Chapter 9

Writing Network Device Drivers

This chapter addresses questions particular to device drivers that run on networked workstations, and is based on the assumption that network device driver writers are familiar with BSD conventions. In particular, it describes how to write an IRIX kernel **ifnet** interface networking device driver. Only issues specific to IRIX are covered here; this section does not describe the complete **ifnet** programmatic interfaces to the system although the sources for a sample skeleton **ifnet** device driver are included at the end of this section.Refer to the following books for more complete information on the BSD kernel protocol stack and device driver conventions:

- Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX® Device Driver*. John Wiley & Sons, 1992.
- Hines, Robert M., and Spence Wilcox. *Device Driver Programming*, UNIX SVR4.2. Englewood Cliffs, New Jersey: UNIX Press, 1992.
- Leffler, Samuel J., et alia. *The Design and Implementation of the 4.3BSD UNIX® Operating System*. Palo Alto, California: Addison-Wesley Publishing Company, 1989.

This chapter contains the following sections:

Preliminary Discussion

This chapter deals with requirements that go beyond STREAMS, namely, how to allow a board to communicate directly with IRIX's native protocol stack.

It is recommended that device driver writers review Chapter 8, "Writing Multiprocessor Device Drivers" before writing a network driver.

IRIX 5.3 and IRIX 6.0, although divergent, accept device drivers that have run on IRIX 5.x.

Note: A forthcoming release, IRIX 6.x, will represent a convergence of the two operating systems (32- and 64-bit). It is expected to be easier, as well as more time-efficient, in many cases, to postpone the development of new network device drivers until IRIX 6.x becomes available.

Caution: Information in this chapter is subject to change without notice.

IRIX Kernel Networking Design

The IRIX kernel networking design is based on the kernel networking framework in 4.3BSD. If you are familiar with the 4.3BSD kernel networking design, then you are already familiar with the IRIX kernel networking design because they are basically the same.

The IRIX networking design is based on the socket interface: *mbufs* are used to exchange messages within the kernel, and device drivers support the TCP/IP internet protocol suite by supporting the *ifnet* interface.

Since the kernel BSD-based networking framework and TCP/IP internet protocol suite implementation have changed little from previous releases of IRIX, porting your *ifnet* device driver to IRIX 5.3 from earlier releases of IRIX should be simple and straightforward.

Figure 9-1 displays the basic IRIX kernel networking architecture.

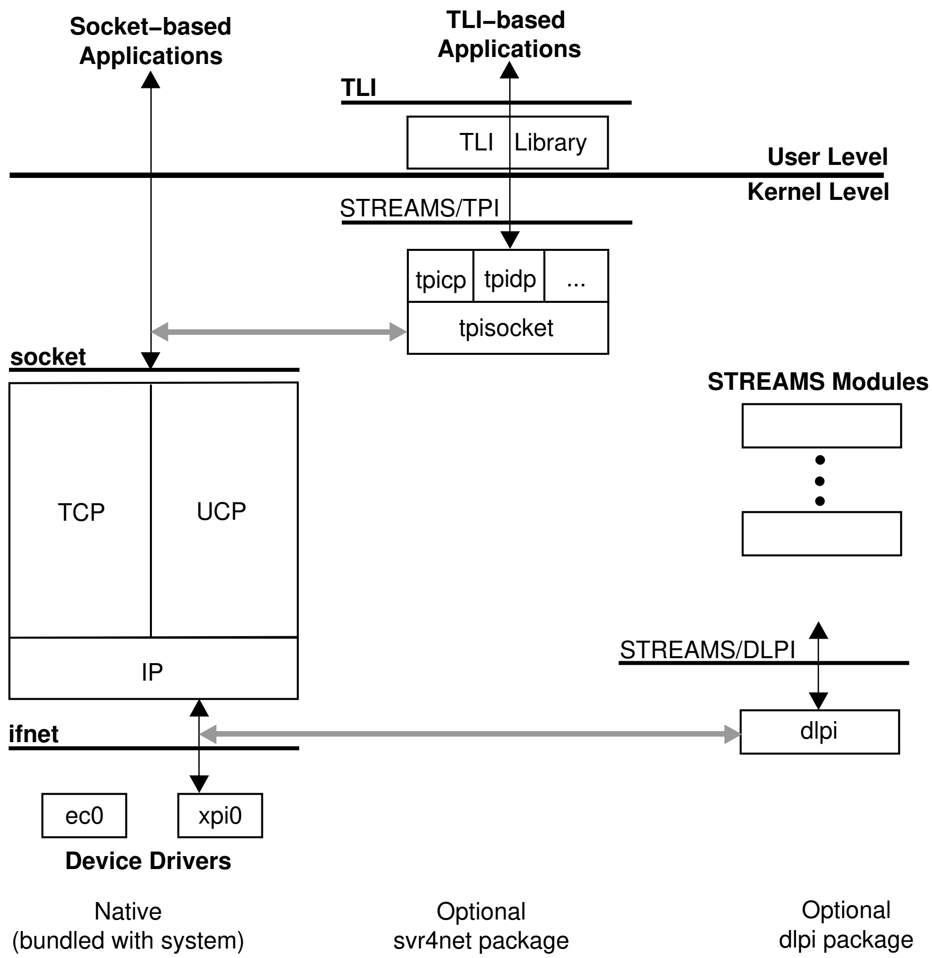

Figure 9-1 IRIX 5.3 Kernel Network Architecture

The left side of the figure shows the native socket-based TCP/IP protocol code, socket layer, and *ifnet*-based device drivers. This portion comes bundled in the basic IRIX system. Socket-based applications such as **rlogin**, **rcp**, NFS client and server, and the socket-based RPC library operate directly over this native networking framework.

