Chapter 16

Network Device Drivers

A network device driver is a kernel-level driver that connects a communications device to the IRIX TCP/IP protocol stack using the *ifnet* interface established by BSD UNIX. This chapter contains these major topics:

- "Overview of Network Drivers" on page 390 gives an overview of the IRIX networking subsystem and the role of an ifnet driver in it.
- "Network Driver Interfaces" on page 392 summarizes the unique interfaces used by an *ifnet* driver.
- "Example ifnet Driver" on page 401 displays the code of a network driver, omitting all device-specific features.

Note: If your interest is in creating a network application based on sockets, TLI, or streams, this chapter offers little but background information. Refer to the *IRIX Network Programming Guide*, document Number 007-0810-050, for a complete review of all application-level services.

Even if your interest is in creating a kernel-level network driver, you should be familiar with the facilities documented in the *IRIX Network Programming Guide*. This chapter assumes that your are familiar with them.

Overview of Network Drivers

A network driver is a kernel-level driver module that connects a communications device such as an Ethernet board to the IRIX implementation of TCP/IP. An overview of the IRIX networking subsystem is shown in Figure 16-1.

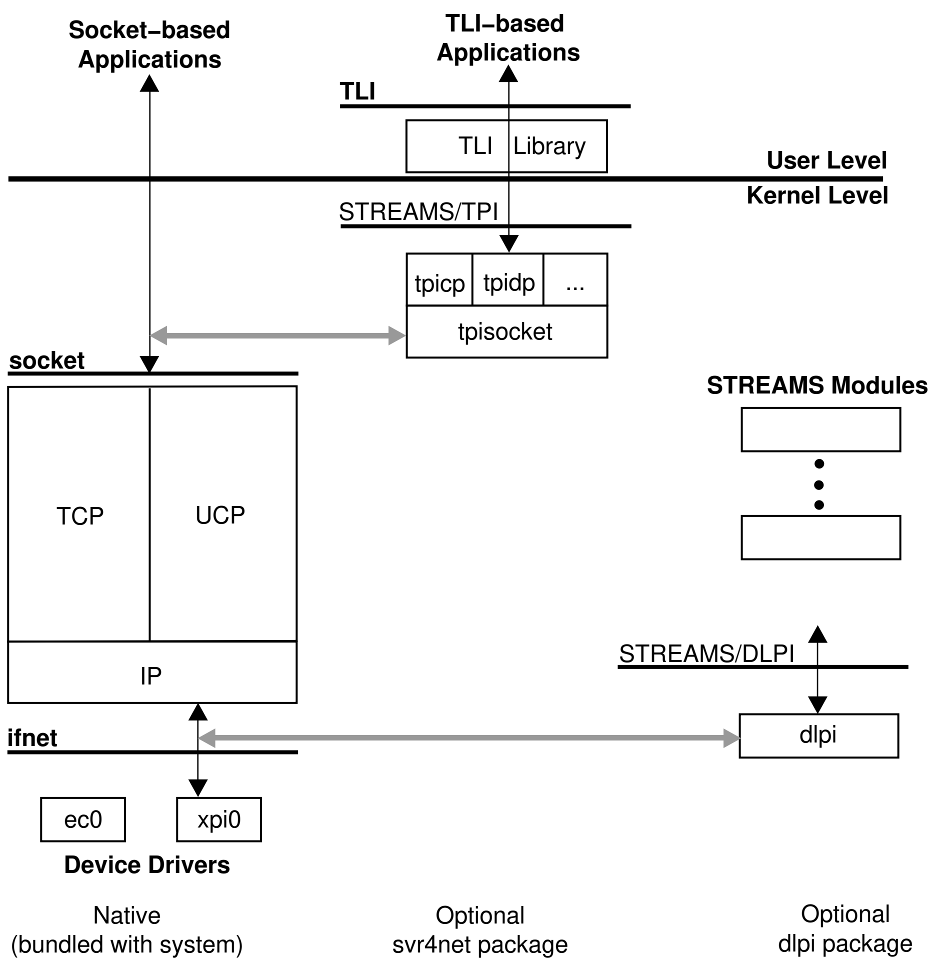

Figure 16-1 Overview of Network Architecture

Application Interfaces

User-level processes access the network in one of three ways:

- using the BSD socket interface (top left of Figure 16-1)
- using the SVR4 TLI interface through compatibility libraries that convert TLI operations into socket operations (top center of Figure 16-1)
- using a STREAMS interface to a STREAMS-based protocol stack (top right of Figure 16-1)

These three interfaces are documented in the *IRIX Network Programming Guide*

The native socket-based TCP/IP protocol code, the socket layer, and a number of *ifnet*-based device drivers are 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.

