第4章　接口：以太网

4.1　引言

在第3章中，我们讨论了所有接口要用到的数据结构及对这些数据结构的初始化。在本章中，我们说明以太网设备驱动程序在初始化后是如何接收和传输帧的。本章的后半部分介绍配置网络设备的通用`ioctl`命令。第5章是SLIP和环回驱动程序。

我们不准备查看整个以太网驱动程序的源代码，因为它有大约 1 000行C代码(其中有一半是一个特定接口卡的硬件细节)，但要研究与设备无关的以太网代码部分，及驱动程序是如何与内核其他部分交互的。

如果读者对一个驱动程序的源代码感兴趣， Net/3版本包括很多不同接口的源代码。要想研究接口的技术规范，就要求能理解设备专用的命令。图 4-1所示的是Net/3提供的各种驱动程序，包括在本章我们要讨论的LANCE驱动程序。

网络设备驱动程序通过`ifnet`结构(图3-6)中的7个函数指针来访问。图 4-2列出了指向我们的三个例子驱动程序的入口点。

输入函数不包含在图4-2中，因为它们是网络设备中断驱动的。中断服务例程的配置与硬件相关，并且超出了本书的范围。我们要识别处理设备中断的函数，但不是这些函数被调用的机制。

设　备	文　件
DEC DEUNA接口	`vax/if/if_de.c`
3Com以太网接口	`vax/if/if_ec.c`
Excelan EXOS 204接口	`vax/if/if_ex.c`
Interlan以太网通信控制器	`vax/if/if_il.c`
Interlan NP100以太网通信控制器	`vax/if/if_ix.c`
Digital Q-BUS to NI适配器	`vax/if/if_qe.c`
CMC ENP-20以太网控制器	`tahoe/if/if_enp.c`
Excelan EXOS 202 (VME) & 203 (QBUS)	`tahoe/if/if_ex.c`
ACC VERSAbus以太网控制器	`tahoe/if/if_ace.c`
AMD 7990 LANCE接口	**`hp300/dev/if_le.c`**
NE2000以太网	`i386/isa/if_ne.c`
Western Digital 8003以太网适配器	`i386/isa/if_we.c`

图4-1　Net/3中可用的以太网驱动程序

ifnet	以 太 网	SLIP	环　回	说　明
`if_init`	`leinit`			硬件初始化
`if_output`	`ether_output`	`slouput`	`looutput`	接收并对传输的帧进行排队
`if_start`	`lestart`			开始传输帧
`if_done`				输出完成 (未用)
`if_ioctl`	`leioctl`	`slioctl`	`lcioctl`	处理来自一个进程的`ioctl`命令
`if_reset`	`lereset`			把设备复位到已知的状态
`if_watchdog`				监视设备故障或收集统计信息

图4-2　例子驱动程序的接口函数

只有函数if_output和if_ioctl被经常地调用。而if_init、if_done和if_reset从来不被调用或仅从设备专用代码调用（例如：leinit直接被leioctl调用）。函数if_start仅被函数ether_output调用。

4.2 代码介绍

以太网设备驱动程序和通用接口ioctl的代码包含在两个头文件和三个C文件中，它们列于图4-3中。

文 件	说 明
net/if_ether.h	以太网结构
net/if.h	ioctl命令定义
net/if_ethersubr.c	通用以太网函数
hp300/dev/if_le.c	LANCE以太网驱动程序
net/if.c	ioctl处理

图4-3 在本章讨论的文件

4.2.1 全局变量

显示在图4-4中的全局变量包括协议输入队列、LANCE接口结构和以太网广播地址。

变 量	数据类型	说 明
arpintrq	struct ifqueue	ARP输入队列
clnlintrq	struct ifqueue	CLNP输入队列
ipintrq	struct ifqueue	IP输入队列
le_softc	struct le_softc[]	LANCE以太网接口
etherbraodcastaddr	u_char[]	以太网广播地址

图4-4 本章介绍的全局变量

le_softc是一个数组，因为这里可以有多个以太网接口。

4.2.2 统计量

结构ifnet中为每个接口收集的统计量如图4-5所示。

图4-6显示了netstat命令的一些输出例子，包括ifnet结构中的一些统计信息。

第1列包含显示为一个字符串的if_name和if_unit。若接口是关闭的（不设置IFF_UP），一个星号显示在这个名字的旁边。在图4-6中，sl0、sl2和sl3是关闭的。

第2列显示的是if_mtu。在表头“Network”和“Address”底下的输出依赖于地址的类型。对于链路层地址，显示了结构sockaddr_dl的sdl_data的内容。对于IP地址，显示了子网和单播地址。其余的列是if_ipackets、if_ierrors、if_opackets、if_oerrors和if_collisions。

- 在输出中冲突的分组大约有3% (942 798 / 23 234 729 = 3%)。
- 这个机器的SLIP输出队列从未满过，因为SLIP接口的输出没有差错。
- 在传输中，LANCE硬件检测到12个以太网的输出差错。其中有些差错可能被视为冲突。

ifnet成员	说　明	用于SNMP
if_collisions	在CSMA接口的冲突数	
if_ibytes	接收到的字节总数	•
if_ierrors	接收到的有输入差错分组数	•
if_imcasts	接收到的多播分组数	•
if_ipackets	在接口接收到的分组数	•
if_iqdrops	被此接口丢失的输入分组数	•
if_lastchange	上一次改变统计的时间	•
if_noproto	指定为不支持协议的分组数	•
if_obytes	发送的字节总数	•
if_oerrors	接口上输出的差错数	•
if_omcasts	发送的多播分组数	•
if_opackets	接口上发送的分组数	•
if_snd.ifq_drops	在输出期间丢失的分组数	•
if_snd.ifq_len	输出队列中的分组数	

图4-5　结构ifnet 中维护的统计

netstat -i output

Name	Mtu	Network	Address	Ipkts	Ierrs	Opkts	Oerrs	Coll
le0	1500	<Link>8.0.9.13.d.33		28680519	814	29234729	12	942798
le0	1500	128.32.33	128.32.33.5	28680519	814	29234729	12	942798
sl0*	296	<Link>		54036	0	45402	0	0
sl0*	296	128.32.33	128.32.33.5	54036	0	45402	0	0
sl1	296	<Link>		40397	0	33544	0	0
sl1	296	128.32.33	128.32.33.5	40397	0	33544	0	0
sl2*	296	<Link>		0	0	0	0	0
sl3*	296	<Link>		0	0	0	0	0
lo0	1536	<Link>		493599	0	493599	0	0
lo0	1536	127	127.0.0.1	493599	0	493599	0	0

图4-6　接口统计的样本

• 硬件检测出814个以太网的输入差错，例如分组太短或错误的检验和。

4.2.3　SNMP变量

图4-7所示的是SNMP接口表(ifTable)中的一个接口项对象(ifEntry)，它包含在每个接口的ifnet结构中。

ISODE SNMP代理从if_type获得ifSpeed，并为ifAdminStatus维护一个内部变量。代理的ifLastChange基于结构ifnet中的if_lastchange，但与代理的启动时间相关，而不是与系统的启动时间相关。代理为ifSpecific返回一个空变量。

接口表，索引=<*ifIndex*>		
SNMP变量	ifnet成员	说　明
ifIndex	if_index	唯一地标识接口
ifDescr	if_name	接口的文本名称
ifType	if_type	接口的类型(例如以太网、SLIP等等)

图4-7　接口表ifTable 的变量

接口表，索引=<*ifIndex*>		
SNMP变量	ifnet成员	说 明
ifMtu	if_mtu	接口的MTU(字节)
ifSpeed	(看正文)	接口的正常速率(每秒比特)
ifPhysAddress	ac_enaddr	媒体地址(来自结构arpcom)
ifAdminStatus	(看正文)	接口的期望状态(IFF_UP标志)
ifOperStatus	if_flags	接口的操作状态(IFF_UP标志)
ifLastChange	(看正文)	上一次统计改变时间
ifInOctets	if_ibytes	输入的字节总数
ifInUcastPkts	if_ipackets -if_imcasts	输入的单播分组数
ifInNUcastPkts	if_imcasts	输入的广播或多播分组数
ifInDiscards	if_iqdrops	因为实现的限制而丢弃的分组数
ifInErrors	if_ierrors	差错的分组数
ifInUnknownProtos	if_noproto	指定为未知协议的分组数
ifOutOctets	if_obytes	输出字节数
ifOutUcastPkts	if_opackets-if_omcasts	输出的单播分组数
ifOutNUcastPkts	if_omcasts	输出的广播或多播分组数
ifOutDiscards	if_snd.ifq_drops	因为实现的限制而丢失的输出分组数
ifOutErrors	if_oerrors	因为差错而丢失的输出分组数
ifOutQLen	if_snd.ifq_len	输出队列长度
ifSpecific	n/a	媒体专用信息的SNMP对象ID(未实现)

