第3章 接 口 层

3.1 引言

本章开始讨论Net/3在协议栈底部的接口层，它包括在本地网上发送和接收分组的硬件与软件。

我们使用术语设备驱动程序来表示与硬件及网络接口 (或仅仅是接口)通信的软件，网络接口是指在一个特定网络上硬件与设备驱动器之间的接口。

Net/3接口层试图在网络协议和连接到一个系统的网络设备的驱动器间提供一个与硬件无关的编程接口。这个接口层为所有的设备提供以下支持：

- 一套精心定义的接口函数；
- 一套标准的统计与控制标志；
- 一个与设备无关的存储协议地址的方法；
- 一个标准的输出分组的排队方法。

这里不要求接口层提供可靠的分组传输，仅要求提供最大努力 (best-effort)的服务。更高协议层必须弥补这种可靠性缺陷。本章说明为所有网络接口维护的通用数据结构。为了说明相关数据结构和算法，我们参考 Net/3中三种特定的网络接口：

1) 一个AMD 7990 LANCE以太网接口：一个能广播局域网的例子。

2) 一个串行线IP(SLIP)接口：一个在异步串行线上的点对点网络的例子。

3) 一个环回接口：一个逻辑网络把所有输出分组作为输入返回。

3.2 代码介绍

通用接口结构和初始化代码可在三个头文件和两个 C文件中找到。在本章说明的设备专用初始化代码可在另外三个C文件中找到。所有的8个文件都列于图3-1中。

文 件	说 明
sys/socket.h	地址结构定义
net/if.h	接口结构定义
net/if_dl.h	链路层结构定义
kern/init_main.c	系统和接口初始化
net/if.c	通用接口代码
net/if_loop.c	环回设备驱动程序
net/if_sl.c	SLIP设备驱动程序
hp300/dev/if_le.c	LANCE以太网设备驱动程序

图3-1 本章讨论的文件

3.2.1 全局变量

在本章中介绍的全局变量列于图 3-2中。

变　量	数据类型	说　明
pdevinit	struct pdevinit[]	伪设备如SLIP和环回接口的初始化参数数组
ifnet	struct ifnet *	ifnet结构的列表的表头
ifnet_addrs	struct ifaddr **	指向链路层接口地址的指针数组
if_indexlim	int	数组ifnet_addrs的大小
if_index	int	上一个配置接口的索引
ifqmaxlen	int	接口输出队列的最大值
hz	int	这个系统的时钟频率(次/秒)

图3-2　本章中介绍的全局变量

3.2.2　SNMP变量

Net/3内核收集了大量的各种联网统计。在大多数章节中，我们都要总结这些统计并说明它们与定义在简单网络管理协议信息库 (SNMP MIB-II)中的标准TCP/IP信息和统计之间的关系。RFC 1213 [McCloghrie and Rose 1991]说明了SNMP MIB-II，它组织成如图3-3所示的10个不同的信息组。

SNMP组	说　明
System	系统通用信息
Interfaces	网络接口信息
Address Translation	网络地址到硬件地址的映射表(不推荐使用)
IP	IP协议信息
ICMP	ICMP协议信息
TCP	TCP协议信息
UDP	UDP协议信息
EGP	EGP协议信息
Transmission	媒体专用信息
SNMP	SNMP协议信息

图3-3　MIB-II中的SNMP组

Net/3并不包括一个SNMP代理。一个针对Net/3的SNMP代理是作为一个进程来实现的，它根据SNMP的要求通过2.2节描述的机制来访问这些内核统计。

Net/3收集大多数MIB-II变量并且能被SNMP代理直接访问，而其他的变量则要通过间接的方式来获得。MIB-II变量分为三类：(1)简单变量，例如一个整数值、一个时间戳或一个字节串；(2)简单变量的列表，例如一个单独的路由项或一个接口描述项；(3)表的列表，例如整个路由表和所有接口实体的列表。

ISODE包含有一个Net/3 SNMP代理例子。ISODE的信息见附录B。

图3-4所示的是一个为SNMP接口组维护的简单变量。我们在后面的图 4-7中描述SNMP接口表。

SNMP变量	Net/3变量	说　明
ifNumber	if_index + 1	if_index是系统中最后一个接口的索引值，并且起始为0；加1来获得系统中接口个数ifNumber

图3-4　在接口组中的一个简单的SNMP变量

3.3 ifnet结构

结构ifnet中包含所有接口的通用信息。在系统初始化期间，分别为每个网络设备分配一个独立的ifnet结构。每个ifnet结构有一个列表，它包含这个设备的一个或多个协议地址。图3-5说明了一个接口和它的地址之间的关系。

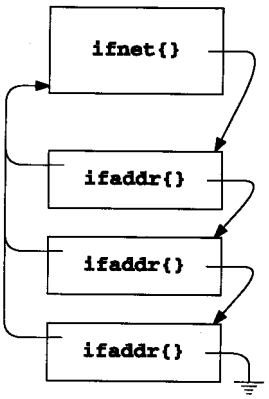

图3-5 每个ifnet结构有一个ifaddr结构的列表

在图3-5中的接口显示了3个存放在ifaddr结构中的协议地址。虽然一些网络接口，例如SLIP，仅支持一个协议；而其他接口，如以太网，支持多个协议并需要多个地址。例如，一个系统可能使用一个以太网接口同时用于Internet和OSI两个协议。一个类型字段标识每个以太网帧的内容，并且因为Internet和OSI协议使用不同的编址方式，以太网接口必须有一个Internet地址和一个OSI地址。所有地址用一个链表链接起来(图3-5右侧的箭头)，并且每个结构包含一个回指指针指向相关的ifnet结构(图3-5左侧的箭头)。

可能一个网络接口支持同一协议的多个地址。例如，在Net/3中可能为一个以太网接口分配两个Internet地址。

这个特点第一次是出现在Net/2中。当为一个网络重编地址时，一个接口有两个IP地址是有用的。在过渡期间，接口可以接收老地址和新地址的分组。

结构ifnet比较大，我们分五个部分来说明：

- 实现信息
- 硬件信息
- 接口统计
- 函数指针
- 输出队列

图3-6所示的是包含在结构ifnet中的实现信息。

if.h

```
80 struct ifnet {
81      struct ifnet *if_next;       /* all struct ifnets are chained */
82      struct ifaddr *if_addrlist; /* linked list of addresses per if */
83      char    *if_name;            /* name, e.g. 'le' or 'lo' */
84      short    if_unit;            /* sub-unit for lower level driver */
85      u_short if_index;            /* numeric abbreviation for this if  */
86      short    if_flags;           /* Figure 3.7 */
87      short    if_timer;           /* time 'til if_watchdog called */
88      int      if_pcount;          /* number of promiscuous listeners */
89      caddr_t if_bpf;              /* packet filter structure */
```

if.h

图3-6 ifnet结构：实现信息

80-82 if_next把所有接口的ifnet结构链接成一个链表。函数if_attach在系统初始化期间构造这个链表。if_addrlist指向这个接口的ifaddr结构列表(图3-16)。每个ifaddr结构存储一个要用这个接口通信的协议的地址信息。

1. 通用接口信息

83-86 if_name是一个短字符串，用于标识接口的类型，而if_unit标识多个相同类型的实例。例如，一个系统有两个SLIP接口，每个都有一个if_name，包含两字节的“s1”和一个if_unit。对第一个接口，if_unit为0；对第二个接口为1。if_index在内核中唯一地标识这个接口，这在sysctl系统调用(见19.14节)以及路由域中要用到。

有时一个接口并不被一个协议地址唯一地标识。例如，几个SLIP连接可以有同样的本地IP地址。在这种情况下，if_index明确地指明这个接口。

if_flags表明接口的操作状态和属性。一个进程能检查所有的标志，但不能改变在图3-7中“内核专用”列中作了记号的标志。这些标志用4.4节讨论的命令SIOCGIFFLAGS和SIOCSIFFLAGS来访问。

if_flags	内核专用	说　　明
IFF_BROADCAST	•	接口用于广播网
IFF_MULTICAST	•	接口支持多播
IFF_POINTOPOINT	•	接口用于点对点网络
IFF_LOOPBACK		接口用于环回网络
IFF_OACTIVE	•	正在传输数据
IFF_RUNNING	•	资源已分配给这个接口
IFF_SIMPLEX	•	接口不能接收它自己发送的数据
IFF_LINK0	见正文	由设备驱动程序定义
IFF_LINK1	见正文	由设备驱动程序定义
IFF_LINK2	见正文	由设备驱动程序定义
IFF_ALLMULTI		接口正接收所有多播分组
IFF_DEBUG		这个接口允许调试
IFF_NOARP		在这个接口上不使用ARP协议
IFF_NOTRAILERS		避免使用尾部封装
IFF_PROMISC		接口接收所有网络分组
IFF_UP		接口正在工作