Compatibility support is included for 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.

A STREAMS pseudo-driver that supports the Data Link Provider Interface (DLPI) for STREAMS-based kernel protocol stacks is delivered in the optional dlpi package.

Protocol Stack Interfaces

A *protocol stack* is the software subsystem that manages data traffic according to the rules of a particular communications protocol. There are two ways in which a protocol stack can be integrated into the IRIX kernel. The TCP/IP stack creates and uses the *ifnet* interface to drivers (bottom left of Figure 16-1) and the socket interface to applications (top left of Figure 16-1).

Alternatively, a stack written to the DLPI architecture can communicate with STREAMS drivers (bottom right of Figure 16-1).

Device Driver Interfaces

A network driver uses the methods and facilities of other kernel-level device drivers, as described in Part III, “Kernel-Level Drivers” of this book. A network driver is compiled and linked like other drivers, configured using the same configuration files, and loaded into the kernel by *lboot* like other drivers.

However, other device drivers support the UNIX filesystem, transferring data in response to calls to their *pfx***read()**, *pfx***write()**, or *pfx***strategy()** entry points. This is not the case with a network driver; it supports protocol stacks, and it transfers data in response to calls from the *ifnet* interface.

Note: If you are working on a network device driver that uses DMA and that can be used in a Challenge or Onyx system, you should read Appendix B, “Challenge DMA with Multiple IO4 Boards.”

Network Driver Interfaces

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: *mbuf* objects 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 BSD-based networking framework and the implementation of the TCP/IP protocol suite have changed little from previous releases of IRIX, porting your *ifnet* device driver to this release of IRIX should be straightforward.

Note: Although the general kernel facilities documented in Chapter 9, “Device Driver/Kernel Interface” are standardized and stable, this is not the case with network interfaces. *The ifnet and other interfaces summarized in this topic are subject to change without notice*.

Kernel Facilities

A network driver is structured like any kernel-level device driver, much as described in Chapter 8, “Structure of a Kernel-Level Driver,” but with the following similarities and differences:

- A network driver is loaded by *lboot* in response to either a USE or VECTOR line in a file in */var/sysgen/system* (see “Configuring a Nonloadable Driver” on page 232).
- A network driver is initialized by a call to either its *pfx***init()** or *pfx***edtinit()** entry point when it is loaded.
- A network driver does not need to provide any other entry points (see “Entry Point Summary” on page 138).
- A network driver does not need to provide a driver flag constant *pfx***devflag** because a network driver is always assumed to be multiprocessor-aware (see “Driver Flag Constant” on page 140).
- Although a network driver can use the kernel functions for synchronization and locking (see “Waiting and Mutual Exclusion” on page 204), it normally does not because the *ifnet* interface includes special-purpose locking facilities that are more convenient (see “Multiprocessor Considerations” on page 396).

Principal ifnet Header Files

The software interface to network facilities is declared in the following important header files:

net/if.h	Basic ifnet facilities and data structures, including the *ifnet* structure, the basic driver interface object.
net/if_types.h	Constants for interface types, used in decoding address headers.
sys/mbuf.h	The *mbuf* structure with related constants and macros, and declarations of functions to allocate, manipulate, and free *mbuf* objects.
net/netisr.h	Declarations related to software interrupts, including **schednetisr()** to schedule an interrupt, and the IP input queue *ipintrq*.
net/multi.h	Routines defining a generic filter for use by drivers whose devices cannot perfectly filter multicast packets.

net/soioctl.h	Socket **ioctl()** function numbers, some of which reach a driver for action.
net/raw.h	The interface to the raw protocol family members *snoop* and *drain*.
net/if_arp.h	Generic ARP declarations.
netinet/if_ether.h	Essential declarations for Ethernet drivers, including ARP protocol for Ethernet.
sys/dlsap_register.h	DLPI interface declarations.

Debugging Facilities

When your driver is operating under a debugging kernel, you can use the facilities of symmon and idbg to display a variety of network-related data structures. See "Preparing the System for Debugging" on page 243, and see "Commands to Display Network-Related Structures" on page 269.

