Tru64 UNIX
Writing Network Device Drivers

Part Number: AA-RNG2A-TE

December 2000

Product Version: Device Driver Kit Version 2.0

Operating System and Version: Tru64 UNIX Version 5.0A or higher

This manual contains information that systems engineers need to write
network device drivers that operate on any bus.

Contents

About This Manual

1 Network Device Driver Environment

2 Defining Device Register Offsets

3 Defining the softc Data Structure

4 Implementing the Configure Section

5 Implementing the Autoconfiguration Support Section (probe)

6 Implementing the Autoconfiguration Support Section (attach)

7 Implementing the unattach Routine

8   Implementing the Initialization Section

9   Implementing the Start Section

10 Implementing a Watchdog Section

11 Implementing the Reset Section

12 Implementing the ioctl Section

13 Implementing the Interrupt Section

14 Network Device Driver Configuration

Index

Figures

Tables

About This Manual

This manual discusses how to write network device drivers for computer systems that run the Compaq Tru64™ UNIX operating system.

Audience

This manual is intended for systems engineers who:

- Use standard library routines to develop programs in the C language

- Know the Bourne shell or some other shell that is based on the UNIX operating system

- Understand basic Tru64 UNIX concepts such as kernel, shell, process, configuration, and autoconfiguration

- Understand how to use the Tru64 UNIX programming tools, compilers, and debuggers

- Develop programs in an environment that involves dynamic memory allocation, linked list data structures, and multitasking

- Understand the hardware device for which the driver is being written

- Understand the basics of the CPU hardware architecture, including interrupts, direct memory access (DMA) operations, and I/O

Before you write a network device driver, we recommend that you be familiar with the networking subsystem that the Tru64 UNIX operating system provides. This manual assumes that you are familiar with the following network interface types:

- Ethernet

- Fiber Distributed Data Interface (FDDI)

- Token Ring

See the Tru64 UNIX *Technical Overview* for descriptions of the data link media.

This manual also assumes that you have some knowledge of the Tru64 UNIX network programming environment, particularly:

- Data link provider interface

- X/Open transport interface

- Sockets

- Socket and XTI programming examples

- TCP specific programming information

- Information for Token Ring driver developers

- Data link interface

See the Tru64 UNIX *Network Programmer's Guide* for descriptions of these topics.

Scope of this Manual

This manual builds on the concepts and topics that are presented in *Writing Device Drivers*, which is the core manual for developing device drivers on Tru64 UNIX. It introduces topics that are specific to writing a device driver for a local area network (LAN) device and that are beyond the scope of the core manual.

In this manual, you can study a network driver called `if_el`. The `if_el` driver supports the driver interface requirements for a LAN device, specifically the 3Com 3C589C series PCMCIA adapter. The `if_el` driver was implemented according to the specifications detailed in *Ethernet III Parallel Tasking ISA, EISA, Micro Channel, and PCMCIA Adapter Drivers Technical Reference* . This specification is published by 3Com Corporation, and the manual part number is 09-0398-002B.

You can access the `if_el` source code in the device driver examples directory (if you have installed it on your system). Ethernet is the network interface type that is associated with the `if_el` driver. However, the explanations point out where differences exist between Ethernet and other network interfaces, including fiber distributed data interface (FDDI) and Token Ring.

The example network driver operates on multiple buses (specifically, the PCMCIA bus and the ISA bus). It uses the common `ifnet` interface to communicate with the upper layers of the Tru64 UNIX operating system. The example does not emphasize any specific types of network device drivers. However, mastering the concepts presented in this manual is useful preparation for writing network device drivers that operate on a variety of buses.

The manual does not discuss:

- How to write STREAMS network device drivers

- Topics associated with wide area networks (WANs)

- How to write an asynchronous transfer mode (ATM) device driver

- Details related to the network programming environment

New and Changed Features

This revision of the manual documents the following new features:

- Enabling support for enhanced hardware management

  Enhanced hardware management (EHM) allows you to modify hardware attributes, such as the type of LAN device, on either a local or a remote system. See *Section 3.2* for more information about how a network device driver uses routines to define and export hardware attributes.

- The `unattach()` routine

  The `unattach()` routine stops the network device and frees resources prior to unloading the device driver or powering off the bus to which the device is attached. See *Chapter 7* for more information.

Organization

This manual is organized as follows:

Chapter 1	Describes the sections that make up a network driver and compares them to the sections that are associated with block and character drivers.
Chapter 2	Describes the device register offset definitions for the `if_el` device driver's associated LAN device, the 3Com 3C5x9 series Ethernet adapter.
Chapter 3	Describes how to define a `softc` data structure, using the `if_el` device driver's `el_softc` structure as an example.
Chapter 4	Describes how to implement a `configure` interface, using the `if_el` device driver's `el_configure()` routine as an example.
Chapter 5	Describes how to implement a `probe` interface and associated routines, using the `if_el` device driver's `el_probe()` routine as an example.
Chapter 6	Describes how to implement an `attach` interface, using the `if_el` device driver's `el_attach()` routine as an example.
Chapter 7	Describes how to implement an `unattach()` routine to stop the device.

Chapter 8	Describes how to implement an `init` interface and associated routines, using the `if_el` device driver's `el_init()` routine as an example.
Chapter 9	Describes how to implement a `start` interface and associated routines, using the `if_el` device driver's `el_start()` routine as an example.
Chapter 10	Describes how to implement a `watchdog` interface, using the `if_el` device driver's `el_watch()` routine as an example.
Chapter 11	Describes how to implement a `reset` interface and associated routines, using the `if_el` device driver's `el_reset()` routine as an example.
Chapter 12	Describes how to implement an `ioctl` interface, using the `if_el` device driver's `el_ioctl()` routine as an example.
Chapter 13	Describes how to implement an interrupt handler, using the `if_el` device driver's `el_intr` interrupt handler as an example.
Chapter 14	Describes the `sysconfigtab` option entries necessary for configuring network device drivers on different bus types.

Related Documentation

The following examples and documents supplement information in this manual.

Examples

The directory `/usr/examples/ddk/src/network` includes the example source files that are used throughout this manual: `if_el.c`, `if_elreg.h`, `files`, and `sysconfigtab`.

Manuals

The following documents provide important information that supplements the information in this manual:

- *Installation Instructions and Release Notes* contains instructions on how to install the Device Driver Kit Version 2.0 product, including source code with examples and user manuals. It also describes changes to the product and documentation since the Device Driver Kit Release 1.0.

- *Writing Device Drivers* contains information that you need to develop device drivers on the Compaq Tru64 UNIX operating system.

- *Writing Kernel Modules* describes topics for all kernel modules such as kernel threads and writing kernel modules in a symmetric multiprocessing (SMP) environment.

- *Writing PCI Bus Device Drivers* describes PCI bus-specific topics, including PCI bus architecture and data structures that PCI bus device drivers use.

- *Writing VMEbus Device Drivers* describes VMEbus-specific topics, including VMEbus architecture and routines that VMEbus device drivers use.

- The *Guide to Preparing Product Kits* describes how to create kernel (device driver) product kits and layered product kits.

- *Kernel Debugging* describes how to use the `dbx`, `kdbx`, and `kdebug` debuggers to find problems in kernel code. It also describes how to write a `kdbx` utility extension and how to create and analyze a crash dump file.

- *Programming Support Tools* describes several commands and utilities in the Tru64 UNIX system, including facilities for text manipulation, macro and program generation, and source file management.

- The *Programmer's Guide* describes the programming environment of the Tru64 UNIX operating system, with an emphasis on the C programming language.

- The *Network Programmer's Guide* describes the Tru64 UNIX network programming environment and provides information on STREAMS programming.

- *System Administration* describes how to configure, use, and maintain the Tru64 UNIX operating system.

Reference Pages

Tru64 UNIX reference pages (also called manpages) contain descriptions of the routines (Section 9r), data structures (Section 9s), loadable services routines (Section 9u), and global variables (Section 9v) that apply to device drivers.

Reader's Comments

Compaq welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32

- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following
  location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is
  printed on the title page of this book and on its back cover.)

- The section numbers and page numbers of the information on which
  you are commenting.

- The version of Tru64 UNIX that you are using.

- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems
or technical support inquiries. Please address technical questions to your
local system vendor or to the appropriate Compaq technical support office.
Information provided with the software media explains how to send problem
reports to Compaq.

Conventions

This manual uses the following conventions:

⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`...`	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
file	Italic (slanted) type indicates variable values, placeholders, and function parameter names.
`buf`	In function definitions and syntax definitions used in driver configuration, this typeface indicates names that you must type exactly as shown.
[ ]	In formal parameter declarations in function definitions and in structure declarations, brackets indicate arrays. Brackets also specify ranges for device minor numbers and device special files in file fragments. However, for the syntax definitions

that are used in driver configuration, these brackets
indicate items that are optional.

| Vertical bars separating items that appear in the
syntax definitions used in driver configuration
indicate that you choose one item from among those
listed.

1

Network Device Driver Environment

A network device is responsible for both transmitting and receiving frames over the network media. Network devices have network device drivers associated with them. A network device driver attaches a network subsystem to a network interface, prepares the network interface for operation, and governs the transmission and reception of network frames over the network interface. Examples of network interface types include Ethernet, Fiber Distributed Data Interface (FDDI), and Token Ring.

Similar to the character and block device drivers that are discussed in *Writing Device Drivers*, a network device driver has the following sections:

- An include files section (Section 1.1)

- A declarations section (Section 1.2)

- A configure section (Section 1.3)

- An autoconfiguration support section (Section 1.4)

- An ioctl section (Section 1.9)

- An interrupt section (Section 1.10)

Similar to a character device driver, a network device driver can also have a reset section (Section 1.8).

Unlike a character or block device driver, a network device driver contains the following network driver-specific sections:

- An initialization section (Section 1.5)

- A start transmit section (Section 1.6)

- A watchdog section (Section 1.7)

- An output section (Section 1.11)

Figure 1–1 shows the sections that a typical network device driver can contain. Network device drivers are not required to have all of these sections, and more complex network drivers can have additional sections. However, all network drivers must have a configure section, and because network device drivers are associated with some device, they also must have a device register header file.

Figure 1–1: Sections of a Network Device Driver

Network Device Driver

```
/* Include Files Section */
 .
 .
 .
 /* Declarations Section */
 .
 .
 .
/* Configure Section */
 .
 .
 .
/* Initialization Section */
 .
 .
 .
/* Autoconfiguration Support Section */
 .
 .
 .
/* Start Transmit Section */
 .
 .
 .
/* Ioctl Section */
 .
 .
 .
/* Interrupt Section */
 .
 .
 .
/* Reset Section */
 .
 .
 .
/* Watchdog Section */
 .
 .
 .
```

ZK-0818U-AI

Unlike for block and character drivers, you do not specify network driver entry points in the `dsent` data structure. This means that a network driver has no exposure into the file system and, therefore, has no entry in the `/dev` directory. Thus, network drivers do not have block and character driver-specific interfaces such as `open`, `close`, `read`, `write`, and `strategy`.

Instead of registering its entry points in a `dsent` data structure, a network driver registers its entry points with the upper layers of the Tru64 UNIX operating system in an `ifnet` data structure. For example, a network driver registers entry points for queueing data for transmission and for starting data transmission.

In addition to storing the entry points for a network driver's associated interfaces, the `ifnet` data structure stores parameter-related information such as the transmission medium and statistics to track the performance of the interface and network.

The `ifnet` data structure also contains a queue of data packets that the network driver sends to the network device. These packets are linked lists of `mbuf` data structures. Each such linked list represents one data packet. Depending on how a network driver fills in certain members of the `ifnet` data structure, the upper-level network code fragments the data to be sent out over a network. In the case of the Ethernet network interface, the upper-level code never hands off to the driver a single packet that exceeds 1514 bytes.

1.1 Include Files Section for a Network Driver

A network device driver includes header files that define data structures and constant values that the driver references. A network device driver includes some of the same files as a block or character device driver, such as `errno.h`. It can also include the header files that are specific to network device drivers. For example:

```
#include <net/net_globals.h>
#include <sys/socket.h>
#include <net/if.h>
#include <net/if_types.h>
```

The following code shows the include files section for the `if_el` device driver:

```
#include <sys/param.h>
#include <sys/systm.h>
#include <sys/mbuf.h>
#include <sys/buf.h>
#include <sys/protosw.h>
#include <sys/socket.h>
#include <sys/vmmac.h>
#include <vm/vm_kern.h>
#include <sys/ioctl.h>        1
#include <sys/errno.h>
#include <sys/time.h>
#include <sys/kernel.h>
#include <sys/proc.h>
#include <sys/sysconfig.h>    2
#include <net/if.h>
#include <net/netisr.h>
#include <net/route.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#include <netinet/in_var.h>
```

```
#include <netinet/ip.h>
#include <netinet/ip_var.h>
#include <netinet/if_ether.h>   3
#include <net/ether_driver.h>
#include <io/common/devdriver.h>   4
#include <hal/cpuconf.h>
#include <kern/thread.h>
#include <kern/sched_prim.h>
#include <kern/lock.h>

#include <io/dec/eisa/eisa.h>   5
#include <io/dec/pcmcia/pcmcia.h>   6
#include <io/dec/pcmcia/cardinfo.h>

#include <io/dec/netif/lan_common.h>   7

#include <io/dec/netif/if_elreg.h>   8
```

1 Includes the `ioctl.h` include file, which defines common `ioctl` com-
   mands. The `ioctl.h` file is located in `/usr/include/sys/ioctl.h`.

2 Includes the `sysconfig.h` header file, which defines the constants that
   all device drivers use during configuration. The `sysconfig.h` file is
   located in `/usr/include/sys/sysconfig.h`.

3 Includes the `if_ether.h` header file, which defines the `ether_header`
   data structure. All network drivers typically include this file.

   If you are writing the network driver for FDDI media, you also include
   the header file `if_fddi.h`. If you are writing the network driver for
   Token Ring media, you also include the header file `if_trn.h`.

4 Includes the `devdriver.h` header file, which defines common device
   driver data structures and constants. The `devdriver.h` file is located
   in `/usr/include/io/common/devdriver.h`.

5 Includes the header file `eisa.h`, which is associated with the ISA bus.

   If you are writing the driver to operate on multiple bus architectures,
   you must include the bus-specific header file. The `if_el` device driver
   is implemented to operate on two buses: the ISA and the PCMCIA.