图3-7 if_flags 值

IFF_BROADCAST和IFF_POINTOPOINT标志是互斥的。

宏IFF_CANTCHANGE是对所有在“内核专用”列中作了记号的标志进行按位“或”操作。

设备专用标志(IFF_LINK*x*)对于一个依赖这个设备的进程可能是可修改的，也可能是不可修改的。例如，图3-29显示了这些标志是如何被SLIP驱动程序定义的。

2. 接口时钟

87 if_timer以秒为单位记录时间，直到内核为此接口调用函数if_watchdog为止。这个函数用于设备驱动程序定时收集接口统计，或用于复位运行不正确的硬件。

3. BSD分组过滤器

88-89 下面两个成员，if_pcount和if_bpf，支持BSD分组过滤器(BPF)。通过BPF，一个进程能接收由此接口传输或接收的分组的备份。当我们讨论设备驱动程序时，还要讨论分组是如何通过BPF的。BPF在第31章讨论。

ifnet结构的下一个部分显示在图3-8中，它用来描述接口的硬件特性。

if.h

```
90      struct if_data {
91  /* generic interface information */
92          u_char  ifi_type;       /* Figure 3.9 */
93          u_char  ifi_addrlen;    /* media address length */
94          u_char  ifi_hdrlen;     /* media header length */
95          u_long  ifi_mtu;        /* maximum transmission unit */
96          u_long  ifi_metric;     /* routing metric (external only) */
97          u_long  ifi_baudrate;   /* linespeed */
                /* other ifnet members */
138 #define if_mtu      if_data.ifi_mtu
139 #define if_type     if_data.ifi_type
140 #define if_addrlen  if_data.ifi_addrlen
141 #define if_hdrlen   if_data.ifi_hdrlen
142 #define if_metric   if_data.ifi_metric
143 #define if_baudrate if_data.ifi_baudrate
```

if.h

图3-8 ifnet 结构：接口特性

Net/3和本书使用第138行~143行的#define语句定义的短语来表示ifnet的成员。

4. 接口特性

90-92 if_type指明接口支持的硬件地址类型。图3-9列出了net/if_types.h中几个公共的if_type值。

if_type	说 明
IFT_OTHER	未指明
IFT_ETHER	以太网
IFT_ISO88023	IEEE 802.3以太网(CSMA/CD)
IFT_ISO88025	IEEE 802.5令牌环
IFT_FDDI	光纤分布式数据接口
IFT_LOOP	环回接口
IFT_SLIP	串行线IP

图3-9 if_type ：数据链路类型

93-94 if_addrlen是数据链路地址的长度，而if_hdrlen是由硬件附加给任何分组的首部的长度。例如，以太网有一个长度为6字节的地址和一个长度为14字节的首部(图4-8)。

95 if_mtu是接口传输单元的最大值：接口在一次输出操作中能传输的最大数据单元的字节数。这是控制网络和传输协议创建分组大小的重要参数。对于以太网来说，这个值是 1500。

96-97 if_metric通常是0；其他更大的值不利于路由通过此接口。 if_baudrate指定接口的传输速率，只有SLIP接口才设置它。

接口统计由图3-10中显示的下一组ifnet接口成员来收集。

5. 接口统计

98-111 这些统计大多数是不言自明的。当分组传输被共享媒体上其他传输中断时，if_collisions加1。if_noproto统计由于协议不被系统或接口支持而不能处理的分组数

(例如：仅支持IP的系统接收到一个OSI分组)。如果一个非IP分组到达一个SLIP接口的输出队列时，if_noproto加1。

if.h

```
 98 /* volatile statistics */
 99         u_long  ifi_ipackets;   /* #packets received on interface */
100         u_long  ifi_ierrors;    /* #input errors on interface */
101         u_long  ifi_opackets;   /* #packets sent on interface */
102         u_long  ifi_oerrors;    /* #output errors on interface */
103         u_long  ifi_collisions; /* #collisions on csma interfaces */
104         u_long  ifi_ibytes;     /* #bytes received */
105         u_long  ifi_obytes;     /* #bytes sent */
106         u_long  ifi_imcasts;    /* #packets received via multicast */
107         u_long  ifi_omcasts;    /* #packets sent via multicast */
108         u_long  ifi_iqdrops;    /* #packets dropped on input, for this
109                                    interface */
110         u_long  ifi_noproto;    /* #packets destined for unsupported
111                                    protocol */
112         struct timeval ifi_lastchange;  /* last updated */
113     } if_data;

                /* other ifnet members */

144 #define if_ipackets if_data.ifi_ipackets
145 #define if_ierrors  if_data.ifi_ierrors
146 #define if_opackets if_data.ifi_opackets
147 #define if_oerrors  if_data.ifi_oerrors
148 #define if_collisions   if_data.ifi_collisions
149 #define if_ibytes   if_data.ifi_ibytes
150 #define if_obytes   if_data.ifi_obytes
151 #define if_imcasts  if_data.ifi_imcasts
152 #define if_omcasts  if_data.ifi_omcasts
153 #define if_iqdrops  if_data.ifi_iqdrops
154 #define if_noproto  if_data.ifi_noproto
155 #define if_lastchange   if_data.ifi_lastchange
```

if.h

图3-10 结构ifnet ：接口统计

这些统计在Net/1中不是ifnet结构的一部分。它们被加入来支持接口的标准SNMP MIB-II变量。

if_iqdrops仅被SLIP设备驱动程序访问。当IF_DROP被调用时，SLIP和其他网络驱动程序把if_snd.ifq_drops (图3-13)加1。在SNMP统计加入前，ifq_drops就已经存在于BSD软件中了。ISODE SNMP代理忽略if_iqdrops而使用if_snd.ifq_drops。

6. 改变时间戳

112-113 if_lastchange记录任何统计改变的最近时间。

Net/3和本书又一次用从144行到155行的#define语句定义的短名来指明ifnet的成员。

结构ifnet的下一个部分，显示在图3-11中，它包含指向标准接口层函数的指针，它们把设备专用的细节从网络层分离出来。每个网络接口实现这些适用于特定设备的函数。

if.h

```
114 /* procedure handles */
115     int     (*if_init)          /* init routine */
116             (int);
117     int     (*if_output)        /* output routine (enqueue) */
118             (struct ifnet *, struct mbuf *, struct sockaddr *,
119              struct rtentry *);
120     int     (*if_start)         /* initiate output routine */
121             (struct ifnet *);
122     int     (*if_done)          /* output complete routine */
123             (struct ifnet *);   /* (XXX not used; fake prototype) */
124     int     (*if_ioctl)         /* ioctl routine */
125             (struct ifnet *, int, caddr_t);
126     int     (*if_reset)
127             (int);              /* new autoconfig will permit removal */
128     int     (*if_watchdog)      /* timer routine */
129             (int);
```

if.h

图3-11 结构ifnet ：接口过程

7. 接口函数

114-129 在系统初始化时，每个设备驱动程序初始化它自己的 ifnet结构，包括7个函数指针。图3-12说明了这些通用函数。

我们在Net/3中常会看到注释/* XXX */它提醒读者这段代码是易混淆的，包括不明确的副作用，或是一个更难问题的快速解决方案。在这里，它指示 if_done 不在Net/3中使用。

函 数	说 明
if_init	初始化接口
if_output	对要传输的输出分组进行排队
if_start	启动分组的传输
if_done	传输完成后的清除(未用)
if_ioctl	处理I/O控制命令
if_reset	复位接口设备
if_watchdog	周期性接口例程

图3-12 结构ifnet ：函数指针

在第4章我们要查看以太网、 SLIP和环回接口的设备专用函数，内核通过 ifnet结构中的这些指针直接调用它们。例如，如果 ifp指向一个ifnet结构，

```
(*ifp->if_start)(ifp)
```

调用这个接口的设备驱动程序的if_start函数。

结构ifnet中剩下的最后一个成员是接口的输出队列，如图 3-13所示。

130-137 if_snd是接口输出分组队列，每个接口有它自己的 ifnet结构，即它自己的输出队列。ifq_head指向队列的第一个分组(下一个要输出的分组)，ifq_tail指向队列最后一个分组，if_len是当前队列中分组的数目，而 ifq_maxlen是队列中允许的缓存最大个数。除非驱动程序修改它，这个最大值被设置为 50(来源于全局整数ifqmaxlen，它在编译期间根据IFQ_MAXLEN初始化而来)。队列作为一个mbuf链的链表来实现。ifq_drops统计