图4-7 (续)

4.3 以太网接口

Net/3以太网设备驱动程序都遵循同样的设计。对于大多数 Unix设备驱动程序来说，都是这样，因为写一个新接口卡的驱动程序总是在一个已有的驱动程序的基础上修改而来的。在本节，我们简要地概述一下以太网的标准和一个以太网驱动程序的设计。我们用 LANCE驱动程序来说明这个设计。

图4-8说明了一个IP分组的以太网封装。

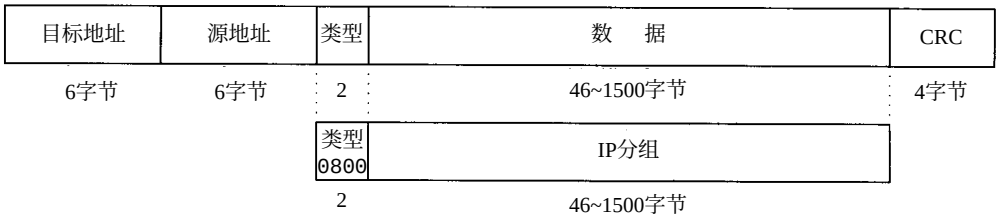

图4-8 一个IP分组的以太网封装

以太网帧包括48 bit的目标地址和源地址，接下来是一个 16 bit的类型字段，它标识这个帧所携带的数据的格式。对于 IP分组，类型是0x0800(2048)。帧的最后是一个32 bit的CRC(循环冗余检验)，它用来检查帧中的差错。

我们所讨论的最初的以太网组帧的标准在1982年由Digital设备公司、Intel公司及施乐公司发布，并作为今天在TCP/IP网络中最常用的格式。另一个可选的格式是IEEE(电气电子工程师协会)规定的802.2和802.3标准。更多的IEEE标准详见[Stallings 1987]。

对于以太网，IP分组的封装由RFC 894[Hornig 1984]规定，而对于802.3网，却由RFC1042[Postel和Reynolds 1988]规定。

我们用48 bit的以太网地址作为硬件地址。IP地址到硬件地址之间的转换用ARP协议(RFC 826 [Plummer 1982])，这个协议在第21章讨论。而硬件地址到IP地址的转换用RARP协议(RFC 903 [Finlayson et al. 1984])。以太网地址有两种类型：单播和多播。一个单播地址描述一个单一的以太网接口，而一个多播地址描述一组以太网接口。一个以太网广播是一个所有接口都接收的多播。以太网单播地址由设备的厂商分配，也有一些设备的地址允许用软件改变。

一些DECNET协议要求标识一个多接口主机的硬件地址，因此DECNET必须能改变一个设备的以太网单播地址。

图4-9列举了以太网接口的数据结构和函数。

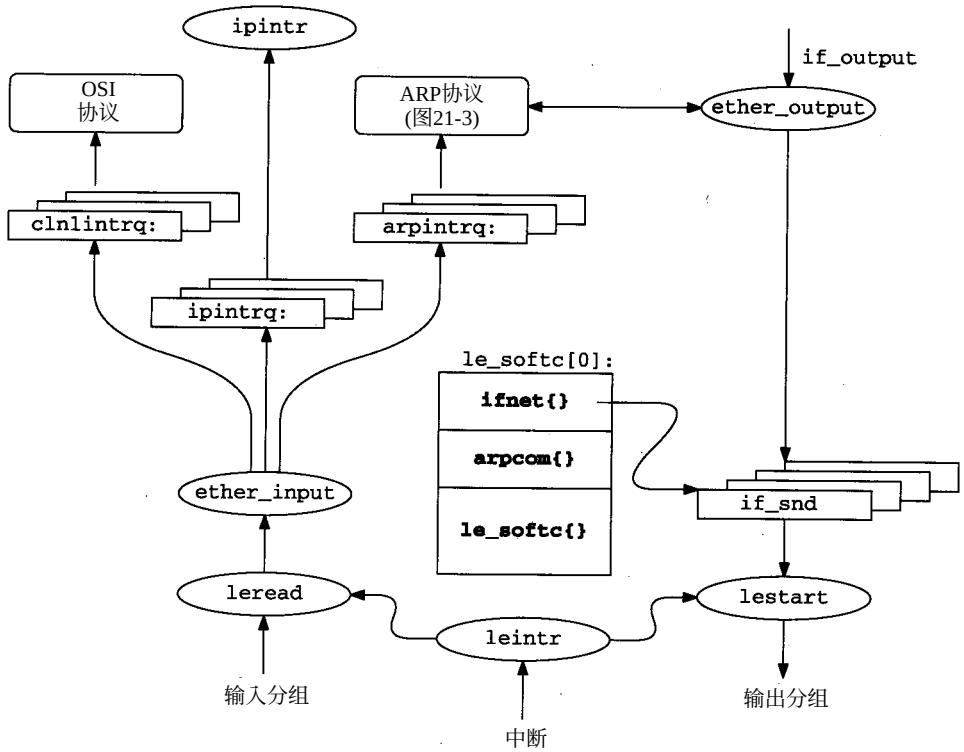

图4-9 以太网设备驱动程序

在图中：用一个椭圆标识一个函数（leintr）、用一个方框标识数据结构(le_softc[0])、le_softc及用圆角方框标识一组函数(ARP协议)。

图4-9左上角显示的是OSI无连接网络层(clnl)协议、IP和ARP的输入队列。对于clnlintrq，我们不打算讲更多，将它包含进来是为了强调ether_input要将以太网帧分用到多个协议队列中。

在技术上，OSI使用无连接网络协议(CLNP而不是CLNL)，但我们使用的是Net/3中的术语。CLNP的官方标准是ISO 8473。[Stallings 1993]对这个标准作了概述。

接口结构le_softc在图4-9的中间。我们感兴趣的是这个结构中的 ifnet和arpcom，其他是LANCE硬件的专用部分。我们在图 3-6中显示了结构ifnet，在图3-26中显示了结构arpcom。

4.3.1 leintr函数

我们从以太网帧的接收开始。现在，假设硬件已初始化并且系统已完成配置，当接口产生一个中断时，leintr被调用。在正常操作中，一个以太网接口接收发送到它的单播地址和以太网广播地址的帧。当一个完整的帧可用时，接口就产生一个中断，并且内核调用 leintr。

在第12章中，我们会看见可能要配置多个以太网接口来接收以太网多播帧 (不同于广播)。

有些接口可以配置为运行在混杂方式。在这种方式下，接口接收所有出现在网络上的帧。在卷1中讨论的tcpdump程序可以使用BPF (BSD分组过滤程序)来利用这种特性。

leintr检测硬件，并且如果有一个帧到达，就调用 leread把这个帧从接口转移到一个mbuf链中(用m_devget)。如果硬件报告一个帧已传输完或发现一个差错 (如一个有错误的检验和)，则leintr更新相应的接口统计，复位这个硬件，并调用 lestart来传输另一个帧。

所有以太网设备驱动程序将它们接收到的帧传给 ether_input做进一步的处理。设备驱动程序构造的 mbuf链不包括以太网首部，以太网首部作为一个独立的参数传递给ether_input。结构ether_header显示在图4-10中。

38-42 以太网CRC并不总是可用。它由接口硬件来计算与检验，接口硬件丢弃到达的 CRC差错帧。以太网设备驱动程序负责 ether_type的网络和主机字节序间的转换。在驱动程序外，它总是主机字节序。

if_ether.h

```
38 struct ether_header {
39     u_char  ether_dhost[6];      /* Ethernet destination address */
40     u_char  ether_shost[6];      /* Ethernet source address */
41     u_short ether_type;          /* Ethernet frame type */
42 };
```

if_ether.h

图4-10 结构ether_header

4.3.2 leread函数

函数leread(图4-11)的开始是由leintr传给它的一个连续的内存缓冲区，并且构造了一个ether_header结构和一个mbuf链。这个链表存储来自以太网帧的数据。 leread还将输入帧传给BPF。

if_le.c

```
528 leread(unit, buf, len)
529 int     unit;
530 char    *buf;
531 int     len;
532 {
533     struct le_softc *le = &le_softc[unit];
```