6 Includes the header files `pcmcia.h` and `cardinfo.h`, which are
   associated with the PCMCIA bus.

7 Includes the `lan_common.h` file, which contains definitions that all
   local area network (LAN) device drivers need.

8 Includes the device register header file. The directory specification you
   make here depends on where you put the device register header file.

1.2 Declarations Section for a Network Driver

The declarations section for a network device driver contains the following
categories of information:

- External and forward declarations (Section 1.2.1)

- Declaration of `softc` and `controller` data structure arrays
  (Section 1.2.2)

- Declaration of the `driver` data structure (Section 1.2.3)

- Definitions of driver-specific macros (Section 1.2.4)

The following sections discuss each of these categories of declarations, using the `if_el` device driver as an example.

The declarations section also contains the definition of the `softc` data structure and declarations for configure-related variables and data structures. Chapter 3 discusses the definition of a network driver's `softc` data structure. Section 4.1 discusses the declarations that are related to configuration.

1.2.1 External and Forward Declarations

The following code shows the external and forward declarations for the `if_el` device driver:

```
int el_configure(cfg_op_t, cfg_attr_t *, size_t, cfg_attr_t *, size_t);  1

static int    el_probe (io_handle_t, struct controller *);  2
static int    el_attach(struct controller *);
static int    el_unattach(struct bus *, struct controller *);
static int    el_init_locked(struct el_softc *, struct ifnet *, int);
static int    el_init(int);
static void   el_start_locked(struct el_softc *, struct ifnet *);
static void   el_start(struct ifnet *);
static int    el_watch(int);
static void   el_reset_locked(struct el_softc *, struct ifnet *, int);
static void   el_reset(int);
static int    el_ioctl(struct ifnet *, u_int, caddr_t);
static int    el_intr(int);
static void   el_rint(struct el_softc *, struct ifnet *);
static void   el_tint(struct el_softc *, struct ifnet *);
static void   el_error(struct el_softc *, struct ifnet *);
static void   el_shutdown(struct el_softc *);
static void   el_card_remove(int, struct el_softc *);
static int    el_isa_reset_all(io_handle_t, int *, struct controller *);
static int    el_isa_activate(io_handle_t, int *, struct controller *);
static unsigned short  el_isa_read_offset(io_handle_t, int);
static void   el_wait(struct el_softc *);
static void   el_autosense_thread(struct el_softc *);
static int    el_card_out(struct el_softc *);
extern struct timeval time;  3
extern task_t first_task;  4
```

1 Declares the function prototype definitions for all exported functions.

2 Declares the driver interfaces for the `if_el` device driver.

3 Declares the external `timeval` data structure called `time`. Various `ioctl` commands use this data structure.

4 Declares a pointer to the external `task_t` data structure called `first_task`. The `task_t` data structure is an opaque data structure; that is, all of its associated members are referenced and manipulated by the Tru64 UNIX operating system and not by the user of kernel threads. Every kernel thread must be part of a task.

The `if_el` driver's `el_probe` interface uses this data structure when it creates a kernel thread.

1.2.2 Declaring softc and controller Data Structure Arrays

The following code shows the declarations for the `el_softc` and `controller` data structure arrays. The system uses these arrays to find out which `softc` and `controller` data structures are associated with a specific 3Com 3C5x9 device. The driver's `el_probe` interface initializes these arrays if the probe operation is successful.

The arrays of `el_softc` and `controller` data structures need to be static for the `if_el` device driver. Be aware that static arrays fix the maximum number of devices that the user can configure on the system.

```
#define  el_MAXDEV 7  1
static struct el_softc *el_softc[el_MAXDEV]={0};  2
static struct controller *el_info[el_MAXDEV]={0};  3

static int el_isa_tag = 0;  4
static int el_isa_reset = 0;  5
decl_simple_lock_info(static, el_lock_info);  6
```

1 Defines a constant called `el_MAXDEV`, which allocates data structures that the `if_el` device driver needs. A maximum of seven instances of the 3C5x9 controller can be on the system. This means that `el_MAXDEV` is the maximum number of controllers that the `if_el` driver can support. This is a small number of instances of the driver, and the data structures themselves are not large, so it is acceptable to allocate for the maximum configuration.

2 Declares an array of pointers to `el_softc` data structures and calls it `el_softc`. The `el_MAXDEV` constant specifies the size of this array.

3 Declares an array of pointers to `controller` data structures and calls it `el_info`. The `el_MAXDEV` constant specifies the size of this array.

4 Declares a variable called `el_isa_tag` and initializes it to the value 0 (zero). The `if_el` driver's `el_isa_activate` interface uses this variable.

5 Declares a variable called `el_isa_reset` and initializes it to the value 0 (zero). The `if_el` driver's `el_probe` interface uses this variable.

6 Uses the `decl_simple_lock_info()` routine to declare a simple lock data structure called `el_lock_info`.

1.2.3 Declaring and Initializing the driver Data Structure

The following code shows how the `if_el` device driver declares and
initializes the `driver` data structure with the names of its entry points:

```
static struct driver eldriver = { 1
    el_probe,
    0,
    el_attach,
    0,
    0,
    0,
    0,
    0,
    "el",
    el_info,
    0,
    0,
    0,
    0,
    0,
    el_unattach,
    0
};
```

[1] Declares and initializes the `driver` data structure called `eldriver`.
Because a network device driver does not have exposure to the file
system, it does not provide `open`, `close`, `read`, `write`, and `strategy`
interfaces. The members of the `driver` data structure that specify
these entry points are initialized to 0 (zero).

The `if_el` driver initializes the following members to nonzero values:

- `probe`, which specifies the driver's `probe` interface, `el_probe`

- `cattach`, which specifies the driver's controller `attach` interface,
  `el_attach`

- `ctlr_name`, which specifies the controller name, `el`

- `ctlr_list`, which specifies a pointer to the array of pointers to
  `controller` data structures, `el_info`

- `ctlr_unattach`, which specifies the driver's controller `unattach`
  interface, `el_unattach`

1.2.4 Defining Driver-Specific Macros

To help you write more portable device drivers, Tru64 UNIX provides the
following kernel routines, which allow you to read from and write to a
control status register (CSR) address without directly accessing its device
registers. These macros call the `read_io_port()` or `write_io_port()`
generic routines.

READ_BUS_D8	Reads a byte (8 bits) from a device register.
READ_BUS_D16	Reads a word (16 bits) from a device register.
READ_BUS_D32	Reads a longword (32 bits) from a device register.
READ_BUS_D64	Reads a quadword (64 bits) from a device register.
WRITE_BUS_D8	Writes a byte (8 bits) to a device register.
WRITE_BUS_D16	Writes a word (16 bits) to a device register.
WRITE_BUS_D32	Writes a longword (32 bits) to a device register.
WRITE_BUS_D64	Writes a quadword (64 bits) to a device register.

The following code shows how the if_el driver uses the READ_BUS_D16,
READ_BUS_D32, WRITE_BUS_D16, and WRITE_BUS_D32 kernel routines to
construct driver-specific macros to perform read and write operations on the
3Com 3C5x9 device:

```
#define READ_CCR(sc)          READ_BUS_D16((sc)->reg4); mb();  1
#define WRITE_CCR(sc, val)    WRITE_BUS_D16((sc)->reg4, (val)); mb();
#define READ_ACR(sc)          READ_BUS_D16((sc)->reg6); mb();
#define WRITE_ACR(sc, val)    WRITE_BUS_D16((sc)->reg6, (val)); mb();
#define WRITE_RCR(sc, val)    WRITE_BUS_D16((sc)->reg8, (val)); mb();
#define WRITE_ECR(sc, val)    WRITE_BUS_D16((sc)->regA, (val)); mb();
#define READ_EDR(sc)          READ_BUS_D16((sc)->regC); mb();
#define WRITE_CMD(sc, val)    WRITE_BUS_D16((sc)->regE, (val)); \
                                     mb(); el_wait((sc))
#define READ_STS(sc)          READ_BUS_D16((sc)->regE); mb();
#define WRITE_DATA(sc, val)   WRITE_BUS_D32((sc)->data, (val)); mb();
#define READ_DATA(sc)         READ_BUS_D32((sc)->data); mb();
#define READ_ND(sc)           READ_BUS_D16((sc)->reg6); mb();
#define WRITE_ND(sc, val)     WRITE_BUS_D16((sc)->reg6, (val)); mb();
#define READ_MD(sc)           READ_BUS_D16((sc)->regA); mb();
#define WRITE_MD(sc, val)     WRITE_BUS_D16((sc)->regA, (val)); mb();
#define READ_TXF(sc)          READ_BUS_D16((sc)->regC); mb();
#define READ_RXF(sc)          READ_BUS_D16((sc)->regA); mb();
#define WRITE_AD1(sc, val)    WRITE_BUS_D16((sc)->reg0, (val)); mb();
#define WRITE_AD2(sc, val)    WRITE_BUS_D16((sc)->reg2, (val)); mb();
#define WRITE_AD3(sc, val)    WRITE_BUS_D16((sc)->reg4, (val)); mb();
#define READ_TXS(sc)          READ_BUS_D16((sc)->regA); mb();
#define WRITE_TXS(sc, val)    WRITE_BUS_D16((sc)->regA, (val)); mb();
#define READ_RXS(sc)          READ_BUS_D16((sc)->reg8); mb();
#define READ_FDP(sc)          READ_BUS_D16((sc)->reg4); mb();
```

1 Constructs driver-specific macros to read from and write to the 3Com
   3C5x9 device's CSRs.

   The first argument to these macros specifies an I/O handle that
   references a device register or memory that is located in bus address
   space (either I/O space or memory space). You can perform standard
   C mathematical operations (addition and subtraction only) on the I/O
   handle. The READ_CCR, WRITE_CCR, and the other macros construct
   the first argument by referencing the I/O handle that is defined in the
   el_softc data structure.

The second argument to the WRITE_CCR and the other write macros specifies the data to be written to the device register in bus address space. These write macros construct the second argument by referencing the val variable. For the if_el driver, this data is typically one of the device register offsets that is defined in the if_elreg.h file.

The read and write driver-specific macros call the mb() kernel routine to perform a memory barrier. The mb() kernel routine ensures that the read or write operation is issued before the CPU executes any subsequent code. See Section 7.5 of the Tru64 UNIX *Writing Device Drivers* manual for more information about the mb() routine and when to use it.

Table 1−1 provides information on the driver-specific macros.

Table 1−1: Driver-Specific Macros

Macro	Description
READ_CCR and WRITE_CCR	Read from and write to the 3Com 3C5x9 device's configuration control register.
READ_ACR and WRITE_ACR	Read from and write to the 3Com 3C5x9 device's address control register.
WRITE_RCR	Write to the 3Com 3C5x9 device's resource configuration register.
WRITE_ECR	Write to the 3Com 3C5x9 device's EEPROM command register.
READ_EDR	Read from the 3Com 3C5x9 device's EEPROM data register.
WRITE_CMD	Write to the 3Com 3C5x9 device's command port registers.
READ_STS	Read from the 3Com 3C5x9 device's I/O status register.
READ_DATA and WRITE_DATA	Read from and write to the 3Com 3C5x9 device's receive data and transmit data registers.
READ_ND and WRITE_ND	Read from and write to the 3Com 3C5x9 device's network diagnostic register.
READ_MD and WRITE_MD	Read from and write to the 3Com 3C5x9 device's media type and status register.
READ_TXF and READ_RXF	Read from the 3Com 3C5x9 device's transmit and receive FIFO registers.
WRITE_AD1, WRITE_AD2, and WRITE_AD3	Set the LAN physical address for the 3Com 3C5x9 device.
READ_TXS and WRITE_TXS	Read from and write to the 3Com 3C5x9 device's transmit status register.

Table 1–1: Driver-Specific Macros (cont.)

Macro	Description
READ_RXS	Read from the 3Com 3C5x9 device's receive status register.
READ_FDP	Read from the 3Com 3C5x9 device's FIFO diagnostic port register.

1.3 Configure Section for a Network Driver

The configure section for a network device driver contains a configure interface. The cfgmgr framework calls the driver's configure interface at system startup to handle static configuration requests. The cfgmgr framework can also call the driver's configure interface to handle user-level requests to dynamically configure, unconfigure, query, and reconfigure a device driver at run time. If you implement the driver as a single binary module, the configure interface can handle both static and dynamic configuration.

1.4 Autoconfiguration Support Section for a Network Driver

The autoconfiguration support section for a network device driver contains the following entry points:

- A probe interface, which determines if the network device exists and is functional on the system
- An attach interface, which establishes communication with the device and initializes the driver's ifnet data structure.

You define the entry point for each of these interfaces in the driver data structure.

1.5 Initialization Section for a Network Driver

The initialization section for a network device driver prepares the network to transmit and receive data packets.

1.6 Start Section for a Network Driver

The start section for a network device driver contains a start interface, which transmits data packets on the network interface. You define the entry point for the start interface in the ifnet data structure. However, before this interface can be called, the network adapter must be enabled for data packet transmission and reception. You enable the network adapter by invoking the SIOCSIFADDR ioctl command.

1.7 Watchdog Section for a Network Driver

The watchdog section for a network device driver contains a `watchdog` interface, which attempts to restart the adapter. The `watchdog` interface is optional in a network device driver. If the network device driver implements it, `watchdog` is called by a kernel thread if the driver's interrupt handler has not shut down the countdown timer within a certain number of seconds of queueing a data packet for transmission from the upper layer. This indicates that the adapter is no longer on line.

1.8 Reset Section for a Network Driver

The reset section for a network device driver contains a `reset` interface. The `reset` interface resets the LAN adapter. This interface is called to restart the device following a network failure. This interface resets all of the counters and local variables. It can also free up and reallocate all of the buffers that the network driver uses.

1.9 ioctl Section for a Network Driver

The ioctl section for network device drivers performs miscellaneous tasks that have nothing to do with data packet transmission and reception. Typically, these tasks relate to turning specific features of the hardware on or off.

The ioctl section contains an `ioctl` interface. You define this entry point in the `ifnet` data structure.

1.10 Interrupt Section for a Network Driver

The interrupt section for a network device driver contains an interrupt handler. The interrupt handler processes network device interrupts. You define the entry point for the interrupt handler by calling the `handler` interfaces. The interrupt handler is called each time that the network interface receives an interrupt. After identifying which type of interrupt was received — transmit or receive — the interrupt handler calls the appropriate routine to process the interrupt.

