owner 则指当前拥有 xmit_lock 自旋锁的 CPU 的编号。

16.1.3 设备驱动功能层

net_device 结构体的成员（属性和函数指针）需要被设备驱动功能层的具体数值和函数赋予。对于具体的设备 xxx，工程师应该编写设备驱动功能层的函数，这些函数形如 xxx_open()、xxx_stop()、xxx_tx()、xxx_ hard_header()、xxx_get_stats()、xxx_tx_timeout()、xxx_poll()等。

由于网络数据包的接收可由中断引发，设备驱动功能层中另一个主体部分将是中断处理函数，它负责读取硬件上接收的数据包并传送给上层协议，可能包含 xxx_interrupt()和 xxx_rx()函数，前者完成中断类型判断等基本的工作，后者则需完成数据包的生成和递交上层等复杂工作。

16.2～16.8 节将对上述 xxx_xxx()函数进行详细地分析并给出参考设计模板。

对于特定的设备，我们还可以定义其相关私有数据和操作，并封装为一个私有信息结构体 xxx_private，让其指针被赋值给 net_device 的 priv 成员。xxx_private 结构体中可包含设备特殊的属性和操作、自旋锁与信号量、定时器以及统计信息等，由工程师自定义。

16.1.4 网络设备与媒介层

网络设备与媒介层直接对应于实际的硬件设备。为了给设备的物理配置和寄存器

操作一个更一般的描述，我们可以定义一组宏和一组访问设备内部寄存器的函数，具体的宏和函数与特定的硬件紧密相关。代码清单 16.3 所示为相应的设计范例。

代码清单 16.3　网络设备底层硬件操作

```
1   /*  寄存器定义 */
2   #define DATA_REG 0x0004
3   #define CMD_REG 0x0008
4
5   /* 寄存器读写函数 */
6   static u16 xxx_readword(u32 base_addr, int portno)
7   {
8     ...  /*读取寄存器的值并返回*/
9   }
10
11  static void xxx_writeword(u32 base_addr, int portno, u16 value)
12  {
13    ...  /*向寄存器写入数值*/
14  }
```

16.2 网络设备驱动的注册与注销

网络设备驱动的注册与注销使用成对出现的 register_netdev()和 unregister_netdev()函数完成，这两个函数的原型为：

```
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
```

这两个函数都接收一个 net_device 结构体指针为参数，可见 net_device 数据结构在网络设备驱动中的核心地位。

net_device 的生成和成员的赋值并非一定要由工程师逐个亲自动手完成，可以利用下面的函数帮助我们填充：

```
struct net_device *alloc_netdev(int sizeof_priv, const char *name,
void(*setup)
    (struct net_device*));
struct net_device *alloc_etherdev(int sizeof_priv);
```

alloc_netdev()函数生成一个 net_device 结构体，对其成员赋值并返回该结构体的指针。第一个参数为设备私有成员的大小，第二个参数为设备名，第三个参数为 net_device 的 setup()函数指针。setup()函数接收的参数也为 struct net_device 指针，用于预置 net_device 成员的值。

alloc_etherdev()是 alloc_netdev()针对以太网的“快捷”函数，这从 alloc_etherdev()的源代码可以看出，如代码清单 16.4 所示。

代码清单 16.4　alloc_etherdev()函数

```
1  struct net_device *alloc_etherdev(int sizeof_priv)
2  {
3    /* 以 ether_setup 为 alloc_netdev 的 setup 参数 */
4    return alloc_netdev(sizeof_priv, "eth%d", ether_setup);
5  }
```

上述代码中的第 4 行传入 alloc_netdev ()函数的第三个参数为 ether_setup()函数地址，ether_setup()是由 Linux 内核提供的一个对以太网设备 net_device 结构体中公有成员快速赋值的函数。

完成与 alloc_enetdev()和 alloc_etherdev()函数相反功能，即释放 net_device 结构体的函数为：

```
void free_netdev(struct net_device *dev);
```

net_device 结构体的分配和网络设备驱动注册需在网络设备驱动程序的模块加载函数中进行，而 net_device 结构体的释放和网络设备驱动的注销则需在模块卸载函数中完成，如代码清单 16.5 所示。

代码清单 16.5　网络设备驱动程序的模块加载函数模板

```
1  int xxx_init_module(void)
2  {
3    ...
4    /* 分配 net_device 结构体并对其成员赋值 */
5    xxx_dev   =   alloc_netdev(sizeof(struct   xxx_priv),   "sn%d",
xxx_init);
6    if (xxx_dev == NULL)
7    ... /* 分配 net_device 失败 */
8
9    /* 注册 net_device 结构体 */
10     if ((result = register_netdev(xxx_dev)))
11     ...
12 }
13
14 void xxx_cleanup(void)
15 {
16   ...
17   /* 注销 net_device 结构体 */
18   unregister_netdev(xxx_dev);
19   /* 释放 net_device 结构体 */
20   free_netdev(xxx_dev);
21 }
```

16.3 网络设备的初始化

网络设备的初始化主要需要完成如下几个方面的工作。

- 进行硬件上的准备工作，检查网络设备是否存在，如果存在，则检测设备所

使用的硬件资源。

- 进行软件接口上的准备工作，分配 net_device 结构体并对其数据和函数指针成员赋值。
- 获得设备的私有信息指针并初始化其各成员的值。如果私有信息中包括自旋锁或信号量等并发或同步机制，则需对其进行初始化。

对 net_device 结构体成员及私有数据的赋值都可能需要与硬件初始化工作协同进行，即硬件检测出了相应的资源，需要根据检测结果填充 net_device 结构体成员和私有数据。

一个网络设备驱动初始化函数的模板如代码清单 16.6 所示，具体的设备驱动初始化函数并不一定完全和本模板一样，但是其本质过程是一致的。

代码清单 16.6　网络设备驱动的初始化函数模板

```
1   void xxx_init(struct net_device *dev)
2   {
3     /*设备的私有信息结构体*/
4     struct xxx_priv *priv;
5
6     /* 检查设备是否存在和设备所使用的硬件资源 */
7     xxx_hw_init();
8
9     /* 初始化以太网设备的公用成员 */
10    ether_setup(dev);
11
12    /*设置设备的成员函数指针*/
13    dev->open = xxx_open;
14    dev->stop = xxx_release;
15    dev->set_config = xxx_config;
16    dev->hard_start_xmit = xxx_tx;
17    dev->do_ioctl = xxx_ioctl;
18    dev->get_stats = xxx_stats;
19    dev->change_mtu = xxx_change_mtu;
20    dev->rebuild_header = xxx_rebuild_header;
21    dev->hard_header = xxx_header;
22    dev->tx_timeout = xxx_tx_timeout;
23    dev->watchdog_timeo = timeout;
24
25    /*如果使用 NAPI，设置 pool 函数*/
26    if (use_napi)
27    {
28      dev->poll = xxx_poll;
29    }
30
31    /* 取得私有信息，并初始化它*/
32    priv = netdev_priv(dev);
33    ...  /* 初始化设备私有数据区 */
34  }
```

上述代码第 7 行的 xxx_hw_init()函数完成硬件相关的初始化操作，如下所示。

- 探测 xxx 网络设备是否存在。探测的方法类似于数学上的“反证法”，即先假设存在设备 xxx，访问该设备，如果设备的表现与预期的一致，就确定设备存在；否则，假设错误，设备 xxx 不存在。
- 探测设备的具体硬件配置。一些设备驱动编写得非常通用，对于同类的设备使用统一的驱动，我们需要在初始化时探测设备的具体型号。另外，即便是同一设备，在硬件上的配置也可能不一样，我们也可以探测设备所使用的硬件资源。
- 申请设备所需要的硬件资源，如用 request_region()函数进行 I/O 端口的申请等，但是这个过程可以放在设备的打开函数 xxx_open()中完成。