The middle of the figure shows the optionally installed **svr4net** package, which provides compatibility support for user-level applications written to the STREAMS Transport Layer Interface (TLI). **tpisocket** is a kernel library module used by protocol-specific STREAMS pseudo-drivers, such as **tpitcp**, **tpiudp**, and so on, providing a TPI interface above the native kernel sockets-based network protocol stack.

The right side of the figure shows the optionally installed **dlpi** package which provides a STREAMS pseudo-driver that supports the Data Link Provider Interface (DLPI) for STREAMS-based kernel protocol stacks.

Refer to the *IRIX Network Programming Guide* and the SVR4 man pages for **STREAMS**, **TLI**, and **DLPI** programming information.

ifnet Driver Interfaces

The interface definitions and contents of the following #*include* files are subject to change without notice. While the policy is to avoid or minimize driver modifications required as new releases of IRIX become available, no guarantees of source or binary compatibility between releases of the operating system are made for networking drivers.

The primary *ifnet* data structure and routines to manipulate this are defined in *net/if.h*. They are augmented with interface types defined in *net/if_types.h*.

Functions and macros to allocate, manipulate, and free *mbufs* are defined in *sys/mbuf.h*.

The function **schednetisr** to schedule a kernel software interrupt routine, related macros, and a declaration for the IP input queue *ipintrq* are defined in *net/netisr.h*.

Constants and structures for support of the raw protocol family are defined in *net/raw.h.*

Routines defining a generic filter for use by network interfaces whose devices cannot perfectly filter multicast packets are declared in *net/multi.h.*

DLPI interface support routines and structure definitions are in *sys/dlsap_register.h.*

Socket interface **ioctl** definitions are in *net/soioctl.h.*

Ethernet and ARP-related data structures and function prototypes are provided in *netinet/if_ether.h*.

Multiprocessor Issues

Prior to IRIX 5.3, the kernel BSD framework code and TCP/IP protocol executed under a single kernel lock on multiprocessor systems making it a *single-threaded* implementation. In IRIX 5.3, the BSD framework and TCP/IP protocol suite have been *multi-threaded* to support symmetric multiprocessing by the addition of kernel locks protecting critical sections. It now supports multiple, concurrent threads of execution within the TCP/UDP/IP protocol suite, kernel socket layer, and bundled networking device drivers.

These changes are transparent to user-level programs, but, if you've written your own *ifnet*-based networking driver, it requires minor source-level changes in order to run in IRIX 5.3.

In a multi-threaded kernel, raising the processor interrupt level (IPL) by calling one of the **spl** routines, such as **splimp**() or **splnet**(), blocks interrupts from occurring on the local processor; it does not prevent interrupts from occurring on other processors in the system, nor does it prevent other processes on other processors from executing code in your critical section.

Under BSD networking, drivers interface with the protocol stacks by queueing the incoming packets on a per-protocol input queue. On multiprocessor systems, this protocol input queue must be protected by the locking macros defined in the file *net/if.h*.

All the locking macros that protect the input queue are assumed to be called at the proper processor masking level, **splimp**. All input queue locking macros also take an input parameter *ifq*, which is a pointer to the protocol input queue that must be defined as a `struct ifqueue`.

Compilation Flags for MP TCP/IP

For IRIX 5.3, the following flag must be defined in order to enable the macros necessary to run under multi-threaded TCP/IP:

```
-D_MP_NETLOCKS -DMP
```

IFNET_LOCK(*ifp, s*)

Network driver interrupt handlers should call this macro to protect critical data structures against system calls that try to access the same data structures on other processors. This macro acquires the lock that is part of the driver **ifnet** structure pointed to by *ifp*. *s* is the return value of the processor interrupt mask. The lock is held at **splimp** level. The multiprocessor TCP/IP locking scheme in IRIX 5.3 automatically holds this lock when the system enters the driver via a system call. The lock is be released automatically upon returning from the driver to synchronize simultaneous accesses to the driver from multiple processors.

IFNET_UNLOCK(*ifp*,*s*)

This macro is the reverse of **IFNET_LOCK**, **IFNET_UNLOCK** is used to release the lock. *s* is the value previously returned by **IFNET_LOCK**().

IFNET_LOCKNOSPL(*ifp*)

This macro is similar to **IFNET_LOCK** but assumes that **splimp** has been called previously.

IFNET_UNLOCKNOSPL(*ifp*)

This macro is similar to **IFNET_UNLOCK** but does not lower the **spl**.

IFQ_LOCK/UNLOCK(*ifq*)

This macro acquires/releases the lock, which is part of the input queue.

IF_ENQUEUE(*ifq*, *m***)**

This macro acquires the lock on *ifq* and appends the packet pointed to by the mbuf *m*. The lock is released upon return.

Input Queueing Example

This is a code fragment of an interrupt handler that queues an input packet pointed to by *m* onto the **ip** input queue. **schednetisr**() is called to schedule processing of that packet.