Information Sources

Aside from comments in header files, the complete *ifnet* interface and related interfaces have never been documented. In prior years, most people working on *ifnet* drivers have had access to the Berkeley UNIX source distribution and have been able to answer questions by referring to the code.

Referring to the code is an even more common option today, thanks to the release of 4.4BSD-Lite, a software distribution of BSD UNIX that does not require a source license, now widely available at a reasonable price. To obtain a copy, order the following:

- *4.4BSD-Lite Berkely Software Distribution CD-ROM Companion*, published by USENIX and O'Reilly & Associates; ISBN 1-56592-081-3 (US domestic) or ISBN 1-56592-092-9 (non-US).

The *ifnet* source code in this software is functionally compatible with IRIX *ifnet*, although some protocols (for example, *snoop* and *drain*) are not implemented in BSD-Lite.

In many respects, the *ifnet* interfaces and the logic of device drivers is dictated by Internet standards that are published as Requests for Comment (RFCs). The text of all RFCs can be obtained via FTP or with a WWW browser at *ftp://ds.internic.net/rfc/*.

Finally, the IRIX reference pages contain a wealth of detail regarding network interfaces. Some reference pages that are related to the interests of driver designers are listed in Table 16-1.

Table 16-1 Important Reference Pages Related to Network Drivers

Reference Page	Contents
arp(7)	Operation of the ARP protocol, with details of **ioctl()** functions.
drain(7)	Operation of the drain driver, which receives unwanted packets, with details of its **ioctl()** functions.
ethernet(7)	Overview of the IRIX Ethernet drivers, including error messages and the use of VECTOR lines to configure them.
fddi(7)	Cursory overview of IRIX FDDI drivers, with naming conventions.
ifconfig(1)	Management program used to enable and disable network interfaces (drivers) and change their runtime parameters.
netintro(7)	Overview of network facilities; mentions the role of the network interface (driver); has extensive detail on routing **ioctl()** calls.
network(1)	Documents the network initialization script that runs when the system is booted up.
raw(7)	Overview of the Raw protocol family whose members are snoop and drain.
routed(1)	Documents operation of the routing daemon, including **ioctl()** use.
snoop(7)	Operation of the snoop driver, which allows inspection of packets, with details of its **ioctl()** features.
ticlts(7)	Operation and use of the ticlts, ticots, and ticotsord loopback drivers.
tokenring(7)	Overview of the IRIX token-ring drivers, including packet formats.

Network Inventory Entries

When a device driver is initialized, it can install an entry in the system hardware inventory (see “Creating an Inventory Entry” on page 32). It is a good idea for a network driver to do this, because the *netsnoop* program relies on hardware inventory entries. After successfully initializing, a network device driver should call **add_to_inventory()** with the following five parameters (see *sys/invent.h*):

class INV_NETWORK

type The packet type, for example INV_NET_ETHER. See *sys/invent.h* for the possible “types for class network” list.

controller The kind of network controller from the “controllers for network types” list in *sys/invent.h*.

unit Any distinguishing number for this device. The *hinv* command does not decode this field.

state Any characteristic number for this device. The *hinv* command does not decode this field.

Multiprocessor Considerations

Prior to IRIX 5.3, the kernel BSD framework code and TCP/IP protocol stack executed under a single kernel lock, creating a single-threaded implementation. Beginning with IRIX 5.3, the BSD framework and TCP/IP protocol suite have been multi-threaded to support symmetric multiprocessing. The code uses different kernel locks to protect different critical sections.

IRIX now supports multiple, concurrent threads of execution within the TCP/UDP/IP protocol suite, and the kernel socket layer. In addition, network device drivers run on any available CPU, concurrently with the network software, applications, and other drivers.

This means that any ifnet-based network driver must be prepared to run asynchronously and concurrently with other drivers and with the protocol stack.

Ineffective spl() Functions

The **spl*()** functions were the traditional UNIX method of gaining exclusive use of data. In single-threaded ifnet drivers, the **splimp()** or **splnet()** functions were used to get exclusive use of the ifnet structure.

In a multiprocessor, **spl*()** functions like **splimp()** or **splnet()** do block interrupts on the local CPU, but they do not prevent interrupts from occurring on other processors in the system, nor do they prevent other processes on other CPUs from executing code that refers to the same data.

If you are porting a driver from a uniprocessor environment, search for any use of an **spl*()** function and plan to replace it with effective mutual exclusion locking macros.