16.4 网络设备的打开与释放

网络设备的打开函数需要完成如下工作。

- 使能设备使用的硬件资源，申请 I/O 区域、中断和 DMA 通道等。
- 调用 Linux 内核提供的 netif_start_queue()函数，激活设备发送队列。

网络设备的关闭函数需要完成如下工作。

- 调用 Linux 内核提供的 netif_stop_queue()函数，停止设备传输包。
- 释放设备所使用的 I/O 区域、中断和 DMA 资源。

Linux 内核提供的 netif_start_queue()和 netif_stop_queue()两个函数的原型为：

```
void netif_start_queue(struct net_device *dev);
void netif_stop_queue (struct net_device *dev);
```

根据以上分析，可得出如代码清单 16.7 所示的网络设备打开和释放函数的模板。

代码清单 16.7　网络设备打开和释放函数模板

```
1   int xxx_open(struct net_device *dev)
2   {
3     /* 申请端口、IRQ 等，类似于 fops->open */
4     ret = request_irq(dev->irq, &xxx_interrupt, 0, dev->name, dev);
5     ...
6     netif_start_queue(dev);
7     ...
8   }
9
10  int xxx_release(struct net_device *dev)
11  {
12    /* 释放端口、IRQ 等，类似于 fops->close */
13    free_irq(dev->irq, dev);
14    ...
15    netif_stop_queue(dev); /* can't transmit any more */
16    ...
17  }
```

16.5 数据发送流程

从 16.1 节网络设备驱动程序的结构分析可知，Linux 网络子系统在发送数据包时，会调用驱动程序提供的 hard_start_transmit()函数，该函数用于启动数据包的发送。在设备初始化的时候，这个函数指针需被初始化指向设备的 xxx_tx()函数。

网络设备驱动完成数据包发送的流程如下。

（1）网络设备驱动程序从上层协议传递过来的 sk_buff 参数获得数据包的有效数据和长度，将有效数据放入临时缓冲区。

（2）对于以太网，如果有效数据的长度小于以太网冲突检测所要求数据帧的最小长度 ETH_ZLEN，则给临时缓冲区的末尾填充 0。

（3）设置硬件的寄存器，驱使网络设备进行数据发送操作。

完成以上 3 个步骤的网络设备驱动程序的数据包发送函数的模板如代码清单 16.8 所示。

代码清单 16.8　网络设备驱动程序的数据包发送函数模板

```
1   int xxx_tx(struct sk_buff *skb, struct net_device *dev)
2   {
3     int len;
4     char *data, shortpkt[ETH_ZLEN];
5    /* 获得有效数据指针和长度 */
6     data = skb->data;
7     len = skb->len;
8     if (len < ETH_ZLEN)
9     {
10       /* 如果帧长小于以太网帧最小长度，补0 */
11      memset(shortpkt, 0, ETH_ZLEN);
12      memcpy(shortpkt, skb->data, skb->len);
13      len = ETH_ZLEN;
14      data = shortpkt;
15    }
16
17    dev->trans_start = jiffies; /* 记录发送时间戳 */
18
19    /* 设置硬件寄存器让硬件把数据包发送出去 */
20    xxx_hw_tx(data, len, dev);
21    ...
22  }
```

当数据传输超时时，意味着当前的发送操作失败，此时，数据包发送超时处理函数 xxx_tx_ timeout()将被调用。这个函数需要调用 Linux 内核提供的 netif_wake_queue()

函数重新启动设备发送队列，如代码清单 16.9 所示。

代码清单 16.9　网络设备驱动程序的数据包发送超时函数模板

```
1  void xxx_tx_timeout(struct net_device *dev)
2  {
3    ...
4    netif_wake_queue(dev);  /* 重新启动设备发送队列 */
5  }
```

16.6 数据接收流程

网络设备接收数据的主要方法是由中断引发设备的中断处理函数，中断处理函数判断中断类型，如果为接收中断，则读取接收到的数据，分配 sk_buffer 数据结构和数据缓冲区，将接收到的数据复制到数据缓冲区，并调用 netif_rx()函数将 sk_buffer 传递给上层协议。代码清单 16.10 所示为完成这一过程的函数模板。

代码清单 16.10　网络设备驱动的中断处理函数模板

```
1  static void xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
2  {
3    ...
4    switch (status &ISQ_EVENT_MASK)
5    {
6      case ISQ_RECEIVER_EVENT:
7        /* 获取数据包 */
8        xxx_rx(dev);
9        break;
10       /* 其他类型的中断 */
11   }
12 }
13 static void xxx_rx(struct xxx_device *dev)
14 {
15   ...
16   length = get_rev_len (...);
17   /* 分配新的套接字缓冲区 */
18   skb = dev_alloc_skb(length + 2);
19
```

```
20    skb_reserve(skb, 2); /* 对齐 */
21    skb->dev = dev;
22
23    /* 读取硬件上接收到的数据 */
24    insw(ioaddr + RX_FRAME_PORT, skb_put(skb, length), length >> 1);
25    if (length &1)
26      skb->data[length - 1] = inw(ioaddr + RX_FRAME_PORT);
27
28    /* 获取上层协议类型 */
29    skb->protocol = eth_type_trans(skb, dev);
30
31    /* 把数据包交给上层 */
32    netif_rx(skb);
33
34    /* 记录接收时间戳 */
35    dev->last_rx = jiffies;
36    ...
37  }
```

从上述代码的第 4～7 行可以看出，当设备的中断处理程序判断中断类型为数据包接收中断时，它调用第 13～37 定义的 xxx_rx()函数完成更深入的数据包接收工作。xxx_rx()函数代码中的第 16 行从硬件读取到接收数据包有效数据的长度，第 17～20 行分配 sk_buff 和数据缓冲区，第 23～26 行读取硬件上接收到的数据并放入数据缓冲区，第 28～29 行解析接收数据包上层协议的类型，最后，第 31～32 行代码将数据包上交给上层协议。

如果是 NAPI 兼容的设备驱动，则可以通过 poll 方式接收数据包。这种情况下，我们需要为该设备驱动提供 xxx_poll()函数，如代码清单 16.11 所示。

代码清单 16.11　网络设备驱动的 poll 函数模板

```
1   static int xxx_poll(struct net_device *dev, int *budget)
2   {
3     int npackets = 0, quota = min(dev->quota,  *budget);
4     struct sk_buff *skb;
5     struct xxx_priv *priv = netdev_priv(dev);
6     struct xxx_packet *pkt;
7
8     while (npackets < quota && priv->rx_queue)
9     {
10      /*从队列中取出数据包*/
11      pkt = xxx_dequeue_buf(dev);
12
13      /*接下来的处理，和中断触发的数据包接收一致*/
14      skb = dev_alloc_skb(pkt->datalen + 2);
15      if (!skb)
16      {
17        ...
18        continue;
19      }
```

```
    20      skb_reserve(skb, 2);
    21      memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
    22      skb->dev = dev;
    23      skb->protocol = eth_type_trans(skb, dev);
    24      /*调用 netif_receive_skb 而不是 net_rx 将数据包交给上层协议*/
    25      netif_receive_skb(skb);
    26
    27      /*更改统计数据 */
    28      priv->stats.rx_packets++;
    29      priv->stats.rx_bytes += pkt->datalen;
    30      xxx_release_buffer(pkt);
    31    }
    32    /* 处理完所有数据包*/
    33    *budget -= npackets;
    34    dev->quota -= npackets;
    35
    36    if (!priv->rx_queue)
    37    {
    38      netif_rx_complete(dev);
    39      xxx_enable_rx_int (…); /* 再次使能网络设备的接收中断 */
    40      return 0;
    41    }
    42
    43    return 1;
    44  }
```