因为队列满而丢弃的分组数。图3-14列出了那些访问队列的宏和函数。

if.h

```
130     struct ifqueue {
131         struct mbuf *ifq_head;
132         struct mbuf *ifq_tail;
133         int     ifq_len;        /* current length of queue */
134         int     ifq_maxlen;     /* maximum length of queue */
135         int     ifq_drops;      /* packets dropped because of full queue */
136     } if_snd;                   /* output queue */
137 };
```

if.h

图3-13 结构ifnet ：输出队列

函 数	说 明
IF_QFULL	*ifq*是否满 int **IF_QFULL**(struct ifqueue *ifq);
IF_DROP	IF_DROP仅将与*ifq*关联的ifq_drops加1。这个名字会引起误导：调用者丢弃这个分组 void **IF_DROP**(struct ifqueue *ifq);
IF_ENQUEUE	把分组*m*追加到*ifq*队列的后面。分组通过mbuf首部中的m_nextpkt链接在一起 void **IF_ENQUEUE**(struct ifqueue *ifq, struct mbuf *m);
IF_PREPEND	把分组*m*插入到*ifq*队列的前面 void **IF_PREPEND**(struct ifqueue *ifq, struct mbuf *m);
IF_DEQUEUE	从*ifq*队列中取走第一个分组。*m*指向取走的分组，若队列为空，则*m*为空值 void **IF_DEQUEUE**(struct ifqueue *ifq, struct mbuf *m);
if_qflush	丢弃队列*ifq*中的所有分组，例如，当一个接口被关闭了 void **if_qflush**(struct ifqueue *ifq);

图3-14 fiqueue 例程

前5个例程是定义在net/if.h中的宏，最后一个例程if_qflush是定义在net/if.c中的一个函数。这些宏经常出现在下面这样的程序语句中：

```
s = splimp();
if (IF_QFULL(inq)) {
    IF_DROP(inq);           /* queue is full, drop new packet */
    m_freem(m);
} else
    IF_ENQUEUE(inq, m);     /* there is room, add to end of queue */
splx(s);
```

这段代码试图把一个分组加到队列中。如果队列满， IF_DROP把ifq_drops加1，并且分组被丢弃。可靠协议如TCP会重传丢弃的分组。使用不可靠协议(如UDP)的应用程序必须自己检测和处理重传。

访问队列的语句被splimp和splx括起来，阻止网络中断，并且防止在不确定状态时网络中断服务例程访问此队列。

在splx之前调用m_freem，是因为这段mbuf代码有一个临界区运行在splimp级别上。若在m_freem前调用splx，在m_freem中进入另一个临界区(2.5节)是浪费效率的。

3.4 ifaddr结构

我们要看的下一个结构是接口地址结构，ifaddr，它显示在图3-15中。每个接口维护一个ifaddr结构的链表，因为一些数据链路，如以太网，支持多于一个的协议。一个单独的ifaddr结构描述每个分配给接口的地址，通常每个协议一个地址。支持多地址的另一个原因是很多协议，包括TCP/IP，支持为单个物理接口指派多个地址。虽然Net/3支持这个特性，但很多TCP/IP实现并不支持。

if.h

```
217 struct ifaddr {
218         struct  ifaddr *ifa_next;       /* next address for interface */
219         struct  ifnet *ifa_ifp;         /* back-pointer to interface */
220         struct  sockaddr *ifa_addr;     /* address of interface */
221         struct  sockaddr *ifa_dstaddr;  /* other end of p-to-p link */
222 #define ifa_broadaddr   ifa_dstaddr     /* broadcast address interface */
223         struct  sockaddr *ifa_netmask;  /* used to determine subnet */
224         void    (*ifa_rtrequest)();     /* check or clean routes */
225         u_short ifa_flags;              /* mostly rt_flags for cloning */
226         short   ifa_refcnt;             /* references to this structure */
227         int     ifa_metric;             /* cost for this interface */
228 };
```

if.h

图3-15 结构ifaddr

217-219 结构ifaddr通过ifa_next把分配给一个接口的所有地址链接起来，它还包括一个指回接口的ifnet结构的指针ifa_ifp。图3-16显示了结构ifnet与ifaddr之间的关系。

220 ifa_addr指向接口的一个协议地址，而ifa_netmask指向一个位掩码，它用于选择ifa_addr中的网络部分。地址中表示网络部分的比特在掩码中被设置为1，地址中表示主机的部分被设置为0。两个地址都存放在sockaddr结构中(3.5节)。图3-38显示了一个地址及其掩码结构。对于IP地址，掩码选择IP地址中的网络和子网部分。

221-223 ifa_dstaddr(或它的别名ifa_broadaddr)指向一个点对点链路上的另一端的接口协议地址或指向一个广播网中分配给接口的广播地址(如以太网)。接口的ifnet结构中互斥的两个标志IFF_BROADCAST和IFF_POINTOPOINT(图3-7)指示接口的类型。

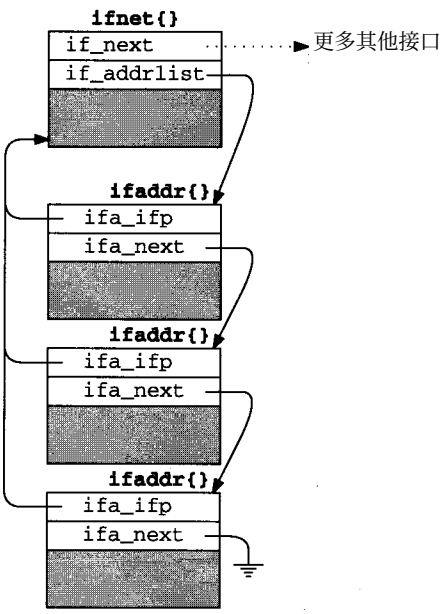

图3-16 结构ifnet 和ifaddr

224-228 ifa_rtrequest、ifa_flags和ifa_metric支持接口的路由查找。

ifa_refcnt统计对结构ifaddr的引用。宏IFAFREE仅在引用计数降到0时才释放这个结构，例如，当地址被命令SIOCDIFADDRioctl删除时。结构ifaddr使用引用计数是因为接口和路由数据结构共享这些结构。

如果有其他对ifaddr的引用，IFAFREE将计数器加1并返回。这是一个通用的方法，除了最后一个引用外，它避免了每次都调用一个函数的开销。如果是最后一个引用，IFAFREE调用函数ifafree，来释放这个结构。

3.5 sockaddr结构

一个接口的编址信息不仅仅只包括一个主机地址。 Net/3在通用的sockaddr结构中维护主机地址、广播地址和网络掩码。通过使用一个通用的结构，将硬件与协议专用的地址细节相对于接口层隐藏起来。

图3-17显示的是这个结构的当前定义及早期 BSD版的定义 —— 结构osockaddr。图3-18说明了这些结构的组织。

socket.h

```
120 struct sockaddr {
121     u_char  sa_len;             /* total length */
122     u_char  sa_family;          /* address family (Figure 3.19) */
123     char    sa_data[14];        /* actually longer; address value */
124 };

271 struct osockaddr {
272     u_short sa_family;          /* address family (Figure 3.19) */
273     char    sa_data[14];        /* up to 14 bytes of direct address */
274 };
```

socket.h

图3-17 结构sockaddr 和osockaddr

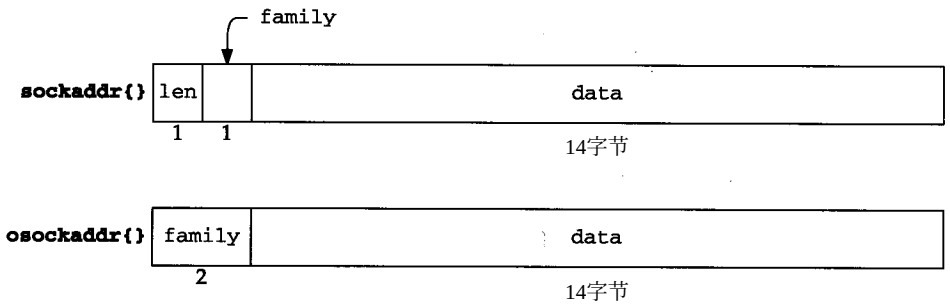

图3-18 结构sockaddr 和osockaddr(省略了前缀sa_)

在很多图中，我们省略了成员名中的公共前缀。在这里，我们省略了 sa_前缀。

1. Sockaddr结构

120-124 每个协议有它自己的地址格式。 Net/3在一个sockaddr结构中处理通用的地址。sa_len指示地址的长度(OSI和Unix域协议有不同长度的地址), sa_family指示地址的类型。图3-19列出了地址族(address family)常量，其中包括我们遇到的。