1.11 Output Section for a Network Driver

The output section for a network device driver formats a data packet for transmission on the network. The `ether_output()` routine formats data packets for Tru64 UNIX network drivers. Despite its name, `ether_output()` handles the frame formats for Ethernet, token ring, and FDDI. After it has properly formatted the data packet, `ether_output()` enqueues the packet on the driver's send queue and calls the driver's start

interface to transmit the data. All network drivers *must* set the output
member of the `ifnet` data structure to `ether_output`.

2

Defining Device Register Offsets

The device register header file defines the device register offsets for the device. The `if_elreg.h` file is the device register header file for the `if_el` device driver. It defines the device register offsets for the 3Com 3C5x9 series Ethernet adapter. Specifically, the `if_elreg.h` file contains the following categories of device registers:

- Interrupt and status register (Section 2.1)

- Command port registers (Section 2.2)

- Window 0 configuration registers (Section 2.3)

- Window 3 configuration registers (Section 2.4)

- Window 1 operational registers (Section 2.5)

- Window 4 diagnostic registers (Section 2.6)

- EEPROM data structure definition (Section 2.7)

Your network device might have different device registers. However, this device register header file can serve as an example of how to set up device register offset definitions. See your network device documentation to learn about control and status registers for your device.

2.1 Interrupt and Status Register Offset Definitions

The following code shows the offset definitions for the interrupt and status register. The `if_el` device driver reads these offsets from the interrupt and status register. The `CMD_ACKINT`, `CMD_SINTMASK`, and `CMD_ZINTMASK` commands either set or clear the bits.

```
#define STS_PORT        0xe   1

#define S_IL            (1)      2
#define S_AF            (1<<1)   3
#define S_TC            (1<<2)   4
#define S_TA            (1<<3)   5
#define S_RC            (1<<4)   6
#define S_RE            (1<<5)   7
#define S_IR            (1<<6)   8
#define S_US            (1<<7)   9
#define S_IP            (1<<12)   10
#define CURWINDOW(x)    ((x>>13)&0x7)   11
```

1 Defines the offset for the I/O port of the interrupt and status register. This register can be set to one or more of the bit values.

2 Defines the interrupt latch bit position.

3 Defines the adapter failure bit position.

4 Defines the transmit complete bit position.

5 Defines the transmit available bit position.

6 Defines the receive complete bit position.

7 Defines the receive early bit position.

8 Defines the interrupt request bit position.

9 Defines the update statistics bit position.

10 Defines the command in-progress bit position.

11 Defines the current window number bit position.

2.2 Command Port Register Offset Definitions

The following code shows the offset definitions for the command port
register. Bits 0:10 contain optional parameter bits and bits 11:15 contain
the command.

```
#define CMD_PORT          0xe   1

#define CMD_RESET                 (0x0)          2
#define CMD_WINDOW0               ((0x1<<11)+0x0)    3
#define CMD_WINDOW1               ((0x1<<11)+0x1)    4
#define CMD_WINDOW2               ((0x1<<11)+0x2)    5
#define CMD_WINDOW3               ((0x1<<11)+0x3)    6
#define CMD_WINDOW4               ((0x1<<11)+0x4)    7
#define CMD_WINDOW5               ((0x1<<11)+0x5)    8
#define CMD_WINDOW6               ((0x1<<11)+0x6)    9
#define CMD_START2                (0x2<<11)      10
#define CMD_RXDIS                 (0x3<<11)      11
#define CMD_RXENA                 (0x4<<11)      12
#define CMD_RXRESET               (0x5<<11)      13
#define CMD_RXDTP                 (0x8<<11)      14
#define CMD_TXENA                 (0x9<<11)      15
#define CMD_TXDIS                 (0xa<<11)      16
#define CMD_TXRESET               (0xb<<11)      17
#define CMD_REQINT                (0xc<<11)      18
#define CMD_ACKINT                (0xd<<11)      19
#define CMD_SINTMASK              (0xe<<11)      20
#define CMD_ZINTMASK              (0xf<<11)      21
#define CMD_FILTER                (0x10<<11)     22
enum rx_filter {   23
    RF_IND     =0x1,
    RF_GRP     =0x2,
    RF_BRD     =0x4,
    RF_PRM     =0x8
};
#define CMD_RXEARLY               (0x11<<11)     24
#define CMD_TXAVAILTHRESH         (0x12<<11)     25
#define CMD_TXSTARTTHRESH         (0x13<<11)     26
#define CMD_STATSENA              (0x15<<11)     27
#define CMD_STATSDIS              (0x16<<11)     28
#define CMD_STOP2                 (0x17<<11)     29
#define CMD_RXRECTHRESH           (0x18<<11)     30
```

```
#define CMD_POWERUP            (0x1b<<11)  31
#define CMD_POWERDOWN          (0x1c<<11)  32
#define CMD_POWERAUTO          (0x1d<<11)  33
```

1 Defines the offset for the I/O port of the command port register.

2 Defines the reset command bit position.

3 Defines the window selector for commands that are used to set up the device.

4 Defines the window selector for commands that control the operation of the device.

5 Defines the window selector for specifying the hardware address of the device.

6 Defines the window selector for the device's first-in/first-out (FIFO) buffer.

7 Defines the window selector for commands that are used for diagnostic purposes.

8 Defines a second window selector for commands that are used for diagnostic purposes.

9 Defines the window selector for commands that are related to gathering device statistics.

10 Defines the start 10Base2 Ethernet cable command.

11 Defines the receive (RX) disable command.

12 Defines the receive (RX) enable command.

13 Defines the receive (RX) reset command.

14 Defines the receive (RX) discard top packet command.

15 Defines the transmit (TX) enable command.

16 Defines the transmit (TX) disable command.

17 Defines the transmit (TX) reset command.

18 Defines the request interrupt command.

19 Defines the acknowledge interrupt command.

20 Defines the set interrupt mask command.

21 Defines the clear interrupt command.

22 Defines the receive (RX) filter command.

23 Defines an enumerated data type called `rx_filter`. The `if_el` device driver can assign one of the following values to `CMD_FILTER`:

`RF_IND`	Individual address
`RF_GRP`	Group address
`RF_BRD`	Broadcast address
`RF_PRM`	Promiscuous address

24 Defines the receive (RX) early threshold command.

25 Defines the transmit (TX) available threshold command.

26 Defines the transmit (TX) start threshold command.

27 Defines the statistics enable command.

28 Defines the statistics disable command.

29 Defines the stop 10Base2 Ethernet cable command.

30 Defines the receive (RX) reclaim threshold command.

31 Defines the power-up command.

32 Defines the power-down command.

33 Defines the power-auto command.

2.3 Window 0 Configuration Register Offset Definitions

The window 0 configuration registers include such registers as manufacturer ID and adapter ID, as shown in Figure 2–1.

Figure 2–1: Window 0 Configuration Registers

Register	Constant
Manufacturer ID Register	W0_MID
Adapter ID Register	W0_AID
Configuration Control Register	W0_CCR
Address Control Register	W0_ACR
Resource Configuration Register	W0_RCR
EEPROM Command Register	W0_ECR
EEPROM Data Register	W0_EDR

ZK-1267U-AI

The following code shows the offset definitions for the registers that make up the window 0 configuration register:

```
#define W0_MID      0x0  1
#define W0_AID      0x2  2
#define W0_CCR      0x4  3
enum w0_ccr {  4
   CCR_PCMCIA=0x4000,
   CCR_AUI=0x2000,
   CCR_10B2=0x1000,
   CCR_ENDEC=0x0100,
   CCR_RESET=0x4,
   CCR_ENA=0x1
};
#define W0_ACR      0x6  5
enum w0_acr {  6
   ACR_10BT=0x0000,
   ACR_10B5=0x4000,
   ACR_10B2=0xc000,
   ACR_ROMS=0x3000,
   ACR_ROMB=0x0f00,
   ACR_ASE= 0x0080,
   ACR_BASE=0x001f
};
#define W0_RCR      0x8  7
enum w0_rcr {  8
   RCR_IRQ=0xf000,
   RCR_RSV=0x0f00
};
#define W0_ECR      0xa  9
enum w0_ecr {  10
   ECR_EBY=0x8000,
   ECR_TST=0x4000,
   ECR_CMD=0x00ff,
   ECR_READ= 0x0080,
```

```
    ECR_WRITE=0x0040,
    ECR_ERASE=0x00c0,
    ECR_EWENA=0x0030,
    ECR_EWDIS=0x0000,
    ECR_EAR=  0x0020,
    ECR_WAR=  0x0010
};
#define W0_EDR     0xc  11
```

1 Defines the offset for the manufacturer ID register.

2 Defines the offset for the adapter ID register.

3 Defines the offset for the configuration control register.

4 Defines an enumerated data type called `w0_ccr`. The `if_el`
   device driver can assign one of the following values to `W0_CCR` (the
   configuration control register):

CCR_PCMCIA	If set, this is a PCMCIA bus. Otherwise, it is an ISA bus.
CCR_AUI	If set, the attachment unit interface (AUI) is available.
CCR_10B2	If set, the 10Base2 receiver is available.
CCR_ENDEC	If set, the internal encode/decode (ENDEC) loopback is used.
CCR_RESET	Reset adapter.
CCR_ENA	Enable adapter.

5 Defines the offset for the address control register.

6 Defines an enumerated data type called `w0_acr`. The `if_el` device
   driver can assign one of the following values to `W0_ACR` (the address
   control register):

ACR_10BT	If set, the information transmission rate is at 10 Mb/sec for the Ethernet unshielded twisted-pair cable wires.
ACR_10B5	If set, the information transmission rate is at 10 Mb/sec for the Ethernet thick coaxial cable wire. The length between repeaters is 500 meters.
ACR_10B2	If set, the information transmission rate is at 10 Mb/sec for the Ethernet thin coaxial cable wire. The length between repeaters is 200 meters.
ACR_ROMS	Represents the read-only memory size for the ISA bus.
ACR_ROMB	Represents the read-only memory base for the ISA bus.
ACR_ASE	Represents the autoselect mode.
ACR_BASE	Represents the I/O base address.

7 Defines the offset for the resource configuration register.

8️⃣ Defines an enumerated data type called `w0_rcr`. The `if_el` device driver can assign one of the following bits to `W0_RCR` (the resource configuration register):

`RCR_IRQ`	Represents the interrupt request (IRQ).
`RCR_RSV`	Represents a reserved field.

9️⃣ Defines the offset for the EEPROM command register.

🔟 Defines an enumerated data type called `w0_ecr`. The `if_el` device driver can assign one of the following bits to `W0_ECR` (the EEPROM command register):

`ECR_EBY`	Indicates that the EEPROM is busy.
`ECR_TST`	Indicates that the EEPROM is in test mode.
`ECR_CMD`	Represents EEPROM command bits.
`ECR_READ`	Represents an EEPROM read command.
`ECR_WRITE`	Represents an EEPROM write command.
`ECR_ERASE`	Represents an EEPROM erase command.
`ECR_EWENA`	Represents an EEPROM enable erase or write command.
`ECR_EWDIS`	Represents an EEPROM disable erase or write command.
`ECR_EAR`	Represents an EEPROM erase all registers command.
`ECR_WAR`	Represents an EEPROM write all registers command.

1️⃣1️⃣ Defines the offset for the EEPROM data register.

2.4 Window 3 Configuration Register Offset Definitions

The window 3 configuration registers consist of the additional setup information registers shown in Figure 2–2.

Figure 2–2: Window 3 Configuration Registers

Register	Constant
Additional Setup Information 2 Register	W3_ASI2
Additional Setup Information 0 Register	W3_ASI0

ZK-1268U-AI

The following code shows the offset definitions for the registers that are associated with the window 3 configuration registers:

```
#define W3_ASI2      0x2   1
#define W3_ASI0      0x0   2
enum w3_asi {   3
   ASI_IAS_ISA=0x00040000,
   ASI_IAS_PNP=0x00080000,
   ASI_IAS_BOT=0x000c0000,
   ASI_IAS_NON=0x00000000,
   ASI_PAR_35 =0x00000000,
   ASI_PAR_13 =0x00010000,
   ASI_PAR_11 =0x00020000,
   ASI_RS     =0x00000030,
   ASI_RW     =0x00000008,
   ASI_RSIZE8 =0x00000001,
   ASI_RSIZE32=0x00000002
};
```

1 Defines the offset for the additional setup information register 2.

2 Defines the offset for the additional setup information register 0.

3 Defines an enumerated data type called `w3_asi`. The `if_el` device driver can assign one of the following values to `w3_ASI2` and `w3_ASI0` (the additional setup information registers):

ASI_IAS_ISA	Activates ISA bus contention.
ASI_IAS_PNP	Activates ISA bus PNP.
ASI_IAS_BOT	Activates ISA bus contention and PNP.
ASI_IAS_NON	Indicates neither ISA nor PNP activation.
ASI_PAR_35	Uses the RAM partition 3 TX to 5 RX (3:5).
ASI_PAR_13	Uses the RAM partition 1 TX to 3 RX (1:3).
ASI_PAR_11	Uses the RAM partition 1 TX to 1 RX (1:1).
ASI_RS	Indicates the RAM speed.
ASI_RW	Indicates the RAM width (which will always be 0 to 8 bits).

`ASI_RSIZE8`	Indicates a RAM size of 8 kilobytes (the default).
`ASI_RSIZE32`	Indicates a RAM size of 32 kilobytes.

2.5 Window 1 Operational Register Offset Definitions

The window 1 operational registers include such registers as the receive
status, the transmit status, and the request interrupt registers, as shown in
Figure 2–3.

Figure 2–3: Window 1 Operational Registers

Register	Constant
Receive Status Register	W1_RXSTAT
Transmit Status Register	W1_TXSTAT
Request Interrupt After Transmit Completion Register	TX_INT
Receive Data Register	W1_RXDATA
Transmit Data Register	W1_TXDATA
Free Transmit Bytes Register	W1_FREETX

ZK-1269U-AI

The following code shows the offset definitions for the window 1 operational
registers:

```
#define W1_RXSTAT            0x8  1
enum w1_rxstat {  2
    RX_IC=0x8000,
    RX_ER=0x4000,
    RX_EM=0x3800,
    RX_EOR=0x0000,
    RX_ERT=0x1800,
    RX_EAL=0x2000,
    RX_ECR=0x2800,
    RX_EOS=0x0800,
    RX_BYTES=0x7ff
};
#define W1_TXSTAT            0xb  3
enum w1_txstat {  4
    TX_CM=0x80,
    TX_IS=0x40,
    TX_JB=0x20,
    TX_UN=0x10,
    TX_MC=0x08,
    TX_OF=0x04,
    TX_RE=0x02
```

```
};
#define TX_INT            0x8000  5
#define W1_RXDATA         0x0     6
#define W1_TXDATA         0x0     7
#define W1_FREETX         0xc     8
```

1   Defines the offset for the receive status register.

2   Defines an enumerated data type called `w1_rxstat`. The `if_el` device
    driver can assign one of the following values to `W1_RXSTAT` (the receive
    status register):

RX_IC	Indicates an incomplete operation.
RX_ER	Indicates an error in the operation.
RX_EM	If any of the bits are set in the mask, indicates that an error has occurred.
RX_EOR	Indicates an overrun error in the operation.
RX_ERT	Indicates a run-time error.
RX_EAL	Indicates an alignment error.
RX_ECR	Indicates a CRC error.
RX_EOS	Indicates an oversize error.
RX_BYTES	Mask used to determine the number of bytes received.

