5

Interfaces: SLIP and Loopback

5.1 Introduction

In Chapter 4 we looked at the Ethernet interface. In this chapter we describe the SLIP and loopback interfaces, as well as the `ioctl` commands used to configure all network interfaces. The TCP compression algorithm used by the SLIP driver is described in Section 29.13. The loopback driver is straightforward and we discuss it here in its entirety.

Figure 5.1, which also appeared as Figure 4.2, lists the entry points to our three example drivers.

`ifnet`	Ethernet	SLIP	Loopback	Description
`if_init`	`leinit`			initialize hardware
`if_output`	`ether_output`	`sloutput`	`looutput`	accept and queue packet for transmission
`if_start`	`lestart`			begin transmission of frame
`if_done`				output complete (unused)
`if_ioctl`	`leioctl`	`slioctl`	`loioctl`	handle `ioctl` commands from a process
`if_reset`	`lereset`			reset the device to a known state
`if_watchdog`				watch the device for failures or collect statistics

Figure 5.1 Interface functions for the example drivers.

5.2 Code Introduction

The files containing code for SLIP and loopback drivers are listed in Figure 5.2.

File	Description
net/if_slvar.h	SLIP definitions
net/if_sl.c	SLIP driver functions
net/if_loop.c	loopback driver

Figure 5.2 Files discussed in this chapter.

Global Variables

The SLIP and loopback interface structures are described in this chapter.

Variable	Datatype	Description
sl_softc	struct sl_softc []	SLIP interface
loif	struct ifnet	loopback interface

Figure 5.3 Global variables introduced in this chapter.

sl_softc is an array, since there can be many SLIP interfaces. loif is not an array, since there can be only one loopback interface.

Statistics

The statistics from the ifnet structure described in Chapter 4 are also updated by the SLIP and loopback drivers. One other variable (which is not in the ifnet structure) collects statistics; it is shown in Figure 5.4.

Variable	Description	Used by SNMP
tk_nin	#bytes received by any serial interface (updated by SLIP driver)	

Figure 5.4 tk_nin variable.

5.3 SLIP Interface

A SLIP interface communicates with a remote system across a standard asynchronous serial line. As with Ethernet, SLIP defines a standard way to frame IP packets as they are transmitted on the serial line. Figure 5.5 shows the encapsulation of an IP packet into a SLIP frame when the IP packet contains SLIP's reserved characters.

Packets are separated by the SLIP END character 0xc0. If the END character appears in the IP packet, it is prefixed with the SLIP ESC character 0xdb and transmitted as 0xdc instead. When the ESC character appears in the IP packet, it is prefixed with the ESC character 0xdb and transmitted as 0xdd.

Since there is no type field in SLIP frames (as there is with Ethernet), SLIP is suitable only for carrying IP packets.

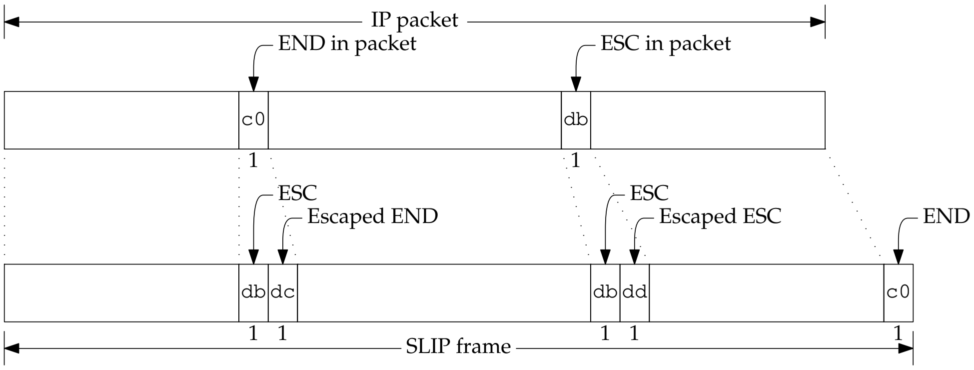

Figure 5.5 SLIP encapsulation of an IP packet.

SLIP is described in RFC 1055 [Romkey 1988], where its many weaknesses and nonstandard status are also stated. Volume 1 contains a more detailed description of SLIP encapsulation.

The Point-to-Point Protocol (PPP) was designed to address SLIP's problems and to provide a standard method for transmitting frames across a serial link. PPP is defined in RFC 1332 [McGregor 1992] and RFC 1548 [Simpson 1993]. Net/3 does not contain an implementation of PPP, so we do not discuss it in this text. See Section 2.6 of Volume 1 for more information regarding PPP. Appendix B describes where to obtain a reference implementation of PPP.

The SLIP Line Discipline: `SLIPDISC`

In Net/3 the SLIP interface relies on an asynchronous serial device driver to send and receive the data. Traditionally these device drivers have been called TTYs (teletypes). The Net/3 TTY subsystem includes the notion of a *line discipline* that acts as a filter between the physical device and I/O system calls such as `read` and `write`. A line discipline implements features such as line editing, newline and carriage-return processing, tab expansion, and more. The SLIP interface appears as a line discipline to the TTY subsystem, but it does not pass incoming data to a process reading from the device and does not accept outgoing data from a process writing to the device. Instead, the SLIP interface passes incoming packets to the IP input queue and accepts outgoing packets through the `if_output` function in SLIP's `ifnet` structure. The kernel identifies line disciplines by an integer constant, which for SLIP is `SLIPDISC`.

Figure 5.6 shows a traditional line discipline on the left and the SLIP discipline on the right. We show the process on the right as `slattach` since it is the program that initializes a SLIP interface. The details of the TTY subsystem and line disciplines are outside the scope of this text. We present only the information required to understand the workings of the SLIP code. For more information about the TTY subsystem see [Leffler et al. 1989]. Figure 5.7 lists the functions that implement the SLIP driver. The middle columns indicate whether the function implements line discipline features, network interface features, or both.

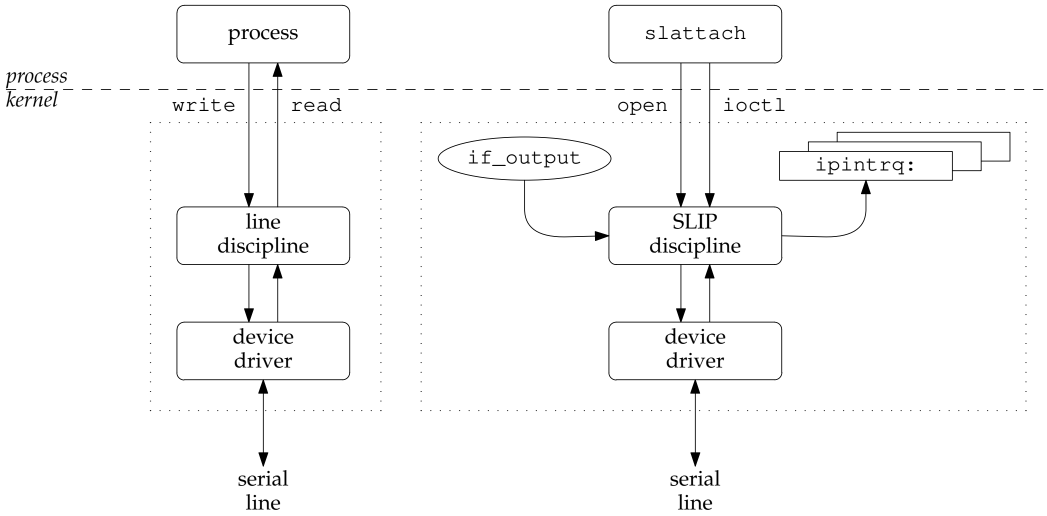

Figure 5.6 The SLIP interface as a line discipline.