The code assumed to be already at **splimp**().

```
{
    ...

      ifq = &ipintrq; /* the ip protocol queue */

 /*
 * If queue is full, we drop the packet.
 */
 IFQ_LOCK(ifq);
 if (IF_QFULL(ifq)) {
 m_freem(m);
 IF_DROP(ifq);
 IFQ_UNLOCK(ifq);
 return(-1);
 }

 IF_ENQUEUE_NOLOCK(ifq, m);
 schednetisr(NETISR_IP); /* schedule ip interrupt */
 IFQ_UNLOCK(ifq);
 return(0);
}
```

Interrupt Handler Example

The following is an example of an Ethernet interrupt handler.

```
/*
 * Ethernet interface interrupt.
 */
if_etintr(int unit)
{
 ETIO io;
 struct et_info *ei;
 register int s = splimp(); /* get the spin lock */

 ASSERT(unit == 0);
 ei = &et_info;
 io = ei->ei_io;

 if (io == 0) { /* ignore early interrupts */
     printf("et0: early interrupt\n");
     splx(s);
     return 1;
 }
 IFNET_LOCKNOSPL(&ei->ei_if);
 et_poll(ei);
 IFNET_UNLOCKNOSPL(&ei->ei_if);
 splx(s);
}
```

ifnet Device Driver Example

This is a skeleton *ifnet* driver for IRIX 5.3 meant to demonstrate *ifnet* driver entry points, data structures, required **ioctls**, address format conventions, kernel utility routines, and locking primitives.

Note: These kernel data structures and routines are subject to change without notice. "XXX" is used to designate places where device-specific, bus-specific, or driver-specific code sections are required.

```
/*
 * Locking strategy:
 * IFNET_LOCK() and IFNET_UNLOCK() acquire/release the
 * lock on a given ifnet structure. IFQ_LOCK() and
 * IFQ_UNLOCK() acquire/release the lock on a given ifqueue
 * structure. The ifnet or ifqueue lock must be held while
 * modifying any fields within the associated data
 * structure. The ifnet lock is also held to singlethread
 * portions of the device driver. The driver xxinit,
 * xxreset, xxoutput, xxwatchdog, and xxioctl entry points
 * are called with IFNET_LOCK() already acquired thus only
 * a single thread of execution is allowed in these
 * portions of the driver for each interface. It is the
 * driver's responsibility to call IFNET_LOCK() within its
 * xxintr() and other private routines to singlethread any
 * other critical sections.  It is also the driver's
 * responsibility to acquire the ifq lock by calling
 * IFQ_LOCK() before attempting to enqueue onto the IP
 * input queue "ipintrq".
 *
 * Notes:
 * - don't forget appropriate machine-specific cache flushing operations
 *    (refer to IRIX Device Driver Programming guide)
 * - declare pointers to device registers as "volatile"
 * - compile on multiprocessor systems with "-D_MP_NETLOCKS -DMP"
 *
 * Caveat Emptor:
 * No guarantees are made wrt correctness nor completeness
 * of this source.
 *
 * Copyright 1994 Silicon Graphics, Inc.  All rights reserved.
 */
#ident "$Revision: 1.0$"
```

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/sysmacros.h>
#include <sys/cmn_err.h>
#include <sys/debug.h>
#include <sys/edt.h>
#include <sys/errno.h>
#include <sys/tcp-param.h>
#include <sys/mbuf.h>
#include <sys/immu.h>
#include <sys/sbd.h>
#include <sys/ddi.h>
#include <sys/cpu.h>
#include <sys/invent.h>
#include <net/if.h>
#include <net/if_types.h>
#include <net/netisr.h>
#include <netinet/if_ether.h>
#include <net/raw.h>
#include <net/multi.h>
#include <netinet/in_var.h>
#include <net/soioctl.h>
#include <sys/dlsap_register.h>
XXX

/*
 * driver-specific and device-specific data structure
 * declarations and definitions might go here.
 */

#define    SK_MAX_UNITS    8
#define    SK_MTU        4096
#define    SK_DOG        (2*IFNET_SLOWHZ) /* watchdog duration in seconds */
#define    SK_IFT        (IFT_FDDI)    /* refer to <net/if_types.h> */
#define    SK_INV        (INV_NET_FDDI)    /* refer to <sys/invent.h> */

#define    INV_FDDI_SK    (23)        /* refer to <sys/invent.h> */

#define    IFF_ALIVE        (IFF_UP|IFF_RUNNING)
#define    iff_alive(flags)    (((flags) & IFF_ALIVE) == IFF_ALIVE)
#define iff_dead(flags)        (((flags) & IFF_ALIVE) != IFF_ALIVE)

#define    SK_ISBROAD(addr)    (!bcmp((addr), &skbroadcastaddr, SKADDRLEN))
#define    SK_ISGROUP(addr)    ((addr)[0] & 01)
```

```
/*
 * XXX media-specific definitions of address size and header format.
 */

#define    SKADDRLEN    (6)
#define    SKHEADERLEN    (sizeof (struct skheader))

/*
 * Our fictional media has an IEEE 802-looking header..
 */
struct skaddr {
    u_int8_t sk_vec[SKADDRLEN];
};
struct skheader {
    struct skaddr sh_dhost;
    struct skaddr sh_shost;
    u_int16_t sh_type;
};
struct skaddr skbroadcastaddr = {
    0xff, 0xff, 0xff, 0xff, 0xff, 0xff
};

/*
 * Each interface is represented by a private
 * network interface data structure that maintains
 * the device hardware resource addresses, pointers
 * to device registers, allocated dma_alloc maps,
 * lists of mbufs pending transmit or reception, etc, etc.
 * XXX We use ARP and have an 802 address.
 */
struct sk_info {
    struct arpcom si_ac;        /* common ifnet and arp */
    struct skaddr si_ouraddr;    /* our individual media address */
    struct mfilter si_filter;    /* AF_RAW sw snoop filter */
    struct rawif si_rawif;        /* raw snoop interface */
    int si_unit;
    int si_flags;
    int si_initdone;
    XXX
};
struct sk_info sk_info[SK_MAX_UNITS];

#define    si_if    si_ac.ac_if
XXX
```

```
#define    sktoifp(si) (&(si)->si_ac.ac_if)
#define ifptosk(ifp)((struct sk_info *)ifp)

#define    WORDALIGNED(p)    (p & (sizeof(int)-1) == 0)