3   Defines the offset for the transmit status register.

4   Defines an enumerated data type called `w1_txstat`. The `if_el`
    device driver can assign one of the following values to `W1_TXSTAT` (the
    transmit status register):

TX_CM	Indicates that the transmission completed.
TX_IS	Indicates that the device should interrupt when a transmission is successfully completed.
TX_JB	Indicates a jabber error.
TX_UN	Indicates an underrun. This is a serious error that requires a reset.
TX_MC	Indicates the maximum number of collisions that occurred.
TX_OF	Indicates an overflow error.
TX_RE	Not currently used.

5   Defines the offset for the request interrupt after completion register.

6   Defines the offset for the receive data register.

7   Defines the offset for the transmit data register.

8 Defines the offset for the free transmit bytes register.

2.6 Window 4 Diagnostic Register Offset Definitions

The window 4 operational registers include such registers as the media type
and status register and the network diagnostic port register, as shown in
Figure 2–4.

Figure 2–4: Window 4 Diagnostic Registers

Register	Constant
Media Type and Status Register	W4_MEDIA
Network Diagnostic and Status Register	W4_NET

ZK-1270U-AI

The following code shows the definitions for the window 4 diagnostic
registers:

```
#define W4_MEDIA              0xa   1
enum w4_media {   2
    MD_TPE   =0x8000,
    MD_COAXE =0x4000,
    MD_RES1  =0x2000,
    MD_SQE   =0x1000,
    MD_VLB   =0x0800,
    MD_PRD   =0x0400,
    MD_JAB   =0x0200,
    MD_UNSQ  =0x0100,
    MD_LBE   =0x0080,
    MD_JABE  =0x0040,
    MD_CS    =0x0020,
    MD_COLL  =0x0010,
    MD_SQEE  =0x0008,
    MD_NCRC  =0x0004
};
#define W4_NET                0x6   3
enum w4_net {   4
    ND_EXT   =0x8000,
    ND_ENDEC =0x4000,
    ND_ECL   =0x2000,
    ND_LOOP  =0x1000,
    ND_TXE   =0x0800,
    ND_RXE   =0x0400,
    ND_TXB   =0x0200,
    ND_TXRR  =0x0100,
    ND_STATE =0x0080,
    ND_REV   =0x003e,
    ND_LOW   =0x0001
};
```

1 Defines the offset for the media type and status register.

2 Defines an enumerated data type called `w4_media`. The `if_el` device driver can assign one of the following values to `W4_MEDIA` (the media type and status register):

`MD_TPE`	Indicates that 10BaseT cable is enabled.
`MD_COAXE`	Indicates that 10Base2 cable is enabled.
`MD_RES1`	Reserved.
`MD_SQE`	Indicates that SQE is present.
`MD_VLB`	Indicates that a valid link beat was detected.
`MD_PRD`	Indicates that polarity reversal was detected.
`MD_JAB`	Indicates that jabber was detected.
`MD_UNSQ`	Indicates unsequelch.
`MD_LBE`	Indicates that link beat was enabled.
`MD_JABE`	Indicates that jabber was enabled.
`MD_CS`	Indicates that carrier sense was detected.
`MD_COLL`	Indicates that collisions occurred.
`MD_SQEE`	Indicates that SQE stats were enabled.
`MD_NCRC`	Indicates that the CRC strip was disabled.

3 Defines the offset for the network diagnostic port register.

4 Defines an enumerated data type called `w4_net`. The `if_el` device driver can assign one of the following values to `W4_NET` (the network diagnostic port register):

`ND_EXT`	Indicates external loopback.
`ND_ENDEC`	Indicates encode/decode (ENDEC) loopback.
`ND_ECL`	Indicates Ethernet controller loopback.
`ND_LOOP`	Indicates FIFO loopback.
`ND_TXE`	Indicates that TX is enabled.
`ND_RXE`	Indicates that RX is enabled.
`ND_TXB`	Indicates that TX is busy.
`ND_TXRR`	Indicates that TX reset is required.
`ND_STATE`	Indicates that statistics are enabled.
`ND_REV`	Indicates the ASIC revision.
`ND_LOW`	Not currently used.

2.7 EEPROM Data Structure Definition

The following code shows the definition for the `w3_eeprom` data structure. This data structure stores information about the 3Com 3C5x9 device.

```
struct w3_eeprom { 1
   unsigned short addr[3];
   unsigned short pid;
   unsigned short mandata[3];
   unsigned short mid;
   unsigned short addrconf;
   unsigned short resconf;
   unsigned short oem[3];
   unsigned short swinfo;
   unsigned short compat;
   unsigned short cs1;
   unsigned short cw2;
   unsigned short res1;
   unsigned int   icw;
   unsigned short swinfo2;
   unsigned short res[2];
   unsigned short cs2;
   unsigned short pnp[40];
};
```

1 Defines an EEPROM data structure called `w3_eeprom`. This data structure has the following members:

addr	Contains the local area network (LAN) address.
pid	Contains the product ID.
mandata	Contains manufacturing data.
mid	Contains the manufacturer ID.
addrconf	Contains the address configuration.
resconf	Contains the resource configuration.
oem	Contains original equipment manufacturer (OEM) address fields.
swinfo	Contains software information.
compat	Contains a compatibility word.
cs1	Contains the first part of the checksum.
cw2	Contains a second compatibility word.
res1	Reserved.
icw	Contains an internal configuration word.
swinfo2	Contains secondary software information.
res	Reserved.
cs2	Contains the second part of the checksum.
pnp	Contains plug-and-play data.

3

Defining the softc Data Structure

All network device drivers define a `softc` data structure to contain the software context of the network device driver and to allow the driver interfaces to share information.

A `softc` data structure contains the following information:

- Common information (Section 3.1)
- Enhanced hardware management (EHM) support (Section 3.2)
- Media state information (Section 3.3)
- Base register definition (Section 3.4)
- Multicast table information (Section 3.5)
- Interrupt handler ID declaration (Section 3.6)
- CSR pointer information (Section 3.7)
- FIFO maintenance information (Section 3.8)
- Bus-specific information (Section 3.9)
- Broadcast flag definition (Section 3.10)
- Debug flag definition (Section 3.11)
- Interrupt and timeout statistics (Section 3.12)
- Autosense kernel thread context information (Section 3.13)
- Polling context flag definition (Section 3.14)
- `w3_eeprom` data structure definition (Section 3.15)
- Simple lock data structure declaration (Section 3.16)

Figure 3–1 shows a typical `softc` data structure.

Figure 3–1: Typical softc Data Structure

*	Common Information
*	Enhanced Hardware Management Information
**	Media State Information
*	Base Register
*	Multicast Table Information
*	Interrupt Handler ID
*	CSR Pointer Information
**	FIFO Maintenance Information
**	Bus-Specific Information
**	Broadcast Flag
**	Debug Flag
**	Interrupt and Timeout Information
**	Autosense Kernel Thread Context Information
**	Polling Context Flag
**	w3_eeprom Structure
*	Simple Lock Structure

ZK-1273U-AI

A single asterisk denotes information that all network device drivers provide
in the associated `softc` data structure, and a double asterisk denotes
information that is specific to the hardware or bus.

3.1 Defining Common Information

The common information in a local area network driver's `softc` data
structure is contained in the `ether_driver` data structure, which consists
of information such as counter blocks, media state, media values, and so
forth. The following code shows the declaration and definition of the common
information in the `if_el` device driver's `el_softc` data structure. Make
sure that the common part of your `softc` data structure has the same
declaration and definitions.

```
struct el_softc {
    struct     ether_driver *is_ed;  1
```

```
#define is_ac    is_ed->ess_ac    2
#define ztime    is_ed->ess_ztime  3
#define ctrblk   is_ed->ess_ctrblk  4
#define is_if    is_ac.ac_if    5
#define is_addr  is_ac.ac_enaddr  6
```

1 Declares an instance of the `ether_driver` data structure and calls
   it `is_ed`. All network drivers must have an `ether_driver` data
   structure. By convention, a pointer to this data structure is the first
   element in the `softc` data structure.

2 Maps the `ess_ac` member of the `ether_driver` data structure to
   the alternate name `is_ac`. The `ess_ac` member is referred to as the
   "Ethernet common part" and is actually an instance of the `arpcom` data
   structure. Figure 3–2 shows the `is_ac` alternate name and associated
   mapping.

3 Maps the `ess_ztime` member of the `ether_driver` data structure to
   the alternate name `ztime`. The `ess_ztime` member stores the time
   counters that were last zeroed. Figure 3–2 shows the `ztime` alternate
   name and associated mapping.

4 Maps the `ess_ctrblk` member of the `ether_driver` data structure
   to the alternate name `ctrblk`. The `ess_ctrblk` member is referred
   to as the "counter block" and is actually an instance of the `estat` data
   structure. Figure 3–2 shows the `ctrblk` alternate name and associated
   mapping.

   You must define this line in your network device driver if you plan to
   use `ADD_RECV_MPACKET`, `ADD_RECV_PACKET`, `ADD_XMIT_MPACKET`,
   and `ADD_XMIT_PACKET` for maintaining LAN device counters. Each of
   these macros references the `ctrblk` alternate name.

5 Maps the `ac_if` member of the `arpcom` data structure to the alternate
   name `is_if`. The `ac_if` member is referred to as the "network-visible
   interface" and is actually an instance of the `ifnet` data structure.
   Figure 3–2 shows the `is_if` alternate name and associated mapping.

6 Maps the `ac_enaddr` member of the `arpcom` data structure to the
   alternate name `is_addr`. The name `ac_enaddr` is actually an alternate
   name for `ac_hwaddr`, which is the name of the actual member of
   the `arpcom` data structure that stores the hardware address. The
   `if_ether.h` file defines the `ac_enaddr` alternate name. Figure 3–2
   shows the `is_addr` alternate name and associated mapping.

Figure 3–2: Mapping Alternate Names

ZK-1274U-AI
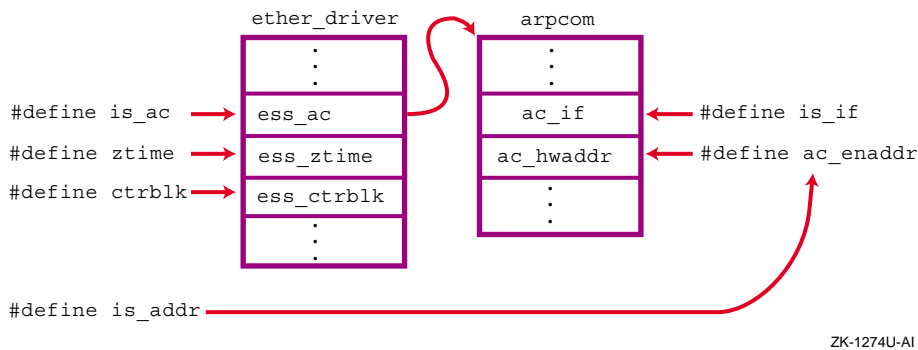
3.2 Enabling Support for Enhanced Hardware Management

Enhanced hardware management (EHM) is a feature of Tru64 UNIX
Version 5.0 that allows a system administrator to view, and possibly modify,
various attributes of the hardware on either a local or a remote system.
To support this facility, device drivers and bus drivers must provide their
specific, predefined attributes to a centralized management entity. Examples
of these attributes for network drivers include the type of LAN device, its
hardware address, the type of media it is attached to, and how fast it can
operate. The LAN subsystem supplies access routines for defining and
exporting these attributes.

To use these routines, a network driver must declare a `net_hw_mgmt` data
structure as shown by the following code:

```
struct net_hw_mgmt    ehm;  1
```

1   Declares a `net_hw_mgmt` data structure and calls it `ehm`.

3.3 Defining Media State Information

The media state information contained in a network driver's `softc` data
structure consists of information about the `lan_media` data structure.
The following code shows the declaration and definition of the media state
information in the `if_el` device driver's `el_softc` data structure:

```
struct  lan_media      lan_media; 1
#define lm_media_mode  lan_media.lan_media_mode 2
#define lm_media_state lan_media.lan_media_state 3
#define lm_media       lan_media.lan_media 4
```

1   Declares a `lan_media` data structure and calls it `lan_media`. The
    `lan_media` data structure contains media state values.

2   Defines an alternate name for referencing the `lan_media_mode`
    member of the `lan_media` data structure. The value that is stored in

`lan_media_mode` usually reflects how the media is to be selected. (In contrast, the value that is stored in the `lan_media` member reflects the current setting of the device.) Typically, you set this member in the driver's `probe` interface to the media mode constant that identifies the mode for the media.

The `lan_common.h` file defines two enumerated data types called `media_types` and `media_modes`. You can set the `lan_media_mode` member to one of the following values, which are defined by the `media_types` and `media_modes` enumerated data types:

`LAN_MEDIA_UTP`	The mode for the media is unshielded twisted-pair cable.
`LAN_MEDIA_BNC`	The mode for the media is thin wire.
`LAN_MEDIA_STP`	The mode for the media is shielded twisted pair cable.
`LAN_MEDIA_FIBER`	The mode for the media is any fiber-based media.
`LAN_MEDIA_AUI`	The mode for the media is the attachment unit interface (AUI).
`LAN_MEDIA_4PAIR`	The mode for the media is four-pair cable.
`LAN_MODE_AUTOSENSE`	The hardware determines the media.

3 Defines an alternate name for referencing the `lan_media_state` member of the `lan_media` data structure. The `lan_media_state` member will be set only if `lan_media_mode` has the value `LAN_MODE_AUTOSENSE`. This member is typically set in the driver's `probe()` routine.