上述代码第 3 行中的 dev->quota 是当前 CPU 能够从所有接口中接收数据包的最大数目，budget 是在初始化阶段分配给接口的 weight 值，poll 函数必须接收两者之间的最小值，表示轮询函数本次要处理的数据包个数。第 8 行的 while()循环读取设备的接收缓冲区，读取数据包并提交给上层。这个过程和中断触发的数据包接收过程一致，但是最后使用 netif_receive_skb()函数而非 netif_rx()函数将数据包提交给上层。这里体现出了中断处理机制和轮询机制之间的差别。

当网络设备接收缓冲区中的数据包都被读取完后（即 priv->rx_queue 为 NULL），一个轮询过程结束，第 38 行代码调用 netif_rx_complete()把当前指定的设备从 poll 队列中清除，第 39 行代码再次启动网络设备的接收中断。

虽然 NAPI 兼容的设备驱动以 poll 方式接收数据包，但是仍然需要首次数据包接收中断来触发 poll 过程。与数据包的中断接收方式不同的是，以轮询方式接收数据包时，当第一次中断发生后，中断处理程序要禁止设备的数据包接收中断，如代码清单 16.12 所示。

代码清单 16.12　网络设备驱动的 poll 中断处理函数模板

```
    1   static  void  xxx_poll_interrupt(int  irq,  void  *dev_id,  struct
pt_regs *regs)
    2   {
    3     switch (status &ISQ_EVENT_MASK)
    4     {
    5       case ISQ_RECEIVER_EVENT:
    6         … /* 获取数据包 */
    7         xxx_disable_rx_int(...);  /* 禁止接收中断 */
    8         netif_rx_schedule(dev);
    9         break;
```

```
10      …  /* 其他类型的中断 */
11    }
12  }
```

上述代码第 8 行的 netif_rx_schedule()函数被轮询方式驱动的中断程序调用，将设备的 poll 方法添加到网络层的 poll 处理队列中，排队并且准备接收数据包，最终触发一个 NET_RX_SOFTIRQ 软中断，通知网络层接收数据包。图 16.4 所示为 NAPI 驱动程序各部分的调用关系。

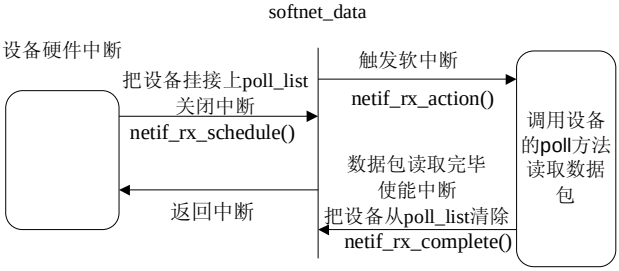


图 16.4　NAPI 调用关系

16.7 网络连接状态

网络适配器硬件电路可以检测出链路上是否有载波，载波反映了网络的连接是否正常。网络设备驱动可以通过 netif_carrier_on()和 netif_carrier_off()函数改变设备的连接状态，如果驱动检测到连接状态发生变化，也应该以 netif_carrier_on()和 netif_carrier_off()函数显式地通知内核。

除了 netif_carrier_on()和 netif_carrier_off()函数以外，另一个函数 netif_carrier_ok()可用于向调用者返回链路上的载波信号是否存在。

这几个函数都接收一个 net_device 设备结构体指针为参数，原型分别为：

```
void netif_carrier_on(struct net_device *dev);
void netif_carrier_off(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
```

网络设备驱动程序中往往设置一个定时器来对链路状态进行周期性地检查。当定时器到期之后，在定时器处理函数中读取物理设备的相关寄存器获得载波状态，从而更新设备的连接状态，如代码清单 16.13 所示。

代码清单 16.13　网络设备驱动用定时器周期检查链路状态

```
1   static void xxx_timer(unsigned long data)
2   {
3     struct net_device *dev = (struct net_device*)data;
4     u16 link;
5     …
6     if (!(dev->flags &IFF_UP))
7     {
8       goto set_timer;
9     }
```

```
10
11    /* 获得物理上的连接状态 */
12    if (link = xxx_chk_link(dev))
13    {
14      if (!(dev->flags &IFF_RUNNING))
15      {
16        netif_carrier_on(dev);
17        dev->flags |= IFF_RUNNING;
18        printk(KERN_DEBUG "%s: link up\n", dev->name);
19      }
20    }
21    else
22    {
23      if (dev->flags &IFF_RUNNING)
24      {
25        netif_carrier_off(dev);
26        dev->flags &= ~IFF_RUNNING;
27        printk(KERN_DEBUG "%s: link down\n", dev->name);
28      }
29    }
30
31    set_timer:
32    priv->timer.expires = jiffies + 1 * HZ;
33    priv->timer.data = (unsigned long)dev;
34    priv->timer.function = &xxx_timer; /* timer handler */
35    add_timer(&priv->timer);
36  }
```

上述代码第 12 行调用的 xxx_chk_link()函数用于读取网络适配器硬件的相关寄存器以获得链路连接状态，具体实现由硬件决定。当链路连接上时，第 16 行的 netif_carrier_on()函数显式地通知内核链路正常；反之，则第 25 行的 netif_carrier_off()则同样显式地通知内核链路失去连接。

此外，从上述源代码还可以看出，定时器处理函数会不停地利用第 31～35 行代码启动新的定时器以实现周期检测的目的。那么最初启动定时器的地方在哪里呢？很显然，它最适合在设备的打开函数中完成，如代码清单 16.14 所示。

代码清单 16.14　在网络设备驱动的打开函数中初始化定时器

```
1   static int xxx_open(struct net_device *dev)
2   {
3     struct xxx_priv *priv = (struct xxx_priv*)dev->priv;
4
5     ...
6     priv->timer.expires = jiffies + 3 * HZ;
7     priv->timer.data = (unsigned long)dev;
8     priv->timer.function = &xxx_timer; /* timer handler */
9     add_timer(&priv->timer);
10    ...
```

```
11  }
```

16.8 参数设置和统计数据

在网络设备的驱动程序中还提供一些方法供系统对设备的参数进行设置或读取设备相关的信息。

当用户调用 ioctl()函数，并指定 SIOCSIFHWADDR 命令时，意味着要设置这个设备的 MAC 地址。一般来说，这个操作没有太大的意义。设置网络设备的 MAC 地址可用如代码清单 16.15 所示的模板。

代码清单 16.15　设置网络设备的 MAC 地址

```
1   static int set_mac_address(struct net_device *dev, void *addr)
2   {
3       if (netif_running(dev))
4           return -EBUSY;  /* 设备忙 */
5
6       /* 设置以太网的 MAC 地址 */
7       xxx_set_ mac(dev, addr);
8
9       return 0;
10  }
```