当指明为AF_UNSPEC时，一个sockaddr的内容要根据情况而定。大多数情况下，它包含一个以太网硬件地址。

成员sa_len和sa_family允许协议无关代码操作来自多个协议的变长的 sockaddr结构。剩下的成员sa_data，包含一个协议相关格式的地址。 sa_data定义为一个14字节的数组，但当sockaddr结构覆盖更大的内存空间时， sa_data可能会扩展到253字节。 sa_len

仅有一个字节，因此整个地址，包括 sa_len和sa_family必须不超过256字节。

这是C语言的一种通用技术，它允许程序员把一个结构中的最后一个成员看成是可变长的。

sa_family	协 议
AF_INET	Internet
AF_ISO,AF_OSI	OSI
AF_UNIX	Unix
AF_ROUTE	路由表
AF_LINK	数据链路
AF_UNSPEC	(见正文)

图3-19 sa_family 常量

每个协议定义一个专用的sockaddr结构，该结构复制成员sa_len和sa_family，但按那个协议的要求来定义成员sa_data。存储在sa_data中的地址是一个传输地址；它包含足够的信息来标识同一主机上的多个通信端点。在第 6章我们要查看Internet地址结构sockaddr_in，它包含了一个IP地址和一个端口号。

2. Osockaddr结构

271-274 结构osockaddr是4.3BSD Reno版本以前的sockaddr定义。因为在这个定义中一个地址的长度不是显式地可用，所以它不能用来写处理可变长地址的协议无关代码。 OSI协议使用可变长地址，为了包括 OSI协议，使得在Net/3的sockaddr定义中有了我们所见的改变。结构osockaddr是为了支持对以前编译的程序的二进制兼容。

在本书中我们省略了二进制兼容代码。

3.6 ifnet与ifaddr的专用化

结构ifnet和ifaddr包含适用于所有网络接口和协议地址的通用信息。为了容纳其他设备和协议专用信息，每个设备定义了并且每个协议分配了一个专用化版本的 ifnet和ifaddr结构。这些专用化的结构总是包含一个 ifnet或ifaddr结构作为它们的第一个成员，这样无须考虑其他专用信息就能访问这些公共信息。

多数设备驱动程序通过分配一个专用化的 ifnet结构的数组来处理同一类型的多个接口，但其他设备(例如环回设备)仅处理一个接口。图3-20所示的是我们的例子接口的专用化 ifnet结构的组织。

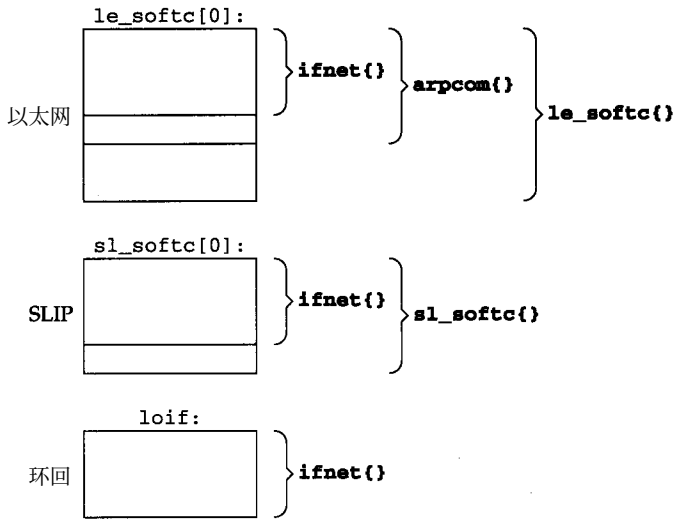

图3-20 设备相关的结构中的 ifnet 结构的组织

注意，每个设备的结构以一个ifnet开始，接下来全是设备相关的数据。环回接口只声名了一个ifnet结构，因为它不要求任何设备相关的数据。在图3-20中，我们显示的以太网和SLIP驱动程序的结构softc带有数组下标0，因为两个设备都支持多个接口。任何给定类型的接口的最大个数由内核建立时的配置参数来限制。

结构arpcom(图3-26)对于所有以太网设备是通用的，并且包含地址解析协议(ARP)和以太网多播信息。结构le_softc(图3-25)包含专用于LANCE以太网设备驱动器的其他信息。

每个协议把每个接口的地址信息存储在一个专用化的ifaddr结构的列表中。以太网协议使用一个in_ifaddr结构(6.5节)，而OSI协议使用一个iso_ifaddr结构。另外，当接口被初始化时，内核为每个接口分配了一个链路层地址，它在内核中标识这个接口。

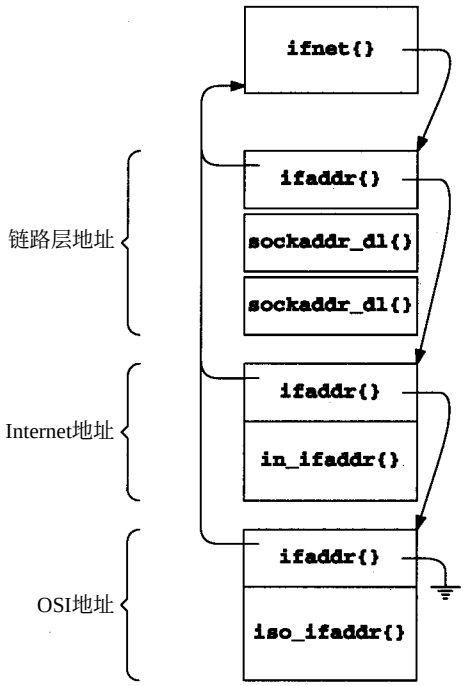

图3-21 一个包含链路层地址、Internet地址和OSI地址的接口地址列表

内核通过分配一个ifaddr结构和两个sockaddr_dl结构(一个是链路层地址本身，一个是地址掩码)来构造一个链路层地址。结构sockaddr_dl可被OSI、ARP和路由算法访问。图3-21显示的是一个带有一个链路层地址、一个Internet地址和一个OSI地址的以太网接口。3.11节说明了链路层地址(ifaddr和两个sockaddr_ld结构)的构造和初始化。

3.7 网络初始化概述

所有我们说明的结构是在内核初始化时分配和互相链接起来的。在本节我们大致概述一下初始化的步骤。在后面的章节，我们说明特定设备的初始化步骤和特定协议的初始化步骤。

有些设备，例如SLIP和环回接口，完全用软件来实现。这些伪设备用存储在全局pdevinit数组中的一个pdevinit结构来表示(图3-22)。在内核配置期间构造这个数组。例如：

```
struct pdevinit pdevinit[] = {
    { slattach, 1 },
    { loopattach, 1 },
    { 0, 0 }
};
```

device.h

```
120 struct pdevinit {
121     void    (*pdev_attach) (int);   /* attach function */
122     int     pdev_count;             /* number of devices */
123 };
```

device.h

图3-22 结构pdevinit

120-123 对于SLIP和环回接口，在结构pdevinit中，pdev_attach分别被设置为

slattach和loopattach。当调用这个attach函数时，pdev_count作为传递的唯一参数，它指定创建的设备个数。只有一个环回设备被创建，但如果管理员适当配置 SLIP项可能有多个SLIP设备被创建。

网络初始化函数从main开始显示在图3-23中。

init_main.c

```
70 main(framep)
71 void   *framep;
72 {
                        /* nonnetwork code */
96      cpu_startup();              /* locate and initialize devices */
                        /* nonnetwork code */
172     /* Attach pseudo-devices. (e.g., SLIP and loopback interfaces) */
173     for (pdev = pdevinit; pdev->pdev_attach != NULL; pdev++)
174         (*pdev->pdev_attach) (pdev->pdev_count);
175     /*
176      * Initialize protocols.  Block reception of incoming packets
177      * until everything is ready.
178      */
179     s = splimp();
180     ifinit();                   /* initialize network interfaces */
181     domaininit();               /* initialize protocol domains */
182     splx(s);
                        /* nonnetwork code */
231     /* The scheduler is an infinite loop. */
232     scheduler();
233     /* NOTREACHED */
234 }
```

init_main.c

图3-23 main 函数：网络初始化

70-96 cpu_startup查找并初始化所有连接到系统的硬件设备，包括任何网络接口。

97-174 在内核初始化硬件设备后，它调用包含在 pdevinit数组中的每个pdev_attach函数。

175-234 ifinit和domaininit完成网络接口和协议的初始化，并且 scheduler开始内核进程调度。ifinit和domaininit在第7章讨论。

在下面几节中，我们说明以太网、 SLIP和环回接口的初始化。

3.8 以太网初始化

作为cpu_startup的一部分，内核查找任何连接的网络设备。这个进程的细节超出了本书的范围。一旦一个设备被识别，一个设备专用的初始化函数就被调用。图 3-24显示的是我们的3个例子接口的初始化函数。