Function	Network Interface	Line Discipline	Description
slattach	•		initialize and attach sl_softc structures to ifnet list
slinit	•		initialize the SLIP data structures
sloutput	•		queue outgoing packets for transmission on associated TTY device
slioctl	•		process socket ioctl requests
sl_btom	•		convert a device buffer to an mbuf chain
slopen		•	attach sl_softc structure to TTY device and initialize driver
slclose		•	detach sl_softc structures from TTY device, mark interface as down, and release memory
sltioctl		•	process TTY ioctl commands
slstart	•	•	dequeue packet and begin transmitting data on TTY device
slinput	•	•	process incoming byte from TTY device, queue incoming packet if an entire frame has been received

Figure 5.7 The functions in the SLIP device driver.

The SLIP driver in Net/3 supports compression of TCP packet headers for better throughput. We discuss header compression in Section 29.13, so Figure 5.7 omits the functions that implement this feature.

The Net/3 SLIP interface also supports an escape sequence. When detected by the receiver, the sequence shuts down SLIP processing and returns the device to the standard line discipline. We omit this processing from our discussion.

Figure 5.8 shows the complex relationship between SLIP as a line discipline and SLIP as a network interface.

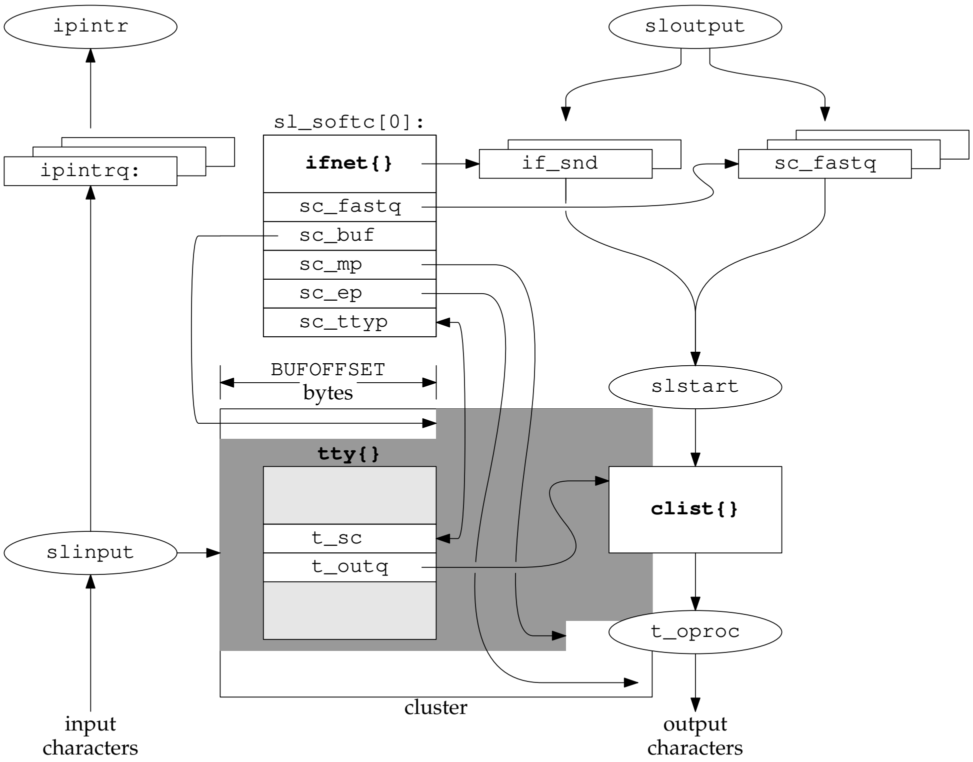

Figure 5.8 SLIP device driver.

In Net/3 sc_ttyp and t_sc point to the tty structure and the sl_softc[0] structure. Instead of cluttering the figure with two arrows, we use a double-ended arrow positioned at each pointer to illustrated the two links between the structures.

Figure 5.8 contains a lot of information:

- The network interface is represented by the sl_softc structure and the TTY device by the tty structure.
- Incoming bytes are stored in the cluster (shown behind the tty structure). When a complete SLIP frame is received, the enclosed IP packet is put on the ipintrq by slinput.
- Outgoing packets are dequeued from if_snd or sc_fastq, converted to SLIP frames, and passed to the TTY device by slstart. The TTY buffers outgoing bytes in the clist structure. The t_oproc function drains and transmits the bytes held in the clist structure.

SLIP Initialization: `slopen` and `slinit`

We discussed in Section 3.7 how `slattach` initializes the `sl_softc` structures. The interface remains initialized but inoperative until a program (usually `slattach`) opens a TTY device (e.g., `/dev/tty01`) and issues an `ioctl` command to replace the standard line discipline with the SLIP discipline. At this point the TTY subsystem calls the line discipline's open function (in this case `slopen`), which establishes the association between a particular TTY device and a particular SLIP interface. `slopen` is shown in Figure 5.9.

if_sl.c

```
181 int
182 slopen(dev, tp)
183 dev_t   dev;
184 struct tty *tp;
185 {
186     struct proc *p = curproc;   /* XXX */
187     struct sl_softc *sc;
188     int     nsl;
189     int     error;

190     if (error = suser(p->p_ucred, &p->p_acflag))
191         return (error);

192     if (tp->t_line == SLIPDISC)
193         return (0);

194     for (nsl = NSL, sc = sl_softc; --nsl >= 0; sc++)
195         if (sc->sc_ttyp == NULL) {
196             if (slinit(sc) == 0)
197                 return (ENOBUFS);
198             tp->t_sc = (caddr_t) sc;
199             sc->sc_ttyp = tp;
200             sc->sc_if.if_baudrate = tp->t_ospeed;
201             ttyflush(tp, FREAD | FWRITE);
202             return (0);
203         }
204     return (ENXIO);
205 }
```

if_sl.c

Figure 5.9 The `slopen` function.

181–193 Two arguments are passed to `slopen`: `dev`, a kernel device identifier that `slopen` does not use; and `tp`, a pointer to the `tty` structure associated with the TTY device. First some precautions: if the process does not have superuser privileges, or if the TTY's line discipline is set to `SLIPDISC` already, `slopen` returns immediately.

194–205 The `for` loop searches the array of `sl_softc` structures for the first unused entry, calls `slinit` (Figure 5.10), joins the `tty` and `sl_softc` structures by `t_sc` and `sc_ttyp`, and copies the TTY output speed (`t_ospeed`) into the SLIP interface. `ttyflush` discards any pending input or output data in the TTY queues. `slopen` returns `ENXIO` if a SLIP interface structure is not available, or 0 if it was successful.

Notice that the first available `sl_softc` structure is associated with the TTY device. There need not be a fixed mapping between TTY devices and SLIP interfaces if the system has more than one SLIP line. In fact, the mapping depends on the order in which `slattach` opens and closes the TTY devices.