上述程序首先用 netif_running()宏判断设备是否正在运行，如果是，则意味着设备忙，此时不允许设置 MAC 地址；否则，调用 xxx_set_ mac()函数在网络适配器硬件内写入新的 MAC 地址。这要求设备在硬件上支持 MAC 地址的修改，而实际上，许多设备并不提供修改 MAC 地址的接口。

netif_running()宏的定义为：

```
#define netif_running(dev) (dev->flags & IFF_UP)
```

当用户调用 ioctl()函数时，若命令为 SIOCSIFMAP（如在控制台中运行网络配置命令 ifconfig 就会引发这一调用），系统会调用驱动程序的 set_config()函数。

系统会向 set_config()函数传递一个 ifmap 结构体，该结构体中主要包含用户欲设置的设备要使用的 I/O 地址、中断等信息。注意，并非 ifmap 结构体中给出的所有修改都是可以被接受的。实际上，大多数设备并不宜包含 set_config()函数。set_config()函数的例子如代码清单 16.16 所示。

代码清单 16.16　网络设备驱动的 set_config 函数模板

```
1   int xxx_config(struct net_device *dev, struct ifmap *map)
2   {
3     if (netif_running(dev))   /* 不能设置一个正在运行状态的设备 */
4       return  - EBUSY;
5
6     /* 假设不允许改变 I/O 地址 */
7     if (map->base_addr != dev->base_addr)
```

```
8     {
9       printk(KERN_WARNING "xxx: Can't change I/O address\n");
10      return  - EOPNOTSUPP;
11    }
12
13    /* 假设允许改变 IRQ */
14    if (map->irq != dev->irq)
15    {
16      dev->irq = map->irq;
17    }
18
19    return 0;
20  }
```

上述代码中的 set_config()函数接受 IRQ 的修改，拒绝设备 I/O 地址的修改。具体的设备是否接收这些信息的修改，要视硬件的设计而定。

如果用户调用 ioctl()时，命令类型在 SIOCDEVPRIVATE 和 SIOCDEVPRIVATE+15 之间，系统会调用驱动程序的 do_ioctl()函数，进行设备专用数据的设置。这个设置大多数情况下也并不需要。

驱动程序还应提供 get_stats()函数用以向用户反馈设备状态和统计信息，该函数返回的是一个 net_device_stats 结构体，如代码清单 16.17 所示。

代码清单 16.17　网络设备驱动的 get_stats 函数模板

```
1   struct net_device_stats *xxx_stats(struct net_device *dev)
2   {
3     struct xxx_priv *priv = netdev_priv(dev);
4     return  &priv->stats;
5   }
```

net_device_stats 结构体定义在内核的 include/linux/netdevice.h 文件中，它包含了比较完整的统计信息，如代码清单 16.18 所示。

代码清单 16.18　net_device_stats 结构体

```
1   struct net_device_stats
2   {
3     unsigned long rx_packets;         /* 收到的数据包数 */
4     unsigned long tx_packets;         /* 发送的数据包数 */
5     unsigned long rx_bytes;       /* 收到的字节数  */
6     unsigned long tx_bytes;       /* 发送的字节数  */
7     unsigned long rx_errors;          /* 收到的错误数据包数 */
8     unsigned long tx_errors;          /* 发生发送错误的数据包数 */
9     ...
10  };
```

上述代码清单只是列出了 net_device_stats 包含的主项目统计信息，实际上，这些项目还可以进一步细分，net_device_stats 中的其他信息给出了更详细的子项目统计，详见 Linux 源代码。

net_device_stats 结构体适宜包含在设备的私有信息结构体中，而其中统计信息的修改则应该在设备驱动的与发送和接收相关的具体函数中完成，这些函数包括中断处理程序、数据包发送函数、数据包发送超时函数和数据包接收相关函数等。我们应该在这些函数中添加相应的代码，如代码清单 16.19 所示。

代码清单 16.19　net_device_stats 结构体中统计信息的维护

```
1  /* 发送超时函数 */
2  void xxx_tx_timeout(struct net_device *dev)
3  {
4    struct xxx_priv *priv = netdev_priv(dev);
5    ...
6    priv->stats.tx_errors++;  /* 发送错误包数加 1 */
7    ...
8  }
9
10 /* 中断处理函数 */
11 static void xxx_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
12 {
13   switch (status &ISQ_EVENT_MASK)
14   {
15     ...
16     case ISQ_TRANSMITTER_EVENT:  /
17       priv->stats.tx_packets++;   /* 数据包发送成功，tx_packets 信息加
1 */
18       netif_wake_queue(dev); /* 通知上层协议 */
19       if ((status &(TX_OK | TX_LOST_CRS | TX_SQE_ERROR |
20       TX_LATE_COL | TX_16_COL)) != TX_OK) /*读取硬件上的出错标志*/
21       {
22         /* 根据错误的不同情况，对 net_device_stats 的不同成员加 1 */
23         if ((status &TX_OK) == 0)
24           priv->stats.tx_errors++;
25         if (status &TX_LOST_CRS)
26           priv->stats.tx_carrier_errors++;
27         if (status &TX_SQE_ERROR)
28           priv->stats.tx_heartbeat_errors++;
29         if (status &TX_LATE_COL)
30           priv->stats.tx_window_errors++;
31         if (status &TX_16_COL)
32           priv->stats.tx_aborted_errors++;
33       }
34       break;
35     case ISQ_RX_MISS_EVENT:
36       priv->stats.rx_missed_errors += (status >> 6);
37       break;
38     case ISQ_TX_COL_EVENT:
39       priv->stats.collisions += (status >> 6);
40       break;
41   }
42 }
```

上述代码的第 6 行意味着在发送数据包超时时，将发生发送错误的数据包数增 1。

而第 13～41 行则意味着当网络设备中断产生时，中断处理程序读取硬件的相关信息以决定修改 net_device_ stats 统计信息中的哪些项目和子项目，并将相应的项目增 1。

16.9 CS8900 网卡设备驱动实例

16.9.1 CS8900 网卡硬件描述

CS8900 芯片是 Cirrus Logic 公司生产的一种以太网处理芯片，提供 ISA 总线接口，内部集成了 E^2PROM 控制器、802.3 MAC 控制器、片内 RAM 和 10BASE-T 收发滤波器，支持 I/O 和内存模式两种模式，可采用 DMA 方式与主机以 16 位或 8 位数据长度进行数据交换。图 16.5 所示为 CS8900 的内部结构框架。

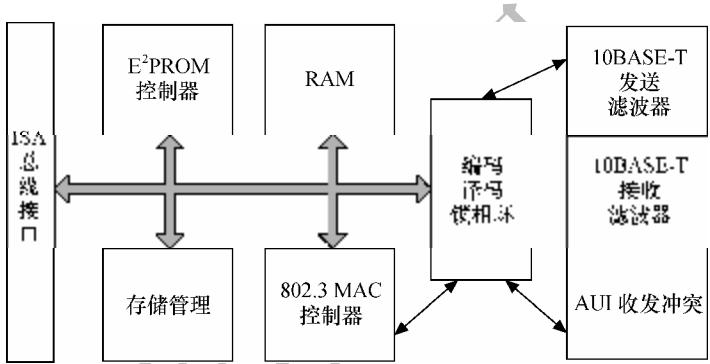


图 16.5 CS8900 以太网芯片的内部结构

在一般的嵌入式系统硬件原理设计时，CS8900 可以直接挂接在 CPU 的存储器总线上。将 CS8900A 网卡连接在 S3C2410A ARM 处理器的数据总线、控制总线和地址总线上的原理如图 16.6 所示。