每个设备驱动程序为一个网络接口初始化一个专用化的ifnet结构，并调用if_attach把这个结构插入到接口链表中。显示在图 3-25中的结构le_softc是我们的例子以太网驱动程序的专用化 ifnet结构(图3-20)。

设　备	初始化函数
LANCE以太网	leattach
SLIP	slattach
环回	loopattach

图3-24　网络接口初始化函数

1. le_softc结构

69-95　在if_le.c中声明了一个le_softc结构(有NLE成员)的数组。每个结构的第一个成员是 sc_ac，一个arpcom结构，它对于所有以太网接口都是通用的，接下来是设备专用成员。宏 sc_if和sc_addr简化了对结构ifnet及存储在结构arpcom(sc_ac)中的以太网地址的访问，如图3-26所示。

if_le.c

```
69 struct le_softc {
70     struct arpcom sc_ac;          /* common Ethernet structures */
71 #define sc_if   sc_ac.ac_if       /* network-visible interface */
72 #define sc_addr sc_ac.ac_enaddr /* hardware Ethernet address */

                    /* device-specific members */

95 } le_softc[NLE];
```

if_le.c

图3-25　结构le_softc

if_ether.h

```
95 struct arpcom {
96     struct ifnet ac_if;           /* network-visible interface */
97     u_char  ac_enaddr[6];         /* ethernet hardware address */
98     struct in_addr ac_ipaddr;     /* copy of ip address - XXX  */
99     struct ether_multi *ac_multiaddrs;  /* list of ether multicast addrs */
100    int     ac_multicnt;          /* length of ac_multiaddrs list */
101 };
```

if_ether.h

图3-26　结构arpcom

2. arpcom结构

95-101　结构arpcom的第一个成员ac_if是一个ifnet结构，如图3-20所示。ac_enaddr是以太网硬件地址，它是在 cpu_startup期间内核检测设备时由 LANCE设备驱动程序从硬件上复制的。对于我们的例子驱动程序，这发生在函数 leattach中(图3-27)。ac_ipaddr是上一个分配给此设备的 IP地址。我们在6.6节讨论地址的分配，可以看到一个接口可以有多个IP地址。见习题6.3。ac_multiaddrs是一个用结构ether_multi表示的以太网多播地址的列表。ac_multicnt统计这个列表的项数。多播列表在第 12章讨论。

图3-27所示的是LANCE以太网驱动程序的初始化代码。

106-115　内核在系统中每发现一个LANCE卡都调用一次leattach。

只有一个指向一个hp_device结构的参数，它包含了 HP专用信息，因为它是专为HP工作站编写的驱动程序。

le指向此卡的专用化ifnet结构(图3-20)，ifp指向这个结构的第一个成员 sc_if，一个通用的ifnet结构。图3-27并不包括设备专用初始化代码，它在本书中不予讨论。

if_le.c

```
106 leattach(hd)
107 struct hp_device *hd;
108 {
109     struct lereg0 *ler0;
110     struct lereg2 *ler2;
111     struct lereg2 *lemem = 0;
112     struct le_softc *le = &le_softc[hd->hp_unit];
113     struct ifnet *ifp = &le->sc_if;
114     char   *cp;
115     int     i;

                    /* device-specific code */

126     /*
127      * Read the ethernet address off the board, one nibble at a time.
128      */
129     cp = (char *) (lestd[3] + (int) hd->hp_addr);
130     for (i = 0; i < sizeof(le->sc_addr); i++) {
131         le->sc_addr[i] = (*++cp & 0xF) << 4;
132         cp++;
133         le->sc_addr[i] |= *++cp & 0xF;
134         cp++;
135     }
136     printf("le%d: hardware address %s\n", hd->hp_unit,
137            ether_sprintf(le->sc_addr));

                    /* device-specific code */

150     ifp->if_unit = hd->hp_unit;
151     ifp->if_name = "le";
152     ifp->if_mtu = ETHERMTU;
153     ifp->if_init = leinit;
154     ifp->if_reset = lereset;
155     ifp->if_ioctl = leioctl;
156     ifp->if_output = ether_output;
157     ifp->if_start = lestart;
158     ifp->if_flags = IFF_BROADCAST | IFF_SIMPLEX | IFF_MULTICAST;
159     bpfattach(&ifp->if_bpf, ifp, DLT_EN10MB, sizeof(struct ether_header));
160     if_attach(ifp);
161     return (1);
162 }
```

if_le.c

图3-27 函数leattach

3. 从设备复制硬件地址

126-137 对于LANCE设备，由厂商指派的以太网地址在这个循环中以每次半个字节 (4 bit) 从设备复制到sc_addr(即sc_ac.ac_enaddr —— 见图3-26)。

lestd是一个设备专用的位移表，用于定位hp_addr的相关信息，hp_addr指向LANCE专用信息。

通过printf语句将完整的地址输出到控制台，来指示此设备存在并且可操作。

4. 初始化ifnet结构

150-157 leattach从hp_device结构把设备单元号复制到if_unit来标识同类型的多

个接口。这个设备的if_name是“ le ”；if_mtu为1500字节(ETHERMTU)，以太网的最大传输单元；if_init、if_reset、if_ioctl、if_output和it_start都指向控制网络接口的通用函数的设备专用实现。4.1节说明这些函数。

158 所有的以太网设备都支持IFF_BROADCAST。LANCE设备不接收它自己发送的数据，因此被设置为IFF_SIMPLEX。支持多播的设备和硬件还要设置IFF_MULTICAST。

159-162 bpfattach登记有BPF的接口，在图31-8中说明。函数if_attach把初始化了的ifnet结构插入到接口的链表中(3.11节)。

3.9 SLIP初始化

依赖标准异步串行设备的SLIP接口在调用cpu_startup时初始化。当main直接通过SLIP的pdevinit结构中的指针pdev_attach调用slattach时，SLIP伪设备被初始化。

每个SLIP接口由图3-28中的一个sl_softc结构来描述。

if_slvar.h

```
43 struct sl_softc {
44     struct ifnet sc_if;          /* network-visible interface */
45     struct ifqueue sc_fastq;     /* interactive output queue */
46     struct tty *sc_ttyp;         /* pointer to tty structure */
47     u_char *sc_mp;               /* pointer to next available buf char */
48     u_char *sc_ep;               /* pointer to last available buf char */
49     u_char *sc_buf;              /* input buffer */
50     u_int   sc_flags;            /* Figure 3.29 */
51     u_int   sc_escape;           /* =1 if last char input was FRAME_ESCAPE */
52     struct slcompress sc_comp;   /* tcp compression data */
53     caddr_t sc_bpf;              /* BPF data */
54 };
```

if_slvar.h

图3-28 结构sl_softc

43-54 与所有接口结构一样，sl_softc有一个ifnet结构并且后面跟着设备专用信息。

除了在ifnet结构中的输出队列外，一个SLIP设备还维护另一个队列sc_fastq，它用于要求低时延服务的分组——典型地由交互应用产生。

sc_ttyp指向关联的终端设备。指针sc_buf和sc_ep分别指向一个接收SLIP分组的缓存的第一个字节和最后一个字节。sc_mp指向下一个接收字节的地址，并在另一个字节到达时向前移动。

SLIP定义的4个标志显示在图3-29中。

常 量	sc_softc成员	说 明
SC_COMPRESS	sc_if.if_flags	IFF_LINK0；压缩TCP通信
SC_NOICMP	sc_if.if_flags	IFF_LINK1；禁止ICMP通信
SC_AUTOCOMP	sc_if.if_flags	IFF_LINK2；允许TCP自动压缩
SC_ERROR	sc_flags	检测到错误；丢弃接收帧