/*
 * The start of an mbuf containing an input frame
 */
struct sk_ibuf {
    struct ifheader sib_ifh;
    struct snoopheader sib_snoop;
    struct skheader sib_skh;
};
#define    SK_IBUFSZ    (sizeof (struct sk_ibuf))

/*
 * Multicast filter request for SIOCADDMULTI/SIOCDELMULTI .
 */
struct mfreq {
    union mkey *mfr_key;    /* pointer to socket ioctl arg */
    mval_t    mfr_value;    /* associated value */
};

static void skedtinit(struct edt *e);
static int sk_init(int unit);
static void sk_reset(struct sk_info *si);
static void sk_intr(int unit);
static int sk_output(struct ifnet *ifp, struct mbuf *m, struct sockaddr *dst);
static void sk_input(struct sk_info *si, struct mbuf *m, int totlen);
static int sk_ioctl(struct ifnet *ifp, int cmd, void *data);
static void sk_watchdog(int unit);
static void sk_stop(struct sk_info *si);
static int sk_start(struct sk_info *si, int flags);
static int sk_add_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_del_da(struct sk_info *si, union mkey *key, int ismulti);
static int sk_dahash(char *addr);
static int sk_dlp(struct sk_info *si, int port, int encap, struct mbuf *m, int
len);
XXX

extern struct ifqueue ipintrq;    /* ip input queue */
extern struct ifnet loif;    /* loopback driver if */

/*
```

```
 * EDT initialization routine.
 */
static void
skedtinit(struct edt *e)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int    unit;
    XXX

    /*
     * Refer to writing xxedtinit() routine descriptions
     * in VME/GIO sections of the Device Driver Programming
     * guide for:
     *
     * - probing the device
     * - configuring the slot config register
     * - registering our interrupt handler
     */
    XXX

    /*
     * Driver-specific actions that might go here:
     *
     * - allocate an unused unit number and initialize
     *   that sk_info structure.
     * - call sk_reset to disable the device
     * - allocate shared host/device memory
     * - allocating VME dma mapping registers
     * -
     */
    XXX

    if (showconfig)
        printf("sk%d: hardware MAC address %s\n",
            si->si_unit,
            sk_sprintf(si->si_ouraddr));

    /*
     * XXX your address translation protocol goes here.
     * Save a copy of our MAC address in the arpcom structure.
     */
    bcopy((caddr_t)&si->si_ouraddr, (caddr_t)si->si_ac.ac_enaddr,
        SKADDRLEN);
```

```
/*
 * Initialize ifnet structure with our name, type, mtu size,
 * supported flags, pointers to our entry points,
 * and attach to the available ifnet drivers list.
 */
ifp = sktoifp(si);
ifp->if_name = "sk";
ifp->if_unit = unit;
ifp->if_type = SK_IFT;
ifp->if_mtu = SK_MTU;
ifp->if_flags = IFF_BROADCAST | IFF_MULTICAST | IFF_NOTRAILERS;
ifp->if_init = (int (*)(int))sk_init;
ifp->if_output = sk_output;
ifp->if_ioctl = (int (*)(struct ifnet*, int, void*))sk_ioctl;
ifp->if_watchdog = sk_watchdog;
if_attach(ifp);

/*
 * Allocate a multicast filter table with an initial
 * size of 10.  See <net/multi.h> for a description
 * of the support for generic sw multicast filtering.
 * Use of these mf routines is purely optional -
 * if you're not supporting multicast addresses or
 * your device does perfect filtering or you think
 * you can roll your own better, feel free.
 */
if (!mfnew(&si->si_filter, 10))
    cmn_err(CE_PANIC, "sk_edtinit: no memory for frame filter\n");

/*
 * Initialize the raw socket interface.  See <net/raw.h>
 * and the man pages for descriptions of the SNOOP
 * and DRAIN raw protocols.
 */
rawif_attach(&si->si_rawif, &si->si_if,
    (caddr_t) &si->si_ouraddr,
    (caddr_t) &skbroadcastaddr,
    SKADDRLEN,
    SKHEADERLEN,
    structoff(skheader, sh_shost),
    structoff(skheader, sh_dhost));

/*
 * for hinv
 */
```

```
    add_to_inventory(INV_NETWORK, SK_INV, INV_FDDI_SK, unit, 0);
}

static int
sk_init(int unit)
{
    struct sk_info *si;
    struct    ifnet *ifp;
    XXX

    si = &sk_info[unit];
    ifp = sktoifp(si);

    ASSERT(IFNET_ISLOCKED(ifp));

    /*
     * Reset the device first, ask questions later..
     */
    sk_reset(si);

    /*
     * - free or reuse any pending xmit/recv mbufs
     * - initialize device configuration registers, etc.
     * - allocate and post receive buffers
     *
     * Refer to Device Driver Programming guide for
     * descriptions on use of kvtophys() (GIO) or
     * dma_map/dma_mapaddr() (VME) routines for
     * obtaining DMA addresses and system-specific
     * issues like flushing caches or write buffers.
     */

    /*
     * enable if_flags device behavior (IFF_DEBUG on/off, etc.)
     */
    XXX

    ifp->if_timer = SK_DOG;    /* turn on watchdog */

    /* turn device "on" now */
    XXX

    return 0;
}
```

```
/*
 * Reset the interface.
 */
static void
sk_reset(struct sk_info *si)
{
    struct ifnet *ifp = sktoifp(si);

    ifp->if_timer = 0;    /* turn off watchdog */

    /*
     * - reset device
     * - reset device receive descriptor ring
     * - free any enqueued transmit mbufs
     * - create device xmit descriptor ring
     */
}

static void
sk_intr(int unit)
{
    register struct sk_info *si;
    struct ifnet *ifp;
    struct mbuf *m;
    struct    ifqueue *ifq;
    int totlen;
    int s;
    int error;
    int port;

    si = &sk_info[unit];
    ifp = sktoifp(si);

