

第 14 章

Linux 网络设备驱动

本章导读

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。

14.1 节讲解 Linux 网络设备驱动的层次结构，描述其中 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。

14.2 ~ 14.8 节主要讲解设备驱动功能层的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。

14.9 节介绍 DM9000 网卡的设备驱动及其在具体开发板上的移植。

14.1 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 14.1 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。

1) 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，不论上层协议是 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。

2) 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。

3) 设备驱动功能层的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。

4) 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在物理上驱动。对于 Linux 系统而言，网络设备和媒介都可以是虚拟的。

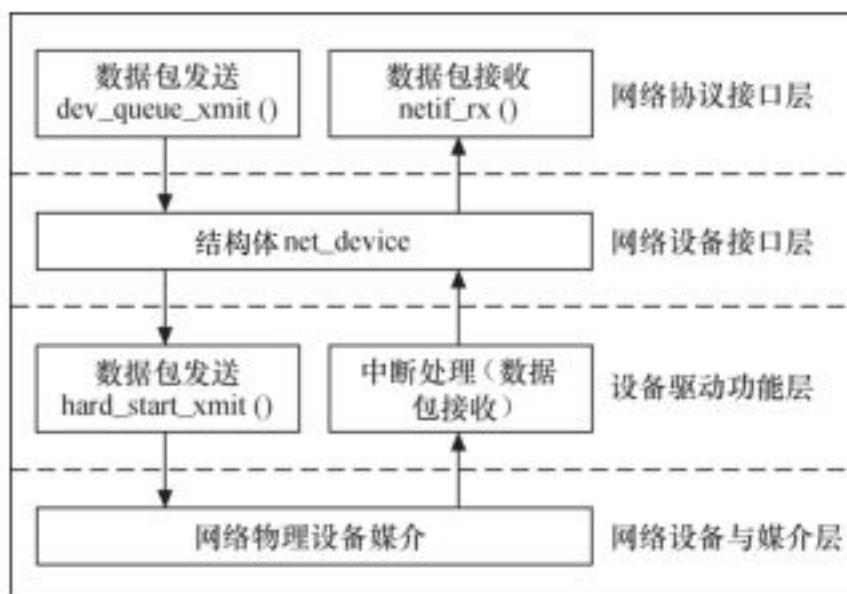


图 14.1 Linux 网络设备驱动程序的体系结构

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。

14.1.1 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透明的数据包发送和接收接口。当上层 ARP 或 IP 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：

```
int 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，它定义于 include/linux/skbuff.h 文件中，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。代码清单 14.1 列出了 sk_buff 结构体中的几个关键数据成员以及描述。

代码清单 14.1 sk_buff 结构体中的几个关键数据成员以及描述

```

1  /**
2   *      struct sk_buff - socket buffer
3   *      @next: Next buffer in list
4   *      @prev: Previous buffer in list
5   *      @len: Length of actual data
6   *      @data_len: Data length
7   *      @mac_len: Length of link layer header
8   *      @hdr_len: writable header length of cloned skb
9   *      @csum: Checksum (must include start/offset pair)
10  *      @csum_start: Offset from skb->head where checksumming should start
11  *      @csum_offset: Offset from csum_start where checksum should be stored
12  *      @priority: Packet queueing priority
13  *      @protocol: Packet protocol from driver
14  *      @inner_protocol: Protocol (encapsulation)
15  *      @inner_transport_header: Inner transport layer header (encapsulation)
16  *      @inner_network_header: Network layer header (encapsulation)
17  *      @inner_mac_header: Link layer header (encapsulation)
18  *      @transport_header: Transport layer header
19  *      @network_header: Network layer header
20  *      @mac_header: Link layer header
21  *      @tail: Tail pointer
22  *      @end: End pointer
23  *      @head: Head of buffer
24  *      @data: Data head pointer
25  */
26
27 struct sk_buff {
28     /* These two members must be first. */
29     struct sk_buff          *next;
30     struct sk_buff          *prev;
31
32     ...
33     unsigned int             len,
34                           data_len;
35     __u16                   mac_len,
36                           hdr_len;
37     ...
38     __u32                   priority;
39     ...
40     __be16                  protocol;
41
42     ...
43

```