图4-11 函数leread

```
534     struct ether_header *et;
535     struct mbuf *m;
536     int     off, resid, flags;

537     le->sc_if.if_ipackets++;
538     et = (struct ether_header *) buf;
539     et->ether_type = ntohs((u_short) et->ether_type);
540     /* adjust input length to account for header and CRC */
541     len = len - sizeof(struct ether_header) - 4;
542     off = 0;

543     if (len <= 0) {
544         if (ledebug)
545             log(LOG_WARNING,
546                 "le%d: ierror(runt packet): from %s: len=%d\n",
547                 unit, ether_sprintf(et->ether_shost), len);
548         le->sc_runt++;
549         le->sc_if.if_ierrors++;
550         return;
551     }
552     flags = 0;
553     if (bcmp((caddr_t) etherbroadcastaddr,
554              (caddr_t) et->ether_dhost, sizeof(etherbroadcastaddr)) == 0)
555         flags |= M_BCAST;
556     if (et->ether_dhost[0] & 1)
557         flags |= M_MCAST;

558     /*
559      * Check if there's a bpf filter listening on this interface.
560      * If so, hand off the raw packet to enet.
561      */
562     if (le->sc_if.if_bpf) {
563         bpf_tap(le->sc_if.if_bpf, buf, len + sizeof(struct ether_header));

564         /*
565          * Keep the packet if it's a broadcast or has our
566          * physical ethernet address (or if we support
567          * multicast and it's one).
568          */

569         if ((flags & (M_BCAST | M_MCAST)) == 0 &&
570             bcmp(et->ether_dhost, le->sc_addr,
571                  sizeof(et->ether_dhost)) != 0)
572             return;
573     }
574     /*
575      * Pull packet off interface.  Off is nonzero if packet
576      * has trailing header; m_devget will then force this header
577      * information to be at the front, but we still have to drop
578      * the type and length which are at the front of any trailer data.
579      */
580     m = m_devget((char *) (et + 1), len, off, &le->sc_if, 0);
581     if (m == 0)
582         return;
583     m->m_flags |= flags;
584     ether_input(&le->sc_if, et, m);
585 }
```

if_le.c

图4-11 (续)

528-539 函数leintr给leread传了三个参数：unit，它标识接收到此帧的特定接口

卡；buf，它指向接收到的帧；len，它是帧的字节数(包括首部和CRC)。

函数将et指向这个缓存的开始，并且将以太网字节序转换成主机字节序，来构造结构ether_header。

540-551 将len减去以太网首部和CRC的大小得到数据的字节数。短分组(runt packet)是一个长度太短的非法以太网帧，它被记录、统计，并被丢弃。

552-557 接下来，目标地址被检测，并判断是不是以太网广播或多播地址。以太网广播地址是一个以太网多播地址的特例；它的每一比特都被设置了。etherbroadcastaddr是一个数组，定义如下：

```
u_char etherbroadcastaddr[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff };
```

> 这是C语言中定义一个48 bit值的简便方法。这项技术仅在我们假设字符是 8 bit值时才起作用——ANSI C并不保证这一点。

bcmp比较etherbroadcastaddr和ether_dhost，若相同，则设置标志M_BCAST。一个以太网多播地址由这个地址的首字节的低位比特来标识，如图 4-12所示。

图4-12 检测一个以太网多播地址

在第12章中，我们会看到并不是所有以太网多播帧都是 IP多播数据报，并且IP必须进一步检测这个分组。

如果这个地址的多播比特被置位，在 mbuf首部中设置M_MCAST。检测的顺序是重要的：首先ether_input将整个48 bit地址和以太网广播地址比较，若不同，则检测标识以太网多播地址的首字节的低位比特(习题4.1)。

558-573 如果接口带有BPF，调用bpf_tap把这个帧直接传给BPF。我们会看见对于SLIP和环回接口，要构造一个特定的BPF帧，因为这些网络没有一个链路层首部(不像以太网)。

当一个接口带有BPF时，它可以配置为运行在混淆模式，并且接收网络上出现的所有以太网帧，而不是通常由硬件接收的帧的子集。如果分组发送给一个不与此接口地址匹配的单播地址，则被leread丢弃。

574-585 m_devget (2.6节)将数据从传给leread的缓存中复制到一个它分配的mbuf链中。传给m_devget的第一个参数指向以太网首部后的第一个字节，它是此帧中的第一个数据字节。如果m_devget内存用完，leread立即返回。另外广播和多播标志被设置在链表中的第一个mbuf中，ether_input处理这个分组。

4.3.3 ether_input函数

函数ether_input显示在图4-13中，它检查结构ether_header来判断接收到的数据的类型，并将接收到的分组加入到队列中等待处理。

1. 广播和多播的识别

196-209 传给ether_input的参数有：ifp，一个指向接收此分组的接口的 ifnet结构的指针；eh，一个指向接收分组的以太网首部的指针； m，一个指向接收分组的指针(不包括以太网首部)。

任何到达不工作接口的分组将被丢弃。可能没有为接口配置一个协议地址，或者接口可能被程序ifconfig(8) (6.6节)显式地禁用了。

if_ethersubr.c

```
196 void
197 ether_input(ifp, eh, m)
198 struct ifnet *ifp;
199 struct ether_header *eh;
200 struct mbuf *m;
201 {
202     struct ifqueue *inq;
203     struct llc *l;
204     struct arpcom *ac = (struct arpcom *) ifp;
205     int     s;

206     if ((ifp->if_flags & IFF_UP) == 0) {
207         m_freem(m);
208         return;
209     }
210     ifp->if_lastchange = time;
211     ifp->if_ibytes += m->m_pkthdr.len + sizeof(*eh);
212     if (bcmp((caddr_t) etherbroadcastaddr, (caddr_t) eh->ether_dhost,
213              sizeof(etherbroadcastaddr)) == 0)
214         m->m_flags |= M_BCAST;
215     else if (eh->ether_dhost[0] & 1)
216         m->m_flags |= M_MCAST;
217     if (m->m_flags & (M_BCAST | M_MCAST))
218         ifp->if_imcasts++;

219     switch (eh->ether_type) {
220     case ETHERTYPE_IP:
221         schednetisr(NETISR_IP);
222         inq = &ipintrq;
223         break;

224     case ETHERTYPE_ARP:
225         schednetisr(NETISR_ARP);
226         inq = &arpintrq;
227         break;

228     default:
229         if (eh->ether_type > ETHERMTU) {
230             m_freem(m);
231             return;
232         }

                    /* OSI code */

307     }

308     s = splimp();
309     if (IF_QFULL(inq)) {
310         IF_DROP(inq);
311         m_freem(m);
312     } else
313         IF_ENQUEUE(inq, m);
314     splx(s);
315 }
```

if_ethersubr.c

图4-13 函数ether_input

210-218 变量time是一个全局的timeval结构，内核用它维护当前时间和日期，它是从Unix新纪元(1970年1月1日00:00:00，协调通用时间[UTC])开始的秒和微秒数。在[Itano and Ramsey 1993]中可以找到对UTC的简要讨论。我们在Net/3源代码中会经常遇到结构timeval：

```
struct timeval {
  long tv_sec;     /* seconds */
  long tv_usec;    /* and microseconds */
};
```

ether_input用当前时间更新if_lastchange，并且把if_ibytes加上输入分组的长度(分组长度加上14字节的以太网首部)。

然后，ether_input再次用leread去判断分组是否为一个广播或多播分组。

有些内核编译时可能没有包括BPF代码，因此测试必须在ether_input中进行。

2. 链路层分用

219-227 ether_input根据以太网类型字段来跳转。对于一个IP分组，schednetisr调度一个IP软件中断，并选择IP输入队列，ipintrq。对于一个ARP分组，调度ARP软件中断，并选择arpintrq。

一个isr是一个中断服务例程。

在原先的BSD版本中，当处于网络中断级别时，ARP分组通过调用arpinput立即被处理。通过分组排队，它们可以在软件中断级别被处理。

如果要处理其他以太网类型，一个内核程序员应在此增加其他情况的处理。或者，一个进程能用BPF接收其他以太网类型。例如，在Net/3中，RARP服务通常用BPF实现。