The `lan_media_state` member can be set to one of the following constants, which are defined in the `lan_common.h` file:

`LAN_MEDIA_STATE_SENSING`	The media is currently in the autosensing state.
`LAN_MEDIA_STATE_DETERMINED`	The media state has been determined.

4 Defines an alternate name for referencing the `lan_media` member of the `lan_media` data structure. The `lan_media` member specifies the currently set media.

The value that is stored in the `lan_media` member is valid in the autosense mode only if the `lan_media_state` member is set to the constant `LAN_MEDIA_STATE_DETERMINED`. The value that is stored in `lan_media` reflects the current setting of the device. (In contrast, the value that is stored in the `lan_media_mode` member usually reflects how the media is to be selected.) Typically, you set the `lan_media`

member in the driver's `probe` interface to the media state constant that identifies the state for the media.

You can set the `lan_media` member to the same constants that are listed for the `lan_media_mode` member in item 2.

3.4 Defining the Base Register

The base register in a network driver's `softc` data structure is a member that represents the base register of the device. The following code shows the declaration of the base register in the `if_el` device driver's `el_softc` data structure. Most network device drivers declare a variable to store the device's base register.

```
vm_offset_t basereg; 1
```

1 Declares a base register member and calls it `basereg`.

3.5 Defining Multicast Table Information

All multicast address information in a network driver's `softc` data structure is encapsulated in the `lan_multi` data structure. The following code shows the declaration of the `lan_multi` data structure in the `if_el` device driver's `el_softc` data structure. Most network device drivers declare this data structure in their `softc` data structure.

```
struct lan_multi is_multi; 1
```

1 Declares a `lan_multi` data structure and calls it `is_multi`.

3.6 Defining the Interrupt Handler ID

The interrupt handler ID in a network driver's `softc` data structure is a variable that stores the interrupt handler ID that the `handler_add()` routine returns. The following code shows the declaration of the interrupt handler ID in the `if_el` device driver's `el_softc` data structure. Make sure that the interrupt handler ID part of your `softc` data structure has a similar declaration.

```
ihandler_id_t *hid; 1
```

1 Declares a pointer to an ID that deregisters the interrupt handlers.

3.7 Defining CSR Pointer Information

The control and status register (CSR) addresses in a network driver's `softc` data structure consist of specific adapter register addresses. These registers generally consist of the base register plus some offset, as defined by the network adapter's hardware specification. Make sure that you *never*

access a CSR directly. The driver-specific macros handle the read and write operations that are made on these device registers.

The following code shows the declarations of the CSR addresses in the `if_el` device driver's `el_softc` data structure. Make sure that the CSR pointer information part of your `softc` data structure has similar declarations.

```
io_handle_t regE; 1
io_handle_t regC;
io_handle_t regA;
io_handle_t reg8;
io_handle_t reg6;
io_handle_t reg4;
io_handle_t reg2;
io_handle_t reg0;
io_handle_t data;
```

1 Declares the CSR addresses for the `if_el` driver. These addresses are computed during the `probe()` routine by adding the specified offset (0xE, 0xC, 0xA, and so forth) to the base address.

3.8 Defining FIFO Maintenance Information

The first-in/first-out (FIFO) maintenance information in the `if_el` driver's `el_softc` data structure consists of a variable that stores a value that the device keeps on board. The following code shows its declaration. This information is hardware-specific, so you can omit it from your network device driver's `softc` data structure.

```
unsigned long  txfree;
```

3.9 Defining Bus-Specific Information

The bus-specific information in a network driver's `softc` data structure consists of information about the bus or buses on which the driver operates. The `if_el` driver operates on the PCMCIA and ISA buses, so that the information in this section reflects these buses.

The following code shows the bus-specific declarations in the `if_el` device driver's `el_softc` data structure. The bus-specific information that is described here may not apply to your network device driver. However, the declarations do give you an idea of some of the information that a network driver needs to keep when operating on the PCMCIA and ISA buses.

```
int     irq;    1
int     iobase; 2
int     isa_tag; 3
int     cardout; 4
int     reprobe; 5
int     ispcmcia; 6
struct  card_info *cinfop; 7
```

1 Contains the interrupt request (IRQ) to use.

2   Contains the I/O base address.

3   Contains a tag value that identifies 3Com 3C5x9 devices on an ISA bus.

4   Contains a value that indicates whether the user has ejected the
PCMCIA card.

5   Contains a value that indicates whether the user has reloaded the
PCMCIA card.

6   Contains a value that indicates whether the card is a PCMCIA card.

7   Declares a pointer to the `card_info` data structure and calls it `cinfop`.
The `card_info` data structure contains information that is necessary
to communicate with the kernel PCMCIA subsystem.

3.10 Defining the Broadcast Flag

The broadcast flag in the `if_el` driver's `el_softc` data structure indicates
whether the device should receive broadcast traffic. This flag is specific to
the `if_el` driver and, therefore, is optional in most network device drivers.
The following code shows the declaration of the broadcast flag in the `if_el`
device driver's `el_softc` data structure:

```
int        is_broadcast;  1
```

1   Contains a boolean value. If true, the broadcast address flag is set.

3.11 Defining the Debug Flag

The debug flag in a network driver's `softc` data structure indicates whether
debug mode is on. The following code shows the declaration of the debug
flag in the `if_el` device driver's `el_softc` data structure. The debug flag
is optional.

```
int        debug;  1
```

1   Contains the status of the debug flag. If the `if_flags` member of
the `ifnet` data structure pointer is set to `IFF_DEBUG`, debug is on.
Otherwise, debug is off.

3.12 Defining Interrupt and Timeout Statistics

The interrupt and timeout statistics in the `if_el` driver's `el_softc` data
structure consists of information about timeout and interrupt events.

The following code shows the declarations of the timeout and interrupt
information in the `if_el` device driver's `el_softc` data structure:

```
unsigned long  txreset;   1
unsigned long  xmit_tmo;  2
unsigned long  tint;      3
unsigned long  rint;      4
```

1 Contains the number of transmitter error resets.

2 Contains the number of times that transmit timeouts occurred. The
   `el_watch()` routine increments this member.

3 Contains the count of transmit interrupts.

4 Contains the count of receive interrupts.

3.13 Defining Autosense Kernel Thread Context Information

The autosense kernel thread context information in the `if_el` driver's
`el_softc` data structure consists of information about the kernel thread
that performs the autosense operation. For the `if_el` driver, this kernel
thread is called `el_autosense_thread`.

The following code shows the declarations of the autosense kernel thread
variables in the `if_el` device driver's `el_softc` data structure. The `if_el`
device driver uses kernel threads to perform the tasks that are related to
autosensing the media. However, you can choose other methods instead of
kernel threads.

```
thread_t    autosense_thread;  1
int         autosense_flag;    2
```

1 Contains the autosense kernel thread ID.

2 Contains the autosense kernel thread blocking flag.

3.14 Defining the Polling Context Flag

A LAN driver typically does not need to perform polling operations. However,
the `if_el` driver provides an example of how polling operations might be
accomplished.

The polling context flag in a network driver's `softc` data structure indicates
whether polling is on or off. The following code shows the declaration of the
polling member in the `if_el` device driver's `el_softc` data structure:

```
int         polling_flag;  1
```

1 Declares a polling context flag member called `polling_flag`. This
   member stores a boolean value of 1 (polling context is on) or 0 (polling
   context is off).

3.15 Defining a Copy of the w3_eeprom Data Structure

The w3_eeprom data structure copy in the if_el driver's el_softc data structure consists of information about the hardware-specific w3_eeprom data structure. The following code shows the declaration of this device-specific data structure. If your device has an EEPROM, you might want to save some or all of its contents in your softc data structure.

```
struct w3_eeprom eeprom;  1
```

1 Declares a copy of the w3_eeprom data structure and calls it eeprom.

3.16 Declaring the Simple Lock Data Structure

A network driver's softc data structure contains the declaration of a simple lock data structure. The if_el driver uses a simple lock to protect the data integrity of the el_softc data structure on multiprocessor systems. It also guarantees the sequence of register accesses that a CPU in a multiprocessor system makes to the adapter. See *Writing Kernel Modules* for more information about locking in an SMP environment.

The following code shows the declaration of the simple lock data structure in the if_el driver's el_softc data structure:

```
decl_simple_lock_data(, el_softc_lock)  1
```

1 Uses the decl_simple_lock_data() routine to declare a simple lock data structure as a member of the el_softc data structure. The simple lock data structure is called el_softc_lock.

4

Implementing the Configure Section

The configure section of a network device driver contains the code that incorporates the device driver into the kernel, either statically or dynamically. In a static configuration, the device driver's `configure` interface registers callback routines, which allow the `cfgmgr` framework to configure the driver into the kernel at a specified point during system startup. In a dynamic configuration, the `configure` interface cooperates with the `cfgmgr` framework to handle user-level requests to dynamically configure, reconfigure, and query a network device driver at run time.

Because these tasks are common to all network drivers, the code has been consolidated into a single routine called `lan_configure()`. Routines with the prefix `lan_` reside in the `lan_common.c` source file. A network driver's `configure()` routine can simply call `lan_configure()` to carry out the following tasks:

- `CFG_OP_CONFIGURE`
- `CFG_OP_RECONFIGURE`
- `CFG_OP_UNCONFIGURE`
- `CFG_OP_QUERY`

The `if_el` driver's `configure` section contains an attributes data structure and the `el_configure()` routine.

The following sections describe how to initialize the `cfg_subsys_attr_t` data structure and how to set up the `el_configure()` routine:

- Declaring configure-related variables and initializing the `cfg_subsys_attr_t` data structure (Section 4.1)
- Setting up the `el_configure()` routine (Section 4.2)

4.1 Declaring Configure-Related Variables and the cfg_subsys_attr_t Data Structure

As part of implementing a device driver's `configure` interface, you declare a number of variables and initialize the `cfg_subsys_attr_t` data structure.

The following code shows the declaration of the variables and the
initialization of the `cfg_subsys_attr_t` data structure for the `if_el`
device driver:

```
static unsigned char el_pcmcia_optiondata[400] = "";   1
static unsigned char el_isa_optiondata[300] = "";   2
static unsigned char el_unused[300] = "";
static int el_polling = 0;   3
static int el_pollint = 16;   4
static int el_configured = 0;   5
static struct lan_config_data el_data = {   6
   LAN_CONFIG_VERSION_ID,
   0,
   &eldriver,
   &el_configured
};

cfg_subsys_attr_t el_attributes[] = {   7
   {"PCMCIA_Option", CFG_ATTR_STRTYPE, CFG_OP_CONFIGURE | CFG_OP_QUERY,
                     (caddr_t)el_pcmcia_optiondata, 0, 400, 0},   8
   {"ISA_Option", CFG_ATTR_STRTYPE, CFG_OP_CONFIGURE | CFG_OP_QUERY,
                     (caddr_t)el_isa_optiondata, 0, 300, 0},   9
   {"Polling", CFG_ATTR_INTTYPE, CFG_OP_QUERY | CFG_OP_CONFIGURE,
                     (caddr_t)&el_polling, 0, 1, sizeof(int)},   10
   {"Polls_Per_Second", CFG_ATTR_INTTYPE, CFG_OP_QUERY | CFG_OP_CONFIGURE,
                     (caddr_t)&el_pollint, 10, 100, sizeof(int)},   11
   {"", 0, 0, 0, 0, 0, 0}   12
};
```

1 Declares a character array called `pcmcia_optiondata` and initializes
   it to the null string. The `pcmcia_optiondata` character array is
   where the `cfgmgr` framework stores the value for the `PCMCIA_Option`
   attribute. The `cfgmgr` framework obtains this value from the
   `/etc/sysconfigtab` database.

2 Declares a character array called `isa_optiondata` and initializes it
   to the null string. The `isa_optiondata` character array is where the
   `cfgmgr` framework stores the value for the `ISA_Option` attribute. The
   `cfgmgr` framework obtains this value from the `/etc/sysconfigtab`
   database.

3 Declares an integer variable called `el_polling` and initializes it to
   the value 0 (zero). The `el_polling` variable is where the `cfgmgr`
   framework stores the style of interrupt processing for the `Polling`
   attribute. The `cfgmgr` framework obtains this value from the
   `/etc/sysconfigtab` database.

4 Declares an integer variable called `el_pollint` and initializes it to the
   value 16. The `el_pollint` variable is where the `cfgmgr` framework
   stores the polls per second for the `Polls_Per_Second` attribute. The
   `cfgmgr` framework obtains this value from the `/etc/sysconfigtab`
   database.

5 Declares an integer variable called `el_configured` and initializes it
   to the value 0 (zero). The driver must increment this variable for each
   successfully configured `el` device.

6   Declares the `lan_config_data` structure, which contains all information specific to the `el` driver. The `lan_configure` common code uses this structure.

7   Declares an array of `cfg_subsys_attr_t` data structures and calls it `el_attributes`.

8   Describes the `PCMCIA_Option` attribute, which specifies the option data for the PCMCIA bus.

9   Describes the `ISA_Option` attribute, which specifies the option data for the ISA bus.

10   Describes the `Polling` attribute, which is specific to this device driver. It indicates the style of interrupt processing. The operation code specifies `CFG_OP_CONFIGURE` and `CFG_OP_QUERY`. This means that the attribute can only be set at configuration time and, after that, only queried. You can specify a value in the `sysconfigtab` file fragment (which is appended to the `/etc/sysconfigtab` database). The `cfgmgr` framework obtains this value from the `/etc/sysconfigtab` database and stores it in the `el_polling` variable.

11   Describes the `Polls_Per_Second` attribute, which is specific to this device driver. It indicates the polls per second for interrupt processing. Similar to the `Polling` attribute, you can only specify a value for this attribute at configuration time. The `cfgmgr` framework obtains this value from the `/etc/sysconfigtab` database and stores it in the `el_pollint` variable.

12   Ends the array by specifying the null string.

4.2 Setting Up the el_configure Routine

The following code shows how to set up the `el_configure()` routine:

```
int el_configure(cfg_op_t op,  1
                  cfg_attr_t *indata,  2
                  size_t indatalen,  3
                  cfg_attr_t *outdata,  4
                  size_t outdatalen)  5

{
  return (lan_configure (op, &el_data));  6
}
```

1   Declares an argument called `op` to contain a constant that describes the configuration operation to be performed on the driver. This argument evaluates to one of the following valid constants: `CFG_OP_CONFIGURE`, `CFG_OP_UNCONFIGURE`, `CFG_OP_QUERY`, or `CFG_OP_RECONFIGURE`.