```

44     __be16          inner_protocol;
45     __u16           inner_transport_header;
46     __u16           inner_network_header;
47     __u16           inner_mac_header;
48     __u16           transport_header;
49     __u16           network_header;
50     __u16           mac_header;
51     /* These elements must be at the end, see alloc_skb() for details. */
52     sk_buff_data_t   tail;
53     sk_buff_data_t   end;
54     unsigned char    *head,
55                           *data;
56     ...
57 };

```

如图 14.1 所示，尤其值得注意的是 head 和 end 指向缓冲区的头部和尾部，而 data 和 tail 指向实际数据的头部和尾部。每一层会在 head 和 data 之间填充协议头，或者在 tail 和 end 之间添加新的协议数据。

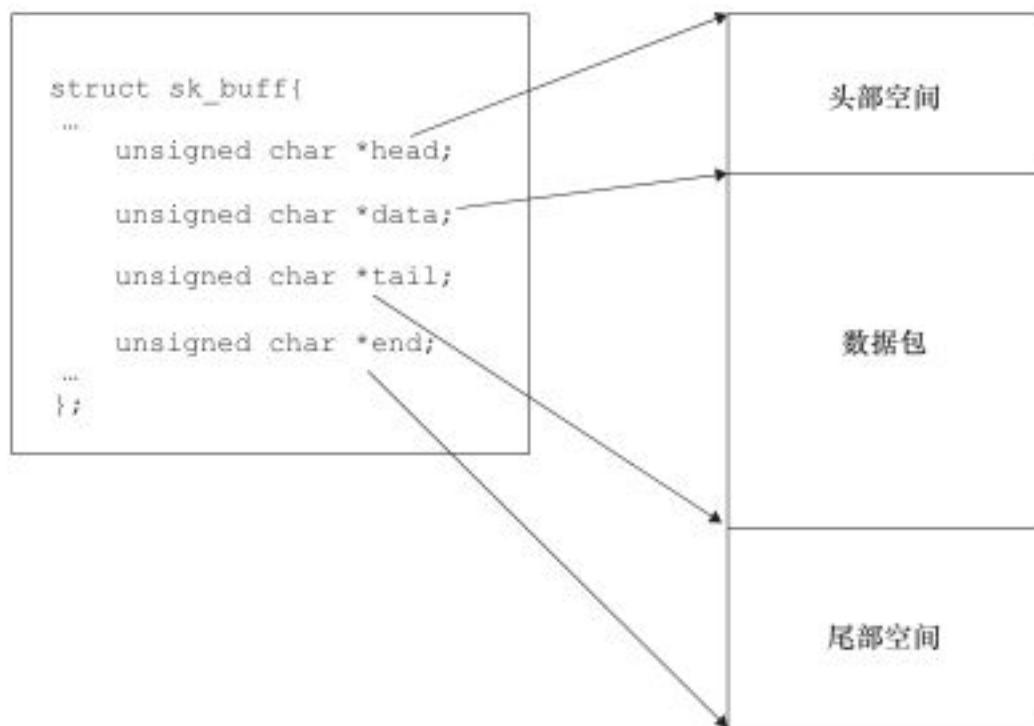


图 14.2 sk_buff 的 head、data、tail、end 指针

下面我们来分析套接字缓冲区涉及的操作函数，Linux 套接字缓冲区支持分配、释放、变更等功能函数。

(1) 分配

Linux 内核中用于分配套接字缓冲区的函数有：

```

struct sk_buff *alloc_skb(unsigned int len, gfp_t 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，通常以 `L1_CACHE_BYTES` 字节（对于 ARM 为 32）对齐，参数 `priority` 为内存分配的优先级。`dev_alloc_skb()` 函数以 `GFP_ATOMIC` 优先级进行 `skb` 的分配，原因是该函数经常在设备驱动的接收中断里被调用。

(2) 释放

Linux 内核中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 `kfree_skb()` 函数，而在网络设备驱动程序中则最好用 `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()` 函数在中断和非中断上下文中皆可采用，它其实是做一个非常简单的上下文判断，然后再调用 `_dev_kfree_skb_irq()` 或者 `dev_kfree_skb()`，这从其代码的实现中也可以看出：

```
void __dev_kfree_skb_any(struct sk_buff *skb, enum skb_free_reason reason)
{
    if (in_irq() || irqs_disabled())
        __dev_kfree_skb_irq(skb, reason);
    else
        dev_kfree_skb(skb);
}
```

(3) 变更

在 Linux 内核中可以用如下函数在缓冲区尾部增加数据：

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

它会导致 `skb->tail` 后移 `len` (`skb->tail += len`)，而 `skb->len` 会增加 `len` 的大小 (`skb->len += len`)。通常，在设备驱动的接收数据处理中会调用此函数。

在 Linux 内核中可以用如下函数在缓冲区开头增加数据：

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

它会导致 `skb->data` 前移 `len` (`skb->data -= len`)，而 `skb->len` 会增加 `len` 的大小 (`skb->len += len`)。与该函数的功能完成相反的函数是 `skb_pull()`，它可以在缓冲区开头移除数据，执行的动作是 `skb->len -= len`、`skb->data += len`。

对于一个空的缓冲区而言，调用如下函数可以调整缓冲区的头部：

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

它会将 `skb->data` 和 `skb->tail` 同时后移 `len`，执行 `skb->data += len`、`skb->tail += len`。内核里存在许多这样的代码：

```
skb = alloc_skb(len+headspace, GFP_KERNEL);
skb_reserve(skb, headspace);
skb_put(skb, len);
memcpy_fromfs(skb->data, data, len);
pass_to_m_protocol(skb);
```

上述代码先分配一个全新的 `sk_buff`，接着调用 `skb_reserve()` 腾出头部空间，之后调用 `skb_put()` 腾出数据空间，然后把数据复制进来，最后把 `sk_buff` 传给协议栈。

14.1.2 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，它定义于 `include/linux/netdevice.h` 文件中，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` 是一个巨大的结构体，定义于 `include/linux/netdevice.h` 中，包含网络设备的属性描述和操作接口，下面介绍其中的一些关键成员。

(1) 全局信息

```
char name[IFNAMESIZE];
```

`name` 是网络设备的名称。

(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;
```

用于存放设备的硬件地址，驱动可能提供了设置 MAC 地址的接口，这会导致用户设置的 MAC 地址等存入该成员，如代码清单 14.2 `drivers/net/ethernet/moxa/moxart_ether.c` 中的 `moxart_set_mac_address()` 函数所示。

代码清单 14.2 `set_mac_address()` 函数