228-307 默认情况处理不识别以太网类型或按802.3标准(例如OSI无连接传输)封装的分组。以太网的type字段和802.3的length字段在一个以太网帧中占用同一位置。两种封装能够分辨出来，因为一个以太网封装的类型范围和802.3封装的长度范围是不同的(图4-14)。我们跳过OSI代码，在[Stallings 1993]中有对OSI链路层协议的说明。

范 围	说 明
0~1500	IEEE 802.3 *length*字段
1501~65535	以太网*type*字段：
2048	IP分组
2045	ARP分组

图4-14 以太网的*type*字段和802.3的*length*字段

有很多其他以太网类型值分配给各种协议；我们没有在图4-14中显示。在RFC 1700 [Reynolds and Postel 1994]中有一个有更多通用类型的列表。

3. 分组排队

308-315 最后，ether_input把分组放置到选择的队列中，若队列为空，则丢弃此分组。我们在图7-23和图21-16中会看到IP和ARP队列的默认限制为每个50个(ipqmaxlen)分组。

当ether_input返回时，设备驱动程序通知硬件它已准备接收下一分组，这时下一分组可能已存在于设备中。当schednetisr调度的软件中断发生时，处理分组输入队列(1.12节)。准确地说，调用ipintr来处理IP输入队列中的分组，调用arpintr来处理ARP输入队列中的分组。

4.3.4 ether_output函数

我们现在查看以太网帧的输出，当一个网络层协议，如 IP，调用此接口ifnet结构中指定的函数if_output时，开始处理输出。所有以太网设备的 if_output是ether_output (图4-2)。ether_output用14字节以太网首部封装一个以太网帧的数据部分，并将它放置到接口的发送队列中。这个函数比较大，我们分 4个部分来说明：

- 验证；
- 特定协议处理；
- 构造帧；
- 接口排队。

图4-15包括这个函数的第一个部分。

if_ethersubr.c

```
49 int
50 ether_output(ifp, m0, dst, rt0)
51 struct ifnet *ifp;
52 struct mbuf *m0;
53 struct sockaddr *dst;
54 struct rtentry *rt0;
55 {
56     short   type;
57     int     s, error = 0;
58     u_char  edst[6];
59     struct mbuf *m = m0;
60     struct rtentry *rt;
61     struct mbuf *mcopy = (struct mbuf *) 0;
62     struct ether_header *eh;
63     int     off, len = m->m_pkthdr.len;
64     struct arpcom *ac = (struct arpcom *) ifp;

65     if ((ifp->if_flags & (IFF_UP | IFF_RUNNING)) != (IFF_UP | IFF_RUNNING))
66         senderr(ENETDOWN);
67     ifp->if_lastchange = time;
68     if (rt = rt0) {
69         if ((rt->rt_flags & RTF_UP) == 0) {
70             if (rt0 = rt = rtalloc1(dst, 1))
71                 rt->rt_refcnt--;
72             else
73                 senderr(EHOSTUNREACH);
74         }
75         if (rt->rt_flags & RTF_GATEWAY) {
76             if (rt->rt_gwroute == 0)
77                 goto lookup;
78             if (((rt = rt->rt_gwroute)->rt_flags & RTF_UP) == 0) {
79                 rtfree(rt);
80                 rt = rt0;
81     lookup:     rt->rt_gwroute = rtalloc1(rt->rt_gateway, 1);

82                 if ((rt = rt->rt_gwroute) == 0)
83                     senderr(EHOSTUNREACH);
84             }
85         }
86         if (rt->rt_flags & RTF_REJECT)
87             if (rt->rt_rmx.rmx_expire == 0 ||
88                 time.tv_sec < rt->rt_rmx.rmx_expire)
89                 senderr(rt == rt0 ? EHOSTDOWN : EHOSTUNREACH);
90     }
```

if_ethersubr.c

图4-15 函数ether_output ：验证

49-64 ether_output的参数有：ifp，它指向输出接口的ifnet结构；m0，要发送的分组；dst，分组的目标地址；rt0，路由信息。

65-67 在ether_output中多次调用宏senderr。

```
#define senderr(e) { error = (e); goto bad;}
```

senderr保存差错码，并跳到函数的尾部 bad，在那里分组被丢弃，并且 ether_output返回error。

如果接口启动并在运行， ether_output更新接口的上次更改时间。否则，返回ENETDOWN。

1. 主机路由

68-74 rt0指向ip_output找到的路由项，并传递给 ether_output。如果从BPF调用ether_output，rt0可以为空。在这种情况下，控制转给图 4-16中的代码。否则，验证路由。如果路由无效，参考路由表，并且当路由不能被找到时，返回 EHOSTUNREACH。这时，rt0和rt指向一个到下一跳目的地的有效路由。

if_ethersubr.c

```
 91      switch (dst->sa_family) {

 92      case AF_INET:
 93          if (!arpresolve(ac, rt, m, dst, edst))
 94              return (0);         /* if not yet resolved */
 95          /* If broadcasting on a simplex interface, loopback a copy */
 96          if ((m->m_flags & M_BCAST) && (ifp->if_flags & IFF_SIMPLEX))
 97              mcopy = m_copy(m, 0, (int) M_COPYALL);
 98          off = m->m_pkthdr.len - m->m_len;
 99          type = ETHERTYPE_IP;
100          break;
101      case AF_ISO:
                                    /* OSI code */
142      case AF_UNSPEC:
143          eh = (struct ether_header *) dst->sa_data;
144          bcopy((caddr_t) eh->ether_dhost, (caddr_t) edst, sizeof(edst));
145          type = eh->ether_type;
146          break;

147      default:
148          printf("%s%d: can't handle af%d\n", ifp->if_name, ifp->if_unit,
149                 dst->sa_family);
150          senderr(EAFNOSUPPORT);
151      }
```

if_ethersubr.c

图4-16 函数ether_output ：网络协议处理

2. 网关路由

75-85 如果分组的下一跳是一个网关 (而不是最终目的)，找到一个到此网关的路由，并且rt指向它。如果不能发现一个网关路由，则返回 EHOSTUNREACH。这时， rt指向下一跳目的地的路由。下一跳可能是一个网关或最终目标地址。

3. 避免ARP泛洪

86-90 当目标方不准备响应 ARP请求时， ARP代码设置标志RTF_REJECT来丢弃到达目标

方的分组。这在图21-24中描述。

ether_output根据此分组的目标地址继续处理。因为以太网设备仅响应以太网地址，要发送一个分组，ether_output必须发现下一跳目的地的IP地址所对应的以太网地址。ARP协议(第21章)用来实现这个转换。图4-16显示了驱动程序是如何访问ARP协议的。

4. IP输出

91-101 ether_output根据目标地址中的sa_family进行跳转。我们在图4-16中仅显示了case为AF_INET、AF_ISO和AF_UNSPEC的代码，而略过了case AF_ISO的代码。

case AF_INET调用arpresolve来决定与目标IP地址相对应的以太网地址。如果以太网地址已存在于ARP高速缓存中，则arpresolve返回1，并且ether_output继续执行。否则，这个IP分组由ARP控制，并且ARP判断地址，从函数in_arpinput调用ether_output。

假设ARP高速缓存包含硬件地址，ether_output检查是否分组要广播，并且接口是否是单向的(例如，它不能接收自己发送的分组)。如果都成立，则m_copy复制这个分组。在执行switch后，这个复制的分组同到达以太网接口的分组一样进行排队。这是广播定义的要求，发送主机必须接收这个分组的一个备份。

我们在第12章会看到多播分组可能会环回到输出接口而被接收。

5. 显式以太网输出

142-146 有些协议，如ARP，需要显式地指定以太网目的地和类型。地址族类常量AF_UNSPEC指出：dst指向一个以太网首部。bcopy复制edst中的目标地址，并把以太网类型设为type。它不必调用arpresolve(如AF_INET)，因为以太网目标地址已由调用者显式地提供了。