The `slinit` function shown in Figure 5.10 initializes the `sl_softc` structure.

if_sl.c

```
156 static int
157 slinit(sc)
158 struct sl_softc *sc;
159 {
160     caddr_t p;

161     if (sc->sc_ep == (u_char *) 0) {
162         MCLALLOC(p, M_WAIT);
163         if (p)
164             sc->sc_ep = (u_char *) p + SLBUFSIZE;
165         else {
166             printf("sl%d: can't allocate buffer\n", sc - sl_softc);
167             sc->sc_if.if_flags &= ~IFF_UP;
168             return (0);
169         }
170     }
171     sc->sc_buf = sc->sc_ep - SLMAX;
172     sc->sc_mp = sc->sc_buf;
173     sl_compress_init(&sc->sc_comp);
174     return (1);
175 }
```

if_sl.c

Figure 5.10 The `slinit` function.

156–175 The `slinit` function allocates an mbuf cluster and attaches it to the `sl_softc` structure with three pointers. Incoming bytes are stored in the cluster until an entire SLIP frame has been received. `sc_buf` always points to the start of the packet in the cluster, `sc_mp` points to the location of the next byte to be received, and `sc_ep` points to the end of the cluster. `sl_compress_init` initializes the TCP header compression state for this link (Section 29.13).

In Figure 5.8 we see that `sc_buf` does not point to the first byte in the cluster. `slinit` leaves room for 148 bytes (`BUFOFFSET`), as the incoming packet may have a compressed header that will expand to fill this space. The bytes that have already been received are shaded in the cluster. We see that `sc_mp` points to the byte just after the last byte received and `sc_ep` points to the end of the cluster. Figure 5.11 shows the relationships between several SLIP constants.

All that remains to make the interface operational is to assign it an IP address. As with the Ethernet driver, we postpone the discussion of address assignment until Section 6.6.

Constant	Value	Description
MCLBYTES	2048	size of an mbuf cluster
SLBUFSIZE	2048	maximum size of an uncompressed SLIP packet—including a BPF header
SLIP_HDRLEN	16	size of SLIP BPF header
BUFOFFSET	148	maximum size of an expanded TCP/IP header plus room for a BPF header
SLMAX	1900	maximum size of a compressed SLIP packet stored in a cluster
SLMTU	296	optimal size of SLIP packet; results in minimal delay with good bulk throughput
SLIP_HIWAT	100	maximum number of bytes to queue in TTY output queue
BUFOFFSET + SLMAX = SLBUFSIZE = MCLBYTES		

Figure 5.11 SLIP constants.

SLIP Input Processing: `slinput`

The TTY device driver delivers incoming characters to the SLIP line discipline one at a time by calling `slinput`. Figure 5.12 shows the `slinput` function but omits the end-of-frame processing, which is discussed separately.

if_sl.c

```
void
slinput(c, tp)
int     c;
struct tty *tp;
{
    struct sl_softc *sc;
    struct mbuf *m;
    int     len;
    int     s;
    u_char  chdr[CHDR_LEN];

    tk_nin++;
    sc = (struct sl_softc *) tp->t_sc;
    if (sc == NULL)
        return;
    if (c & TTY_ERRORMASK || ((tp->t_state & TS_CARR_ON) == 0 &&
                              (tp->t_cflag & CLOCAL) == 0)) {
        sc->sc_flags |= SC_ERROR;
        return;
    }
    c &= TTY_CHARMASK;

    ++sc->sc_if.if_ibytes;

    switch (c) {

    case TRANS_FRAME_ESCAPE:
        if (sc->sc_escape)
            c = FRAME_ESCAPE;
        break;
```

```
553     case TRANS_FRAME_END:
554         if (sc->sc_escape)
555             c = FRAME_END;
556         break;

557     case FRAME_ESCAPE:
558         sc->sc_escape = 1;
559         return;

560     case FRAME_END:

                      /* FRAME_END code (Figure 5.13) */

636     }
637     if (sc->sc_mp < sc->sc_ep) {
638         *sc->sc_mp++ = c;
639         sc->sc_escape = 0;
640         return;
641     }
642     /* can't put lower; would miss an extra frame */
643     sc->sc_flags |= SC_ERROR;

644   error:
645     sc->sc_if.if_ierrors++;
646   newpack:
647     sc->sc_mp = sc->sc_buf = sc->sc_ep - SLMAX;
648     sc->sc_escape = 0;
649 }
```

if_sl.c

Figure 5.12 `slinput` function.

527–545 The arguments to `slinput` are `c`, the next input character; and `tp`, a pointer to the device's `tty` structure. The global integer `tk_nin` counts the incoming characters for all TTY devices. `slinput` converts `tp->t_sc` to `sc`, a pointer to an `sl_softc` structure. If there is no interface associated with the TTY device, `slinput` returns immediately.

The first argument to `slinput` is an integer. In addition to the received character, `c` contains control information sent from the TTY device driver in the high-order bits. If an error is indicated in `c` or the modem-control lines are not enabled and should not be ignored, `SC_ERROR` is set and `slinput` returns. Later, when `slinput` processes the END character, the frame is discarded. The `CLOCAL` flag indicates that the system should treat the line as a local line (i.e., not a dialup line) and should not expect to see modem-control signals.

546–636 `slinput` discards the control bits in `c` by masking it with `TTY_CHARMASK`, updates the count of bytes received on the interface, and jumps based on the received character:

- If `c` is an escaped ESC character and the *previous* character was an ESC, `slinput` replaces `c` with an ESC character.
- If `c` is an escaped END character and the *previous* character was an ESC, `slinput` replaces `c` with an END character.

- If `c` is the SLIP ESC character, `sc_escape` is set and `slinput` returns immediately (i.e., the ESC character is discarded).
- If `c` is the SLIP END character, the packet is put on the IP input queue. The processing for the SLIP frame end character is shown in Figure 5.13.

The common flow of control through this `switch` statement is to fall through (there is no `default` case). Most bytes are data and don't match any of the four cases. Control also falls through the `switch` in the first two cases.

637–649 If control falls through the `switch`, the received character is part of the IP packet. The character is stored in the cluster (if there is room), the pointers are advanced, `sc_escape` is cleared, and `slinput` returns.

If the cluster is full, the character is discarded and `slinput` sets `SC_ERROR`. Control reaches `error` when the cluster is full or when an error is detected in the end-of-frame processing. At `newpack` the cluster pointers are reset for a new packet, `sc_escape` is cleared, and `slinput` returns.

Figure 5.13 shows the `FRAME_END` code omitted from Figure 5.12.