    /*
     * Ignore early interrupts.
     */
    if ((si->si_initdone == 0) || iff_dead(ifp->if_flags)) {
        sk_stop(si);
        return;
    }

    IFNET_LOCK(ifp, s);    /* acquire interface lock */

    /*
     * disable device and return if early interrupt
```

```
     */
    XXX

    /*
     * test and clear device interrupt pending register.
     */
    XXX

    /*
     * process any received packets.
     */
    while (/* XXX received packets available */) {

        /*
         * Do device-specific receive processing here.
         * Allocate and post a replacement receive buffer.
         */
        XXX

        sk_input(si, m, totlen);
    }

    while (/* XXX mbufs completed transmission */) {

        /*
         * Reclaim any completed device transmit resources
         * freeing completed mbufs, checking for errors,
         * and maintaining if_opackets, if_oerrors,
         * if_collisions, etc.
         */
        XXX
    }

    IFNET_UNLOCK(ifp, s);
}

/*
 * Transmit packet.  If the destination is this system or
 * broadcast, send the packet to the loop-back device if
 * we cannot hear ourself transmit.  Return 0 or errno.
 */
static int
sk_output(
    struct ifnet    *ifp,
    struct mbuf *m0,
```

```
    struct sockaddr *dst)
{
    struct    sk_info    *si = ifptosk(ifp);
    struct skaddr *sdst, *ssrc;
    struct skheader *sh;
    struct mbuf *m, *m1, *m2;
    struct mbuf *mloop;
    int error;
    u_int16_t type;
    XXX

    ASSERT(IFNET_ISLOCKED(ifp));

    mloop = NULL;

    if (iff_dead(ifp->if_flags)) {
        error = EHOSTDOWN;
        goto bad;
    }

    /*
     * If snd queue full, try reclaiming some completed
     * mbufs.  If it's still full, then just drop the
     * packet and return ENOBUFS.
     */
    if (IF_QFULL(&si->si_if.if_snd)) {
        while (/* XXX xmits done */) {
            /*
             * Reclaim completed xmit descriptors.
             */
            XXX

            IF_DEQUEUE_NOLOCK(&si->si_if.if_snd, m);
            m_freem(m);
        }
        if (IF_QFULL(&si->si_if.if_snd)) {
            m_freem(m0);
            si->si_if.if_odrops++;
            IF_DROP(&si->si_if.if_snd);
            return (ENOBUFS);
        }
    }

    switch (dst->sa_family) {
    case AF_INET: {
```

```
/*
 * Get room for media header,
 * use this mbuf if possible.
 */
if (!M_HASCL(m0)
    && m0->m_off >= MMINOFF+sizeof(*sh)
    && (sh = mtod(m0, struct skheader*))
    && WORDALIGNED((u_long)sh)) {
    ASSERT(m0->m_off <= MSIZE);
    m1 = 0;
    --sh;
} else {
    m1 = m_get(M_DONTWAIT, MT_DATA);
    if (m1 == NULL) {
        m_freem(m0);
        si->si_if.if_odrops++;
        IF_DROP(&si->si_if.if_snd);
        return (ENOBUFS);
    }
    sh = mtod(m1, struct skheader*);
    m1->m_len = sizeof (*sh);
}

bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);

/*
 * translate dst IP address to media address.
 */
if (!ip_arpresolve(&si->si_ac, m0,
    &((struct sockaddr_in *)dst)->sin_addr,
     (u_char*)&sh->sh_dhost)) {
    m_freem(m1);
    return (0);    /* just wait if not yet resolved */
}

if (m1 == 0) {
    m0->m_off -= sizeof (*sh);
    m0->m_len += sizeof (*sh);
} else {
    m1->m_next = m0;
    m0 = m1;
}

/*
 * Listen to ourself, if we are supposed to.
```

```
         */
        if (SK_ISBROAD(&sh->sh_shost)) {
            mloop = m_copy(m0, sizeof (*sh), M_COPYALL);
            if (mloop == NULL) {
                m_freem(m0);
                si->si_if.if_odrops++;
                IF_DROP(&si->si_if.if_snd);
                return (ENOBUFS);
            }
        }
        break;
    }

    case AF_UNSPEC:
#define    EP    ((struct ether_header *)&dst->sa_data[0])
        /*
         * Translate an ARP packet using RFC-1042.
         * Require the entire ARP packet be in the first mbuf.
         */
        sh = mtod(m0, struct skheader*);
        if (M_HASCL(m0)
            || !WORDALIGNED((u_long)sh)
            || m0->m_len < sizeof(struct ether_arp)
            || m0->m_off < MMINOFF+sizeof(*sh)
            || EP->ether_type != ETHERTYPE_ARP) {
            printf("sk_output: bad ARP output\n");
            m_freem(m0);
            si->si_if.if_oerrors++;
            IF_DROP(&si->si_if.if_snd);
            return (EAFNOSUPPORT);
        }
        ASSERT(m0->m_off <= MSIZE);
        m0->m_len += sizeof(*sh);
        m0->m_off -= sizeof(*sh);
        --sh;

        bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
        bcopy(&EP->ether_dhost[0], &sh->sh_dhost, SKADDRLEN);

        sh->sh_type = EP->ether_type;
# undef EP
        break;

    case AF_RAW:
        /* The mbuf chain contains the raw frame incl header.
```

```
         */
        sh = mtod(m0, struct skheader*);
        if (M_HASCL(m0)
            || m0->m_len < sizeof(*sh)
            || !WORDALIGNED((u_long)sh)) {
            m0 = m_pullup(m0, SKHEADERLEN);
            if (m0 == NULL) {
                si->si_if.if_odrops++;
                IF_DROP(&si->si_if.if_snd);
                return (ENOBUFS);
            };
            sh = mtod(m0, struct skheader*);
        }
        break;

    case AF_SDL:
#define    SCKTP    ((struct sockaddr_sdl *)dst)
        /*
         * Send an 802 packet for DLPI.
         * mbuf chain should already have everything
         * but MAC header.
         */