CS8900 采用了独特的 PacketPage（封包页）机制来实现总线对其内部 4KB RAM 空间的访问。位于 CS8900 PacketPage RAM 空间内的地址只需要通过直接映射到 CPU 内存或 I/O 空间的 16 字节连续的可直接访问的端口中的 PacketPage 指针端口和 PacketPage 数据端口即可间接访问，最大程度地减少了 CS8900 对硬件资源的占用。

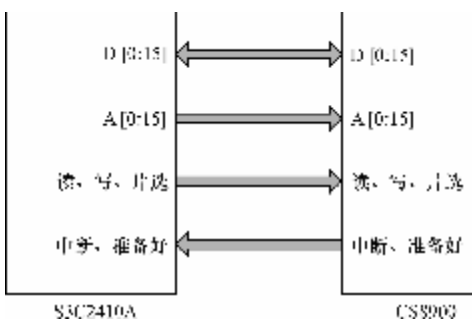


图 16.6 CS8900 与 S3C2410A 连接原理

当 CS8900 处于 I/O 模式下时，可以通过以下几个 PacketPage 空间内的寄存器来控制 CS8900 的行为（括号内给出的是寄存器地址相对于 PacketPage 基地址的偏移）。

- LINECTL(0112H)。

LINECTL 决定 CS8900 的基本配置和物理接口，可选择使用 10BASE-T 接口、AUI 接口或者自动选择。

- RXCTL(0104H)。

RXCTL 控制 CS8900 接收特定数据包，控制是否接收多播、广播和单播包。

- RXCFG(0102H)。

RXCFG 控制 CS8900 接收到特定数据包后引发接收中断，并控制是否使用接收 DMA 和 CRC 校验。

- BUSCT(0116H)。

BUSCT 可控制芯片的工作模式、DMA 方式、是否使能外部中断引脚。

- BUSST(0138H)。

BUSST 标志网络设备的发送状态，如设备是否准备好发送。

- ISQ(0120H)。

ISQ 是网卡芯片的中断状态寄存器。

当 CS8900 复位时，16 字节连续的端口被分配在偏移地址 300H 处，这个地址就是 CS8900 的 I/O 基地址。当然，I/O 基地址可以通过 PacketPage 空间内的相对于 PacketPage 基地址偏移 0020H 地址处的 I/O 基地址寄存器重新设置，但一般没有这个修改的必要。

通过下面的 3 个寄存器可完成数据包的发送和接收（括号内为寄存器地址相对 I/O 基地址的偏移）。

- PORT0(0000H)。

发送或接收数据包时，CPU 都通过 PORT0 写入或读取数据。

- TXCMD(0004H)。

发送控制寄存器，填写相应的发送命令后会引发数据包的发送。

- TXLENG(0006H)。

发送数据包长度寄存器，发送数据时，应该首先写入发送数据长度，然后将数据通过 PORT0 写入芯片。

CS8900 工作时，应首先对其进行初始化，即在寄存器 LINECTL、RXCTL、RCCFG、BUSCT 中写入正确的配置字。

如图 16.7 所示，在 I/O 模式下，CS8900 发送数据包的步骤如下。

（1）向控制寄存器 TXCMD 寄存器写入发送命令。

（2）将发送数据长度写入 TXLENG 寄存器。

（3）读取 PacketPage 空间内的 BUSST 寄存器，确定其第 8 位被设置为 Rdy4TxNOW，即设备处于准备发送状态。

（4）将要发送的数据循环写入 PORT0 寄存器。

完成上述寄存器操作后，CS8900 会将数据组织为以太网帧并添加填充位和 CRC 校验信息，然后将数据转化为比特流传送到网络媒介。

如图 16.8 所示，在 I/O 模式下，CS8900 接收数据包的方法如下。

（1）接收到网络适配器产生的中断，查询相对于 I/O 基地址偏移 0008H 中断状态队列端口，判断中断类型为接收中断。

（2）读 PORT0 寄存器依次获得接收状态 rxStatus、接收数据长度 rxLength。

（3）循环继续对 PORT0 寄存器读取 rxLength 次，获得整个数据包。

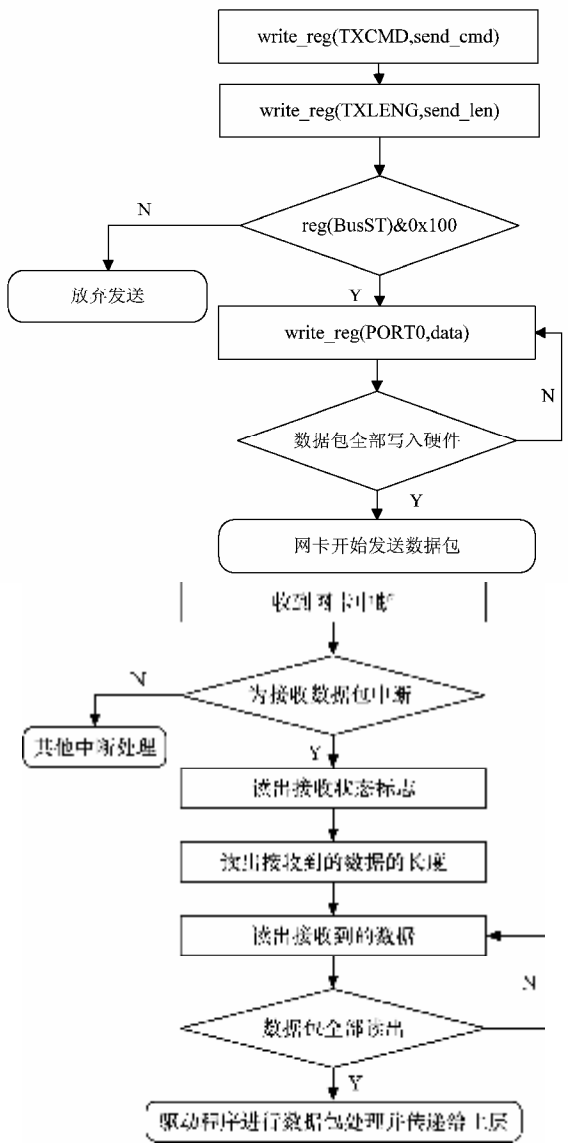


图 16.7　CS8900 数据包发送流程　　图 16.8　CS8900 数据包接收流程

这里所说的 CS8900 处于 I/O 模式并非意味着它一定位于 CPU 的 I/O 空间，实际上，CS8900 I/O 模式下的寄存器仍然映射 ARM 处理器的内存空间。因此，我们直接通过读/写寄存器地址 ioremap()之后的虚拟地址即可完成与代码清单 16.4 对应的 CS8900 寄存器读写函数。

16.9.2 CS8900 网卡驱动设计分析

总的来说，CS8900 是一种具体的网络设备，是通用的抽象的网络设备驱动体系结构和设计模板的例化。

对于一种网络设备的驱动而言，工程师主要需完成设备驱动功能层的设计。在 16.2～16.8 节已经给出了设备驱动功能层主要数据结构和函数的设计模板，因此，在编写 CS8900 的这些数据结构和函数时，实际要完成的工作就是用具体的针对 CS8900 的操作来填充模板，具体包括以下工作。

- 填充 CS8900 的私有信息结构体，把 CS8900 特定的数据放入这个私有结构体中。在 CS8900 驱动程序中，这个数据结构为 struct net_local。
- 填充设备初始化模板，初始化 net_device 结构体，将其注册入内核。net_device 的注册与注销在模块加载与注销函数中完成。在 CS8900 驱动程序中，与此相关的函数有：

```
struct net_device * __init cs89x0_probe(int unit);
int cs89x0_probe1(struct net_device *dev, int ioaddr, int modular);
int init_module(void);
void cleanup_module(void);
```