Multiprocessor Locking Macros

Under BSD networking, drivers interface with the protocol stacks by queueing incoming packets on a per-protocol input queue. In a multiprocessor, each 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 interrupt 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

The _MP_NETLOCKS and MP compiler variables must be defined in order to enable the macros necessary to run under multi-threaded TCP/IP. Make sure the following compiler options are used:

```
-D_MP_NETLOCKS -DMP
```

Mutual Exclusion Macros

The macros for mutual exclusion defined in *net/if.h* are listed in Table 16-2.

Table 16-2 Mutual Exclusion Macros for ifnet Drivers

Macro Prototype	Purpose
IFNET_LOCK(*ifp*, *s*)	Get exclusive use of the structure **ifp*. **splimp()** is called to raise the interrupt level if necessary, and the returned value is saved in *s*.
IFNET_UNLOCK(*ifp*,*s*)	Release use of **ifp*, and return to interrupt level *s*.
IFNET_LOCKNOSPL(*ifp*)	Get exclusive use of the structure **ifp*, but do not call **splimp()** (the driver knows it is already at the appropriate level.)
IFNET_UNLOCKNOSPL(*ifp*)	Release use of **ifp* after use of IFNET_LOCKNOSPL.
IFNET_ISLOCKED(*ifp*)	Test whether **ifp* is locked.
IFQ_LOCK(*ifq*)	Get exclusive use of an input queue **ifq*.
IFQ_UNLOCK(*ifq*)	Release use of **ifq*.
IF_ENQUEUE(*ifq*, *mp*)	Lock the queue **ifq*; post the mbuf **mp*; release the queue.
IF_ENQUEUE_NOLOCK(*ifq*,*mp*)	Post the mbuf **mp* without locking.

The variables used in Table 16-2 are as follows:

ifp Address of a *struct ifnet* to be used exclusively.

s Integer variable to store the current interrupt mask level.

ifq Address of a *struct ifqueue* to be posted.

mp Address of a *struct mbuf* to be posted.

Macro Use

The TCP/IP protocol stack automatically acquires the ifnet structure before calling a network driver routine through that structure. Thus the driver's **init()**, **stop()**, **start()**, **output()**, and **ioctl()** functions do not need to use IFNET_LOCK or IFNET_UNLOCK. Look for expressions

```
ASSERT(IFNET_ISLOCKED(ifp));
```

in the example driver ("Example ifnet Driver" on page 401) to see places where this is the case. Explicit use of IFNET_LOCK is needed in the interrupt handler.

Input Queueing Example

Example 16-1 displays a code fragment of an interrupt handler that queues an input packet pointed to by *m* onto the IP input queue. The function **schednetisr()** is called to schedule processing of that packet. The code is assumed to be already at **splimp()**.

Example 16-1 Input Queueing Using Locking Macros

```
{
...
    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

Example 16-2 displays the skeleton of an Ethernet interrupt handler.

Example 16-2 Interrupt Handling Using Locking Macros

```
/*
 * 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);
}
```

Example ifnet Driver

The code in Example 16-3 represents the skeleton of an ifnet driver, showing its entry points, data structures, required **ioctl**() functions, address format conventions, and its use of kernel utility routines and locking primitives.

A comment beginning “MISSING:” represents a point at which a complete driver would contain code related to the device or bus it manages.

Example 16-3 Skeleton ifnet Driver

```
/*
 * 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
 *    (see “Managing Memory for Cache Coherency” on page 200)
 * - declare pointers to device registers as "volatile"
 * - compile on multiprocessor systems with "-D_MP_NETLOCKS -DMP"
 *
 * 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>
/* MISSING: local includes and defines */
#define    WORDALIGNED(p)    (p & (sizeof(int)-1) == 0)
#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)
/* MISSING: media-specific definitions of address size and header format */
#define    SKADDRLEN    (6)
#define    SKHEADERLEN    (sizeof (struct skheader))
/*
 * Our hypothetical medium has an IEEE 802-like 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.
 * This hypothetical driver uses 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;
    /* MISSING: other per-interface fields */
};
#define    si_if    si_ac.ac_if
#define    sktoifp(si)  (&(si)->si_ac.ac_if)
#define ifptosk(ifp)((struct sk_info *)ifp)
struct sk_info sk_info[SK_MAX_UNITS];
/* MISSING: other device-dependent structures */
/*
 * 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 */
};
/*
```