if_sl.c

```
    case FRAME_END:
        if (sc->sc_flags & SC_ERROR) {
            sc->sc_flags &= ~SC_ERROR;
            goto newpack;
        }
        len = sc->sc_mp - sc->sc_buf;
        if (len < 3)
            /* less than min length packet - ignore */
            goto newpack;

        if (sc->sc_bpf) {
            /*
             * Save the compressed header, so we
             * can tack it on later.  Note that we
             * will end up copying garbage in some
             * cases but this is okay.  We remember
             * where the buffer started so we can
             * compute the new header length.
             */
            bcopy(sc->sc_buf, chdr, CHDR_LEN);
        }
        if ((c = (*sc->sc_buf & 0xf0)) != (IPVERSION << 4)) {
            if (c & 0x80)
                c = TYPE_COMPRESSED_TCP;
            else if (c == TYPE_UNCOMPRESSED_TCP)
                *sc->sc_buf &= 0x4f;    /* XXX */
            /*
             * We've got something that's not an IP packet.
             * If compression is enabled, try to decompress it.
             * Otherwise, if auto-enable compression is on and
             * it's a reasonable packet, decompress it and then
             * enable compression.  Otherwise, drop it.
             */
```

```
            if (sc->sc_if.if_flags & SC_COMPRESS) {
                len = sl_uncompress_tcp(&sc->sc_buf, len,
                                        (u_int) c, &sc->sc_comp);
                if (len <= 0)
                    goto error;
            } else if ((sc->sc_if.if_flags & SC_AUTOCOMP) &&
                       c == TYPE_UNCOMPRESSED_TCP && len >= 40) {
                len = sl_uncompress_tcp(&sc->sc_buf, len,
                                        (u_int) c, &sc->sc_comp);
                if (len <= 0)
                    goto error;
                sc->sc_if.if_flags |= SC_COMPRESS;
            } else
                goto error;
        }
        if (sc->sc_bpf) {
            /*
             * Put the SLIP pseudo-"link header" in place.
             * We couldn't do this any earlier since
             * decompression probably moved the buffer
             * pointer.  Then, invoke BPF.
             */
            u_char *hp = sc->sc_buf - SLIP_HDRLEN;

            hp[SLX_DIR] = SLIPDIR_IN;
            bcopy(chdr, &hp[SLX_CHDR], CHDR_LEN);
            bpf_tap(sc->sc_bpf, hp, len + SLIP_HDRLEN);
        }
        m = sl_btom(sc, len);
        if (m == NULL)
            goto error;

        sc->sc_if.if_ipackets++;
        sc->sc_if.if_lastchange = time;
        s = splimp();
        if (IF_QFULL(&ipintrq)) {
            IF_DROP(&ipintrq);
            sc->sc_if.if_ierrors++;
            sc->sc_if.if_iqdrops++;
            m_freem(m);
        } else {
            IF_ENQUEUE(&ipintrq, m);
            schednetisr(NETISR_IP);
        }
        splx(s);
        goto newpack;
```
if_sl.c

Figure 5.13 `slinput` function: end-of-frame processing.

560–579 `slinput` discards an incoming SLIP packet immediately if `SC_ERROR` was set while the packet was being received or if the packet is less than 3 bytes in length (remember that the packet may be compressed).

If the SLIP interface is tapped by BPF, `slinput` saves a copy of the (possibly compressed) header in the `chdr` array.

580–606 By examining the first byte of the packet, slinput determines if it is an uncompressed IP packet, a compressed TCP segment, or an uncompressed TCP segment. The type is saved in c and the type information is removed from the first byte of data (Section 29.13). If the packet appears to be compressed and compression is enabled, sl_uncompress_tcp attempts to uncompress the packet. If compression is not enabled, auto-enable compression is on, and if the packet is large enough sl_uncompress_tcp is also called. If it is a compressed TCP packet, the compression flag is set.

slinput discards packets it does not recognize by jumping to error. Section 29.13 discusses the header compression techniques in more detail. The cluster now contains a complete uncompressed packet.

607–618 After SLIP has decompressed the packet, the header and data are passed to BPF. Figure 5.14 shows the layout of the buffer constructed by slinput.

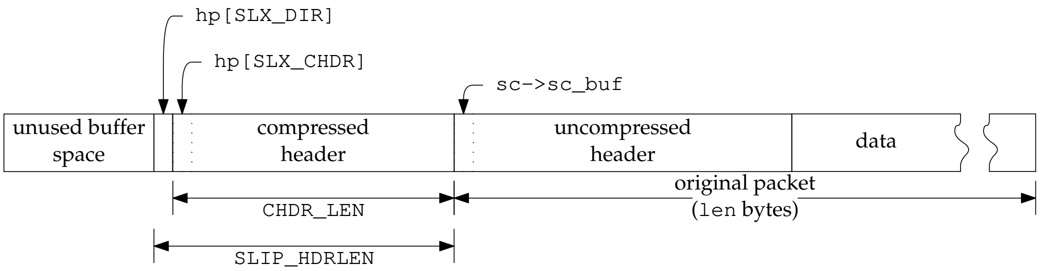

Figure 5.14 SLIP packet in BPF format.

The first byte of the BPF header encodes the direction of the packet, in this case incoming (SLIPDIR_IN). The next 15 bytes contain the compressed header. The entire packet is passed to bpf_tap.

619–635 sl_btom converts the cluster to an mbuf chain. If the packet is small enough to fit in a single mbuf, sl_btom copies the packet from the cluster to a newly allocated mbuf packet header; otherwise sl_btom attaches the cluster to an mbuf and allocates a new cluster for the interface. This is faster than copying from one cluster to another. We do not show sl_btom in this text.

Since only IP packets are transmitted on a SLIP interface, slinput does not have to select a protocol queue (as it does in the Ethernet driver). The packet is queued on ipintrq, an IP software interrupt is scheduled, and slinput jumps to newpack, where it updates the cluster packet pointers and clears sc_escape.

> While the SLIP driver increments if_ierrors if the packet cannot be queued on ipintrq, neither the Ethernet nor loopback drivers increment this statistic in the same situation.

Access to the IP input queue must be protected by splimp even though slinput is called at spltty. Recall from Figure 1.14 that an splimp interrupt can preempt spltty processing.

SLIP Output Processing: `sloutput`

As with all network interfaces, output processing begins when a network-level protocol calls the interface's `if_output` function. For the Ethernet driver, the function is `ether_output`. For SLIP, the function is `sloutput` (Figure 5.15).

if_sl.c

```
int
sloutput(ifp, m, dst, rtp)
struct ifnet *ifp;
struct mbuf *m;
struct sockaddr *dst;
struct rtentry *rtp;
{
    struct sl_softc *sc = &sl_softc[ifp->if_unit];
    struct ip *ip;
    struct ifqueue *ifq;
    int     s;

    /*
     * Cannot happen (see slioctl).  Someday we will extend
     * the line protocol to support other address families.
     */
    if (dst->sa_family != AF_INET) {
        printf("sl%d: af%d not supported\n", sc->sc_if.if_unit,
               dst->sa_family);
        m_freem(m);
        sc->sc_if.if_noproto++;
        return (EAFNOSUPPORT);
    }
    if (sc->sc_ttyp == NULL) {
        m_freem(m);
        return (ENETDOWN);      /* sort of */
    }
    if ((sc->sc_ttyp->t_state & TS_CARR_ON) == 0 &&
        (sc->sc_ttyp->t_cflag & CLOCAL) == 0) {
        m_freem(m);
        return (EHOSTUNREACH);
    }
    ifq = &sc->sc_if.if_snd;
    ip = mtod(m, struct ip *);
    if (sc->sc_if.if_flags & SC_NOICMP && ip->ip_p == IPPROTO_ICMP) {
        m_freem(m);
        return (ENETRESET);     /* XXX ? */
    }
    if (ip->ip_tos & IPTOS_LOWDELAY)
        ifq = &sc->sc_fastq;
    s = splimp();
    if (IF_QFULL(ifq)) {
        IF_DROP(ifq);
        m_freem(m);
        splx(s);
        sc->sc_if.if_oerrors++;
        return (ENOBUFS);
    }
```

```
306     IF_ENQUEUE(ifq, m);
307     sc->sc_if.if_lastchange = time;
308     if (sc->sc_ttyp->t_outq.c_cc == 0)
309         slstart(sc->sc_ttyp);
310     splx(s);
311     return (0);
312 }
```
if_sl.c

Figure 5.15 `sloutput` function.

259–289 The four arguments to `sloutput` are: `ifp`, a pointer to the SLIP `ifnet` structure (in this case an `sl_softc` structure); `m`, a pointer to the packet to be queued for output; `dst`, the next-hop destination for the packet; and `rtp`, a pointer to a route entry. The fourth argument is not used by `sloutput`, but it is required since `sloutput` must match the prototype for the `if_output` function in the `ifnet` structure.