图3-29 SLIP的if_flags 和sc_flags 值

SLIP在ifnet结构中定义了3个接口标志预留给设备驱动程序，另一个标志定义在结构sl_softc中。

sc_escape用于串行线的IP封装机制(5.3节)，而TCP首部压缩信息(29.13节)保留在

sc_comp中。

指针sc_bpf指向SLIP设备的BPF信息。

结构sl_softc由slattach初始化，如图3-30所示。

135-152 不像leattach一次仅初始化一个接口，内核只调用一次 slattach，并且slattach初始化所有的SLIP接口。硬件设备在内核执行 cpu_startup被发现时初始化，而伪设备都是在 main为这个设备调用 pdev_attach函数时被初始化的。一个 SLIP设备的if_mtu为296字节(SLMTU)。这包括标准的20字节IP首部、标准的20字节TCP首部和256字节的用户数据(5.3节)。

if_sl.c

```
135 void
136 slattach()
137 {
138     struct sl_softc *sc;
139     int      i = 0;

140     for (sc = sl_softc; i < NSL; sc++) {
141         sc->sc_if.if_name = "sl";
142         sc->sc_if.if_next = NULL;
143         sc->sc_if.if_unit = i++;
144         sc->sc_if.if_mtu = SLMTU;
145         sc->sc_if.if_flags =
146             IFF_POINTOPOINT | SC_AUTOCOMP | IFF_MULTICAST;
147         sc->sc_if.if_type = IFT_SLIP;
148         sc->sc_if.if_ioctl = slioctl;
149         sc->sc_if.if_output = sloutput;
150         sc->sc_if.if_snd.ifq_maxlen = 50;
151         sc->sc_fastq.ifq_maxlen = 32;
152         if_attach(&sc->sc_if);
153         bpfattach(&sc->sc_bpf, &sc->sc_if, DLT_SLIP, SLIP_HDRLEN);
154     }
155 }
```

if_sl.c

图3-30 函数slattach

一个SLIP网络由位于一个串行通信线两端的两个接口组成。 slattach在if_flags中设置IFF_POINTOPOINT、SC_AUTOCOMP和IFF_MULTICAST。

SLIP接口限制它的输出分组队列if_snd的长度为50，并且它自己的接口队列sc_fastq的长度为32。图3-42显示if_snd队列的长度默认为50(ifqmaxlen)，因此，如果设备驱动程序选择一个长度，这里的初始化是多余的。

以太网设备驱动程序不显式地设置它的输出队列的长度，它依赖于 ifinit (图3-42)把它设置为系统的默认值。

if_attach需要一个指向一个ifnet结构的指针，因此slattach将sc_if的地址传递给if_attach，sc_if是一个第一个成员为结构sl_softc的ifnet结构。

专用程序slattach在内核初始化后运行(从初始化文件/etc/netstart)，并通过打开串行设备和执行ioctl命令(5.3节)添加SLIP接口和一个异步串行设备。

153-155 对于每个SLIP设备，slattach调用bpfattach来登记有BPF的接口。

3.10 环回初始化

最后显示环回接口的初始化。环回接口把输出分组放回到相应的输入队列中。接口没有

相关联的硬件设备。环回伪设备在main通过环回接口的pdevinit结构中的pdev_attach指针直接调用loopattach时初始化。图3-31所示的是函数loopattach。

if_loop.c

```
41 void
42 loopattach(n)
43 int     n;
44 {
45     struct ifnet *ifp = &loif;

46     ifp->if_name = "lo";
47     ifp->if_mtu = LOMTU;
48     ifp->if_flags = IFF_LOOPBACK | IFF_MULTICAST;
49     ifp->if_ioctl = loioctl;
50     ifp->if_output = looutput;
51     ifp->if_type = IFT_LOOP;
52     ifp->if_hdrlen = 0;
53     ifp->if_addrlen = 0;
54     if_attach(ifp);
55     bpfattach(&ifp->if_bpf, ifp, DLT_NULL, sizeof(u_int));
56 }
```

if_loop.c

图3-31 环回接口初始化

41-56 环回if_mtu被设置为1536字节(LOMTU)。在if_flags中设置IFF_LOOPBACK和IFF_MULTICAST。一个环回接口没有链路首部和硬件地址，因此if_hdrlen和if_addrlen被设置为0。if_attach完成ifnet结构的初始化并且bpfattach登记有BPF的环回接口。

环回MTU至少有1576(40 + 3×512)留给一个标准的TCP/IP首部。例如Solaris 2.3环回MTU设置为8232(40 + 8×1024)。这些计算基于Internet协议；而其他协议可能有大于40字节的默认首部。

3.11 if_attach函数

前面显示的三个接口初始化函数都调用if_attach来完成接口的ifnet结构的初始化，并把这个结构插入到先前配置的接口的列表中。在if_attach中，内核也为每个接口初始化并分配一个链路层地址。图3-32说明了由if_attach构造的数据结构。

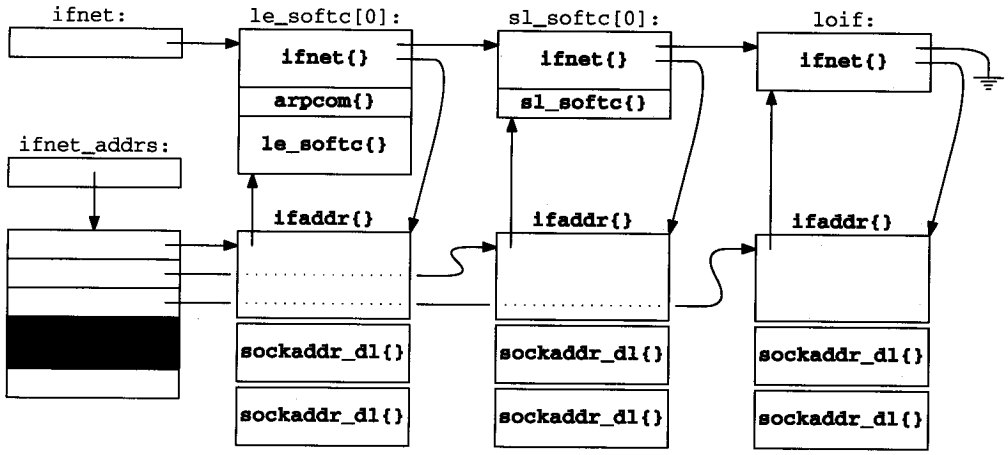

图3-32 ifnet 列表

在图3-32中，if_attach被调用了三次：以一个le_softc结构为参数从leattach调用，以一个sl_softc结构为参数从slattach调用，以一个通用ifnet结构为参数从loopattach调用。每次调用时，它向ifnet列表中添加一个新的ifnet结构，为这个接口创建一个链路层ifaddr结构(包含两个sockaddr_dl结构，图3-33)，并且初始化ifnet_addrs数组中的一项。

if_dl.h

```
55 struct sockaddr_dl {
56     u_char  sdl_len;            /* Total length of sockaddr */
57     u_char  sdl_family;         /* AF_LINK */
58     u_short sdl_index;          /* if != 0, system given index for
59                                    interface */
60     u_char  sdl_type;           /* interface type (Figure 3.9) */
61     u_char  sdl_nlen;           /* interface name length, no trailing 0
62                                    reqd. */
63     u_char  sdl_alen;           /* link level address length */
64     u_char  sdl_slen;           /* link layer selector length */
65     char    sdl_data[12];       /* minimum work area, can be larger;
66                                    contains both if name and ll address */
67 };

68 #define LLADDR(s) ((caddr_t)((s)->sdl_data + (s)->sdl_nlen))
```

if_dl.h

图3-33 结构sockaddr_dl

图3-20显示了包含在le_softc[0]和sl_softc[0]中嵌套的结构。

初始化以后，接口仅配置链路层地址。例如， IP地址直到后面讨论的ifconfig程序才配置(6.6节)。

链路层地址包含接口的一个逻辑地址和一个硬件地址 (如果网络支持，例如le0的一个48 bit以太网地址)。在ARP和OSI协议中要用到这个硬件地址，而一个sockaddr_dl中的逻辑地址包含一个名称和这个接口在内核中的索引数值，它支持用于在接口索引和关联ifaddr结构(ifa_ifwithnet，图6-32)间相互转换的表查找。

结构sockaddr_dl显示在图3-33中。

55-57 回忆图3-18，sdl_len指明了整个地址的长度，而sdl_family指明了地址族类，在此例中为AF_LINK。

58 sdl_index在内核中标识接口。图3-32中的以太网接口会有一个为1的索引，SLIP接口的索引为2，而环回接口的索引为3。全局整数变量if_index包含的是内核最近分配的一个索引值。

60 sdl_type根据这个数据链路地址的ifnet结构的成员if_type进行初始化。

61-68 除了一个数字索引，每个接口有一个由结构ifnet的成员if_name和if_unit组成的文本名称。例如，第一个SLIP接口叫“sl0”，而第二个叫“sl1”。文本名称存储在数组sdl_data的前面，并且sdl_nlen为这个名称的字节长度(在我们的SLIP例子中为3)。

数据链路地址也存储在这个结构中。宏LLADDR将一个指向sockaddr_dl结构的指针转换成一个指向这个文本名称的第一个字节的指针。 sdl_alen是硬件地址的长度。对于一个以太网设备，48 bit硬件地址位于结构sockaddr_dl的这个文本名称的前面。图3-38所示的是一个初始化了的sockaddr_dl结构。