6. 未识别的地址族类

147-151 未识别的地址族类产生一个控制台消息，并且ether_output返回EAFNOSUPPORT。

图4-17所示的是ether_output的下一部分：构造以太网帧。

if_ethersubr.c

```
152     if (mcopy)
153         (void) looutput(ifp, mcopy, dst, rt);
154     /*
155      * Add local net header.  If no space in first mbuf,
156      * allocate another.
157      */
158     M_PREPEND(m, sizeof(struct ether_header), M_DONTWAIT);
159     if (m == 0)
160         senderr(ENOBUFS);
161     eh = mtod(m, struct ether_header *);
162     type = htons((u_short) type);
163     bcopy((caddr_t) &type, (caddr_t) &eh->ether_type,
164         sizeof(eh->ether_type));
165     bcopy((caddr_t)edst, (caddr_t)eh->ether_dhost, sizeof (edst));
166     bcopy((caddr_t)ac->ac_enaddr, (caddr_t)eh->ether_shost,
167         sizeof(eh->ether_shost));
```

if_ethersubr.c

图4-17 函数ether_output：构造以太网帧

7. 以太网首部

152-167 如果在switch中的代码复制了这个分组，这个分组副本同在输出接口上接收到

的分组一样通过调用looutput来处理。环回接口和looutput在5.4节讨论。

M_PREPEND确保在分组的前面有14字节的空间。

大多数协议要在mbuf链表的前面留一些空间，因此，M_PREPEND仅需要调整一些指针(例如，16.7节中UDP输出的sosend和13.6节的igmp_sendreport)。

ether_output用type、edst和ac_enaddr(图3-26)构成以太网首部。ac_enaddr是与此输出接口关联的以太网单播地址，并且是所有从此接口传输的帧的源地址。ether_header用ac_enaddr重写调用者可能在ether_header结构中指定的源地址。这使得伪造一个以太网帧的源地址变得更难。

这时，mbuf包含一个除32 bit CRC以外的完整以太网帧，CRC由以太网硬件在传输时计算。图4-18所示的代码对设备要传送的帧进行排队。

if_ethersubr.c

```
168     s = splimp();
169     /*
170      * Queue message on interface, and start output if interface
171      * not yet active.
172      */
173     if (IF_QFULL(&ifp->if_snd)) {
174         IF_DROP(&ifp->if_snd);
175         splx(s);
176         senderr(ENOBUFS);
177     }
178     IF_ENQUEUE(&ifp->if_snd, m);
179     if ((ifp->if_flags & IFF_OACTIVE) == 0)
180         (*ifp->if_start) (ifp);
181     splx(s);
182     ifp->if_obytes += len + sizeof(struct ether_header);
183     if (m->m_flags & M_MCAST)
184         ifp->if_omcasts++;
185     return (error);

186   bad:
187     if (m)
188         m_freem(m);
189     return (error);
190 }
```

if_ethersubr.c

图4-18 函数ether_output：输出排队

168-185 如果输出队列为空，ether_output丢弃此帧，并返回ENOBUFS。如果输出队列不为空，这个帧放置到接口的发送队列中，并且若接口未激活，接口的if_start函数传输下一帧。

186-190 宏senderr跳到bad，在这里帧被丢弃，并返回一个差错码。

4.3.5 lestart函数

函数lestart从接口输出队列中取出排队的帧，并交给LANCE以太网卡发送。如果设备空闲，调用此函数开始发送帧。ether_output(图4-18)的最后是一个例子，直接通过接口的if_start函数调用lestart。

如果设备忙，当它完成了当前帧的传输时产生一个中断。设备调用lestart来退队并传输下一帧。一旦开始，协议层不再用调用lestart来排队帧，因为驱动程序不断退队并传输

帧，直到队列为空为止。

图4-19所示的是函数lestart。lestart假设已调用splimp来阻塞所有设备中断。

if_le.c

```
325 lestart(ifp)
326 struct ifnet *ifp;
327 {
328     struct le_softc *le = &le_softc[ifp->if_unit];
329     struct letmd *tmd;
330     struct mbuf *m;
331     int      len;

332     if ((le->sc_if.if_flags & IFF_RUNNING) == 0)
333         return (0);

                /* device-specific code */

335     do {

                /* device-specific code */

340         IF_DEQUEUE(&le->sc_if.if_snd, m);
341         if (m == 0)
342             return (0);
343         len = leput(le->sc_r2->ler2_tbuf[le->sc_tmd], m);
344         /*
345          * If bpf is listening on this interface, let it
346          * see the packet before we commit it to the wire.
347          */
348         if (ifp->if_bpf)
349             bpf_tap(ifp->if_bpf, le->sc_r2->ler2_tbuf[le->sc_tmd],
350                     len);

                /* device-specific code */

359     } while (++le->sc_txcnt < LETBUF);
360     le->sc_if.if_flags |= IFF_OACTIVE;
361     return (0);
362 }
```

if_le.c

图4-19 函数lestart

1. 接口必须初始化

325-333 如果接口没有初始化，lestart立即返回。

2. 将帧从输出队列中退队

335-342 如果接口已初始化，下一帧从队列中移去。如果接口输出队列为空，则lestart返回。

3. 传输帧并传递给BPF

343-350 leput将m中的帧复制到leput第一个参数所指向的硬件缓存中。如果接口带有BPF，将帧传给bpf_tap。我们跳过硬件缓存中帧传输的设备专用初始化代码。

4. 如果设备准备好，重复发送多帧

359 当le->sc_txcnt等于LETBUF时，lestart停止给设备传送帧。有些以太网接口能排队多个以太网输出帧。对于LANCE驱动器，LETBUF是此驱动器硬件传输缓存的可用个数，并且le->sc_txcnt保持跟踪有多少个缓存被使用。

5. 将设备标记为忙

360-362 最后，lestart在ifnet结构中设置IFF_OACTIVE来标识这个设备忙于传输帧。

在设备中将多个要传输的帧进行排队有一个负面影响。根据 [Jacobson 1998a]，LANCE芯片能够在两个帧间以很小的时延传输排队的帧。不幸的是，有些 (差的)以太网设备会丢失帧，因为它们不能足够快地处理输入的数据。

在一个应用如NFS中，这会很糟糕地互相影响。 NFS发送大的UDP数据报(经常是超过8192字节)，数据报被IP分片，并在LANCE设备中作为多个以太网帧排队。分片在接收方丢失，当NFS重传整个UDP数据报时，会导致很多未完成的数据报极大的时延。

Jacobson提出Sun的LANCE驱动器一次只排队一个帧就可能避免这一问题。

4.4 ioctl系统调用

ioctl系统调用提供一个通用命令接口，一个进程用它来访问一个设备的标准系统调用所不支持的特性。ioctl的原型为：

```
int ioctl(int fd, unsigned long com,…);
```

fd是一个描述符，通常是一个设备或网络连接。每种类型的描述符都支持它自己的一套ioctl命令，这套命令由第二个参数com来指定。第三个参数在原型中显示为“ ...”，因为它是依赖于被调用的ioctl命令的类型的指针。如果命令要取回信息，第三个参数必须是指向一个足够保存数据的缓存的指针。在本书中，我们仅讨论用于插口描述符的 ioctl命令。

我们显示的系统调用的原型是一个进程进行系统调用的原型。在第 15章中我们会看见在内核中的这个函数还有一个不同的原型。

我们在第17章讨论系统调用ioctl的实现，但在本书的各个部分讨论 ioctl单个命令的实现。

我们讨论的第一个ioctl命令提供对讨论过的网络接口结构的访问。我们总结的本书中所有的ioctl命令如图4-20所示。

命 令	第三个参数	函 数	说 明
SIOCGIFCONF	struct ifconf *	ifconf	获取接口配置清单
SIOCGIFFLAGS	struct ifreq *	ifioctl	获得接口标志
SIOCGIFMETRIC	struct ifreq *	ifioctl	获得接口度量
SIOCSIFFLAGS	struct ifreq *	ifioctl	设置接口标志
SIOCSIFMETRIC	struct ifreq *	ifioctl	设置接口度量

图4-20 接口ioctl 的命令

第一列显示的符号常量标识 ioctl命令(第二个参数， com)。第二列显示传递给第一列所显示的命令的系统调用的第三个参数的类型。第三列是实现这个命令的函数的名称。