`sloutput` ensures that `dst` is an IP address, that the interface is connected to a TTY device, and that the TTY device is operating (i.e., the carrier is on or should be ignored). An error is returned immediately if any of these tests fail.

290–291 The SLIP interface maintains two queues of outgoing packets. The standard queue, `if_snd`, is selected by default.

292–295 If the outgoing packet contains an ICMP message and `SC_NOICMP` is set for the interface, the packet is discarded. This prevents a SLIP link from being overwhelmed by extraneous ICMP packets (e.g., ECHO packets) sent by a malicious user (Chapter 11).

The error code `ENETRESET` indicates that the packet was discarded because of a policy decision (versus a network failure). We'll see in Chapter 11 that the error is silently discarded unless the ICMP message was generated locally, in which case an error is returned to the process that tried to send the message.

> Net/2 returned a 0 in this case. To a diagnostic tool such as `ping` or `traceroute` it would appear as if the packet disappeared since the output operation would report a successful completion.
>
> In general, ICMP messages can be discarded. They are not required for correct operation, but discarding them makes troubleshooting more difficult and may lead to less than optimal routing decisions, poorer performance, and wasted network resources.

296–297 If the TOS field in the outgoing packet specifies low-delay service (`IPTOS_LOWDELAY`), the output queue is changed to `sc_fastq`.

> RFC 1700 and RFC 1349 [Almquist 1992] specify the TOS settings for the standard protocols. Low-delay service is specified for Telnet, Rlogin, FTP (control), TFTP, SMTP (command phase), and DNS (UDP query). See Section 3.2 of Volume 1 for more details.
>
> In previous BSD releases, the `ip_tos` was not set correctly by applications. The SLIP driver implemented TOS queueing by examining the transport headers contained within the IP packet. If it found TCP packets for the FTP (command), Telnet, or Rlogin ports, the packet was queued as if `IPTOS_LOWDELAY` was specified. Many routers continue this practice, since many implementations of these interactive services still do not set `ip_tos`.

298–312 The packet is now placed on the selected queue, the interface statistics are updated, and (if the TTY output queue is empty) `sloutput` calls `slstart` to initiate transmission of the packet.

> SLIP increments `if_oerrors` if the interface queue is full; `ether_output` does not.

Unlike the Ethernet output function (`ether_output`), `sloutput` does not construct a data-link header for the outgoing packet. Since the only other system on a SLIP network is at the other end of the serial link, there is no need for hardware addresses or a protocol, such as ARP, to convert between IP addresses and hardware addresses. Protocol identifiers (such as the Ethernet *type* field) are also superfluous, since a SLIP link carries only IP packets.

`slstart` Function

In addition to the call by `sloutput`, the TTY device driver calls `slstart` when it drains its output queue and needs more bytes to transmit. The TTY subsystem manages its queues through a `clist` structure. In Figure 5.8 the output clist `t_outq` is shown below `slstart` and above the device's `t_oproc` function. `slstart` adds bytes to the queue, while `t_oproc` drains the queue and transmits the bytes.

The `slstart` function is shown in Figure 5.16.

318–358 When `slstart` is called, `tp` points to the device's `tty` structure. The body of `slstart` consists of a single `for` loop. If the output queue `t_outq` is not empty, `slstart` calls the output function for the device, `t_oproc`, which transmits as many bytes as the device will accept. If more than 100 bytes (`SLIP_HIWAT`) remain in the TTY output queue, `slstart` returns instead of adding another packet's worth of bytes to the queue. The output device generates an interrupt when it has transmitted all the bytes, and the TTY subsystem calls `slstart` when the output list is empty.

If the TTY output queue is empty, a packet is dequeued from `sc_fastq` or, if `sc_fastq` is empty, from the `if_snd` queue, thus transmitting all interactive packets before any other packets.

> There are no standard SNMP variables to count packets queued according to the TOS fields. The `XXX` comment in line 353 indicates that the SLIP driver is counting low-delay packets in `if_omcasts`, *not* multicast packets.

359–383 If the SLIP interface is tapped by BPF, `slstart` makes a copy of the output packet before any header compression occurs. The copy is saved on the stack in the `bpfbuf` array .

384–388 If compression is enabled and the packet contains a TCP segment, `sloutput` calls `sl_compress_tcp`, which attempts to compress the packet. The resulting packet type is returned and logically ORed with the first byte in IP header (Section 29.13).

389–398 The compressed header is now copied into the BPF header, and the direction recorded as `SLIPDIR_OUT`. The completed BPF packet is passed to `bpf_tap`.

483–484 `slstart` returns if the `for` loop terminates.

if_sl.c

```
void
slstart(tp)
struct tty *tp;
{
    struct sl_softc *sc = (struct sl_softc *) tp->t_sc;
    struct mbuf *m;
    u_char *cp;
    struct ip *ip;
    int     s;
    struct mbuf *m2;
    u_char  bpfbuf[SLMTU + SLIP_HDRLEN];
    int     len;
    extern int cfreecount;

    for (;;) {
        /*
         * If there is more in the output queue, just send it now.
         * We are being called in lieu of ttstart and must do what
         * it would.
         */
        if (tp->t_outq.c_cc != 0) {
            (*tp->t_oproc) (tp);
            if (tp->t_outq.c_cc > SLIP_HIWAT)
                return;
        }
        /*
         * This happens briefly when the line shuts down.
         */
        if (sc == NULL)
            return;

        /*
         * Get a packet and send it to the interface.
         */
        s = splimp();
        IF_DEQUEUE(&sc->sc_fastq, m);
        if (m)
            sc->sc_if.if_omcasts++;     /* XXX */
        else
            IF_DEQUEUE(&sc->sc_if.if_snd, m);
        splx(s);
        if (m == NULL)
            return;

        /*
         * We do the header compression here rather than in sloutput
         * because the packets will be out of order if we are using TOS
         * queueing, and the connection id compression will get
         * munged when this happens.
         */
        if (sc->sc_bpf) {
            /*
             * We need to save the TCP/IP header before it's
             * compressed.  To avoid complicated code, we just
             * copy the entire packet into a stack buffer (since
```

```
370              * this is a serial line, packets should be short
371              * and/or the copy should be negligible cost compared
372              * to the packet transmission time).
373              */
374             struct mbuf *m1 = m;
375             u_char *cp = bpfbuf + SLIP_HDRLEN;

376             len = 0;
377             do {
378                 int     mlen = m1->m_len;

379                 bcopy(mtod(m1, caddr_t), cp, mlen);
380                 cp += mlen;
381                 len += mlen;
382             } while (m1 = m1->m_next);
383         }
384         if ((ip = mtod(m, struct ip *))->ip_p == IPPROTO_TCP) {
385             if (sc->sc_if.if_flags & SC_COMPRESS)
386                 *mtod(m, u_char *) |= sl_compress_tcp(m, ip,
387                                                       &sc->sc_comp, 1);
388         }
389         if (sc->sc_bpf) {
390             /*
391              * Put the SLIP pseudo-"link header" in place.  The
392              * compressed header is now at the beginning of the
393              * mbuf.
394              */
395             bpfbuf[SLX_DIR] = SLIPDIR_OUT;
396             bcopy(mtod(m, caddr_t), &bpfbuf[SLX_CHDR], CHDR_LEN);
397             bpf_tap(sc->sc_bpf, bpfbuf, len + SLIP_HDRLEN);
398         }

                       /* packet output code */

483     }
484 }
```

if_sl.c

Figure 5.16 `slstart` function: packet dequeueing.