```

1 static int moxart_set_mac_address(struct net_device *ndev, void *addr)
2 {
3     struct sockaddr *address = addr;
4
5     if (!is_valid_ether_addr(address->sa_data))
6         return -EADDRNOTAVAIL;
7
8     memcpy(ndev->dev_addr, address->sa_data, ndev->addr_len);
9     moxart_update_mac_address(ndev);
10
11    return 0;
12 }
```

上述代码完成了 `memcpy()` 以及最终硬件上的 MAC 地址变更。

```
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) 和 `IFF_POINTOPOINT` (接口连接到点到点链路) 等。

(4) 设备操作函数

```
const struct net_device_ops *netdev_ops;
```

该结构体是网络设备的一系列硬件操作行数的集合，它也定义于 include/linux/netdevice.h 中，这个结构体很大，代码清单 14.3 列出了其中的一些基础部分。

代码清单 14.3 net_device_ops 结构体

```

1 struct net_device_ops {
2     int (*ndo_init)(struct net_device *dev);
3     void (*ndo_uninit)(struct net_device *dev);
4     int (*ndo_open)(struct net_device *dev);
5     int (*ndo_stop)(struct net_device *dev);
6     netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb,
7                                   struct net_device *dev);
8     u16 (*ndo_select_queue)(struct net_device *dev,
9                             struct sk_buff *skb,
10                            void *accel_priv,
11                            select_queueFallback_t fallback);
12    void (*ndo_change_rx_flags)(struct net_device *dev,
13                                int flags);
14    void (*ndo_set_rx_mode)(struct net_device *dev);
15    int (*ndo_set_mac_address)(struct net_device *dev,
16                               void *addr);
17    int (*ndo_validate_addr)(struct net_device *dev);
18    int (*ndo_do_ioctl)(struct net_device *dev,
19                         struct ifreq *ifr, int cmd);
20    ...
21 };

```

ndo_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 (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

ndo_start_xmit() 函数会启动数据包的发送，当系统调用驱动程序的 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 (*ndo_tx_timeout)(struct net_device *dev);
```

当数据包的发送超时时，ndo_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措施来恢复网络设备到正常状态。

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

ndo_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 结构体指针。net_device_stats 结构体保存了详细的网络设备流量统计信息，如发送和接收的数据包数、字节数等，详见 14.8 节。

```

int (*ndo_do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
int (*ndo_set_config)(struct net_device *dev, struct ifmap *map);
int (*ndo_set_mac_address)(struct net_device *dev, void *addr);

```

ndo_do_ioctl() 函数用于进行设备特定的 I/O 控制。

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

ndo_set_mac_address() 函数用于设置设备的 MAC 地址。

除了 netdev_ops 以外，在 net_device 中还存在类似于 ethtool_ops、header_ops 这样的操作集：

```

const struct ethtool_ops *ethtool_ops;
const struct header_ops *header_ops;

```

ethtool_ops 成员函数与用户空间 ethtool 工具的各个命令选项对应，ethtool 提供了网卡及网卡驱动管理能力，能够为 Linux 网络开发人员和管理人员提供对网卡硬件、驱动程序和网络协议栈的设置、查看以及调试等功能。

header_ops 对应于硬件头部操作，主要是完成创建硬件头部和从给定的 sk_buff 分析出硬件头部等操作。

(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。

通常情况下，网络设备驱动以中断方式接收数据包，而 poll_controller() 则采用纯轮询方式，另外一种数据接收方式是 NAPI (New API)，其数据接收流程为“接收中断来临→关闭接收中断→以轮询方式接收所有数据包直到收空→开启接收中断→接收中断来临……”内核中提供了如下与 NAPI 相关的 API：

```

static inline void netif_napi_add(struct net_device *dev,
                                  struct napi_struct *napi,
                                  int (*poll)(struct napi_struct *, int),
                                  int weight);
static inline void netif_napi_del(struct napi_struct *napi);

```

以上两个函数分别用于初始化和移除一个 NAPI，netif_napi_add() 的 poll 参数是 NAPI 要调度执行的轮询函数。

```

static inline void napi_enable(struct napi_struct *n);
static inline void napi_disable(struct napi_struct *n);

```

以上两个函数分别用于使能和禁止 NAPI 调度。

```
static inline int napi_schedule_prep(struct napi_struct *n);
```

该函数用于检查 NAPI 是否可以调度，而 napi_schedule() 函数用于调度轮询实例的运行，其原型为：

```
static inline void napi_schedule(struct napi_struct *n);
```

在 NAPI 处理完成的时候应该调用：

```
static inline void napi_complete(struct napi_struct *n);
```

14.1.3 设备驱动功能层

net_device 结构体的成员（属性和 net_device_ops 结构体中的函数指针）需要被设备驱动功能层赋予具体的数值和函数。对于具体的设备 xxx，工程师应该编写相应的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() 和 xxx_tx_timeout() 等。

由于网络数据包的接收可由中断引发，设备驱动功能层中的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到的数据包并传送给上层协议，因此可能包含 xxx_interrupt() 和 xxx_rx() 函数，前者完成中断类型判断等基本工作，后者则需完成数据包的生成及将其递交给上层等复杂工作。

14.2 ~ 14.8 节将对上述函数进行详细分析并给出参考设计模板。

对于特定的设备，我们还可以定义相关的私有数据和操作，并封装为一个私有信息结构体 xxx_private，让其指针赋值给 net_device 的私有成员。在 xxx_private 结构体中可包含设备的特殊属性和操作、自旋锁与信号量、定时器以及统计信息等，这都由工程师自定义。在驱动中，要用到私有数据的时候，则使用在 netdevice.h 中定义的接口：