- 填充设备发送数据包函数模板，把真实的数据包发送硬件操作填充入 xxx_tx()函数，并填充发送超时函数 xxx_tx_timeout()。在 CS8900 驱动程序中，与此相关的函数有：

```
static int net_send_packet(struct sk_buff *skb, struct net_device
*dev);
static void net_timeout(struct net_device *dev);
```

- 填充设备驱动程序的中断处理程序 xxx_interrupt()和具体的数据包接收函数 xxx_rx()，填入真实的硬件操作。在 CS8900 驱动程序中，与此相关的函数有：

```
irqreturn_t net_interrupt(int irq, void *dev_id, struct pt_regs *regs);
void net_rx(struct net_device *dev);
```

- 填充设备打开 xxx_open()与释放 xxx_release()函数代码。在 CS8900 驱动程序中，与此相关的函数有：

```
int net_open(struct net_device *dev);
int net_close(struct net_device *dev);
```

- 填充设备配置与数据统计的具体代码，填充返回设备冲突的 xxx_stats()函数。

图 16.9 所示为 16.2～16.8 节给出的设备驱动功能层抽象设计模板与上述 CS8900 的具体函数和数据结构间的映射关系。

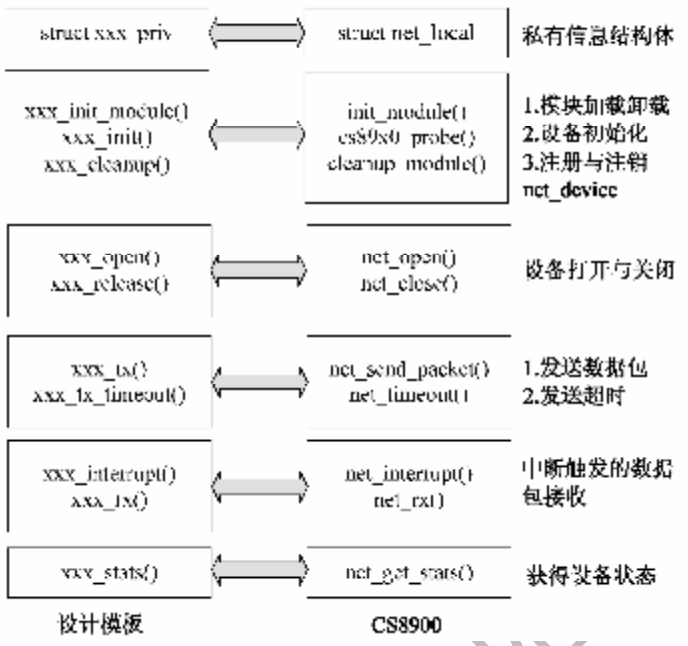


图 16.9　CS8900 驱动与设计模板的映射

下面将具体讲解 CS8900 驱动的若干数据结构和函数，由于整个代码量相当庞大，我们只对其比较复杂的注册与注销、初始化、数据包发送与接口函数进行分析。

16.9.3　CS8900 网卡注册、初始化与注销

在 CS8900 的设备驱动程序中，核心数据结构 net_device 以全局变量的方式定义，其数个成员的初始值都被置为空，如代码清单 16.20 所示。

代码清单 16.20　CS8900 的 net_device 实例

```
1   static struct net_device dev_cs89x0 = {
2           "",
3           0, 0, 0, 0,
4           0, 0,
5           0, 0, 0, NULL, NULL };
```

驱动程序为 CS8900 网卡设备定义的私有信息结构体为 net_local，如代码清单 16.21 所示。

代码清单 16.21　CS8900 私有信息结构体 net_local

```
1   struct net_local {
2       struct net_device_stats stats;   /* 网络设备状态结构体 */
3       int chip_type;   /* 区分芯片类型: CS89x0 */
4       char chip_revision; /* 芯片版本字母，如"A" */
5       int send_cmd; /* 发送命令: TX_NOW, TX_AFTER_381 或 TX_AFTER_ALL
*/
```

```
    ...
    spinlock_t lock;    /* 并发控制自旋锁 */
};
```

当芯片的版本字母不同时，net_local 结构体中记录的 send_cmd 命令将不同。例如，同样是 CS8900 网卡，若芯片版本字母为大于等于“F”，则发送命令为 TX_NOW，而对于 CS8900A，发送命令为 TX_AFTER_ALL。

CS8900 设备驱动程序的初始化由 net_device 结构体中的 init()函数完成，这个函数将在 net_device 被注册时自动被调用。init()函数在 CS8900 网卡的驱动程序中对应于 cs89x0_probe()函数，如代码清单 16.22 所示。

代码清单 16.22　CS8900 设备驱动的初始化函数

```
1   int __init cs89x0_probe(struct net_device *dev) {
2      int i;
3
4      SET_MODULE_OWNER(dev);
5      DPRINTK(1, "cs89x0:cs89x0_probe(0x%x)\n", base_addr);
6
7      BWSCON = (BWSCON & ~(BWSCON_ST3 | BWSCON_WS3 | BWSCON_DW3)) |
8        (BWSCON_ST3 | BWSCON_WS3 | BWSCON_DW(3, BWSCON_DW_16));
9      BANKCON3= BANKCON_Tacs0 | BANKCON_Tcos4 | BANKCON_Tacc14 |
10       BANKCON_Toch1 | BANKCON_Tcah4 | BANKCON_Tacp6 | BANKCON_PMC1;
11
12     set_external_irq(IRQ_CS8900,                EXT_RISING_EDGE,
GPIO_PULLUP_DIS);
13
14     for (i = 0; netcard_portlist[i]; i++) {
15       if (cs89x0_probe1(dev, netcard_portlist[i]) == 0)
16            return 0;
17     }
18     printk(KERN_WARNING "cs89x0: no cs8900 or cs8920 detected."
19        "Be sure to disable PnP with SETUP\n");
20     return -ENODEV;
21  }
```

上述函数第 7～10 行设置 S3C2410A ARM 处理器的片选，第 12 行设置 ARM 与 CS8900 网卡对应的中断，第 14～17 行循环检索 netcard_portlist[ ]数组中定义的基地址处是否存在 CS8900 网卡。对于我们的嵌入式硬件平台，netcard_portlist[ ]数组的定义将很简单，基地址为 vCS8900_BASE + 0x300，因而 netcard_portlist[]数组的定义如代码清单 16.23 所示。

代码清单 16.23　要探测的 CS8900 网卡基地址列表

```
1   static unsigned int netcard_portlist[] __initdata =
2     { vCS8900_BASE + 0x300, 0};
```

代码清单 16.22 中第 15 行调用的 cs89x0_probe1()函数读取 vCS8900_BASE + 0x300 基地址处的 CS8900 相关的寄存器，以“反证法”验证网卡的存在，并获取 CS8900 所使用的硬件资源。代码清单 16.24 所示为 cs89x0_probe1()函数。