        /* sanity check the MAC address */
        if (SCKTP->ssdl_addr_len != SKADDRLEN) {
            m_freem(m0);
            return (EAFNOSUPPORT);
        }

        sh = mtod(m0, struct skheader*);
        if (!M_HASCL(m0)
            && m1->m_off >= MMINOFF+SCKTP_HLEN
            && WORDALIGNED(sh)) {
            ASSERT(m0->m_off <= MSIZE);
            m0->m_len += SCKTP_HLEN;
            m0->m_off -= SCKTP_HLEN;
        } else {
            m1 = m_get(M_DONTWAIT,MT_DATA);
            if (!m1) {
                m_freem(m0);
                si->si_if.if_odrops++;
                IF_DROP(&si->si_if.if_snd);
                return (ENOBUFS);
            }
            m1->m_len = SCKTP_HLEN;
```

```
                m1->m_next = m0;
                m0 = m1;
                sh = mtod(m0, struct skheader*);
            }
            sh->sh_type = htons(ETHERTYPE_IP);
            bcopy(&si->si_ouraddr, &sh->sh_shost, SKADDRLEN);
            bcopy(SCKTP->ssdl_addr, &sh->sh_dhost, SKADDRLEN);
            break;
# undef SCKTP

        default:
            printf("sk_output:  bad af %u\n", dst->sa_family);
            m_freem(m0);
            return (EAFNOSUPPORT);
        }

        /*
         * Check whether snoopers want to copy this packet.
         */
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif, (caddr_t)sh, m0->m_len)) {
            struct mbuf *ms, *mt;
            int len;         /* m0 bytes to copy */
            int lenoff;
            int curlen;

            len = m_length(m0);
            lenoff = 0;
            curlen = len + SK_IBUFSZ;
            if (curlen > MCLBYTES)
                curlen = MCLBYTES;
            ms = m_vget(M_DONTWAIT, MAX(curlen, SK_IBUFSZ), MT_DATA);
            if (ms) {
                IF_INITHEADER(mtod(ms,caddr_t), &si->si_if, SK_IBUFSZ);
                curlen = m_datacopy(m0, lenoff, curlen - SK_IBUFSZ,
                    mtod(ms,caddr_t) + SK_IBUFSZ);
                mt = ms;
                for (;;) {
                    lenoff += curlen;
                    len -= curlen;
                    if (len <= 0)
                        break;
                    curlen = MIN(len, MCLBYTES);
                    m1 = m_vget(M_DONTWAIT, curlen, MT_DATA);
                    if (0 == m1) {
```

```
                m_freem(ms);
                ms = 0;
                break;
            }
            mt->m_next = m1;
            mt = m1;
            curlen = m_datacopy(m0, lenoff, curlen,
                        mtod(m1, caddr_t));
        }
    }
    if (ms == NULL) {
        snoop_drop(&si->si_rawif, SN_PROMISC,
               mtod(m0,caddr_t), m0->m_len);
    } else {
        (void)snoop_input(&si->si_rawif, SN_PROMISC,
                 mtod(m0, caddr_t),
                 ms,
                 (lenoff > SKHEADERLEN)?
                 (lenoff - SKHEADERLEN) : 0);
    }
}

/*
 * Save a copy of the mbuf chain to free later.
 */
IF_ENQUEUE_NOLOCK(&si->si_if.if_snd, m0);

/*
 * Start DMA on the msg.
 * - allocate device-specific xmit resources  (need max
 *   of twice the number of mbufs in the mbuf chain
 *   if we're using physical memory addresses for
 *   GIO assuming worst case that each mbuf crosses
 *   a page boundary.
 */
XXX

if (error)
    goto bad;

ifp->if_opackets++;

if (mloop) {
    si->si_if.if_omcasts++;
    (void) looutput(&loif, mloop, dst);
```

```
    } else if (SK_ISGROUP(sh->sh_dhost.sk_vec))
        si->si_if.if_omcasts++;

    return (0);

bad:
    ifp->if_oerrors++;
    m_freem(m);
    m_freem(mloop);
    return (error);
}

/*
 * deal with a complete input frame in a string of mbufs.
 * mbuf points at a (struct sk_ibuf), totlen is #bytes
 * in user data portion of the mbuf.
 */
static void
sk_input(struct sk_info *si,
    struct mbuf *m,
    int totlen)
{
    struct sk_ibuf *sib;
    struct ifqueue *ifq;
    int snoopflags = 0;
    uint port;

    /*
     * set 'snoopflags' and 'if_ierrors' as appropriate
     */
    XXX

    ifq = NULL;
    sib = mtod(m, struct sk_ibuf*);
    IF_INITHEADER(sib, &si->si_if, SK_IBUFSZ);

    si->si_if.if_ibytes += totlen;
    si->si_if.if_ipackets++;

    /*
     * If it is a broadcast or multicast frame,
     * get rid of imperfectly filtered multicasts.
     */
    if (SK_ISGROUP(sib->sib_skh.sh_dhost.sk_vec)) {
        if (SK_ISBROAD(sib->sib_skh.sh_dhost.sk_vec))
```