The next section of `slstart` (Figure 5.17) discards packets if the system is low on memory, and implements a simple technique for discarding data generated by noise on the serial line. This is the code omitted from Figure 5.16.

399–409 If the system is low on clist structures, the packet is discarded and counted as a collision. By continuing the loop instead of returning, `slstart` quickly discards all remaining packets queued for output. Each iteration discards a packet, since the device still has too many bytes queued for output. Higher-level protocols must detect the lost packets and retransmit them.

410–418 If the TTY output queue is empty, the communication line may have been idle for a period of time and the receiver at the other end may have received extraneous data created by line noise. `slstart` places an extra SLIP END character in the output queue. A 0-length frame or a frame created by noise on the line should be discarded by the SLIP interface or IP protocol at the receiver.

if_sl.c

```
        sc->sc_if.if_lastchange = time;
        /*
         * If system is getting low on clists, just flush our
         * output queue (if the stuff was important, it'll get
         * retransmitted).
         */
        if (cfreecount < CLISTRESERVE + SLMTU) {
            m_freem(m);
            sc->sc_if.if_collisions++;
            continue;
        }
        /*
         * The extra FRAME_END will start up a new packet, and thus
         * will flush any accumulated garbage.  We do this whenever
         * the line may have been idle for some time.
         */
        if (tp->t_outq.c_cc == 0) {
            ++sc->sc_if.if_obytes;
            (void) putc(FRAME_END, &tp->t_outq);
        }
```

if_sl.c

Figure 5.17 `slstart` function: resource shortages and line noise.

Figure 5.18 illustrates this technique for discarding line noise and is attributed to Phil Karn in RFC 1055. In Figure 5.18, the second end-of-frame (END) is transmitted because the line was idle for a period of time. The invalid frame created by the noise and the END byte is discarded by the receiving system.

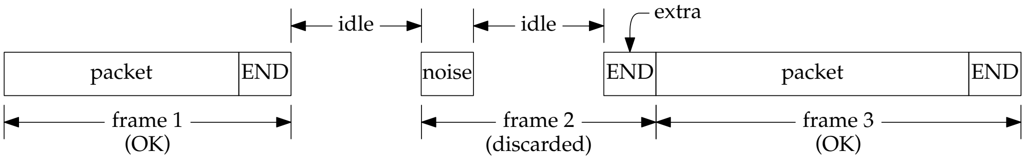

Figure 5.18 Karn's method for discarding noise on a SLIP line.

In Figure 5.19 there is no noise on the line and the 0-length frame is discarded by the receiving system.

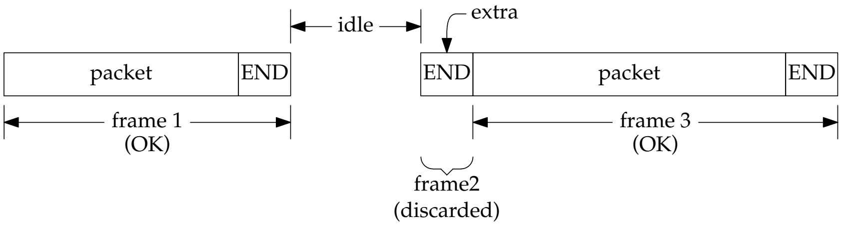

Figure 5.19 Karn's method with no noise.

The next section of `slstart` (Figure 5.20) transfers the data from an mbuf to the output queue for the TTY device.

if_sl.c

```
        while (m) {
            u_char *ep;

            cp = mtod(m, u_char *);
            ep = cp + m->m_len;
            while (cp < ep) {
                /*
                 * Find out how many bytes in the string we can
                 * handle without doing something special.
                 */
                u_char *bp = cp;

                while (cp < ep) {
                    switch (*cp++) {
                    case FRAME_ESCAPE:
                    case FRAME_END:
                        --cp;
                        goto out;
                    }
                }
              out:
                if (cp > bp) {
                    /*
                     * Put n characters at once
                     * into the tty output queue.
                     */
                    if (b_to_q((char *) bp, cp - bp,
                               &tp->t_outq))
                        break;
                    sc->sc_if.if_obytes += cp - bp;
                }
                /*
                 * If there are characters left in the mbuf,
                 * the first one must be special..
                 * Put it out in a different form.
                 */
                if (cp < ep) {
                    if (putc(FRAME_ESCAPE, &tp->t_outq))
                        break;
                    if (putc(*cp++ == FRAME_ESCAPE ?
                             TRANS_FRAME_ESCAPE : TRANS_FRAME_END,
                             &tp->t_outq)) {
                        (void) unputc(&tp->t_outq);
                        break;
                    }
                    sc->sc_if.if_obytes += 2;
                }
            }
            MFREE(m, m2);
            m = m2;
        }
```

if_sl.c

Figure 5.20 `slstart` function: packet transmission.

419–467 The outer `while` loop in this section is executed once for each mbuf in the chain. The middle `while` loop transfers the data from each mbuf to the output device. The inner `while` loop advances `cp` until it finds an END or ESC character. `b_to_q` transfers the bytes between `bp` and `cp`. END and ESC characters are escaped and queued with two calls to `putc`. This middle loop is repeated until all the bytes in the mbuf are passed to the TTY device's output queue. Figure 5.21 illustrates this process with an mbuf containing a SLIP END character and a SLIP ESC character.

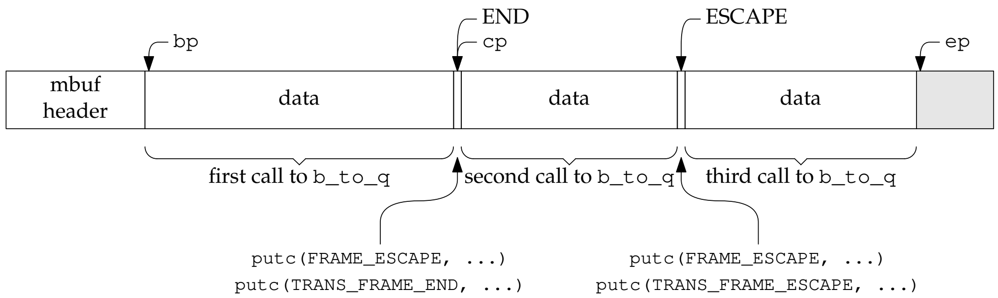

Figure 5.21 SLIP transmission of a single mbuf.

`bp` marks the beginning of the first section of the mbuf to transfer with `b_to_q`, and `cp` marks the end of the first section. `ep` marks the end of the data in the mbuf.

If `b_to_q` or `putc` fail (i.e., data cannot be queued on the TTY device), the `break` causes `slstart` to fall out of the inner `while` loop. The failure indicates that the kernel has run out of clist resources. After each mbuf is copied to the TTY device, or when an error occurs, the mbuf is released, `m` is advanced to the next mbuf in the chain, and the outer `while` loop continues until all the mbufs in the chain have been processed.

Figure 5.22 shows the processing done by `slstart` to complete the outgoing frame.

if_sl.c

```
        if (putc(FRAME_END, &tp->t_outq)) {
            /*
             * Not enough room.  Remove a char to make room
             * and end the packet normally.
             * If you get many collisions (more than one or two
             * a day) you probably do not have enough clists
             * and you should increase "nclist" in param.c.
             */
            (void) unputc(&tp->t_outq);
            (void) putc(FRAME_END, &tp->t_outq);
            sc->sc_if.if_collisions++;
        } else {
            ++sc->sc_if.if_obytes;
            sc->sc_if.if_opackets++;
        }
```

if_sl.c

Figure 5.22 `slstart` function: end-of-frame processing.