```
 * forward declarations of internal subfunctions.
 */
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);
/*
 * global ifnet structures
 */
extern struct ifqueue ipintrq;    /* ip input queue */
extern struct ifnet loif;         /* loopback driver if */
/*
 * EDT initialization routine, called by lboot(1) when processing a VECTOR
 * statement from /var/sysgen/system/*.sm.
 */
static void
skedtinit(struct edt *e)
{
    struct sk_info *si;
    struct ifnet *ifp;
    int    unit;
    /* MISSING: hardware-dependent local variables. */
    /*
     * Depending on the bus attachment, VME, GIO, or EISA,
     * refer to the appropriate part of this book for information on
     * bus-dependent support for probing and initializing a device,
     * allocating interrupt vectors, and registering the interrupt handler.
     */
    /* MISSING: bus- and device-dependent code to initialize the device. */
    /*
     * Other 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
 * - allocate DMA and PIO maps
 */
if (showconfig)
    cmn_err(CE_NOTE,"sk%d: hardware MAC address %s\n",
        si->si_unit,
        sk_sprintf(si->si_ouraddr));
/*
 * MISSING: address translation protocol goes here.
 * Save a copy of the 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));
    /*
     * Add the initialized network interface to the hardware inventory.
     */
    add_to_inventory(INV_NETWORK, SK_INV, INV_FDDI_SK, unit, 0);
}
/*
 * The sk_init() entry point, called ??? */
static int
sk_init(int unit)
{
    struct sk_info *si;
    struct ifnet *ifp;
    /* MISSING: device-dependent local variables */
    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
     *
     * See such topics in Chapter 9, "Device Driver/Kernel Interface"
     * as "Setting Up a DMA Transfer" on page 198, and see the
     * appropriate bus-related section (VME, GIO, EISA) for methods
     * of mapping DMA and PIO addresses.
     */
    /*
     * enable if_flags device behavior (IFF_DEBUG on/off, etc.)
     */
    /* MISSING: general device reset and initialize. */
    ifp->if_timer = SK_DOG;    /* turn on watchdog */
    /* turn device "on" now */
    /* MISSING: open the device for business. */
    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 */
    /* MISSING:
     * - reset device
     * - reset device receive descriptor ring
     * - free any enqueued transmit mbufs
     * - create device xmit descriptor ring
     */
}
/*
 * Interrupt handler.
 */
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;
    }
    /*
     * acquire interface lock
     */
    IFNET_LOCK(ifp, s);
    /*
     * MISSING: disable device and return if early interrupt
```

```
     */
    /*
     * MISSING: test and clear device interrupt pending register.
     */
    /*
     * process any received packets.
     */
    while (/* MISSING: received packets available */) {
        /*
         * MISSING: device-specific receive processing here.
         * Allocate and post a replacement receive buffer.
         */
        sk_input(si, m, totlen);
    }
    while (/* MISSING: mbuf has completed transmission */) {
        /*
         * Reclaim a completed device transmit resource
         * freeing completed mbufs, checking for errors,
         * and maintaining if_opackets, if_oerrors,
         * if_collisions, etc.
         */
    }
    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;
    /* MISSING: other local variables. */
    ASSERT(IFNET_ISLOCKED(ifp));
    mloop = NULL;
```

```
if (iff_dead(ifp->if_flags)) {
    error = EHOSTDOWN;
    goto bad;
}
/*
 * If send 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 (/* MISSING: transmits done */) {
        /*
         * Reclaim completed transmit descriptors.
         */
        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. First, 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);
    } /* switch (dst->sa_family) */

    /*
     * 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);
/*
 * MISSING: 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.
         */
        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;
        /*
         * MISSING: set 'snoopflags' and 'if_ierrors' as appropriate
         */
        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 this example, 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.
     * MISSING: Convert here if necessary.
     */
```

```
    /*
     * 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.
     */
    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;
    /* MISSING: device-dependent local variables */
    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;
        } /* inner switch (ifa->ifa_addr.sa_family) */
        break;
    } /* case SIOCSIFADDR */
    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;
                /* MISSING: enable hw all multicast addrs */
            } 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 SIOCADDMULTI, SIOCDELMULTI */
    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.
             * MISSING: You'll have to invent your own
             * multicast 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;
    } /* case SIOCADD/DELSNOOP */
    /*
     * MISSING: driver-specific ioctl cases 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);

    /* MISSING: poke this hash into device's hw address filter */
    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);
    /* MISSING: disable this hash value from the device if necessary */
    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));
    /* MISSING: poll the device, handle accordingly */
}
/*
 * 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);
}
```