图4-21显示处理ioctl命令的各种函数的组织。带阴影的函数我们在本章中说明。其余

的函数在其他章说明。

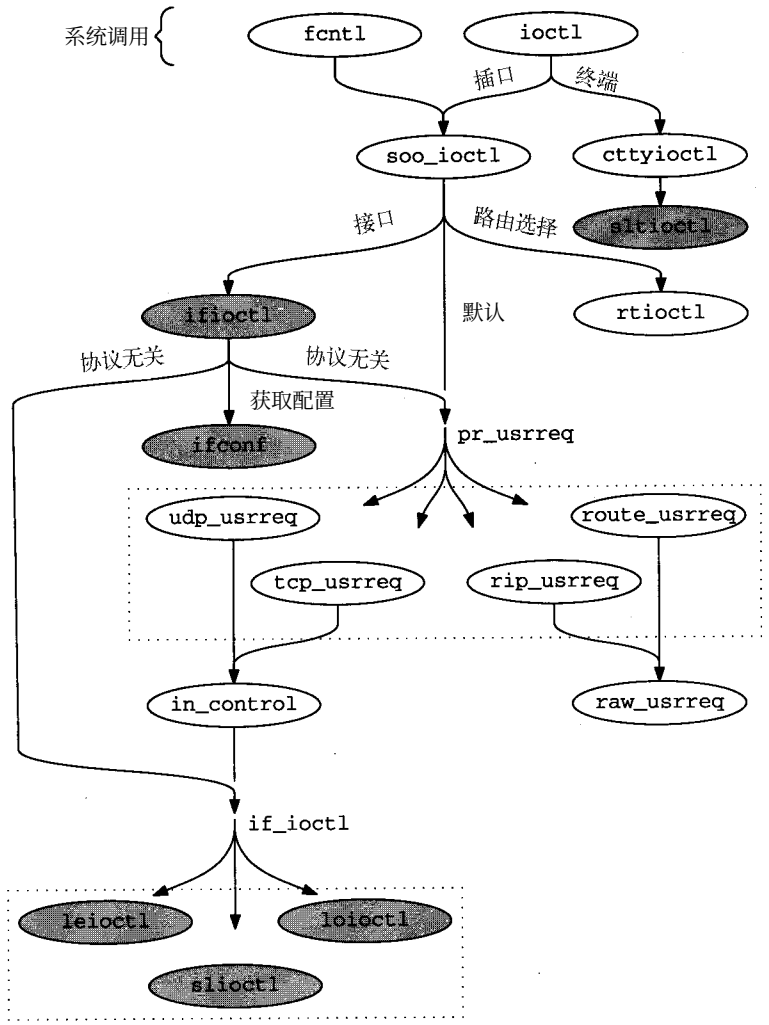

图4-21 在本章说明的ioctl 函数

4.4.1 ifioctl函数

系统调用ioctl将图4-20所列的5种命令传递给图4-22所示的ifioctl函数。

394-405 对于命令SIOCGIFCONF，ifioctl调用ifconf来构造一个可变长ifreq结构的表。

406-410 对于其他ioctl命令，数据参数是指向一个 ifreq结构的指针。 ifunit在ifnet列表中查找名称为进程在ifr->ifr_name中提供的文本名称(例如：“sl0”，“le1”或“lo0”)的接口。如果没有匹配的接口， ifioctl返回ENXIO。剩下的代码依赖于cmd，它们在图4-29中说明。

447-454 如果接口ioctl命令不能被识别，ifioctl把命令发送给与所请求插口关联的协议的用户要求函数。对于IP，这些命令以一个UDP插口发送并调用udp_usrreq。这一类命

令在图6-10中描述。23.10节将详细讨论函数udp_usrreq。

如果控制到达switch语句外，返回0。

if.c

```
394 int
395 ifioctl(so, cmd, data, p)
396 struct socket *so;
397 int     cmd;
398 caddr_t data;
399 struct proc *p;
400 {
401     struct ifnet *ifp;
402     struct ifreq *ifr;
403     int     error;

404     if (cmd == SIOCGIFCONF)
405         return (ifconf(cmd, data));

406     ifr = (struct ifreq *) data;
407     ifp = ifunit(ifr->ifr_name);
408     if (ifp == 0)
409         return (ENXIO);
410     switch (cmd) {

        /* other interface ioctl commands (figures 4.29 and 12.11) */

447     default:
448         if (so->so_proto == 0)
449             return (EOPNOTSUPP);
450         return ((*so->so_proto->pr_usrreq) (so, PRU_CONTROL,
451                                             cmd, data, ifp));
452     }
453     return (0);
454 }
```

if.c

图4-22 函数ifioctl ：综述与SIOCGIFCONF

4.4.2 ifconf函数

ifconf为进程提供一个标准的方法来发现一个系统中的接口和配置的地址。由结构ifreq和ifconf表示的接口信息如图4-23和图4-24所示。

262-279 一个ifreq结构包含在ifr_name中一个接口的名称。在联合中的其他成员被各种ioctl命令访问。通常，用宏来简化对联合的成员的访问语法。

292-300 在结构ifconf中，ifc_len是ifc_buf指向的缓存的字节数。这个缓存由一个进程分配，但由ifconf用一个具有可变长ifreq结构的数组来填充。对于函数ifconf，ifr_addr是结构ifreq中联合的相关成员。每个ifreq结构有一个可变长度，因为ifr_addr(一个sockaddr结构)的长度根据地址的类型而变。必须用结构sockaddr的成员sa_len来定位每项的结束。图4-25说明了ifconf所维护的数据结构。

在图4-25中，左边的数据在内核中，而右边的数据在一个进程中。我们用这个图来讨论图4-26中所示的ifconf函数。

462-474 ifconf的两个参数是：cmd，它被忽略；data，它指向此进程指定的ifconf结构的一个副本。

if.h

```
262 struct  ifreq {
263 #define IFNAMSIZ    16
264     char    ifr_name[IFNAMSIZ];                 /* if name, e.g. "en0" */
265     union {
266         struct  sockaddr ifru_addr;
267         struct  sockaddr ifru_dstaddr;
268         struct  sockaddr ifru_broadaddr;
269         short   ifru_flags;
270         int ifru_metric;
271         caddr_t ifru_data;
272     } ifr_ifru;
273 #define ifr_addr    ifr_ifru.ifru_addr          /* address */
274 #define ifr_dstaddr ifr_ifru.ifru_dstaddr       /* other end of p-to-p link */
275 #define ifr_broadaddr   ifr_ifru.ifru_broadaddr /* broadcast address */
276 #define ifr_flags   ifr_ifru.ifru_flags         /* flags */
277 #define ifr_metric  ifr_ifru.ifru_metric        /* metric */
278 #define ifr_data    ifr_ifru.ifru_data          /* for use by interface */
279 };
```

if.h

图4-23 结构ifreq

if.h

```
292 struct  ifconf {
293     int ifc_len;                    /* size of associated buffer */
294     union {
295         caddr_t ifcu_buf;
296         struct  ifreq *ifcu_req;
297     } ifc_ifcu;
298 #define ifc_buf ifc_ifcu.ifcu_buf   /* buffer address */
299 #define ifc_req ifc_ifcu.ifcu_req   /* array of structures returned */
300 };
```