代码清单 16.24　探测 CS8900 并获取硬件资源

```
1   static int __init cs89x0_probe1(struct net_device *dev, int ioaddr)
2   {
3     struct net_local *lp;
4     unsigned rev_type = 0;
5     int ret;
6
7     /* 初始化设备结构体私有信息 */
8     if (dev->priv == NULL)
9     {
10      dev->priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
11      if (dev->priv == 0)
12      {
13        ret =  - ENOMEM;
14        goto before_kmalloc;
15      }
16      lp = (struct net_local*)dev->priv;
17      memset(lp, 0, sizeof(*lp));
18      spin_lock_init(&lp->lock);
19    }
20    lp = (struct net_local*)dev->priv;
21
22    dev->base_addr = ioaddr;
23    /* 读取芯片类型 */
24    rev_type = readreg(dev, PRODUCT_ID_ADD);
25    lp->chip_type = rev_type &~REVISON_BITS;
26    lp->chip_revision = ((rev_type &REVISON_BITS) >> 8) + 'A';30
27    if (lp->chip_type != CS8900)
28    {
29      printk(__FILE__ ": wrong device driver!\n");
30      ret =  - ENODEV;
31      goto after_kmalloc;
```

```
    }
    /* 根据芯片类型和版本确定正确的发送命令 */
    lp->send_cmd = TX_AFTER_ALL;
    if (lp->chip_type == CS8900 && lp->chip_revision >= 'F')
      lp->send_cmd = TX_NOW;

    reset_chip(dev);

    lp->adapter_cnf = A_CNF_10B_T | A_CNF_MEDIA_10B_T;
    lp->auto_neg_cnf = EE_AUTO_NEG_ENABLE;43
    printk(KERN_INFO   "cs89x0   media   %s%s",   (lp->adapter_cnf
&A_CNF_10B_T) ?
      "RJ-45": "", (lp->adapter_cnf &A_CNF_AUI) ? "AUI" : "");

    /* 设置 CS8900 的 MAC 地址 */
    dev->dev_addr[0] = 0x00;
    dev->dev_addr[1] = 0x00;
    dev->dev_addr[2] = 0xc0;
    dev->dev_addr[3] = 0xff;
    dev->dev_addr[4] = 0xee;
    dev->dev_addr[5] = 0x08;
    set_mac_address(dev, dev->dev_addr);

    /* 设置设备中断号 */
    dev->irq = IRQ_LAN;
    printk(", IRQ %d", dev->irq);

    /* 填充设备结构体的成员函数指针 */
    dev->open = net_open;
    dev->stop = net_close;
    dev->tx_timeout = net_timeout;
    dev->watchdog_timeo = 3 * HZ;
    dev->hard_start_xmit = net_send_packet;
    dev->get_stats = net_get_stats;
    dev->set_multicast_list = set_multicast_list;
    dev->set_mac_address = set_mac_address;

    /* 填充以太网公用数据和函数指针 */
    ether_setup(dev);

    printk("\n");
    DPRINTK(1, "cs89x0_probe1() successful\n");
    return 0;
```

```
74
75    after_kmalloc: kfree(dev->priv);
76    before_kmalloc: return ret;
77  }
```

上述 cs89x0_probe1()函数的流程如下。

（1）第 8～20 行分配设备的私有信息结构体内存并初始化，若分配失败，则直接跳入第 78 行的代码返回。

（2）第 24～26 行从寄存器中读取芯片的具体类型。

（3）第 27～32 行判断芯片类型，若不是 CS8900 则直接跳入第 77 行的代码，释放私有信息结构体并返回。

（4）当芯片类型为 CS8900 时，第 34～69 行完成 net_device 设备结构体的初始化，赋值其属性和函数指针。

下面讲解 CS8900 的模块加载函数，该函数用来注册经过 cs89x0_probe()和 cs89x0_probe1()函数初始化过的 net_device 设备结构体，如代码清单 16.25 所示。

代码清单 16.25　CS8900 网卡设备驱动的模块加载函数

```
1   static int __init init_cs8900a_s3c2410(void)
2   {
3     struct net_local *lp;
4     int ret = 0;
5
6     dev_cs89x0.irq = irq;
7     dev_cs89x0.base_addr = io;
8     dev_cs89x0.init = cs89x0_probe;
9     dev_cs89x0.priv = kmalloc(sizeof(struct net_local), GFP_KERNEL);
10    if (dev_cs89x0.priv == 0)
11    {
12      printk(KERN_ERR "cs89x0.c: Out of memory.\n");
13      return  - ENOMEM;
14    }
15    memset(dev_cs89x0.priv, 0, sizeof(struct net_local));
16    lp = (struct net_local*)dev_cs89x0.priv;
17
18    request_region(dev_cs89x0.base_addr,        NETCARD_IO_EXTENT,
"cs8900a");
19
20    spin_lock_init(&lp->lock);
21
22    /* 设置物理接口的正确类型*/
```

```
23    if (!strcmp(media, "rj45"))
24      lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
25    else if (!strcmp(media, "aui"))
26      lp->adapter_cnf = A_CNF_MEDIA_AUI | A_CNF_AUI;
27    else if (!strcmp(media, "bnc"))
28      lp->adapter_cnf = A_CNF_MEDIA_10B_2 | A_CNF_10B_2;
29    else
30      lp->adapter_cnf = A_CNF_MEDIA_10B_T | A_CNF_10B_T;
31
32    if (duplex ==  - 1)
33      lp->auto_neg_cnf = AUTO_NEG_ENABLE;
34
35    if (io == 0)
36    {
37      printk(KERN_ERR "cs89x0.c: Module autoprobing not allowed.\n");
38      printk(KERN_ERR "cs89x0.c: Append io=0xNNN\n");
39      ret =  - EPERM;
40      goto out;
41    }
42
43    if (register_netdev(&dev_cs89x0) != 0)
44    {
45      printk(KERN_ERR "cs89x0.c: No card found at 0x%x\n", io);
46      ret =  - ENXIO;
47      goto out;
48    }
49    out: if (ret)
50      kfree(dev_cs89x0.priv);
51    return ret;
52  }
```

上述代码第 8 行将 cs89x0_probe()函数赋值给结构体 net_device 的 init 指针，这样，在使用 register_netdev()函数注册 net_device 设备结构体时，cs89x0_probe()函数会被自动调用以完成 net_device 结构体的初始化。

第 18 行调用 request_region()函数为 CS8900 网卡申请了 NETCARD_IO_EXTENT 大小的 I/O 地址区域。

第 43 行完成已经被 cs89x0_probe()函数初始化的 net_device 设备结构体的注册。

CS8900 驱动的模块卸载函数需完成与模块加载函数相反的功能，如代码清单 16.26 所示。

代码清单 16.26　CS8900 网卡设备驱动的模块卸载函数

```
1   static void __exit cleanup_cs8900a_s3c2410(void)
2   {
3     if (dev_cs89x0.priv != NULL)
4     {
5       /* 释放私有信息结构体  */
6       unregister_netdev(&dev_cs89x0);
7       outw(PP_ChipID, dev_cs89x0.base_addr + ADD_PORT);
8       kfree(dev_cs89x0.priv);
9       dev_cs89x0.priv = NULL;
10      /* 释放 CS8900 申请的 I/O 地址区域 */
```

```
    release_region(dev_cs89x0.base_addr, NETCARD_IO_EXTENT);
  }
}
```

上述代码首先注销了 net_device 设备结构体，然后释放 CS8900 的私有信息结构体和在模块加载函数中申请的 I/O 地址区域。

16.9.4 CS8900 网卡发送数据流程

从代码清单 16.24 即 CS8900 的初始化函数可以看出，CS8900 的数据包发送函数指针 hard_ start_xmit 被赋值为 CS8900 驱动程序中的 net_send_packet()，这个函数完成 16.9.1 小节所描述的硬件发送序列，代码清单为 16.27 所示。