```
            m->m_flags |= M_BCAST;
        else {
            if (((si->si_ac.ac_if.if_flags & IFF_ALLMULTI) == 0)
            && !mfethermatch(&si->si_filter,
                sib->sib_skh.sh_dhost.sk_vec, 0)) {
                if (RAWIF_SNOOPING(&si->si_rawif)
                && snoop_match(&si->si_rawif,
                    (caddr_t) &sib->sib_skh, totlen))
                    snoopflags = SN_PROMISC;
                else {
                    m_freem(m);
                    return;
                }
                m->m_flags |= M_MCAST;
            }
        }
        si->si_if.if_imcasts++;
    } else {
        if (RAWIF_SNOOPING(&si->si_rawif)
            && snoop_match(&si->si_rawif,
                (caddr_t) &sib->sib_skh,
                totlen))
            snoopflags = SN_PROMISC;
        else {
            m_freem(m);
            return;
        }
    }

    /*
     *  Set 'port' .  For us, just sh_type.
     */
    port = ntohs(sib->sib_skh.sh_type);

    /*
     * do raw snooping.
     */
    if (RAWIF_SNOOPING(&si->si_rawif)) {
        if (!snoop_input(&si->si_rawif, snoopflags,
                (caddr_t)&sib->sib_skh,
                m,
                (totlen>sizeof(struct skheader)
                 ? totlen-sizeof(struct skheader) : 0))) {
        }
        if (snoopflags)
```

```
        return;

} else if (snoopflags) {
    goto drop;    /* if bad, count and skip it */
}

/*
 * If it is a frame we understand, then give it to the
 * correct protocol code.
 */
switch (port) {
case ETHERTYPE_IP:
    ifq = &ipintrq;
    break;

case ETHERTYPE_ARP:
    arpinput(&si->si_ac, m);
    return;

default:
    if (sk_dlp(si, port, DL_ETHER_ENCAP, m, totlen))
        return;
    break;
}

/*
 * if we cannot find a protocol queue, then flush it down the
 * drain, if it is open.
 */
if (ifq == NULL) {
    if (RAWIF_DRAINING(&si->si_rawif)) {
        drain_input(&si->si_rawif,
                port,
                (caddr_t)&sib->sib_skh.sh_dhost.sk_vec,
                m);
    } else
        m_freem(m);
    return;
}

/*
 * Put it on the IP protocol queue.
 */
if (IF_QFULL(ifq)) {
    si->si_if.if_iqdrops++;
```

```
        si->si_if.if_ierrors++;
        IF_DROP(ifq);
        goto drop;
    }
    IF_ENQUEUE(ifq, m);
    schednetisr(NETISR_IP);
    return;

drop:
    m_freem(m);
    if (RAWIF_SNOOPING(&si->si_rawif))
        snoop_drop(&si->si_rawif, snoopflags,
               (caddr_t)&sib->sib_skh, totlen);
    if (RAWIF_DRAINING(&si->si_rawif))
        drain_drop(&si->si_rawif, port);

}

/*
 * See if a DLPI function wants a frame.
 */
static int
sk_dlp(struct sk_info *si,
    int port,
    int encap,
    struct mbuf *m,
    int len)
{
    dlsap_family_t *dlp;
    struct mbuf *m2;
    struct sk_ibuf *sib;

    if ((dlp = dlsap_find(port, encap)) == NULL)
        return (0);

    /*
     * The DLPI code wants the entire MAC and LLC headers.
     * It needs the total length of the mbuf chain to reflect
     * the actual data length, not to be extended to contain
     * a fake, zeroed LLC header which keeps the snoop code from
     * crashing.
     */
    if ((m2 = m_copy(m, 0, len+sizeof(struct skheader))) == NULL)
        return (0);
```

```
    if (M_HASCL(m2)) {
        m2 = m_pullup(m2, SK_IBUFSZ);
        if (m2 == NULL)
            return (0);
    }
    sib = mtod(m2, struct sk_ibuf*);

    /*
     * The DLPI code wants the MAC address in canonical bit order.
     * Convert here if necessary.
     */
    XXX

    /*
     * The DLPI code wants the LLC header, if present,
     * not to be hidden with the MAC header.  Decrement
     * LLC header size from ifh_hdrlen if necessary.
     */
    XXX

    if ((*dlp->dl_infunc)(dlp, &si->si_if, m2, &sib->sib_skh)) {
        m_freem(m);
        return (1);
    }
    m_freem(m2);
    return (0);
}

/*
 * Process an ioctl request.
 * Return 0 or errno.
 */
static int
sk_ioctl(
    struct ifnet *ifp,
    int cmd,
    void *data)
{
    struct sk_info *si;
    int error = 0;
    int flags;
    XXX

    ASSERT(IFNET_ISLOCKED(ifp));
```

```
si = ifptosk(ifp);

switch (cmd) {
case SIOCSIFADDR:
{
    struct ifaddr *ifa = (struct ifaddr *)data;

    switch (ifa->ifa_addr.sa_family) {
    case AF_INET:
        sk_stop(si);
        si->si_ac.ac_ipaddr = IA_SIN(ifa)->sin_addr;
        sk_start(si, ifp->if_flags);
        break;

    case AF_RAW:
        /*
         * Not safe to change addr while the
         * board is alive.
         */
        if (!iff_dead(ifp->if_flags))
            error = EINVAL;
        else {
            bcopy(ifa->ifa_addr.sa_data,
                si->si_ac.ac_enaddr, SKADDRLEN);
            error = sk_start(si, ifp->if_flags);
        }
        break;

    default:
        error = EINVAL;
        break;
    }
    break;
}

case SIOCSIFFLAGS:
{
    flags = ((struct ifreq *)data)->ifr_flags;

    if (((struct ifreq*)data)->ifr_flags & IFF_UP)
        error = sk_start(si, flags);
    else
        sk_stop(si);
    break;
}
```