```
static inline void *netdev_priv(const struct net_device *dev);
```

比如在驱动 drivers/net/ethernet/davicom/dm9000.c 的 dm9000_probe() 函数中，使用 alloc_etherdev(sizeof(struct board_info)) 分配网络设备，board_info 结构体就成了这个网络设备的私有数据，在其他函数中可以简单地提取这个私有数据，例如：

```
static int
dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
    unsigned long flags;
    board_info_t *db = netdev_priv(dev);
    ...
}
```

14.2 网络设备驱动的注册与注销

网络设备驱动的注册与注销由 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不一定要由工程师亲自动手逐个完成，可以利用下面的宏帮助我们填充：

```
#define alloc_netdev(sizeof_priv, name, setup) \
    alloc_netdev_mqs(sizeof_priv, name, setup, 1, 1)
#define alloc_etherdev(sizeof_priv) alloc_etherdev_mq(sizeof_priv, 1)
#define alloc_etherdev_mq(sizeof_priv, count) alloc_etherdev_mqs(sizeof_priv,
    count, count)
```

`alloc_netdev` 以及 `alloc_etherdev` 宏引用的 `alloc_netdev_mqs()` 函数的原型为：

```
struct net_device *alloc_netdev_mqs(int sizeof_priv, const char *name,
    void (*setup)(struct net_device *),
    unsigned int txqs, unsigned int rxqs);
```

`alloc_netdev_mqs()` 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，第四、五个参数为要分配的发送和接收子队列的数量。`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。

`free_netdev()` 完成与 `alloc_enetdev()` 和 `alloc_etherdev()` 函数相反的功能，即释放 `net_device` 结构体的函数：

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

`net_device` 结构体的分配和网络设备驱动的注册需在网络设备驱动程序初始化时进行，而 `net_device` 结构体的释放和网络设备驱动的注销在设备或驱动被移除的时候执行，如代码清单 14.4 所示。

代码清单 14.4 网络设备驱动程序的注册和注销

```
1 static int xxx_register(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 static void xxx_unregister(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 }

```

14.3 网络设备的初始化

网络设备的初始化主要需要完成以下几个方面的工作。

- 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。

网络设备驱动的初始化函数模板如代码清单 14.5 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，但其本质过程是一致的。

代码清单 14.5 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    ndev->netdev_ops      = &xxx_netdev_ops;
14    ndev->ethtool_ops     = &xxx_ethtool_ops;
15    dev->watchdog_timeo = timeout;
16
17    /* 取得私有信息，并初始化它 */

```

```

18     priv = netdev_priv(dev);
19     ... /* 初始化设备私有数据区 */
20 }

```

上述代码第 7 行的 `xxx_hw_init()` 函数完成的与硬件相关的初始化操作如下。

- 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。

14.4 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 14.6 所示的网络设备打开和释放函数的模板。

代码清单 14.6 网络设备打开和释放函数模板

```

1 static 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 static 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 }

```

14.5 数据发送流程

从 14.1 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以指向设备的 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模板如代码清单 14.7 所示。

代码清单 14.7 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     if (xxx_send_available(...)) { /* 发送队列未满, 可以发送 */
6         /* 获得有效数据指针和长度 */
7         data = skb->data;
8         len = skb->len;
9         if (len < ETH_ZLEN) {
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         if (avail) /* 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         } else {
22             netif_stop_queue(dev);
23             ...
24         }
25     }

```

这里特别要强调第22行对netif_stop_queue()的调用，当发送队列为满或因其他原因来不及发送当前上层传下来的数据包时，则调用此函数阻止上层继续向网络设备驱动传递数据包。当忙于发送的数据包被发送完成后，在以TX结束的中断处理中，应该调用netif_wake_queue()唤醒被阻塞的上层，以启动它继续向网络设备驱动传送数据包。

当数据传输超时时，意味着当前的发送操作失败或硬件已陷入未知状态，此时，数据包发送超时处理函数xxx_tx_timeout()将被调用。这个函数也需要调用由Linux内核提供的netif_wake_queue()函数以重新启动设备发送队列，如代码清单14.8所示。

代码清单14.8 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 }
```

从前文可知，netif_wake_queue()和netif_stop_queue()是数据发送流程中要调用的两个非常重要的函数，分别用于唤醒和阻止上层向下传送数据包，它们的原型定义于include/linux/netdevice.h中，如下：

```

static inline void netif_wake_queue(struct net_device *dev);
static inline void netif_stop_queue(struct net_device *dev);
```

14.6 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14.9所示为完成这个过程的函数模板。

代码清单14.9 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)
2 {
3     ...
4     switch (status & ISQ_EVENT_MASK) {
5         case ISQ_RECEIVER_EVENT:
6             /* 获取数据包 */
7             xxx_rx(dev);
8             break;
9             /* 其他类型的中断 */
10    }
11 }
12 static void xxx_rx(struct xxx_device *dev)
13 {
```