Net/3不使用sdl_slen。

if_attach更新两个全局变量。第一个是if_index，它存放系统中的最后一个接口的索引；第二个是ifnet_addrs，它指向一个ifaddr指针的数组。这个数组的每项都指向一个接口的链路层地址。这个数组提供对系统中每个接口的链路层地址的快速访问。

函数if_attach较长，并且有几个奇怪的赋值语句。从图 3-34开始，我们分5个部分讨论这个函数。

59-74 if_attach有一个参数：ifp，这是一个指向ifnet结构的指针，由网络设备驱动程序初始化。Net/3在一个链表中维护所有这些ifnet结构，全局指针ifnet指向这个链表的首部。while循环查找链表的尾部，并将链表尾部的空指针的地址存储到 P中。在循环后，新ifnet结构被接到这个 ifnet链表的尾部， if_index加1，并且将新索引值赋给 ifp ->if_index。

if.c

```
59 void
60 if_attach(ifp)
61 struct ifnet *ifp;
62 {
63     unsigned socksize, ifasize;
64     int     namelen, unitlen, masklen, ether_output();
65     char    workbuf[12], *unitname;
66     struct ifnet **p = &ifnet;  /* head of interface list */
67     struct sockaddr_dl *sdl;
68     struct ifaddr *ifa;
69     static int if_indexlim = 8; /* size of ifnet_addrs array */
70     extern void link_rtrequest();

71     while (*p)                  /* find end of interface list */
72         p = &((*p)->if_next);
73     *p = ifp;
74     ifp->if_index = ++if_index; /* assign next index */

75     /* resize ifnet_addrs array if necessary */
76     if (ifnet_addrs == 0 || if_index >= if_indexlim) {
77         unsigned n = (if_indexlim <<= 1) * sizeof(ifa);
78         struct ifaddr **q = (struct ifaddr **)
79                     malloc(n, M_IFADDR, M_WAITOK);

80         if (ifnet_addrs) {
81             bcopy((caddr_t) ifnet_addrs, (caddr_t) q, n / 2);
82             free((caddr_t) ifnet_addrs, M_IFADDR);
83         }
84         ifnet_addrs = q;
85     }
```

if.c

图3-34 函数if_attach ：分配接口索引

1. 必要时调整ifnet_addrs数组的大小

75-85 第一次调用if_attach时，数组ifnet_addrs不存在，因此要分配16(16=8<<1)项的空间。当数组满时，一个两倍大的新数组被分配，并且老数组中的项被复制到新的数组中。

if_indexlim是if_attach私有的一个静态变量。if_indexlim通过<<=操作符来更新。

图3-34中的函数malloc和free不是同名的标准C库函数。内核版的第二个参数指明一个

类型，内核中可选的诊断代码用它来检测程序错误。如果malloc的第三个参数为M_WAITOK，且函数需要等待释放的可用内存，则阻塞调用进程。如果第三个参数为 M_DONTWAIT，则当内存不可用时，函数不阻塞并返回一个空指针。

函数if_attach的下一部分显示在图3-35中，它为接口准备一个文本名称并计算链路层地址的长度。

if.c

```
86      /* create a Link Level name for this device */
87      unitname = sprint_d((u_int) ifp->if_unit, workbuf, sizeof(workbuf));
88      namelen = strlen(ifp->if_name);
89      unitlen = strlen(unitname);

90      /* compute size of sockaddr_dl structure for this device */
91  #define _offsetof(t, m) ((int)((caddr_t)&((t *)0)->m))
92      masklen = _offsetof(struct sockaddr_dl, sdl_data[0]) +
93                unitlen + namelen;
94      socksize = masklen + ifp->if_addrlen;
95  #define ROUNDUP(a) (1 + (((a) - 1) | (sizeof(long) - 1)))
96      socksize = ROUNDUP(socksize);
97      if (socksize < sizeof(*sdl))
98          socksize = sizeof(*sdl);
99      ifasize = sizeof(*ifa) + 2 * socksize;
```

if.c

图3-35 if_attach 函数：计算链路层地址大小

2. 创建链路层名称并计算链路层地址的长度

86-99 if_attach用if_unit和if_name组装接口的名称。函数sprint_d将if_unit的数值转换成一个串并存储到workbuf中。masklen是sockaddr_dl数组中sdl_data前面的信息所占用的字节数加上这个接口的文本名称的大小(namelen + unitlen)。ROUNDUP函数对socksize进行上舍入，socksize是masklen加上硬件地址长度(if_addrlen)，上舍入为一个长整型(ROUNDUP)。如果它小于一个sockaddr_dl结构的长度，就使用标准的sockaddr_dl结构。ifasize是一个ifaddr结构的大小加上两倍的 socksize，因此，它能容纳结构sockaddr_dl。

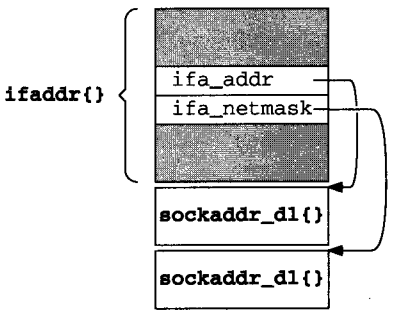

图3-36 在if_attach 中分配的链路层地址和掩码

在函数的下一个部分中，if_attach分配结构并将结构连接起来，如图3-36所示。

图3-36中，在ifaddr结构与两个sockaddr_dl结构间有一个空隙来说明它们分配在一个连续的内存中但没有定义在一个C结构中。

像图3-36所示的组织还出现在结构in_ifaddr中；这个结构的通用ifaddr部分中的指针指向在这个结构的设备专用部分中的专用化sockaddr结构，在本例中是结构sockaddr_dl。图3-37所示的是这些结构的初始化。

3. 地址

100-116 如果有足够的内存可用，bzero把新结构清零，并且sdl指向紧接着ifnet结构的第一个sockaddr_dl。若没有可用内存，代码被忽略。

sdl_len被设置为结构sockaddr_dl的长度，并且sdl_family被设置为AF_LINK。

用if_name和unitname组成的文本名称存放在 sdl_data中，而它的长度存放在sdl_nlen中。接口的索引被复制到sdl_index中，而接口的类型被复制到sdl_type中。分配的结构被插入到数组ifnet_addrs中，并通过ifa_ifp和ifa_addrlist链接到结构ifnet。最后，结构sockaddr_dl用ifa_addr连接到ifnet结构。以太网接口用arp_rtrequest取代默认函数link_rtrequest。环回接口装入函数loop_rtrequest。我们在第19章和第21章讨论ifa_rtrequest和arp_rtrequest。而linkrtrequest和loop_rtrequest留给读者自己去研究。以上完成了第一个sockaddr_dl结构的初始化。

if.c

```
100     if (ifa = (struct ifaddr *) malloc(ifasize, M_IFADDR, M_WAITOK)) {
101         bzero((caddr_t) ifa, ifasize);
102         /* First: initialize the sockaddr_dl address */
103         sdl = (struct sockaddr_dl *) (ifa + 1);
104         sdl->sdl_len = socksize;
105         sdl->sdl_family = AF_LINK;
106         bcopy(ifp->if_name, sdl->sdl_data, namelen);
107         bcopy(unitname, namelen + (caddr_t) sdl->sdl_data, unitlen);
108         sdl->sdl_nlen = (namelen += unitlen);
109         sdl->sdl_index = ifp->if_index;
110         sdl->sdl_type = ifp->if_type;
111         ifnet_addrs[if_index - 1] = ifa;
112         ifa->ifa_ifp = ifp;
113         ifa->ifa_next = ifp->if_addrlist;
114         ifa->ifa_rtrequest = link_rtrequest;
115         ifp->if_addrlist = ifa;
116         ifa->ifa_addr = (struct sockaddr *) sdl;
117         /* Second: initialize the sockaddr_dl mask */
118         sdl = (struct sockaddr_dl *) (socksize + (caddr_t) sdl);
119         ifa->ifa_netmask = (struct sockaddr *) sdl;
120         sdl->sdl_len = masklen;
121         while (namelen != 0)
122             sdl->sdl_data[--namelen] = 0xff;
123     }
```

if.c

图3-37 函数if_attach ：分配并初始化链路层地址

4. 掩码

117-123 第二个sockaddr_dl结构是一个比特掩码，用来选择出现在第一个结构中的文本名称。ifa_netmask从结构ifaddr指向掩码结构(在这里是选择接口文本名称而不是网络掩码)。while循环把与名称对应的那些字节的每个比特都置为1。