```
    case SIOCADDMULTI:
    case SIOCDELMULTI:
    {
#define MKEY ((union mkey*)data)
        int allmulti;

        /*
         * Convert an internet multicast socket address
         * into an 802-type address.
         */
        error = ether_cvtmulti((struct sockaddr *)
            data, &allmulti);
        if (0 == error) {
            if (allmulti) {
                if (SIOCADDMULTI == cmd)
                    si->si_if.if_flags |= IFF_ALLMULTI;
                else
                    si->si_if.if_flags &= ~IFF_ALLMULTI;
                /* XXX enable hw all multicast addrs */
                XXX
            } else {
                bitswapcopy(MKEY->mk_dhost, MKEY->mk_dhost,
                    sizeof (MKEY->mk_dhost));
                if (SIOCADDMULTI == cmd)
                    error = sk_add_da(si, MKEY, 1);
                else
                    error = sk_del_da(si, MKEY, 1);
            }
        }
        break;
#undef MKEY
    }

    case SIOCADDSNOOP:
    case SIOCDELSNOOP:
    {
#define SF(nm) ((struct skheader*)&(((struct snoopfilter *)data)->nm))
        /*
         * raw protocol snoop filter.  See <net/raw.h>
         * and <net/multi.h> and the snoop(7P) man page.
         */
        u_char *a;
        union mkey key;
```

```
        a = &SF(sf_mask[0])->sh_dhost.sk_vec[0];
        if (!SK_ISBROAD(a)) {
            /*
             * cannot filter on device unless mask is trivial.
             */
            error = EINVAL;
        } else {
            /*
             * Filter individual destination addresses.
             * Use a different address family to avoid
             * damaging an ordinary multi-cast filter.
             * XXX You'll have to invent your own
             * mulicast filter routines if this doesn't
             * fit your address size or needs.
             */
            a = &SF(sf_match[0])->sh_dhost.sk_vec[0];
            key.mk_family = AF_RAW;
            bcopy(a, key.mk_dhost, sizeof (key.mk_dhost));

            if (cmd == SIOCADDSNOOP)
                error = sk_add_da(si, &key, SK_ISGROUP(a));
            else
                error = sk_del_da(si, &key, SK_ISGROUP(a));
        }
        break;
    }

    /*
     * XXX add any driver-specific ioctls here.
     */

    default:
        error = EINVAL;
    }

    return (error);
}

/*
 * Add a destination address.
 * Add address to the sw multicast filter table and to
 * our hw device address (if applicable).
 */
static int
sk_add_da(
```

```
    struct sk_info *si,
    union mkey *key,
    int ismulti)
{
    struct mfreq mfr;

    /*
     * mfmatchcnt() looks up key in our multicast filter
     * and, if found, just increments its refcnt and
     * returns true.
     */
    if (mfmatchcnt(&si->si_filter, 1, key, 0))
        return (0);

    mfr.mfr_key = key;
    mfr.mfr_value = (mval_t) sk_dahash(key->mk_dhost);
    if (!mfadd(&si->si_filter, key, mfr.mfr_value))
        return (ENOMEM);

    /* poke this hash into device's hw address filter */
    XXX

    return (0);
}

/*
 * Delete an address filter. If key is unassociated, do nothing.
 * Otherwise delete software filter first, then hardware filter.
 */
sk_del_da(
    struct sk_info *si,
    union mkey *key,
    int ismulti)
{
    struct mfreq mfr;

    /*
     * Decrement refcnt of this address in our multicast filter
     * and reclaim the entry if refcnt == 0.
     */
    if (mfmatchcnt(&si->si_filter, -1, key, &mfr.mfr_value))
        return (0);
    mfdel(&si->si_filter, key);

    /* disable this hash value from the device if necessary */
```

```
    XXX

    return (0);
}

/*
 * compute a hash value for destination addr
 */
static int
sk_dahash(char *addr)
{
    int    hv;

    hv = addr[0] ^ addr[1] ^ addr[2] ^ addr[3] ^ addr[4] ^ addr[5];
    return (hv & 0xff);
}

/*
 * Periodically poll the device for input packets
 * in case an interrupt gets lost or the device
 * somehow gets wedged.  Reset if necessary.
 */
static void
sk_watchdog(int unit)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int s;

    si = &sk_info[unit];
    ifp = sktoifp(si);

    ASSERT(IFNET_ISLOCKED(ifp));

    XXX
}

/*
 * Disable the interface.
 */
static void
sk_stop(struct sk_info *si)
{
    struct ifnet *ifp = sktoifp(si);
```

```
    ASSERT(IFNET_ISLOCKED(ifp));

    ifp->if_flags &= ~IFF_ALIVE;

    /*
     * Mark an interface down and notify protocols
     * of the transition.
     */
    if_down(ifp);

    sk_reset(si);
}

/*
 * Enable the interface.
 */
static int
sk_start(
    struct sk_info *si,
    int flags)
{
    struct ifnet *ifp = sktoifp(si);
    int    error;

    ASSERT(IFNET_ISLOCKED(ifp));

    error = sk_init(si->si_unit);
    if (error || (ifp->if_addrlist == NULL))
        return error;
    ifp->if_flags = flags | IFF_ALIVE;

    /*
     * Broadcast an ARP packet, asking who has addr
     * on interface ac.
     */
    arpwhohas(&si->si_ac, &si->si_ac.ac_ipaddr);

    return (0);
}
```