```

14 ...
15 length = get_rev_len (...);
16 /* 分配新的套接字缓冲区 */
17 skb = dev_alloc_skb(length + 2);
18
19 skb_reserve(skb, 2); /* 对齐 */
20 skb->dev = dev;
21
22 /* 读取硬件上接收到的数据 */
23 insw(ioaddr + RX_FRAME_PORT, skb_put(skb, length), length >> 1);
24 if (length &1)
25     skb->data[length - 1] = inw(ioaddr + RX_FRAME_PORT);
26
27 /* 获取上层协议类型 */
28 skb->protocol = eth_type_trans(skb, dev);
29
30 /* 把数据包交给上层 */
31 netif_rx(skb);
32
33 /* 记录接收时间戳 */
34 dev->last_rx = jiffies;
35 ...
36 }

```

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 12 ~ 36 行定义的 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 15 行从硬件读取到接收数据包有效数据的长度，第 16 ~ 19 行分配 sk_buff 和数据缓冲区，第 22 ~ 25 行读取硬件上接收到的数据并放入数据缓冲区，第 27 ~ 28 行解析接收数据包上层协议的类型，最后，第 30 ~ 31 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。在这种情况下，我们需要为该设备驱动提供作为 netif_napi_add() 参数的 xxx_poll() 函数，如代码清单 14.10 所示。

代码清单 14.10 网络设备驱动的 xxx_poll() 函数模板

```

1 static int xxx_poll(struct napi_struct *napi, int budget)
2 {
3     int npackets = 0;
4     struct sk_buff *skb;
5     struct xxx_priv *priv = container_of(napi, struct xxx_priv, napi);
6     struct xxx_packet *pkt;
7
8     while (npackets < budget && priv->rx_queue) {
9         /* 从队列中取出数据包 */
10        pkt = xxx_dequeue_buf(dev);
11
12        /* 接下来的处理和中断触发的数据包接收一致 */
13        skb = dev_alloc_skb(pkt->datalen + 2);
14        ...

```

```

15     skb_reserve(skb, 2);
16     memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
17     skb->dev = dev;
18     skb->protocol = eth_type_trans(skb, dev);
19     /* 调用 netif_receive_skb，而不是 net_rx，将数据包交给上层协议 */
20     netif_receive_skb(skb);
21
22     /* 更改统计数据 */
23     priv->stats.rx_packets++;
24     priv->stats.rx_bytes += pkt->datalen;
25     xxx_release_buffer(pkt);
26     npackets++;
27 }
28 if (npackets < budget) {
29     napi_complete(napi);
30     xxx_enable_rx_int(...); /* 再次启动网络设备的接收中断 */
31 }
32 return npackets;
33 }

```

上述代码中的 budget 是在初始化阶段分配给接口的 weight 值，xxx_poll() 函数每次只能接收最多 budget 个数据包。第 8 行的 while() 循环读取设备的接收缓冲区，同时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，但是最后使用的是 netif_receive_skb() 函数而不是 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。

当一个轮询过程结束时，第 29 行代码调用 napi_complete() 宣布这一消息，而第 30 行代码则再次启动网络设备的接收中断。

虽然 NAPI 兼容的设备驱动以 xxx_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这个过程。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并调度 NAPI，如代码清单 14.11 所示。

代码清单 14.11 网络设备驱动的 poll 中断处理

```

1 static void xxx_interrupt(int irq, void *dev_id)
2 {
3     switch (status & ISQ_EVENT_MASK) {
4     case ISQ_RECEIVER_EVENT:
5         ... /* 获取数据包 */
6         xxx_disable_rx_int(...); /* 禁止接收中断 */
7         napi_schedule(&priv->napi);
8         break;
9         ... /* 其他类型的中断 */
10    }
11 }

```

上述代码第 7 行的 napi_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，从而通知网络层接收数据包。图 14.3 所示为 NAPI 驱动程序各部分的调用关系。

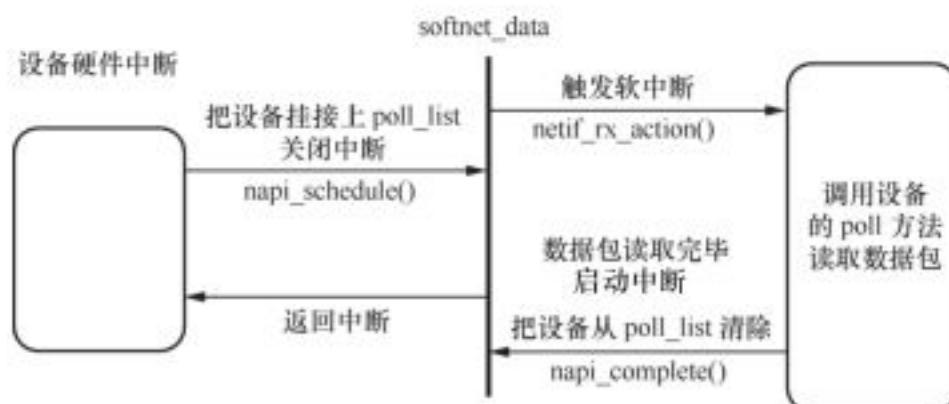


图 14.3 NAPI 驱动程序各部分的调用关系

在支持 NAPI 的网络设备驱动中，通常还会进行如下与 NAPI 相关的工作。

- 1) 在私有数据结构体（如 xxx_priv）中增加一个成员：