2   Declares a pointer to a `cfg_attr_t` data structure called `indata`, which consists of input to the `el_configure()` routine. The `cfgmgr` framework fills in this data structure. The `cfg_attr_t` data structure

represents a variety of information, including the `if_el` driver's interrupt polling requirements.

3 Declares an argument called `indatalen` to store the size of this input data structure. This argument represents the number of `cfg_attr_t` data structures included in `indata`.

4 Declares an argument for user-defined configuration operations, which can occur when the `cfgmgr` framework calls the driver's `configure` interface with the `CFG_OP_USERDEFINED` operation code. Typically, this argument is not used.

5 Declares the size of the `outdata` argument. Typically, this argument is not used.

6 Calls the LAN common driver code to configure the device (either statically or dynamically).

5

Implementing the Autoconfiguration Support Section (probe)

The autoconfiguration support section contains the code that implements a network device driver's `probe` interface. A network device driver's `probe` interface determines whether the network device exists and is functional on a given system. The bus configuration code calls the driver's `probe` interface.

The `if_el` driver operates on the ISA and PCMCIA bus. For the PCMCIA bus, it provides a driver-specific routine that is called when a user removes the card from the slot. For the ISA bus, the driver provides routines to reset, activate, and read from hardware registers. These routines are specific to the `if_el` device driver. To learn how the driver handles these tasks, see the source listing in the examples directory that is installed with the device driver kit.

The following sections describe how to use the `probe` interface:

- Implementing the `el_probe()` routine (Section 5.1)

- Implementing the `el_shutdown()` routine (Section 5.2)

- Implementing the `el_autosense_thread()` routine (Section 5.3)

5.1 Implementing the el_probe Routine

The `el_probe()` routine performs the following tasks:

- Checks the maximum number of devices that the driver supports (Section 5.1.2)

- Performs bus-specific tasks (Section 5.1.3)

- Allocates memory for the `softc` and `ether_driver` data structures (Section 5.1.4 and Section 5.1.5)

- Initializes the enhanced hardware management data structure (Section 5.1.6)

- Computes the control and status register addresses (Section 5.1.7)

- Sets bus-specific data structure members (Section 5.1.8)

- If this is the first time the device has been probed, copies data from the EEPROM, reads and saves the device's physical address and starts the autosense kernel thread to determine the media type (Section 5.1.9)

- For subsequent probe operations, reads the EEPROM to determine if the hardware address (and thus the adapter) has changed (Section 5.1.10)

- Registers the interrupt handler (Section 5.1.11)

- Saves the `controller` and `softc` data structure pointers (Section 5.1.12)

- Tries to allocate another `controller` data structure (Section 5.1.13)

- Registers the shutdown routine (Section 5.1.14)

5.1.1 Setting Up the el_probe Routine

The following code shows how to set up the `el_probe()` routine:

```
static int el_probe (io_handle_t io_handle,  1
                     struct controller *ctlr)  2
{
    struct el_softc *sc;  3
    int unit = ctlr->ctlr_num, i, j, isatag=0, status, multi_func_flag=0;  4
    struct handler_intr_info el_intr_info;  5
    ihandler_t el_ihandle;  6
    struct card_info *card_infop = (struct card_info *)(ctlr->card_info_ptr);  7
    io_handle_t reg;  8
    struct e_port port_sel;  9
    struct irq irq_sel;  10
    unsigned short *ed;
    unsigned char *ee;
    struct tuple_info *tuple_infop;  11
    struct tuple_data_info tuple_data;
    struct tuple_data_info *tuple_data_infop;
```

1 Declares an argument that specifies an I/O handle that you can use to reference a device register or memory that is located in bus address space (either I/O space or memory space). This I/O handle references the device's I/O address space for the bus where the read operation originates (in calls to the `read_io_port()` routine) and where the write operation occurs (in calls to the `write_io_port()` routine). The bus configuration code passes this I/O handle to the driver's `probe` interface during device autoconfiguration.

2 Declares a pointer to a `controller` data structure for this controller. This data structure contains such information as the controller type, the controller name, and the current status of the controller. The bus configuration code passes this initialized `controller` data structure to the driver's `probe` and `attach` interfaces. A device driver typically uses the `ctlr_num` member of the `controller` data structure as an index to identify the instance of the controller a request is for.

3 Declares a pointer to the `el_softc` data structure and calls it `sc`.

4 Declares a `unit` variable and initializes it to the controller number for this controller. This controller number identifies the specific 3Com 3C5x9 controller that is being probed. The controller number is

contained in the `ctlr_num` member of the `controller` data structure for this 3Com 3C5x9 device.

5 Declares a `handler_intr_info` data structure called `el_intr_info`. The `handler_intr_info` data structure is a generic data structure that contains interrupt handler information for buses that are connected to a device controller. Using the `handler_intr_info` data structure makes the driver more portable across different bus architectures.

6 Declares an `ihandler_t` data structure called `el_ihandle` to contain information for the `if_el` device driver's interrupt service routine registration.

7 Declares a pointer to a `card_info` data structure called `card_infop` and initializes it to the specific `card_info` data structure for this controller. This data structure is associated with PCMCIA devices only.

The bus configuration code passes this `card_info` data structure through the `controller` data structure's `conn_priv[2]` member. The `pcmcia.h` file defines `card_info_ptr` as `conn_priv[2]`.

8 Declares a variable called `reg` that stores the I/O handle that is passed to the driver's `el_probe()` routine.

9 Declares an `e_port` data structure called `port_sel`. This data structure is associated with the EISA and ISA buses. The `e_port` data structure describes bus I/O port information. The bus configuration code initializes the members of the `e_port` data structure during device autoconfiguration. Device drivers call the `get_config()` routine to obtain information from the members of the `e_port` data structure.

10 Declares an `irq` data structure called `irq_sel`. The `irq` data structure specifies EISA/ISA bus interrupt channel characteristics that are assigned to a device. The bus configuration code initializes the members of the `irq` data structure during device autoconfiguration. Device drivers call the `get_config()` routine to obtain information from the members of the `irq` data structure.

11 Declares the `tuple_*` data structures. For more information and definitions of the `tuple_info` and `tuple_data_info` data structures, see the `/usr/sys/include/io/dec/pcmcia/cardinfo.h` file and `tuple_info()` and `tuple_data_info()`.

5.1.2 Checking the Maximum Number of Devices That the Driver Supports

The following code shows how to check for the maximum number of devices that the `if_el` device driver supports:

```
if (unit >= el_MAXDEV) { 1
    printf("el%d: el_probe: unit exceeds max supported devices\n",
           unit);
    return(0); 2
}
```

1 If the `unit` variable exceeds the maximum number of devices that the `if_el` driver supports, calls the `printf()` routine to display an appropriate message on the console terminal. The `printf()` routine also displays the controller number that is stored in the `unit` variable. The `el_probe()` routine stores the controller number in this variable by referencing the `ctlr_num` member of the `controller` data structure pointer.

The `el_MAXDEV` constant defines the maximum number of controllers that the `if_el` driver can support.

2 Returns the value 0 (zero) to indicate that the probe operation failed.

5.1.3 Performing Bus-Specific Tasks

The following code shows how the `el_probe()` routine performs tasks that are specific to the PCMCIA and ISA buses. Only network device drivers that operate on the PCMCIA and ISA buses perform these tasks. Your `probe` interface performs tasks that are related to the bus on which your network driver operates. See the bus-specific manual for information on data structures for that bus.

```
switch (ctlr->bus_hd->bus_type) { 1
 case BUS_PCMCIA: 2

    reg = io_handle+card_infop->io_addr[0]; 3

    multi_func_flag = card_infop->card_option->multi_func_flag; 4

    if (!multi_func_flag)
    {

    if (READ_BUS_D16(reg+W0_MID) != 0x6d50) { 5

        WRITE_BUS_D16(reg+CMD_PORT, CMD_RESET);
        DELAY(1000);

        if (READ_BUS_D16(reg+W0_MID) != 0x6d50) { 6
            printf("el%d: EtherLink III not found on bus\n", unit);
            return(0); 7
        }
      }
    }
    break;
```

```
    case BUS_ISA:  8
        if (get_config(ctlr, RES_PORT, NULL, &port_sel, 0) >= 0) {  9
            reg = port_sel.base_address;  10
        } else {  11
            printf("el%d: Can't get assigned IOBASE\n",unit);
            return(0);
        }

        if (get_config(ctlr, RES_IRQ, NULL, &irq_sel, 0) < 0) {  12
            printf("el%d: Can't get assigned IRQ\n", unit);
            return(0);
        }

        if (el_isa_reset++ == 0)  13
          el_isa_reset_all(reg, &isatag, ctlr);

        if (el_isa_activate(reg, &isatag, ctlr)) {  14
            printf("el%d: 3C509 not present or not responding at 0x%x\n",
                    unit, reg);
            return(0);
        }
        break;

  default:  15
      printf("el%d: Unrecognized bus type\n", unit);
      return(0);
      break;
 }
```

1 Determines which bus the `if_el` driver operates on by examining the constant that the bus configuration code has stored in the `bus_type` member. The `el_probe()` routine references this value through the `controller` data structure pointer's `bus_hd` member. This pointer is the data structure that is associated with this 3Com 3C5x9 device.

2 Performs tasks related to the PCMCIA bus if `bus_type` evaluates to the constant `BUS_PCMCIA`.

3 Adds the I/O handle to the base address of the card and stores it in the `reg` variable. The `reg` variable becomes an argument in subsequent calls to the read and write macros.

4 Determines whether the card is a multifunction card or a single-function card.

5 Calls the `READ_BUS_D16` macro to read a word (16 bits) from a device register that is located in the bus I/O address space. This read operation verifies that the EtherLink III card is attached.

   If the data that `READ_BUS_D16` returns is not equal to 0x6d50, calls the `WRITE_BUS_D16` and `DELAY` macros. The `WRITE_BUS_D16` macro writes a word (16 bits) to a device register that is located in the bus I/O address space. This specific write operation resets the card. The `DELAY` macro spins, waiting the specified number of microseconds before continuing execution.

6   Calls the `READ_BUS_D16` macro a second time to determine whether the EtherLink III is attached. If the data returned by `READ_BUS_D16` is not 0x6d50, calls the `printf()` routine to display an appropriate message on the console terminal.

7   Returns the value 0 (zero) to indicate that the probe operation failed.

8   Performs tasks related to the ISA bus if `bus_type` evaluates to the constant `BUS_ISA`.

9   Calls the `get_config()` routine to obtain the base I/O address for the device.

10   If `get_config()` is successful, stores the base I/O address in the `reg` variable.

11   If `get_config()` is unsuccessful, calls the `printf()` routine to display an appropriate message on the console terminal, then returns the value 0 (zero) to indicate that the probe operation failed.

12   Calls the `get_config()` routine to obtain the interrupt request (IRQ) line for the device. If `get_config()` is not successful, `el_probe()` calls the `printf()` routine to display an appropriate message on the console terminal, then returns the value 0 (zero) to indicate that the probe operation failed.

13   If this is the first ISA 3Com 3C5x9 adapter seen in the system, calls the `el_isa_reset_all()` routine to reset all 3Com 3C5x9 adapters on the ISA bus once to clear any bad state data.

14   Calls the `el_isa_activate()` routine to attempt to activate the lowest addressed adapter on the bus and to configure it with the given base address. If the attempt fails, `el_probe()` calls the `printf()` routine to display an appropriate message on the console terminal, then returns the value 0 (zero) to indicate that the probe operation failed. See the `if_el` source file (in the examples directory that is installed with the device driver kit) for a listing of the `el_isa_activate()` routine.

15   If the driver is not operating on either the PCMCIA or ISA bus, calls the `printf()` routine to display an appropriate message on the console terminal, then returns the value 0 (zero) to indicate that the probe operation failed.

5.1.4 Allocating Memory for the softc Data Structure

The following code shows how the `el_probe()` routine allocates memory for the `if_el` device driver's `softc` data structure. If the device has already been probed, the driver does not need to allocate the data structure. This can happen if the user removed and then reinserted the device, an operation that is only possible for PCMCIA versions of the adapter.

```
    if (el_softc[unit]) {  1
        sc = el_softc[unit];
        sc->cardout = 0;
        sc->reprobe = 1;
    } else {  2
        MALLOC(sc, void*, sizeof(struct el_softc), M_DEVBUF, M_WAIT | M_ZERO);
        if (!sc) {  3
            printf("el%d: el_probe: failed to get buffer memory for softc\n",
                    unit);
            return(0);  4
        }
    }
```

1. If the user removed and returned the PCMCIA card to its slot:

   - Locates the existing `el_softc` data structure for this device. The
     controller number (which is stored in the `unit` variable) is used as
     an index into the array of `el_softc` data structures to determine
     which `el_softc` data structure is associated with this 3Com 3C5x9
     device.

   - Sets the `cardout` member of the `el_softc` data structure to the
     value 0 (zero) to indicate that the PCMCIA card is not currently
     removed from its slot.

   - Sets the `reprobe` member of the `el_softc` data structure to the
     value 1 to indicate that the PCMCIA card was reinserted into its slot.

2. If this is an ISA device or if the user did not remove and replace the
   card, calls the `MALLOC` macro to allocate memory for the `el_softc`
   data structure.

3. If `MALLOC` could not allocate the memory, calls the `printf()` routine to
   display an appropriate message on the console terminal. The `printf()`
   routine also displays the controller number for the device.

4. Returns the value 0 (zero) to the bus configuration code to indicate
   that the probe operation failed.

5.1.5 Allocating the ether_driver Data Structure

The following code shows how the `el_probe()` routine calls `if_alloc()`
to allocate the `ether_driver` data structure for this device. `if_alloc()`
returns an `ether_driver` data structure, which contains the `ifnet` data
structure, and initializes the `if_name`, `if_unit`, and `if_index` fields.
Make sure that your driver allocates its `ether_driver` data structure in
the same way.

```
    sc->is_ed = if_alloc("el", unit, sizeof(struct ether_driver));  1
    CLEAR_LAN_COUNTERS(sc->is_ed);  2
```

1. Calls a routine that returns an `ether_driver` data structure and
   initializes the `ifnet` portion of it.

|2| Initializes all Ethernet statistics counters in the `ether_driver` data structure to 0 (zero).