if.h

图4-24 结构ifconf

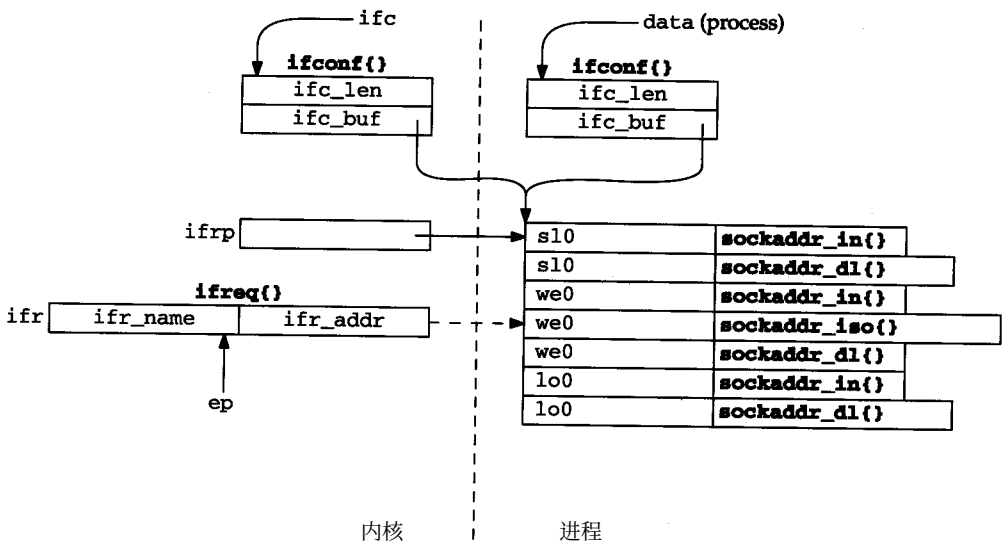

图4-25 ifconf 数据结构

if.c

```
462 int
463 ifconf(cmd, data)
464 int     cmd;
465 caddr_t data;
466 {
467     struct ifconf *ifc = (struct ifconf *) data;
468     struct ifnet *ifp = ifnet;
469     struct ifaddr *ifa;
470     char    *cp, *ep;
471     struct ifreq ifr, *ifrp;
472     int     space = ifc->ifc_len, error = 0;
473     ifrp = ifc->ifc_req;
474     ep = ifr.ifr_name + sizeof(ifr.ifr_name) - 2;
475     for (; space > sizeof(ifr) && ifp; ifp = ifp->if_next) {
476         strncpy(ifr.ifr_name, ifp->if_name, sizeof(ifr.ifr_name) - 2);
477         for (cp = ifr.ifr_name; cp < ep && *cp; cp++)
478             continue;
479         *cp++ = '0' + ifp->if_unit;
480         *cp = '\0';
481         if ((ifa = ifp->if_addrlist) == 0) {
482             bzero((caddr_t) & ifr.ifr_addr, sizeof(ifr.ifr_addr));
483             error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
484                             sizeof(ifr));
485             if (error)
486                 break;
487             space -= sizeof(ifr), ifrp++;
488         } else
489             for (; space > sizeof(ifr) && ifa; ifa = ifa->ifa_next) {
490                 struct sockaddr *sa = ifa->ifa_addr;
491                 if (sa->sa_len <= sizeof(*sa)) {
492                     ifr.ifr_addr = *sa;
493                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
494                                     sizeof(ifr));
495                     ifrp++;
496                 } else {
497                     space -= sa->sa_len - sizeof(*sa);
498                     if (space < sizeof(ifr))
499                         break;
500                     error = copyout((caddr_t) & ifr, (caddr_t) ifrp,
501                                     sizeof(ifr.ifr_name));
502                     if (error == 0)
503                         error = copyout((caddr_t) sa,
504                                         (caddr_t) & ifrp->ifr_addr, sa->sa_len);
505                     ifrp = (struct ifreq *)
506                         (sa->sa_len + (caddr_t) & ifrp->ifr_addr);
507                 }
508                 if (error)
509                     break;
510                 space -= sizeof(ifr);
511             }
512     }
513     ifc->ifc_len -= space;
514     return (error);
515 }
```

if.c

图4-26 函数ifconf

ifc是强制为一个ifconf结构指针的data。ifp从ifnet(列表头)开始遍历接口列表,

而ifa遍历每个接口的地址列表。 cp和ep控制构造在ifr中的接口文本名称， ifr是一个ifreq结构，它在接口名称和地址复制到进程的缓存前保存接口名称和地址。 ifrq指向这个缓存，并且在每个地址被复制后指向下一个。 space是进程缓存中剩余字节的个数， cp用来搜寻名称的结尾，而ep标志接口名称数字部分最后的可能位置。

475-488 for循环遍历接口列表。对于每个接口，文本名称被复制到 ifr_name，在ifr_name的后面跟着if_unit数的文本表示。如果没有给接口分配地址，一个全 0的地址被构造，所得的ifreq结构被复制到进程中，并减小space，增加ifrp。

489-515 如果接口有一个或多个地址，用 for循环来处理每个地址。地址加到 ifr中的接口名称中，然后ifr被复制到进程中。长度超过标准 sockaddr结构的地址不放到ifr中，并且直接复制到进程。在复制完每个地址后，调整 space和ifrp的值。所有接口处理完后，更新缓存长度 (ifc->ifc_len)，并且ifconf返回。系统调用 ioctl负责将结构ifconf中新的内容复制回进程中的结构 ifconf。

4.4.3 举例

图4-27显示了以太网、SLIP和环回接口被初始化后的接口结构的配置。

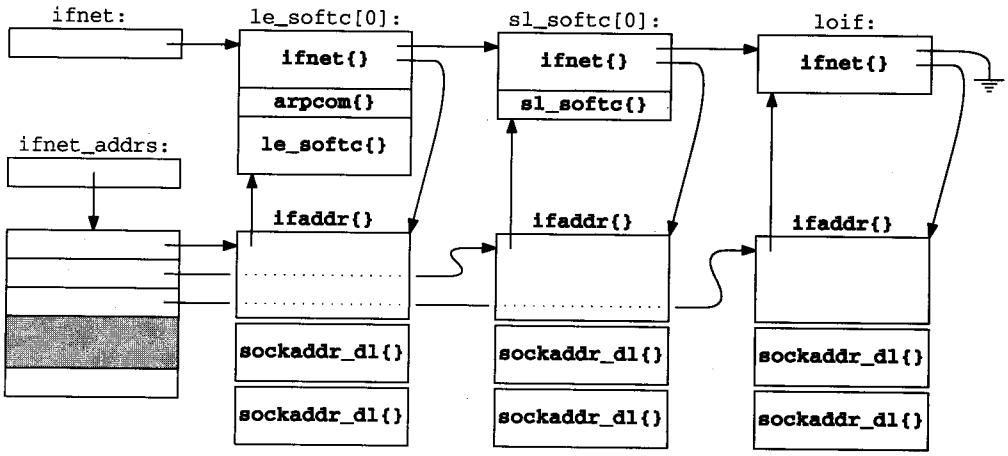

图4-27 接口和地址数据结构

图4-28显示了以下代码执行后的ifc和buffer的内容。

```
struct ifconf ifc;      /* SIOCGIFCONF adjusts this */
char buffer[144];       /* contains interface addresses when ioctl returns */
int s;                  /* any socket */

ifc.ifc_len = 144;
ifc.ifc_buf = buffer;
if (ioctl(s, SIOCGIFCONF, &ifc) < 0 ) {
    perror("ioctl failed");
    exit(1);
}
```

这里对命令SIOCGIFCONF操作的插口的类型没有限制，如我们所看到的，这个命令返回所有协议族类的地址。

在图4-28中，因为在缓存中返回的三个地址仅占用 108 (3×36)字节， ioctl将ifc_len

由144改为108。返回三个sockaddr_dl地址，并且这个缓存后面的36字节未用。每项的前16个字节包含接口的文本名称。在这里，这16字节中只有3个字节被使用。

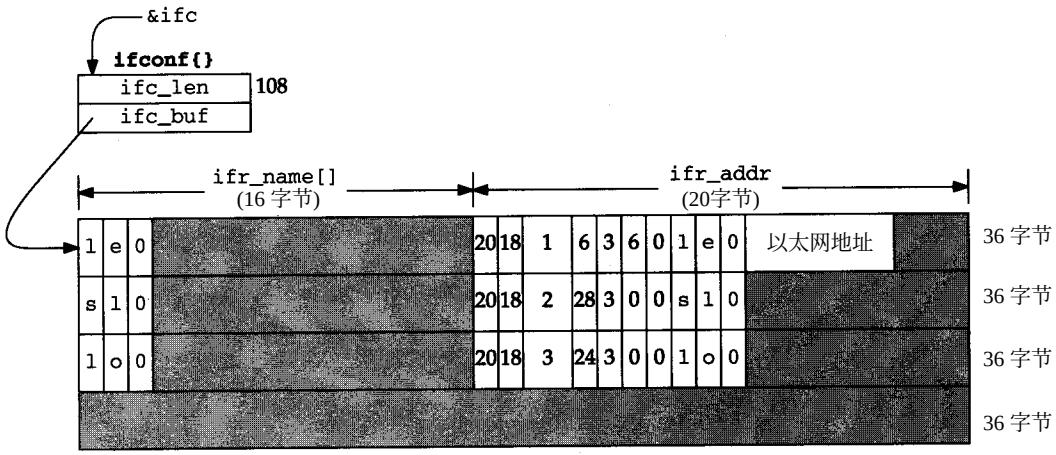

图4-28 SIOCGIFCONF 命令返回的数据

ifr_addr为一个sockaddr结构的形式，因此第一个值为长度(20字节)，且第二个值为地址的类型(18，AF_LINK)。接下来的一个值为sdl_index，与sdl_type一样，对于每个接口，它是不同的(与IFT_ETHER、IFT_SLIP和IFT_LOOP相对应的值为6、28和24)。

下面三个值为sa_nlen(文本名称的长度)、sa_alen(硬件地址的长度)及sa_slen(未用)。对于所有三项，sa_nlen都为3。以太网地址的sa_alen为6，而SLIP和环回接口的sa_alen为0。sa_slen总是为0。

最后，是接口的文本名称，其后面是硬件地址(仅对于以太网)。SLIP和环回接口在sockaddr_dl结构中不存放一个硬件级地址。

在此例中，仅返回sockaddr_dl地址(因为在图4-27中没有配置其他地址类型)，因此缓存中的每项大小一样。如果为每个接口配置其他地址(例如：IP或OSI地址)，它们会同sockaddr_dl地址一起返回，并且每项的大小根据返回的地址类型的不同而不同。