```
struct napi_struct napi;
```

在代码中就可以方便地使用 container_of() 通过 NAPI 成员反向获得对应的 xxx_priv 指针。

- 2) 通常会在设备驱动初始化时调用：

```
netif_napi_add(dev, napi, xxx_poll, XXX_NET_NAPI_WEIGHT);
```

- 3) 通常会在 net_device 结构体的 open() 和 stop() 成员函数中分别调用 napi_enable() 和 napi_disable()。

14.7 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 设备结构体指针作为参数，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);
```

在网络设备驱动程序中可采取一定的手段来检测和报告链路状态，最常见的方法是采用中断，其次可以设置一个定时器来对链路状态进行周期性的检查。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以获得载波状态，从而更新设备的连接状态，如代码清单 14.12 所示。

代码清单 14.12 网络设备驱动用定时器周期性检查链路状态

```
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         goto set_timer;
8
9     /* 获得物理上的连接状态 */
10    if (link = xxx_chk_link(dev)) {
11        if (!(dev->flags &IFF_RUNNING)) {
12            netif_carrier_on(dev);
13            dev->flags |= IFF_RUNNING;
14            printk(KERN_DEBUG "%s: link up\n", dev->name);
15        }
16    } else {
17        if (dev->flags &IFF_RUNNING) {
18            netif_carrier_off(dev);
19            dev->flags &= ~IFF_RUNNING;
20            printk(KERN_DEBUG "%s: link down\n", dev->name);
21        }
22    }
23
24    set_timer;
25    priv->timer.expires = jiffies + 1 * Hz;
26    priv->timer.data = (unsigned long)dev;
27    priv->timer.function = &xxx_timer; /* timer handler */
28    add_timer(&priv->timer);
29 }
```

上述代码第 10 行调用 `xxx_chk_link()` 函数来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 12 行的 `netif_carrier_on()` 函数显式地通知内核链路正常；反之，第 18 行的 `netif_carrier_off()` 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 24 ~ 28 行代码启动新的定时器以实现周期性检测的目的。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 14.13 所示。

代码清单 14.13 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 = netdev_priv(dev);
4
5     ...
6     priv->timer.expires = jiffies + 3 * Hz;
7     priv->timer.data = (unsigned long)dev;
8     priv->timer.function = &xxx_timer; /* 定时器处理函数 */
9     add_timer(&priv->timer);
10    ...
11 }

```

14.8 参数设置和统计数据

网络设备的驱动程序还提供一些供系统对设备的参数进行设置或读取设备相关信息的方法。

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。设置网络设备的 MAC 地址可用如代码清单 14.14 所示的模板。

代码清单 14.14 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：

```

static inline bool netif_running(const struct net_device *dev)
{
    return test_bit(__LINK_STATE_START, &dev->state);
}

```

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，并不是 ifmap 结构体中给出的所有修改都是可以接受的。实际上，大多数设备并不适合包含 set_config() 函数。set_config() 函数的例子如代码清单 14.15 所示。

代码清单 14.15 网络设备驱动的 set_config 函数模板

```

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

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。具体的设备是否接受这些信息的修改，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，以进行设备专用数据的设置。这个设置在大多数情况下也并不需要。

驱动程序还应提供 get_stats() 函数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 14.16 所示。

代码清单 14.16 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     ...
4     return &dev->stats;
5 }
```

有的网卡硬件比较强大，可以从硬件的寄存器中读出一些统计信息，如 rx_missed_errors、tx_aborted_errors、rx_dropped、rx_length_errors 等。这个时候，我们应该从硬件寄存器读取统

计信息，填充到 net_device 的 stats 字段中，并返回。具体例子可见 drivers/net/ethernet/adaptec/starfire.c 中的 get_stats() 函数。

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 14.17 所示。

代码清单 14.17 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 包含的主要项目统计信息，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 结构体已经内嵌在与网络设备对应的 net_device 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 14.18 所示。

代码清单 14.18 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     ...
5     dev->stats.tx_errors++;           /* 发送错误包数加 1 */
6     ...
7 }
8
9 /* 中断处理函数 */
10 static void xxx_interrupt(int irq, void *dev_id)
11 {
12     struct net_device *dev = dev_id;
13     switch (status &ISQ_EVENT_MASK) {
14     ...
15     case ISQ_TRANSMITTER_EVENT:      /
16         dev->stats.tx_packets++;    /* 数据包发送成功，tx_packets 信息加 1 */
17         netif_wake_queue(dev);      /* 通知上层协议 */
18         if ((status &(TX_OK | TX_LOST_CRS | TX_SQE_ERROR |
19             TX_LATE_COL | TX_16_COL)) != TX_OK) ( /* 读取硬件上的出错标志 */

```

```

20         /* 根据错误的不同情况，对 net_device_stats 的不同成员加 1 */
21         if ((status & TX_OK) == 0)
22             dev->stats.tx_errors++;
23         if (status & TX_LOST_CRS)
24             dev->stats.tx_carrier_errors++;
25         if (status & TX_SQE_ERROR)
26             dev->stats.tx_heartbeat_errors++;
27         if (status & TX_LATE_COL)
28             dev->stats.tx_window_errors++;
29         if (status & TX_16_COL)
30             dev->stats.tx_aborted_errors++;
31     }
32     break;
33 case ISQ_RX_MISS_EVENT:
34     dev->stats.rx_missed_errors += (status >> 6);
35     break;
36 case ISQ_TX_COL_EVENT:
37     dev->stats.collisions += (status >> 6);
38     break;
39 }
40 }