5.1.6 Initializing the Enhanced Hardware Management Data Structure

The following code shows how the `el_probe()` routine initializes the data structure for enhanced hardware management (EHM) support:

```
    lan_ehm_init(&sc->ehm, NET_EHM_VERSION_ID);  1
}
```

|1| Initializes the `net_hw_mgmt` data structure. This data structure contains the current and default attribute values for this device as well as other information that EHM requires. The `lan_ehm_init()` routine allocates all necessary storage and performs basic initialization of the EHM data structure. Make sure that your driver makes this call as well.

5.1.7 Computing the CSR Addresses

The following code shows how the `el_probe()` routine determines the addresses of the `if_el` device's control and status (CSR) registers:

```
    sc->regE = reg+0xe;  1
    sc->regC = reg+0xc;  2
    sc->regA = reg+0xa;
    sc->reg8 = reg+0x8;
    sc->reg6 = reg+0x6;
    sc->reg4 = reg+0x4;
    sc->reg2 = reg+0x2;
    sc->reg0 = reg+0x0;
    sc->data = reg+0x0;

    sc->basereg = reg;  3
```

|1| Fills in the `regE` member of the `el_softc` data structure for this 3Com 3C5x9 device. The value that is stored in `regE` consists of the I/O handle plus a byte offset. The `el_probe()` routine computes this address according to the requirements of the PCMCIA bus and the ISA bus.

|2| This line and the subsequent lines compute and save other `if_el` device register addresses in the `el_softc` data structure.

|3| Stores the I/O handle in the `basereg` member of the `el_softc` data structure for this 3Com 3C5x9 device.

5.1.8 Setting Bus-Specific Data Structure Members

The following code shows how the `el_probe()` routine sets members for the bus-specific data structures that are associated with the PCMCIA and ISA buses. See the bus-specific manual for information on data structures for the bus on which your driver operates.

```
    switch (ctlr->bus_hd->bus_type) {  1
      case BUS_PCMCIA:  2
```

```
        sc->irq = 3;  3
        sc->iobase = 0;  4

        sc->ispcmcia = 1;  5
        sc->cinfop =card_infop;

        pcmcia_register_event_callback(card_infop->socket_vnum,  6
                                       CARD_REMOVAL_EVENT,
                                       (caddr_t)el_card_remove,
                                       (caddr_t)sc);

        if (multi_func_flag)  7
           lan_set_attribute(sc->ehm.current_val, NET_MODEL_NDX, "3C562");
        else
           lan_set_attribute(sc->ehm.current_val, NET_MODEL_NDX, "3C589");
        break;

    case BUS_ISA:  8
        sc->irq = irq_sel.channel;  9

        sc->isa_tag = isatag;  10

        sc->iobase = ((reg-0x200)/0x10)&0x1f;  11

        lan_set_attribute(sc->ehm.current_val, NET_MODEL_NDX, "3C509");  12
        break;
    }
```

1 Evaluates the `bus_type` member of the `bus` data structure for this
   3Com 3C5x9 device.

2 Performs tasks that are related to the PCMCIA bus if `bus_type`
   evaluates to `BUS_PCMCIA`.

3 Sets the interrupt request (IRQ) to the value 3.

4 Sets the I/O base of the program card to the value 0 (zero).

5 Indicates that this is a PCMCIA unit and saves the card information
   pointer.

6 Calls the `pcmcia_register_event_callback()` routine. See the
   `if_el` source file (in the examples directory that is installed with the
   device driver kit) for a listing of this routine.

7 Sets the model identification attribute for enhanced hardware
   management support.

8 Performs tasks that are related to the ISA bus.

9 Saves the interrupt request (IRQ) from the ISA bus configuration code.

10 Saves the tag from the activation process.

11 Computes the I/O base to give to the device.

12 Sets the model identification attribute for enhanced hardware
   management support.

5.1.9 Handling First-Time Probe Operations

If the device has not already been probed, the `el_probe()` routine performs
the following tasks:

- Reads the EEPROM and saves it to a temporary data structure

- Reads and saves the device's physical address

- Starts the autosense thread to determine the media type

The following code shows how the `el_probe()` routine performs these
tasks:

```
if (!sc->reprobe) {  1

    if (multi_func_flag) {  2

        bzero((caddr_t)&tuple_data, sizeof(struct tuple_data_info));
        tuple_data_infop = &tuple_data;
        tuple_infop = (struct tuple_info *)&tuple_data;
        tuple_infop->socket = (short) card_infop->socket_vnum;
        tuple_infop->attributes = 0;
        tuple_infop->DesiredTuple = 0x88;
        status = GetFirstTuple(tuple_infop);

        if (status == SUCCESS) {
            tuple_data_infop->TupleOffset = 0;
            tuple_data_infop->TupleDataMax = (u_short)TUPLE_DATA_MAX;
            status = GetTupleData(tuple_data_infop);
            if (status == SUCCESS) {
                ee = (unsigned char *)&sc->eeprom;
                for (i = 0; i < (sizeof(struct w3_eeprom)); i++) {
                    *ee = tuple_data_infop->TupleData[i];
                    ee++;
                }
            } else {
                printf("el%d: Can't read multifunction card's eeprom.\n",
                        unit);
                if (sc->ispcmcia)
                    pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                                      CARD_REMOVAL_EVENT,
                                                      (caddr_t)el_card_remove);
                if_dealloc(sc->is_ed);
                lan_ehm_free(&sc->ehm);
                FREE(sc, M_DEVBUF);
                return(0);
            }
        } else {
            printf("el%d: Can't read multifunction card's eeprom.\n",
                    unit);
            if (sc->ispcmcia)
                pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                                  CARD_REMOVAL_EVENT,
                                                  (caddr_t)el_card_remove);
            if_dealloc(sc->is_ed);
            lan_ehm_free(&sc->ehm);
            FREE(sc, M_DEVBUF);
            return(0);
        }
    } else {  3

        ed = (unsigned short *)&sc->eeprom;
```

```
    for (i=0; i<(sizeof(struct w3_eeprom)/2); i++) {
        WRITE_ECR(sc, ECR_READ+i);
        DELAY(1000);
        *ed = READ_EDR(sc);
        ed++;
    }
}

    for (i=0; i<3; i++) {  4
        j = sc->eeprom.addr[i];
        sc->is_addr[(i*2)] = (j>>8) & 0xff;
        sc->is_addr[(i*2)+1] = (j) & 0xff;
    }

    sc->lm_media_mode = LAN_MODE_AUTOSENSE;  5
    sc->lm_media_state = LAN_MEDIA_STATE_SENSING;  6
    sc->lm_media = LAN_MEDIA_UTP;  7
    sc->autosense_thread = kernel_thread_w_arg(first_task,  8
                                               el_autosense_thread,
                                               (void *)sc);
    if (sc->autosense_thread == NULL) {  9
        printf("el%d: Can't create autosense thread.\n", unit);
        if (sc->ispcmcia)  10
            pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                             CARD_REMOVAL_EVENT,
                                             (caddr_t)el_card_remove);
        if_dealloc(sc->is_ed);  11
        lan_ehm_free(&sc->ehm);  12
        FREE(sc, M_DEVBUF);  13
        return(0);  14
    }
```

1 Determines whether the device has already been probed, which
   indicates that the device is operating on a PCMCIA bus and that the
   user has put the card back into the slot. In this case, the driver does
   not need to redo much of the initial probe work and will skip to the code
   shown in Section 5.1.10.

2 If this is a multifunction card, reads the EEPROM data and saves it in
   `sc->eeprom`. If this is a multifunction PC card, the EEPROM data is
   located in the card information data structure.

3 If this is not a multifunction PC card, the EEPROM data is read directly
   from the card and saved in the `el_softc` data structure.

4 Saves the 48-bit physical address of the device into the `is_addr`
   member of the `el_softc` data structure for this 3Com 3C5x9 device.

5 Sets the media mode to the constant `LAN_MODE_AUTOSENSE`. This
   constant indicates that the driver hardware determines the media
   automatically.

6 Sets the media state to the constant `LAN_MEDIA_STATE_SENSING`. This
   constant indicates that the media is currently in the autosensing state.

7 Sets the currently set media to the constant `LAN_MEDIA_UTP`.
   This constant indicates that the mode for the media is unshielded
   twisted-pair cable.

8 Calls the `kernel_thread_w_arg()` routine to create and start a
   kernel thread with timeshare scheduling. A kernel thread that is
   created with timeshare scheduling means that its priority degrades if it
   consumes an inordinate amount of CPU resources. Make sure that your
   device driver calls `kernel_thread_w_arg()` only for long-running
   tasks and always attaches a kernel thread to the first task.

   The `kernel_thread_w_arg()` routine returns a pointer to the `thread`
   data structure for the newly created thread. The device driver stores
   this pointer in the `autosense_thread` member of the `el_softc` data
   structure.

9 If the value that `kernel_thread_w_arg()` returns is NULL, then the
   thread could not be created. At this point, the `el_probe()` routine
   must undo previous work and return a failure indication to the caller.

10 For PCMCIA versions of the card, unregisters the callback routine that
    was previously registered.

11 Deallocates the `ether_driver` data structure for this device.

12 Frees up any memory that was allocated for enhanced hardware
    management and unregisters this card from the hardware management
    database.

13 Calls the `FREE` macro, which frees the memory that was previously
    allocated for the `el_softc` data structure.

14 Returns the value 0 (zero) to indicate that the probe operation failed.

5.1.10 Handling Subsequent Probe Operations

If the device had already been probed, the `if_el` device driver reads the
EEPROM to determine whether the hardware address has changed. The
following code shows how the `el_probe()` routine performs these tasks:

```
} else {
    struct w3_eeprom ee_copy;
    unsigned char tmp_addr[8];
    struct ifreq ifr;
    struct ifnet *ifp = &sc->is_if

    if (multi_func_flag) {  1

        bzero((caddr_t)&tuple_data, sizeof(struct tuple_data_info));
        tuple_data_infop = &tuple_data;
        tuple_infop = (struct tuple_info *)&tuple_data;
        tuple_infop->socket = (short) card_infop->socket_vnum;
        tuple_infop->attributes = 0;
        tuple_infop->DesiredTuple = 0x88;
        status = GetFirstTuple(tuple_infop);

        if (status == SUCCESS) {
            tuple_data_infop->TupleOffset = 0;
            tuple_data_infop->TupleDataMax = (u_short)TUPLE_DATA_MAX;
            status = GetTupleData(tuple_data_infop);
```

```
        if (status == SUCCESS) {
            ee = (unsigned char *)&ee_copy;
            for (i = 0; i < (sizeof(struct w3_eeprom)); i++) {
                *ee = tuple_data_infop->TupleData[i];
                ee++;
            }
        } else {
            printf("el%d: Can't read multifunction card's eeprom.\n",
                    unit);
            if (sc->ispcmcia)
            pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                              CARD_REMOVAL_EVENT,
                                              (caddr_t)el_card_remove);
            return(0);
        }
    } else {
        printf("el%d: Can't read multifunction card's eeprom.\n",
                unit);
        if (sc->ispcmcia)
        pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                          CARD_REMOVAL_EVENT,
                                          (caddr_t)el_card_remove);
        return(0);
    }
    } else { 2

 ed = (unsigned short *)&ee_copy;
 for (i=0; i<(sizeof(struct w3_eeprom)/2); i++) {
     WRITE_ECR(sc, ECR_READ+i);
     DELAY(1000);
     *ed = READ_EDR(sc);
     ed++;
 }
}

 if (bcmp(sc->eeprom.addr, ee_copy.addr, 6)) { 3

     for (i=0; i<3; i++) { 4
         j = sc->eeprom.addr[i];
         tmp_addr[(i*2)] = (j>>8) & 0xff;
         tmp_addr[(i*2)+1] = (j) & 0xff;
     }

     if (bcmp(tmp_addr, sc->is_addr, 6) == 0) { 5

         for (i=0; i<3; i++) { 6
             j = ee_copy.addr[i];
             tmp_addr[(i*2)] = (j>>8) & 0xff;
             tmp_addr[(i*2)+1] = (j) & 0xff;
         }

         bzero(&ifr, sizeof(struct ifreq));
         bcopy(tmp_addr, ifr.ifr_addr.sa_data, 6);
         bcopy(tmp_addr, sc->is_addr, 6); 7
         if (((struct arpcom *)ifp)->ac_flag & AC_IPUP) { 8
             rearpwhohas((struct arpcom *)ifp);
         }
         if_sphyaddr(ifp, &ifr); 9
         pfilt_newaddress(sc->is_ed.ess_enetunit, sc->is_addr); 10
     }
```

```
        bcopy(&ee_copy, &sc->eeprom, sizeof(struct w3_eeprom)); 11
    }
}
```

1   If this is a multifunction card, reads the EEPROM data and saves it in a temporary data structure, `ee_copy`. If this is a 3Com 3C562 multifunction PC card, the EEPROM data is located in the card information data structure.

2   If this is not a multifunction PC card, the EEPROM data is read directly from the card and saved in the `el_sofc` data structure.

3   Calls the `bcmp()` routine to compare the EEPROM address from the first probe operation to the EEPROM address of the current probe operation.

4   If the EEPROM address has changed, converts the original EEPROM address to its canonical form.

5   Compares the original EEPROM address to the hardware address that is currently in effect. If they are different, then a previously specified hardware address was used that was different from the address that was found in the EEPROM. In this case, the alternate address is still in effect and no further action needs to be taken.

6   If the original EEPROM address is the same as the hardware address that is currently in effect, uses the hardware address that was found in the EEPROM. Because the EEPROM has changed (because the old `if_el` adapter was removed and a new one inserted), it will be necessary to broadcast the new EEPROM hardware address onto the network to inform the network that there has been a change. This section of code converts the hardware address from the current EEPROM to canonical form in preparation for the broadcast message.

7   Saves the new hardware address in the `is_addr` member of the `el_softc` data structure.

8   If an IP address has been configured for this interface, informs the network that there is a new hardware address for the IP address by sending out an ARP packet.