图3-38所示的是我们以太网接口例子的两个初始化了的 sockaddr_dl结构。它的if_name为“ le ”，if_unit为0，if_index为1。

图3-38中所示的是ether_ifattach对这个结构初始化后的地址(图3-41)。

图3-39所示的是第一个接口被if_attach连接后的结构。

在if_attach的最后，以太网设备的函数ether_ifattach被调用，如图3-40所示。

124-127 开始不调用ether_ifattach (例如：从leattach)，是因为它要把以太网硬件地址复制到if_attach分配的sockaddr_dl中。

XXX注释表示作者发现在此处插入代码比修改所有的以太网驱动程序要容易。

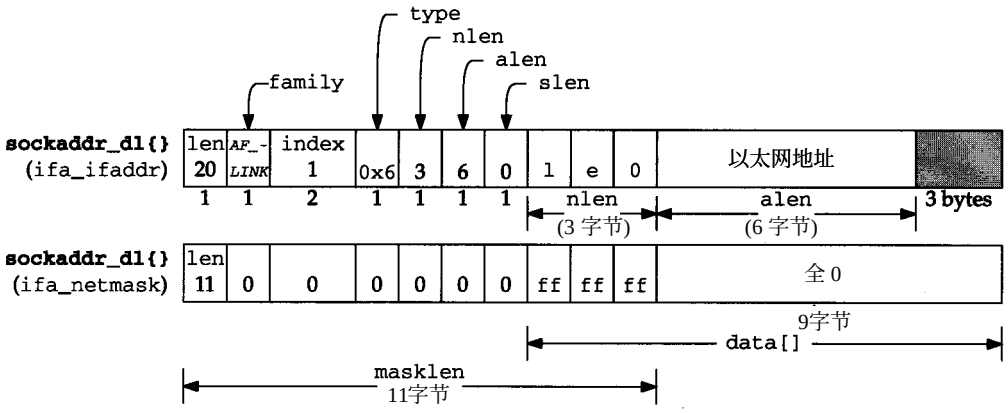

图3-38 初始化了的以太网 sockaddr_dl 结构(省略了前缀 sdl_)

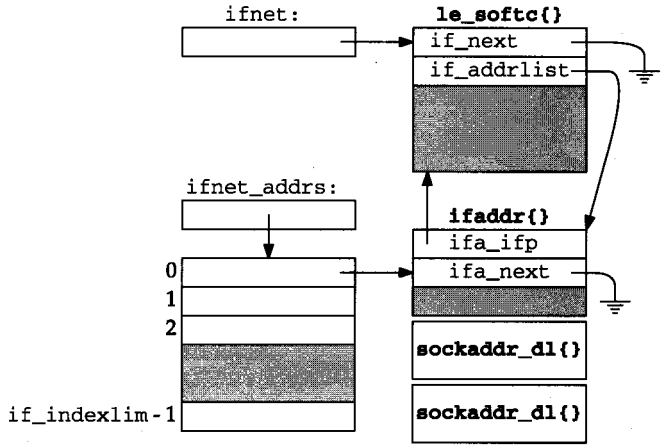

图3-39 第一次调用 if_attach 后的 ifnet 和 sockaddr_dl 结构

if.c

```
124     /* XXX -- Temporary fix before changing 10 ethernet drivers */
125     if (ifp->if_output == ether_output)
126         ether_ifattach(ifp);
127 }
```

if.c

图3-40 函数 if_attach ：以太网初始化

if_ethersubr.c

```
338 void
339 ether_ifattach(ifp)
340 struct ifnet *ifp;
341 {
342     struct ifaddr *ifa;
343     struct sockaddr_dl *sdl;

344     ifp->if_type = IFT_ETHER;
345     ifp->if_addrlen = 6;
346     ifp->if_hdrlen = 14;
347     ifp->if_mtu = ETHERMTU;
348     for (ifa = ifp->if_addrlist; ifa; ifa = ifa->ifa_next)
349         if ((sdl = (struct sockaddr_dl *) ifa->ifa_addr) &&
```

图3-41 函数 ether_ifattach

```
350             sdl->sdl_family == AF_LINK) {
351             sdl->sdl_type = IFT_ETHER;
352             sdl->sdl_alen = ifp->if_addrlen;
353             bcopy((caddr_t) ((struct arpcom *) ifp)->ac_enaddr,
354                   LLADDR(sdl), ifp->if_addrlen);
355             break;
356         }
357 }
```

if_ethersubr.c

图3-41 (续)

5. ether_ifattach函数

函数ether_ifattach执行对所有以太网设备通用的ifnet结构的初始化。

338-357 对于一个以太网设备，if_type为IFT_ETHER，硬件地址为6字节长，整个以太网首部有14字节，而以太网MTU为1500 (ETHERMTU)。

leattach已经指派了MTU，但其他以太网设备驱动程序可能没有执行这个初始化。

4.3节讨论以太网帧组织的更多细节。for循环定位接口的链路层地址，然后初始化结构sockaddr_dl中的以太网硬件地址信息。在系统初始化时，以太网地址被复制到结构arpcom中，现在被复制到链路层地址中。

3.12 ifinit函数

接口结构被初始化并链接到一起后，main (图3-23)调用ifinit，如图3-42所示。

if.c

```
43 void
44 ifinit()
45 {
46     struct ifnet *ifp;

47     for (ifp = ifnet; ifp; ifp = ifp->if_next)
48         if (ifp->if_snd.ifq_maxlen == 0)
49             ifp->if_snd.ifq_maxlen = ifqmaxlen;     /* set default length */
50     if_slowtimo(0);
51 }
```

if.c

图3-42 函数ifinit

43-51 for循环遍历接口列表，并把没有被接口的attach函数设置的每个接口输出队列的最大长度设置为50 (ifqmaxlen)。

输出队列的大小关键要考虑的是发送最大长度数据报的分组的个数。例如以太网，若一个进程调用sendto发送65 507字节的数据，它被分片为45个数据报片，并且每个数据报片被放进接口的输出队列。若队列非常小，由于队列没有空间，进程可能不能发送大的数据报。

if_slowtimo启动接口的监视计时器。当一个接口时钟到期，内核会调用这个接口的把关定时器函数。一个接口可以提前重设时钟来阻止把关定时器函数的调用，或者，若不需要把关定时器函数，则可以把if_timer设置为0。图3-43所示的是函数if_slowtimo。

338-343 if_slowtimo函数有一个参数arg没有使用，但慢超时函数的原型(7.4节)要求有这个参数。

if.c

```
338 void
339 if_slowtimo(arg)
340 void   *arg;
341 {
342     struct ifnet *ifp;
343     int      s = splimp();

344     for (ifp = ifnet; ifp; ifp = ifp->if_next) {
345         if (ifp->if_timer == 0 || --ifp->if_timer)
346             continue;
347         if (ifp->if_watchdog)
348             (*ifp->if_watchdog) (ifp->if_unit);
349     }
350     splx(s);
351     timeout(if_slowtimo, (void *) 0, hz / IFNET_SLOWHZ);
352 }
```

if.c

图3-43 函数if_slowtimo

344-352 if_slowtimo忽略if_timer为0的接口；若if_timer不等于0，if_slowtimo把if_timer减1，并在这个时钟到达0时调用这个接口关联的if_watchdog函数。在调用if_slowtimo时，分组处理进程被splimp阻塞。返回前，ip_slowtimo调用timeout，来以hz/IFNET_SLOWHZ时钟频率调度对它自己的调用。hz是1秒钟内时钟滴答数(通常是100)。它在系统初始化时设置，并保持不变。因为IFNET_SLOWHZ被定义为1，因此内核每赫兹调用一次if_slowtimo，即每秒一次。

函数timeout调度的函数被内核的函数callout回调。详见[Leffler er al. 1989]。

3.13 小结

在本章我们研究了结构ifnet和ifaddr，它们被分配给在系统初始化时发现的每一个网络接口。结构ifnet链接成ifnet链表。每个接口的链路层地址被初始化，并被加到ifnet结构的地址链表中，还存放到数组if_addrs中。

我们讨论了通用sockaddr结构及其成员sa_family和sa_len，它们标识每个地址的类型和长度。我们还查看了一个链路层地址的sockaddr_dl结构的初始化。

在本章中，我们还介绍了在全书中要用到的三个网络接口例子。

习题

3.1 很多Unix系统中的netstat程序列出网络接口及其配置信息。在你接触的系统中试一下命令netstat -i。那个网络接口的名称(if_name)是什么？传输单元的最大长度(if_mtu)是多少？

3.2 在if_slowtimo (图3-43)中，调用splimp和splx出现在循环的外面。与把这些调用放到循环内部相比，这样安排有何优缺点？

3.3 为什么SLIP的交互队列比它的标准输出队列要短？

3.4 为什么if_hdrlen和if_addrlen不在slattach中初始化？

3.5 为SLIP和环回设备画一个与图3-38类似的图。