```

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加 1。而第 13 ~ 38 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，并将相应的项目加 1。

14.9 DM9000 网卡设备驱动实例

14.9.1 DM9000 网卡硬件描述

DM9000 是开发板采用的网络芯片，是一个高度集成且功耗很低的高速网络控制器，可以和 CPU 直连，支持 10/100MB 以太网连接，芯片内部自带 4KB 双字节的 SRAM（3KB 用来发送，13KB 用来接收）。针对不同的处理器，接口支持 8 位、16 位和 32 位。DM9000 一般直接挂在外面的内存总线上。

14.9.2 DM9000 网卡驱动设计分析

DM9000 网卡驱动位于内核源代码的 drivers/net/dm9000.c 中，它基于平台驱动架构，代码清单 14.19 抽取了它的主干。其核心工作是实现了前文所述 net_device 结构体中的 hard_start_xmit()、open()、stop()、set_multicast_list()、do_ioctl()、tx_timeout() 等成员函数，并借助中断辅助进行网络数据包的收发，另外它也实现了 ethtool_ops 中的成员函数。特别注意代码中的黑体部分，它标明了关键的数据收发流程。

代码清单 14.19 DM9000 网卡驱动

```

1 static const struct ethtool_ops dm9000_ethtool_ops = {
2     .get_drvinfo          = dm9000_get_drvinfo,
3     .get_settings         = dm9000_get_settings,
4     .set_settings         = dm9000_set_settings,
5     .get_msghandle        = dm9000_get_msghandle,
6     .set_msghandle        = dm9000_set_msghandle,
7     ...
8 };
9
10 /* Our watchdog timed out. Called by the networking layer */
11 static void dm9000_timeout(struct net_device *dev)
12 {
13     ...
14     netif_stop_queue(dev);
15     dm9000_init_dm9000(dev);
16     dm9000_unmask_interrupts(db);
17     /* We can accept TX packets again */
18     dev->trans_start = jiffies; /* prevent tx timeout */
19     netif_wake_queue(dev);
20
21     ...
22 }
23
24 static int
25 dm9000_start_xmit(struct sk_buff *skb, struct net_device *dev)
26 {
27     ...
28     /* TX control: First packet immediately send, second packet queue */
29     if (db->tx_pkt_cnt == 1) {
30         dm9000_send_packet(dev, skb->ip_summed, skb->len);
31     } else {
32         /* Second packet */
33         db->queue_pkt_len = skb->len;
34         db->queue_ip_summed = skb->ip_summed;
35         netif_stop_queue(dev);
36     }
37
38     spin_unlock_irqrestore(&db->lock, flags);
39
40     /* free this SKB */
41     dev_consume_skb_any(skb);
42
43     return NETDEV_TX_OK;
44 }
45
46 static void dm9000_tx_done(struct net_device *dev, board_info_t *db)
47 {
48     int tx_status = ior(db, DM9000_NSR); /* Got TX status */
49