468–482 Control reaches this code when the outer `while` loop has finished queueing the bytes on the output queue. The driver sends a SLIP END character, which terminates the frame.

If an error occurred while queueing the bytes, the outgoing frame is invalid and is detected by the receiving system because of an invalid checksum or length.

Whether or not the frame is terminated because of an error, if the END character does not fit on the output queue, the *last* character on the queue is discarded and `slstart` ends the frame. This guarantees that an END character is transmitted. The invalid frame is discarded at the destination.

SLIP Packet Loss

The SLIP interface provides a good example of a best-effort service. SLIP discards packets if the TTY is overloaded; it truncates packets if resources are unavailable after the packet transmission has started, and it inserts extraneous null packets to detect and discard line noise. In each of these cases, no error message is generated. SLIP depends on IP and the transport layers to detect damaged and missing packets.

On a router forwarding packets from a fast interface such as Ethernet to a low-speed SLIP line, a large percentage of packets are discarded if the sender does not recognize the bottleneck and respond by throttling back the data rate. In Section 25.11 we'll see how TCP detects and responds to this condition. Applications using a protocol without flow control, such as UDP, must recognize and respond to this condition on their own (Exercise 5.8).

SLIP Performance Considerations

The MTU of a SLIP frame (`SLMTU`), the clist high-water mark (`SLIP_HIWAT`), and SLIP's TOS queueing strategies are all designed to minimize the delay inherent in a slow serial link for interactive traffic.

1. A small MTU improves the delay for interactive data (such as keystrokes and echoes), but hurts the throughput for bulk data transfer. A large MTU improves bulk data throughput, but increases interactive delays. Another problem with SLIP links is that a single typed character is burdened with 40 bytes of TCP and IP header information, which increases the communication delay.

   The solution is to pick an MTU large enough to provide good interactive response time and decent bulk data throughput, and to compress TCP/IP headers to reduce the per-packet overhead. RFC 1144 [Jacobson 1990a] describes a compression scheme and the timing calculations that result in selecting an MTU of 296 for a typical 9600 bits/sec asynchronous SLIP link. We describe Compressed SLIP (CSLIP) in Section 29.13. Sections 2.10 and 7.2 of Volume 1 summarize the timing considerations and illustrate the delay on SLIP links.

2. If too many bytes are buffered in the clist (because `SLIP_HIWAT` is set too high), the TOS queueing will be thwarted as new interactive traffic waits behind the large amount of buffered data. If SLIP passes 1 byte at a time to the TTY driver

(because `SLIP_HIWAT` is set too low), the device calls `slstart` for each byte and the line is idle for a brief period of time after each byte is transferred. Setting `SLIP_HIWAT` to 100 minimizes the amount of data queued at the device and reduces the frequency at which the TTY subsystem must call `slstart` to approximately once every 100 characters.

3. As described, the SLIP driver provides TOS queueing by transmitting interactive traffic from the `sc_fastq` queue before other traffic on the standard interface queue, `if_snd`.

`slclose` Function

For completeness, we show the `slclose` function, which is called when the `slattach` program closes SLIP's TTY device and terminates the connection to the remote system.

if_sl.c

```
210 void
211 slclose(tp)
212 struct tty *tp;
213 {
214     struct sl_softc *sc;
215     int     s;

216     ttywflush(tp);
217     s = splimp();               /* actually, max(spltty, splnet) */
218     tp->t_line = 0;
219     sc = (struct sl_softc *) tp->t_sc;
220     if (sc != NULL) {
221         if_down(&sc->sc_if);
222         sc->sc_ttyp = NULL;
223         tp->t_sc = NULL;
224         MCLFREE((caddr_t) (sc->sc_ep - SLBUFSIZE));
225         sc->sc_ep = 0;
226         sc->sc_mp = 0;
227         sc->sc_buf = 0;
228     }
229     splx(s);
230 }
```

if_sl.c

Figure 5.23 `slclose` function.

210–230 `tp` points to the TTY device to be closed. `slclose` flushes any remaining data out to the serial device, blocks TTY and network processing, and resets the TTY to the default line discipline. If the TTY device is attached to a SLIP interface, the interface is shut down, the links between the two structures are severed, the mbuf cluster associated with the interface is released, and the pointers into the now-discarded cluster are reset. Finally, `splx` reenables the TTY and network interrupts.

`sltioctl` Function

Recall that a SLIP interface has two roles to play in the kernel:

- as a network interface, and
- as a TTY line discipline.

Figure 5.7 indicated that `slioctl` processes `ioctl` commands issued for a SLIP interface through a socket descriptor. In Section 4.4 we showed how `ifioctl` calls `slioctl`. We'll see a similar pattern for `ioctl` commands that we cover in later chapters.

Figure 5.7 also indicated that `sltioctl` processes `ioctl` commands issued for the TTY device associated with a SLIP network interface. The one command recognized by `sltioctl` is shown in Figure 5.24.

Command	Argument	Function	Description
`SLIOCGUNIT`	`int *`	`sltioctl`	return interface unit associated with the TTY device

Figure 5.24 `sltioctl` commands.

The `sltioctl` function is shown in Figure 5.25.

if_sl.c

```
236 int
237 sltioctl(tp, cmd, data, flag)
238 struct tty *tp;
239 int     cmd;
240 caddr_t data;
241 int     flag;
242 {
243     struct sl_softc *sc = (struct sl_softc *) tp->t_sc;

244     switch (cmd) {
245     case SLIOCGUNIT:
246         *(int *) data = sc->sc_if.if_unit;
247         break;

248     default:
249         return (-1);
250     }
251     return (0);
252 }
```

if_sl.c

Figure 5.25 `sltioctl` function.

236–252 The `t_sc` pointer in the `tty` structure points to the associated `sl_softc` structure. The unit number of the SLIP interface is copied from `if_unit` to `*data`, which is eventually returned to the process (Section 17.5).

`if_unit` is initialized by `slattach` when the system is initialized, and `t_sc` is initialized by `slopen` when the `slattach` program selects the SLIP line discipline for the TTY device. Since the mapping between a TTY device and a SLIP `sl_softc` structure is established at run time, a process can discover the interface structure selected by the `SLIOCGUNIT` command.

5.4 Loopback Interface

Any packets sent to the loopback interface (Figure 5.26) are immediately queued for input. The interface is implemented entirely in software.

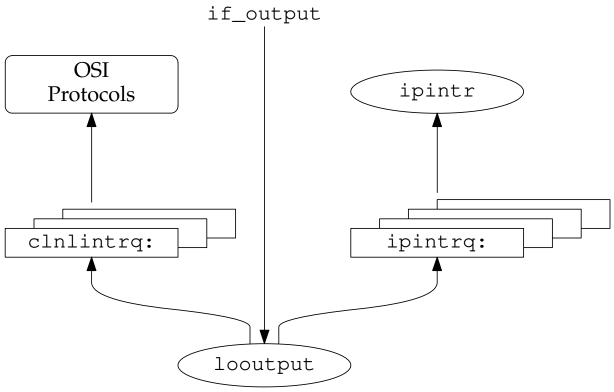

Figure 5.26 Loopback device driver.