4.4.4 通用接口ioctl命令

图4-20中剩下的四个接口命令(SIOCGIFFLAGS、SIOCGIFMETRIC、SIOCSIFFLAGS和SIOCSIFMETRIC)由函数ifioctl处理。图4-29所示的是处理这些命令的case语句。

1. SIOCGIFLAGS和SIOCGIFMETRIC

410-416 对于两个SIOCG*xxx*命令，ifioctl将每个接口的if_flags或if_metric值复制到ifreq结构中。对于标志，使用联合的成员ifr_flags；而对于度量，使用成员ifr_metric(图4-23)。

2. SIOCSIFFLAGS

417-429 为改变接口的标志，调用进程必须有超级用户权限。如果进程正在关闭一个运行的接口或启动一个未运行的接口，分别调用if_down和if_up。

3. 忽略标志IFF_CANTCHAGE

430-434 回忆图3-7，有些接口标志不能被进程改变。表达式(ifp->if_flags&

IFF_CANTCHANGE)清除能被进程改变的接口标志，而表达式 (ifr->ifr_flags~IFF_CANTCHANGE)清除在请求中不被进程改变的标志。这两个表达式进行或运算并作为新值保存在ifp->if_flags中。在返回前，请求被传递给与设备相关联的 if_ioctl函数(例如：LANCE驱动器的leioctl —— 图4-31)。

if.c

```
410     switch (cmd) {
411     case SIOCGIFFLAGS:
412         ifr->ifr_flags = ifp->if_flags;
413         break;

414     case SIOCGIFMETRIC:
415         ifr->ifr_metric = ifp->if_metric;
416         break;

417     case SIOCSIFFLAGS:
418         if (error = suser(p->p_ucred, &p->p_acflag))
419             return (error);
420         if (ifp->if_flags & IFF_UP && (ifr->ifr_flags & IFF_UP) == 0) {
421             int     s = splimp();
422             if_down(ifp);
423             splx(s);
424         }
425         if (ifr->ifr_flags & IFF_UP && (ifp->if_flags & IFF_UP) == 0) {
426             int     s = splimp();
427             if_up(ifp);
428             splx(s);
429         }
430         ifp->if_flags = (ifp->if_flags & IFF_CANTCHANGE) |
431             (ifr->ifr_flags & ~IFF_CANTCHANGE);
432         if (ifp->if_ioctl)
433             (void) (*ifp->if_ioctl) (ifp, cmd, data);
434         break;

435     case SIOCSIFMETRIC:
436         if (error = suser(p->p_ucred, &p->p_acflag))
437             return (error);
438         ifp->if_metric = ifr->ifr_metric;
439         break;
```

if.c

图4-29 函数ifioctl ：标志和度量

4. SIOCSIFMETRIC

435-439 改变接口的度量要容易些；进程同样要有超级用户权限， ifioctl将接口新的度量复制到if_metric中。

4.4.5 if_down和if_up函数

利用程序ifconfig，一个管理员可以通过命令 SIOCSIFFLAGS设置或清除标志IFF_UP来启用或禁用一个接口。图4-30显示了函数if_down和if_up的代码。

292-302 当一个接口被关闭时， IFF_UP标志被清除并且对与接口关联的每个地址用pfctlinput(7.7节)发送命令PRC_IFDOWN。这给每个协议一个机会来响应被关闭的接口。有些协议，如OSI，要使用接口来终止连接。对于IP，如果可能，要通过其他接口为连接进行重新路由。TCP和UDP忽略失效的接口，并依赖路由协议去发现分组的可选路径。

if_qflush忽略接口的任何排队分组。rt_ifmsg通知路由系统发生的变化。TCP自动重传丢失的分组；UDP应用必须自己显式地检测这种情况，并对此作出响应。

308-315 当一个接口被启用时，IFF_UP标志被设置，并且rt_ifmsg通知路由系统接口状态发生变化。

if.c

```
292 void
293 if_down(ifp)
294 struct ifnet *ifp;
295 {
296     struct ifaddr *ifa;

297     ifp->if_flags &= ~IFF_UP;
298     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
299         pfctlinput(PRC_IFDOWN, ifa->ifa_addr);
300     if_qflush(&ifp->if_snd);
301     rt_ifmsg(ifp);
302 }

308 void
309 if_up(ifp)
310 struct ifnet *ifp;
311 {
312     struct ifaddr *ifa;

313     ifp->if_flags |= IFF_UP;
314     rt_ifmsg(ifp);
315 }
```

if.c

图4-30 函数if_down 和if_up

4.4.6 以太网、SLIP和环回

我们看图4-29中处理SIOCSIFFLAGS命令的代码，ifioctl调用接口的if_ioctl函数。在我们的三个例子接口中，函数 slioctl和loioctl为这个被ifioctl忽略的命令返回EINVAL。图4-31显示了函数leioctl及LANCE以太网驱动程序的SIOCSIFFLAGS命令的处理。

if_le.c

```
614 leioctl(ifp, cmd, data)
615 struct ifnet *ifp;
616 int     cmd;
617 caddr_t data;
618 {
619     struct ifaddr *ifa = (struct ifaddr *) data;
620     struct le_softc *le = &le_softc[ifp->if_unit];
621     struct lereg1 *ler1 = le->sc_r1;
622     int     s = splimp(), error = 0;

623     switch (cmd) {

                /* SIOCSIFADDR code (Figure 6.28) */

638     case SIOCSIFFLAGS:
```

图4-31 函数leioctl ：SIOCSIFFLAGS

```
639         if ((ifp->if_flags & IFF_UP) == 0 &&
640             ifp->if_flags & IFF_RUNNING) {
641             LERDWR(le->sc_r0, LE_STOP, ler1->ler1_rdp);
642             ifp->if_flags &= ~IFF_RUNNING;
643         } else if (ifp->if_flags & IFF_UP &&
644                   (ifp->if_flags & IFF_RUNNING) == 0)
645             leinit(ifp->if_unit);
646         /*
647          * If the state of the promiscuous bit changes, the interface
648          * must be reset to effect the change.
649          */
650         if (((ifp->if_flags ^ le->sc_iflags) & IFF_PROMISC) &&
651             (ifp->if_flags & IFF_RUNNING)) {
652             le->sc_iflags = ifp->if_flags;
653             lereset(ifp->if_unit);
654             lestart(ifp);
655         }
656         break;

            /* SIOCADDMULTI and SIOCDELMULTI code (Figure 12.31) */

672     default:
673         error = EINVAL;
674     }
675     splx(s);
676     return (error);
677 }
```

if_le.c

图4-31 (续)

614-623 leioctl把第三个参数data转换为一个ifaddr结构的指针，并保存在ifa中。le指针引用下标为ifp->if_unit的le_softc结构。基于cmd的switch语句构成了这个函数的主体。

638-656 在图4-31中仅显示了case SIOCSIFFLAGS。这次ifioctl调用leioctl，接口标志被改变。显示的代码强制物理接口进入标志所配置的状态。如果要关闭接口 (没有设置IFF_UP)，但接口正在工作，则关闭接口。若要启动未操作的接口，接口被初始化并重启。

如果混淆比特被改变，那么就关闭接口，复位，并重启来实现这种变化。

仅当要求改变IFF_PROMISC比特时包含异或和IFF_PROMISC的表达式才为真。

672-677 处理未识别命令的default情况分支发送EINVAL，并在函数的结尾将它返回。

4.5 小结

在本章中，我们说明了LANCE以太网设备驱动程序的实现，这个驱动程序在全书中多处引用。我们还看到了以太网驱动程序如何检测输入中的广播地址和多播地址，如何检测以太网和802.3封装，以及如何将输入的帧分用到相应的协议队列中。在第 21章中我们会看到IP地址(单播、广播和多播)是如何在输出转换成正确的以太网地址。

最后，我们讨论了协议专用的ioctl命令，它用来访问接口层数据结构。

习题

4.1 在leread中，当接收到一个广播分组时，总是设置标志 M_MCAST(除了M_BCAST外)。与ether_input的代码比较，为什么在leread和ether_input中设置此标志？它至关重要吗？哪个正确？

4.2 在ether_input(图4-13)中，如果交换广播地址和多播地址检测次序会发生什么情况？如果在检测多播地址的if语句前加上一个else会发生什么情况？