代码清单 16.27 CS8900 网卡驱动的发送函数

```
1  static int net_send_packet(struct sk_buff *skb, struct net_device
*dev)
2  {
3    struct net_local *lp = (struct net_local*)dev->priv;
4
5    writereg(dev, PP_BusCTL, 0x0);
6    writereg(dev, PP_BusCTL, readreg(dev, PP_BusCTL) | ENABLE_IRQ);
7
8    spin_lock_irq(&lp->lock);/* 使用自旋锁阻止多个数据包被同时写入硬件
*/
9    netif_stop_queue(dev);
10
11   /* 初始化硬件发送序列 */
12   writeword(dev, TX_CMD_PORT, lp->send_cmd);
13   writeword(dev, TX_LEN_PORT, skb->len);
14
15   /* 检测硬件是否处于发送 READY 状态 */
16   if ((readreg(dev, PP_BusST) &READY_FOR_TX_NOW) == 0)
17   {
18     spin_unlock_irq(&lp->lock);
19     DPRINTK(1, "cs89x0: Tx buffer not free!\n");
20     return 1;
21   }
22
23   writeblock(dev, skb->data, skb->len);/* 将数据写入硬件 */
24
25   spin_unlock_irq(&lp->lock);       /* 解锁自旋锁 */
26   dev->trans_start = jiffies;       /* 记录发送开始的时间戳 */
27   dev_kfree_skb(skb);               /* 释放 sk_buff 和数据缓冲区 */
28
29   return 0;
30 }
```

上述代码第 12～23 行完成硬件操作，即按照发送序列在 CS8900 的寄存器中写入相应的命令或数据。整个硬件发送序列都处于第 8 行锁定的自选锁的控制之下，直到

数据包被写入硬件，才在代码的第 25 行对自旋锁解除锁定。

第 26 行记录最后数据包发送的时间戳。

第 27 行释放了从上层协议传递下来的 sk_buff 套接字缓冲区和数据缓冲区。

当发送数据包超时时，CS8900 驱动程序的数据包发送超时函数将被调用，它重新启动设备发送队列，如代码清单 16.28 所示。

代码清单 16.28　CS8900 网卡驱动的发送超时函数

```
1   static void net_timeout(struct net_device *dev)
2   {
3       DPRINTK(1, "%s: transmit timed out, %s?\n", dev->name,
4           tx_done(dev) ? "IRQ conflict ?" : "network cable problem");
5
6       net_close(dev);
7       writereg(dev,   PP_SelfCTL,   readreg(dev,   PP_SelfCTL)   |
POWER_ON_RESET);
8       net_open(dev);
9   }
```

上述代码处理发送超时的策略是：第 6 行停止网卡，第 7 行进行网卡硬复位，在复位后第 8 行再次启动 CS8900 网卡。

16.9.5　CS8900 网卡接收数据流程

CS8900 网卡设备的数据包接收流程由中断引发，在中断服务程序中会判断中断的类型，如果是接收到数据包中断，则完成套接字缓冲区的创建和填充，并调用 netif_rx()函数将套接字缓冲区传递给上层协议，如代码清单 16.29 所示。

代码清单 16.29　CS8900 网卡驱动的中断处理函数

```
1   static void net_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
2   {
3     struct net_device *dev = dev_id;
4     struct net_local *lp;
5     int ioaddr, status;
6
7     ioaddr = dev->base_addr;
8     lp = (struct net_local*)dev->priv;
9
10    /* 读取中断事件类型 */
11    while ((status = readword(dev, ISQ_PORT)))
12    {
13      DPRINTK(4, "%s: event=%04x\n", dev->name, status);
14      switch (status &ISQ_EVENT_MASK)
15      {
```

```
16        case ISQ_RECEIVER_EVENT:
17          /* 获得数据包 */
18          net_rx(dev);
19          break;
20        ... /* 其他类型中断 */
21      }
22    }
23  }
```

上述程序的第 18 行意味着网卡中断为接收中断时，调用 net_rx()函数完成数据包的获取与向上层协议传递。因此，net_rx()函数相当关键，如代码清单 16.30 所示。

代码清单 16.30　CS8900 网卡驱动的数据包接收函数

```
1   static void net_rx(struct net_device *dev)
2   {
3     struct net_local *lp = (struct net_local*)dev->priv;
4     struct sk_buff *skb;
5     int status, length;
6
7     int ioaddr = dev->base_addr;
8
9     status = inw(ioaddr + RX_FRAME_PORT);
10    if ((status &RX_OK) == 0)
11    {
12      count_rx_errors(status, lp);
13      return ;
14    }
15
16    length = inw(ioaddr + RX_FRAME_PORT);/* 读取接收数据包的长度 */
17
18    /* 分配新的套接字缓冲区和数据缓冲区 */
19    skb = dev_alloc_skb(length + 2);
20    if (skb == NULL)
21    {
22      lp->stats.rx_dropped++; /* 分配失败，统计被丢弃的包数 */
23      return ;
24    }
25    skb_reserve(skb, 2);
26    skb->len = length;
27    skb->dev = dev;29
28    readblock(dev, skb->data, skb->len); /* 将硬件中读取数据包放入数据
缓冲区 */
29
30    skb->protocol = eth_type_trans(skb, dev);/* 解析协议类型 */

31    netif_rx(skb);       /* 传递给上层协议 */
32
33    dev->last_rx = jiffies;   /* 记录最后收到数据包的时间戳 */
34    /* 统计接收数据包数和接收字节数 */
35    lp->stats.rx_packets++;
```

```
36    lp->stats.rx_bytes += length;
37  }
```

上述代码的第 16 行从硬件中读取收到的数据包长度 length，根据这一结果，第 19 行分配套接字缓冲区和数据缓冲区，第 28 行从硬件中读取数据包到分配的数据缓冲区中，第 30 行利用内核提供的 eth_type_trans()函数解析收到数据包的网络层协议类型并赋值给 skb->protocol，第 31 行利用网络协议接口层的 netif_rx()函数将数据包递交给上层协议。程序中最后的第 33～36 行记录接收时间戳并完成信息统计。

16.10 总结

Linux 网络设备驱动体系结构的层次化设计实现了对上层协议接口的统一和硬件驱动的对下层多样化硬件设备的可适应。程序员需要完成的工作集中在设备驱动功能层，网络设备接口层 net_device 结构体的存在将千变万化的网络设备得以抽象，使得设备功能层中除数据包接收以外的主体工作都由填充 net_device 的属性和函数指针完成。

在分析 net_device 数据结构的基础上，本章给出了设备驱动功能层设备初始化、数据包收发、打开和释放等函数的设计模板，这些模板对实际设备驱动的开发具有直接指导意义。有了这些模板，我们在设计具体设备的驱动时，不再需要关心程序的体系，而可以集中精力于硬件操作本身。从对 CS8900 网卡的驱动分析可知，其驱动的主要函数和数据结构和模板可以良好地映射。

在 Linux 网络子系统和设备驱动中，套接字缓冲区 sk_buff 发挥着巨大的作用，是所有数据流动的载体。网络设备驱动和上层协议之间也依赖此结构进行数据包交互，因此，我们要特别牢记它的操作方法。

- 实验设备：http://www.embedu.org/courses/course5.htm

推荐课程：华清远见-嵌入式 Linux 短期高端培训班

- 嵌入式 Linux 应用开发班：

  http://www.farsight.com.cn/courses/TS-LinuxApp.htm

- 嵌入式 Linux 系统开发班：

  http://www.farsight.com.cn/courses/TS-LinuxEMB.htm

- 嵌入式 Linux 驱动开发班：

  http://www.farsight.com.cn/courses/TS-LinuxDriver.htm