`looutput`, the `if_output` function for the loopback interface, places outgoing packets on the input queue for the protocol specified by the packet's destination address.

We already saw that `ether_output` may call `looutput` to queue a copy of an outgoing broadcast packet when the device has set `IFF_SIMPLEX`. In Chapter 12, we'll see that multicast packets may be also be looped back in this way. `looutput` is shown in Figure 5.27.

if_loop.c

```
int
looutput(ifp, m, dst, rt)
struct ifnet *ifp;
struct mbuf *m;
struct sockaddr *dst;
struct rtentry *rt;
{
    int     s, isr;
    struct ifqueue *ifq = 0;

    if ((m->m_flags & M_PKTHDR) == 0)
        panic("looutput no HDR");
    ifp->if_lastchange = time;
    if (loif.if_bpf) {
        /*
         * We need to prepend the address family as
         * a four byte field.  Cons up a dummy header
```

```
         * to pacify bpf.  This is safe because bpf
         * will only read from the mbuf (i.e., it won't
         * try to free it or keep a pointer a to it).
         */
        struct mbuf m0;
        u_int   af = dst->sa_family;

        m0.m_next = m;
        m0.m_len = 4;
        m0.m_data = (char *) &af;

        bpf_mtap(loif.if_bpf, &m0);
    }
    m->m_pkthdr.rcvif = ifp;

    if (rt && rt->rt_flags & (RTF_REJECT | RTF_BLACKHOLE)) {
        m_freem(m);
        return (rt->rt_flags & RTF_BLACKHOLE ? 0 :
                rt->rt_flags & RTF_HOST ? EHOSTUNREACH : ENETUNREACH);
    }
    ifp->if_opackets++;
    ifp->if_obytes += m->m_pkthdr.len;
    switch (dst->sa_family) {
    case AF_INET:
        ifq = &ipintrq;
        isr = NETISR_IP;
        break;

    case AF_ISO:
        ifq = &clnlintrq;
        isr = NETISR_ISO;
        break;

    default:
        printf("lo%d: can't handle af%d\n", ifp->if_unit,
               dst->sa_family);
        m_freem(m);
        return (EAFNOSUPPORT);
    }
    s = splimp();
    if (IF_QFULL(ifq)) {
        IF_DROP(ifq);
        m_freem(m);
        splx(s);
        return (ENOBUFS);
    }
    IF_ENQUEUE(ifq, m);
    schednetisr(isr);
    ifp->if_ipackets++;
    ifp->if_ibytes += m->m_pkthdr.len;
    splx(s);
    return (0);
}
```

if_loop.c

Figure 5.27 The `looutput` function.

57–68 The arguments to `looutput` are the same as those to `ether_output` since both are called indirectly through the `if_output` pointer in their `ifnet` structures: `ifp`, a pointer to the outgoing interface's `ifnet` structure; `m`, the packet to send; `dst`, the destination address of the packet; and `rt`, routing information. If the first mbuf on the chain does not contain a packet, `looutput` calls `panic`.

Figure 5.28 shows the logical layout for a BPF loopback packet.

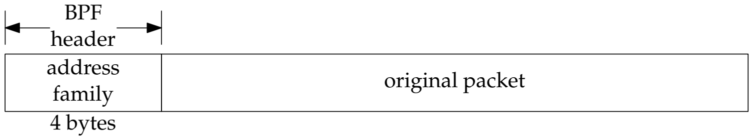

Figure 5.28 BPF loopback packet: logical format.

69–83 The driver constructs the BPF loopback packet in `m0` on the stack and connects `m0` to the mbuf chain containing the original packet. Note the unusual declaration of `m0`. It is an *mbuf*, not a pointer to an mbuf. `m_data` in `m0` points to `af`, which is also allocated on the stack. Figure 5.29 shows this arrangement.

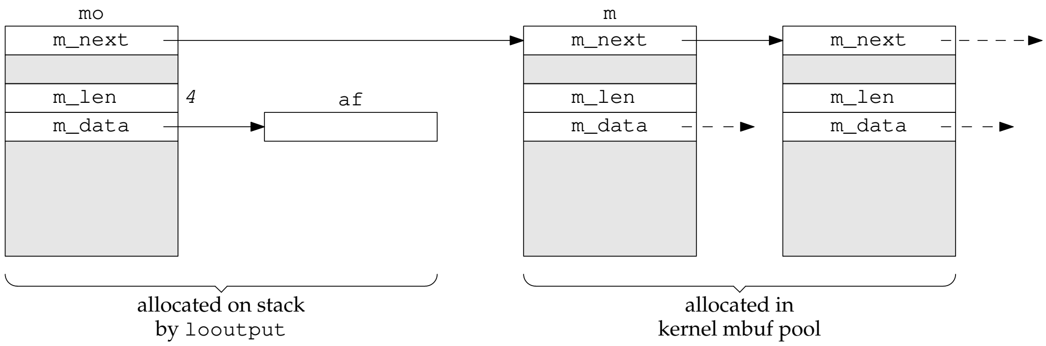

Figure 5.29 BPF loopback packet: mbuf format.

`looutput` copies the destination's address family into `af` and passes the new mbuf chain to `bpf_mtap`, which processes the packet. Contrast this to `bpf_tap`, which accepts the packet in a single contiguous buffer not in an mbuf chain. As the comment indicates, BPF never releases mbufs in a chain, so it is safe to pass `m0` (which points to an mbuf on the stack) to `bpf_mtap`.

84–89 The remainder of `looutput` contains *input* processing for the packet. Even though this is an output function, the packet is being looped back to appear as input. First, `m->m_pkthdr.rcvif` is set to point to the receiving interface. If the caller provided a routing entry, `looutput` checks to see if it indicates that the packet should be rejected (`RTF_REJECT`) or silently discarded (`RTF_BLACKHOLE`). A black hole is implemented by discarding the mbuf and returning 0. It appears to the caller as if the packet has been transmitted. To reject a packet, `looutput` returns `EHOSTUNREACH` if the route is for a host and `ENETUNREACH` if the route is for a network.

The various `RTF_`*xxx* flags are described in Figure 18.25.

90–120 `looutput` then selects the appropriate protocol input queue and software interrupt by examining `sa_family` in the packet's destination address. It then queues recognized packets and schedules a software interrupt with `schednetisr`.

5.5 Summary

We described the two remaining interfaces to which we refer throughout the text: `sl0`, a SLIP interface, and `lo0`, the standard loopback interface.

We showed the relationship between the SLIP interface and the SLIP line discipline, described the SLIP encapsulation method, and discussed TOS processing for interactive traffic and other performance considerations for the SLIP driver.

We showed how the loopback interface demultiplexes outgoing packets based on their destination address and places the packet on the appropriate input queue.

Exercises

5.1 Why does the loopback interface not have an input function?

5.2 Why do you think `mo` is allocated on the stack in Figure 5.27?

5.3 Perform an analysis of SLIP characteristics for a 19,200 bps serial line. Should the SLIP MTU be changed for this line?

5.4 Derive a formula to select a SLIP MTU based on the speed of the serial line.

5.5 What happens if a packet is too large to fit in SLIP's input buffer?

5.6 An earlier version of `slinput` did not set `SC_ERROR` when a packet overflowed the input buffer. How would the error be detected in this case?

5.7 In Figure 4.31 `le` is initialized by indexing the `le_softc` array with `ifp->if_unit`. Can you think of another method for initializing `le`?

5.8 How can a UDP application recognize when its packets are being discarded because of a bottleneck in the network?