```

```

50         if (tx_status & (NSR_TX2END | NSR_TX1END)) {
51             /* One packet sent complete */
52             db->tx_pkt_cnt--;
53             dev->stats.tx_packets++;
54
55             if (netif_msg_tx_done(db))
56                 dev_dbg(db->dev, "tx done, NSR %02x\n", tx_status);
57
58             /* Queue packet check & send */
59             if (db->tx_pkt_cnt > 0)
60                 dm9000_send_packet(dev, db->queue_ip_summed,
61                                     db->queue_pkt_len);
62             netif_wake_queue(dev);
63         }
64     }
65
66     static void
67     dm9000_rx(struct net_device *dev)
68     {
69         ...
70
71         /* Check packet ready or not */
72         do {
73             ...
74
75             /* Move data from DM9000 */
76             if (GoodPacket &&
77                 ((skb = netdev_alloc_skb(dev, RxLen + 4)) != NULL)) {
78                 skb_reserve(skb, 2);
79                 rdptr = (u8 *) skb_put(skb, RxLen - 4);
80
81                 /* Read received packet from RX SRAM */
82
83                 (db->inblk) (db->io_data, rdptr, RxLen);
84                 dev->stats.rx_bytes += RxLen;
85
86                 /* Pass to upper layer */
87                 skb->protocol = eth_type_trans(skb, dev);
88                 if (dev->features & NETIF_F_RXCSUM) {
89                     if (((rxbyte & 0xlc) << 3) & rxbyte) == 0)
90                         skb->ip_summed = CHECKSUM_UNNECESSARY;
91                     else
92                         skb_checksum_none_assert(skb);
93                 }
94                 netif_rx(skb);
95                 dev->stats.rx_packets++;
96
97             }...
98         } while (rxbyte & DM9000_PKT_RDY);
99     }
100 }
```

```
101 static irqreturn_t dm9000_interrupt(int irq, void *dev_id)
102 {
103     ...
104     /* Received the coming packet */
105     if (int_status & ISR_PRS)
106         dm9000_rx(dev);
107
108     /* Trnasmit Interrupt check */
109     if (int_status & ISR PTS)
110         dm9000_tx_done(dev, db);
111
112     ...
113     return IRQ_HANDLED;
114 }
115
116 static int
117 dm9000_open(struct net_device *dev)
118 {
119     ...
120
121     /* Initialize DM9000 board */
122     dm9000_init_dm9000(dev);
123
124     if (request_irq(dev->irq, dm9000_interrupt, irqflags, dev->name, dev))
125         return -EAGAIN;
126     ...
127
128     mii_check_media(&db->mii, netif_msg_link(db), 1);
129     netif_start_queue(dev);
130
131     return 0;
132 }
133
134 static int
135 dm9000_stop(struct net_device *ndev)
136 {
137     ...
138
139     netif_stop_queue(ndev);
140     netif_carrier_off(ndev);
141
142     /* free interrupt */
143     free_irq(ndev->irq, ndev);
144
145     dm9000_shutdown(ndev);
146
147     return 0;
148 }
149
150 static const struct net_device_ops dm9000_netdev_ops = {
151     .ndo_open          = dm9000_open,
```

```

152     .ndo_stop          = dm9000_stop,
153     .ndo_start_xmit   = dm9000_start_xmit,
154     .ndo_tx_timeout    = dm9000_timeout,
155     .ndo_set_rx_mode   = dm9000_hash_table,
156     .ndo_do_ioctl      = dm9000_ioctl,
157     .ndo_change_mtu    = eth_change_mtu,
158     .ndo_set_features   = dm9000_set_features,
159     .ndo_validate_addr  = eth_validate_addr,
160     .ndo_set_mac_address = eth_mac_addr,
161 #ifdef CONFIG_NET_POLL_CONTROLLER
162     .ndo_poll_controller = dm9000_poll_controller,
163 #endif
164 };
165
166
167 /*
168  * Search DM9000 board, allocate space and register it
169  */
170 static int
171 dm9000_probe(struct platform_device *pdev)
172 {
173     ...
174
175     /* Init network device */
176     ndev = alloc_etherdev(sizeof(struct board_info));
177     ...
178
179     /* setup board info structure */
180     db = netdev_priv(ndev);
181
182     ...
183
184     ...
185     /* driver system function */
186     ether_setup(ndev);
187
188     ndev->netdev_ops      = &dm9000_netdev_ops;
189     ndev->watchdog_timeo = msecs_to_jiffies(watchdog);
190     ndev->ethtool_ops     = &dm9000_ethtool_ops;
191
192     ...
193     ret = register_netdev(ndev);
194 }
195
196 static int
197 dm9000_drv_remove(struct platform_device *pdev)
198 {
199     struct net_device *ndev = platform_get_drvdata(pdev);
200
201     unregister_netdev(ndev);

```

```

203     dm9000_release_board(pdev, netdev_priv(ndev));
204     free_netdev(ndev);           /* free device structure */
205     ...
206 }
207
208 #ifdef CONFIG_OF
209 static const struct of_device_id dm9000_of_matches[] = {
210     { .compatible = "davicom,dm9000", },
211     { /* sentinel */ }
212 };
213 MODULE_DEVICE_TABLE(of, dm9000_of_matches);
214 #endif
215
216 static struct platform_driver dm9000_driver = {
217     .driver = {
218         .name     = "dm9000",
219         .owner    = THIS_MODULE,
220         .pm       = &dm9000_drv_pm_ops,
221         .of_match_table = of_match_ptr(dm9000_of_matches),
222     },
223     .probe   = dm9000_probe,
224     .remove  = dm9000_drv_remove,
225 };
226
227 module_platform_driver(dm9000_driver);

```

DM9000 驱动的实现与具体 CPU 无关，在将该驱动移植到特定电路板时，只需要在板文件中为与板上 DM9000 对应的平台设备的寄存器和数据基地址进行赋值，并指定正确的 IRQ 资源即可，代码清单 14.20 给出了在 arch/arm/mach-at91/board-sam9261ek.c 板文件中对 DM9000 添加的内容。

代码清单 14.20 board-sam9261ek 板文件中的 DM9000 的平台设备

```

1 static struct resource dm9000_resource[] = {
2     [0] = {
3         .start  = AT91_CHIPSELECT_2,
4         .end    = AT91_CHIPSELECT_2 + 3,
5         .flags  = IORESOURCE_MEM
6     },
7     [1] = {
8         .start  = AT91_CHIPSELECT_2 + 0x44,
9         .end    = AT91_CHIPSELECT_2 + 0xFF,
10        .flags  = IORESOURCE_MEM
11    },
12    [2] = {
13        .flags  = IORESOURCE_IRQ
14        | IORESOURCE_IRQ_LOWEDGE | IORESOURCE_IRQ_HIGHEDGE,
15    }
16 };

```

```

17
18 static struct dm9000_plat_data dm9000_platdata = {
19     .flags          = DM9000_PLATF_16BITONLY | DM9000_PLATF_NO_EEPROM,
20 };
21
22 static struct platform_device dm9000_device = {
23     .name           = "dm9000",
24     .id             = 0,
25     .num_resources = ARRAY_SIZE(dm9000_resource),
26     .resource       = dm9000_resource,
27     .dev            = {
28         .platform_data = &dm9000_platdata,
29     }
30 };

```

14.10 总结

对 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进行抽象，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，而可以将精力集中于硬件操作本身。

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，它是所有数据流动的载体。网络设备驱动和上层协议之间也基于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。