9   Marks this new hardware address as the link address for this interface.

10   Informs the packet filter of the new hardware address.

11   Saves the EEPROM contents in the `el_softc` data structure.

5.1.11 Registering the Interrupt Handler

The following code shows how the `el_probe()` routine registers the interrupt handler. The *Writing Device Drivers* manual provides detailed information on the data structures and routines that relate to the

registration of interrupt handlers. All network device drivers are required to register interrupt handlers.

```
el_intr_info.configuration_st = (caddr_t)ctlr;  1
el_intr_info.intr = el_intr;  2
el_intr_info.param = (caddr_t)unit;  3
el_intr_info.config_type = CONTROLLER_CONFIG_TYPE;  4
if (ctlr->bus_hd->bus_type == BUS_PCMCIA)  5
    el_intr_info.config_type |= SHARED_INTR_CAPABLE;

el_ihandle.ih_bus = ctlr->bus_hd;  6
el_ihandle.ih_bus_info = (char *)&el_intr_info;  7

sc->hid = handler_add(&el_ihandle);  8
if (sc->hid == (ihandler_id_t *)(NULL)) {  9
    printf("el%d: interrrupt handler add failed\n", unit);
    if (sc->ispcmcia)
        pcmcia_unregister_event_callback(card_infop->socket_vnum,
                                  CARD_REMOVAL_EVENT,
                                  (caddr_t)el_card_remove);
    if_dealloc(sc->is_ed);
    lan_ehm_free(&sc->ehm);
    FREE(sc, M_DEVBUF);
    return(0);
}
```

1 Sets the `configuration_st` member of the `el_intr_info` data structure to the pointer to the `controller` data structure for this 3Com 3C5x9 device.

2 Sets the `intr` member of the `el_intr_info` data structure to `el_intr`, which is the `if_el` device driver's interrupt handler.

3 Sets the `param` member of the `el_intr_info` data structure to the controller number for the `controller` data structure for this 3Com 3C5x9 device.

4 Sets the `config_type` member of the `el_intr_info` data structure to the constant `CONTROLLER_CONFIG_TYPE`, which identifies the `if_el` driver type as a controller driver.

5 If the `if_el` driver operates on the PCMCIA bus, indicates that the `if_el` driver can handle shared interrupts.

6 Sets the `ih_bus` member of the `el_ihandle` data structure to the `bus` data structure for the `if_el` device driver. The `bus` data structure is referenced through the `bus_hd` member of the `controller` data structure for this 3Com 3C5x9 device.

7 Sets the `ih_bus_info` member of the `el_ihandle` data structure to the address of the bus-specific information data structure, `el_intr_info`.

8 Calls the `handler_add()` routine to register the device driver's interrupt handler and its associated `ihandler_t` data structure with the bus-specific interrupt-dispatching algorithm.

This routine returns an opaque `ihandler_id_t` key, which is a unique number that identifies the interrupt handler to be acted on by subsequent calls to `handler_del`, `handler_disable`, and `handler_enable`. The `hid` member of the `el_softc` data structure stores this key.

9. If the return value from `handler_add` equals NULL, the `if_el` driver failed to register an interrupt handler for the `if_el` device. This is a fatal error, and the `if_el` driver will undo all previous operations and return an error to the caller.

5.1.12 Saving the controller and softc Data Structure Pointers

The following code shows how the `el_probe()` routine saves the `controller` and `el_softc` data structure pointers. All `probe` interfaces perform this task.

```
el_softc[unit] = sc;   1
el_info[unit] = ctlr;  2
```

1. Saves the `el_softc` data structure pointer for this instance of the 3Com 3C5x9 device in the array of `el_softc` data structures. The unit number is the offset to the data structure within the `el_softc` array.

2. Saves the `controller` data structure pointer for this instance of the 3Com 3C5x9 device in the array of `controller` data structures.

5.1.13 Trying to Allocate Another controller Data Structure

The following code shows how the `el_probe()` routine attempts to allocate another `controller` data structure. You make this call so that a driver can support multiple devices.

```
if (!sc->reprobe && lan_create_controller(&el_data) != ESUCCESS) {   1
    printf("el%d: WARNING: create_controller failed\n", unit);
}
```

1. If this is the first time that the device has been probed, calls the `lan_create_controller()` routine to try to create a second `controller` data structure. If `lan_create_controller()` fails, calls the `printf()` routine to display a message. (Routines that begin with `lan_` reside in the `lan_common.c` source file.)

5.1.14 Registering the shutdown Routine

The following code shows how the `el_probe()` routine registers its `shutdown()` routine. The kernel calls this routine when the system shuts down. The driver can specify an argument for the kernel to pass to the routine at that time.

```
  if (!sc->reprobe)
     drvr_register_shutdown(el_shutdown, (void*)sc, DRVR_REGISTER);  1
  return(~0);
}
```

1 Registers the `shutdown()` routine and directs the kernel to pass a pointer to the driver's `softc` data structure to the routine. The `shutdown()` routine is important for those devices that perform DMA-related operations.

5.2 Implementing the el_shutdown Routine

The driver's `shutdown()` routine shuts down the controller. The kernel calls all registered `shutdown()` routines when the system shuts down.

The `el_probe()` routine registers a `shutdown()` routine called `el_shutdown()`. The `if_el` device driver implements the routine as follows:

```
static void el_shutdown(struct el_softc *sc)  1
{
   WRITE_CMD(sc, CMD_RESET);  2
   DELAY(1000);  3
}
```

1 Specifies the argument that the kernel passes to the routine, which is a pointer to the driver's `el_softc` data structure. The driver specifies this argument when it registers the `shutdown()` routine in its `probe` interface.

2 Calls the `WRITE_CMD` macro to write data to the command port register. In this call, the `el_softc` data structure for this 3Com 3C5x9 device contains the I/O handle to reference the device's command register. The data to be written is the `CMD_RESET` bit, which resets the device.

3 Calls the `DELAY` macro to delay the execution of `el_shutdown()` for 1 millisecond before continuing execution. This gives the reset command time to complete.

5.3 Implementing the el_autosense_thread Routine

The `if_el` device driver implements a driver-specific routine called `el_autosense_thread()` to determine the mode of the network interface. The `el_probe()` routine calls `el_autosense_thread()` during device autoconfiguration.

To determine the mode, `el_autosense_thread()` tries to send a test data packet in each of the possible modes. When it successfully transmits the data packet, it sets the network interface to that mode. The `lm_media_mode`, `lm_media`, and `lm_media_state` members of the `el_softc` data structure keep track of the progress of the autosensing procedure, as follows:

- The value of the `lm_media_mode` member determines whether the `el_autosense_thread()` will automatically determine the network interface, or whether the user specified the type of media.

- The `lm_media` member specifies the current media. This member changes each time that the driver uses a different medium to try to transmit a packet. The `if_el` device driver can set this member to any of the following values:

`LAN_MEDIA_UTP`	The media is unshielded twisted-pair cable.
`LAN_MEDIA_BNC`	The media is thin wire.
`LAN_MEDIA_AUI`	The media is the attachment unit interface (AUI).

- The `lm_media_state` member specifies the current state of the autosensing procedure, as follows:

`LAN_MEDIA_STATE_SENSING`	The driver is trying to determine the media mode.
`LAN_MEDIA_STATE_DETERMINED`	The media state has been determined.

The `el_autosense_thread()` routine is implemented as a kernel thread. It performs the following tasks:

- Blocks until awakened (Section 5.3.2)

- Tests for the termination flag (Section 5.3.3)

- Starts up statistics (Section 5.3.4)

- Enters the packet transmit loop (Section 5.3.5)

- Saves counters prior to the transmit operation (Section 5.3.6)

- Allocates memory for a test packet (Section 5.3.7)

- Uses the default from the ROM (Section 5.3.8)

- Sets the media setting in the hardware (Section 5.3.9)

- Builds a test packet to transmit (Section 5.3.10)

- Transmits the test packet (Section 5.3.11)

- Sets a timer for the current kernel thread (Section 5.3.12)

- Tests for loss of carrier (Section 5.3.13)

- Determines whether packets were transmitted successfully (Section 5.3.14)

- Prints debug information (Section 5.3.15)

- Sets up new media to try if transmit was unsuccessful (Section 5.3.16)

- Establishes media if transmit was successful (Section 5.3.17)

5.3.1 Setting Up the el_autosense_thread Routine

The following code shows how to set up the `el_autosense_thread()` routine:

```
unsigned char el_junk_msg[] = {  1
   0xaa, 0x00, 0x04, 0xff, 0xff, 0xff, 0, 0, 0, 0, 0, 0, 0x60, 0x06,
   't', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 'j', 'u', 'n', 'k',
   ' ', 'a', 'u', 't', 'o', 's', 'e', 'n', 's', 'e', ' ', 'm',
   'e', 's', 's', 'a', 'g', 'e', '.'
};
#define EL_JUNK_SIZE 46
#define EL_AUTOSENSE_PASSES 3*10
static void el_autosense_thread(struct el_softc *sc)  2
{
   struct ifnet *ifp = &sc->is_if;  3
   unsigned long prev_tint, prev_tmo, prev_err;
   struct mbuf *m;
   int good_xmits, wait, s, i, link_beat, passes;
   unsigned long wait_flag=0;
```

1 Defines the message to transmit when trying to determine the mode of the device.

2 Declares a pointer to the `el_softc` data structure and calls it `sc`.

3 Declares a pointer to an `ifnet` data structure and calls it `ifp`. This line also initializes `ifp` to the address of the `ifnet` data structure for this 3Com 3C5x9 device. The `ifnet` data structure is referenced through the `is_if` member of the `el_softc` data structure pointer. The `is_if` name is an alternate name for the `ac_if` member of the `arpcom` data structure. The `ac_if` member is referred to as the network-visible interface and is actually the instance of the `ifnet` data structure for this 3Com 3C5x9 device.

5.3.2 Blocking Until Awakened

The following code shows how the `el_autosense_thread()` routine blocks until awakened:

```
   while(1) {
       assert_wait((vm_offset_t)&sc->autosense_flag, TRUE);  1
       thread_block();
```

1 Waits for some process to indicate when to proceed with the autosense test.

5.3.3 Testing for the Termination Flag

The following code shows how the `el_autosense_thread()` routine tests
for the termination flag:

```
while (thread_should_halt(sc->autosense_thread)) {  1
        printf("el%d: Autosense thread exiting\n", ifp->if_unit);
        thread_halt_self();  2
}
```

1 Performs an initial test for the termination flag. The termination
   flag would have been set if another kernel thread had called the
   `thread_terminate()` routine for the `el_autosense_thread()`
   routine.

2 The `thread_halt_self()` routine performs the work that is
   associated with a variety of asynchronous traps (ASTs) for a kernel
   thread that terminates itself. A kernel thread terminates itself by
   calling the `thread_halt_self()` routine. The `thread_halt_self()`
   routine does not return to the caller.

5.3.4 Starting Up Statistics

The following code shows how the `el_autosense_thread()` routine starts
up statistics:

```
s = splimp();  1
simple_lock(&sc->el_softc_lock);
WRITE_CMD(sc, CMD_STATSENA);
simple_unlock(&sc->el_softc_lock);
splx(s);
```

1 Starts up statistics to test for the loss of the carrier during the transmit
   operation.

5.3.5 Entering the Packet Transmit Loop

The following code shows how the `el_autosense_thread()` routine enters
the packet transmit loop:

```
good_xmits = passes = 0;  1
sc->lm_media_state = LAN_MEDIA_STATE_SENSING;
while (good_xmits < 5) {
    while (thread_should_halt(sc->autosense_thread)) {
            printf("el%d: Autosense thread exiting\n", ifp->if_unit);
            s = splimp();
            simple_lock(&sc->el_softc_lock);
            WRITE_CMD(sc, CMD_STATSDIS);
```

```
                    simple_unlock(&sc->el_softc_lock);
                    splx(s);
                    thread_halt_self();
            }
```

[1] Enters a loop for transmitting a packet and determining if it succeeds.
     A packet must go out twice successfully for media selection to succeed.
     This algorithm probably will not work in all cases.

5.3.6 Saving Counters Prior to the Transmit Operation

The following code shows how the `el_autosense_thread()` routine saves
counters prior to the transmit operation:

```
prev_tint= sc->tint;
prev_err = ifp->if_oerrors;
prev_tmo = sc->xmit_tmo;
```

5.3.7 Allocating Memory for a Test Packet

The following code shows how the `el_autosense_thread()` routine
allocates memory for a test packet:

```
MGETHDR(m, M_WAIT, MT_DATA);
if ((passes++ > EL_AUTOSENSE_PASSES) || (m == NULL)) {
    if (m) {
        m_freem(m);
        printf("el%d: Autosense thread cannot determine media\n",
                ifp->if_unit);
        printf("el%d: Use lan_config to configure if necessary\n",
                ifp->if_unit);
    } else {
        printf("el%d: Autosense thread cannot get xmit buffer\n",
                ifp->if_unit);
    }
```

5.3.8 Using the Default from the ROM

The following code shows how the `el_autosense_thread()` routine uses
the default media setting from ROM. This code sequence signifies a last
resort if the driver is unable to determine the media.

```
switch (sc->eeprom.addrconf & 0xc) {  [1]
 case ACR_10B5:
    if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
        sc->lm_media = LAN_MEDIA_AUI;
    break;
 case ACR_10B2:
    if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
        sc->lm_media = LAN_MEDIA_BNC;
    break;
 case ACR_10BT:
 default:
    if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
        sc->lm_media = LAN_MEDIA_UTP;
 break;
}
 printf("el%d: Used %s setting from eeprom\n",
```

```
        ifp->if_unit, lan_media_strings_10[sc->lm_media]);
    good_xmits = 100;
```

[1] Uses the default from ROM.

5.3.9 Setting the Media in the Hardware

The following code shows how the `el_autosense_thread()` routine sets
the media setting in the hardware:

```
el_reset(ifp->if_unit); [1]
break; [2]
} else {
```

[1] Directs the hardware to use the media setting that was selected in the
    previous section.

[2] Breaks out of the packet transmit loop because the media setting has
    been determined.

5.3.10 Building the Test Packet

The following code shows how the `el_autosense_thread()` routine builds
a test packet to transmit:

```
bcopy(el_junk_msg, mtod(m, caddr_t), EL_JUNK_SIZE); [1]
bcopy(sc->is_addr, mtod(m, caddr_t), 6); [2]
bcopy(sc->is_addr, mtod(m, caddr_t)+6, 6); [3]
m->m_pkthdr.len = m->m_len = EL_JUNK_SIZE;
```

[1] Loads the junk message into the `mbuf` data structure.

[2] Sets the destination address as the address of the adapter.

[3] Sets the source address as the address of the adapter.

5.3.11 Transmitting the Test Packet

The following code shows how the `el_autosense_thread()` routine
transmits the test packet:

```
s = splimp();
simple_lock(&sc->el_softc_lock);
IF_ENQUEUE(&ifp->if_snd, m);
el_start_locked(sc, ifp);
simple_unlock(&sc->el_softc_lock);
splx(s);
```

5.3.12 Setting a Timer for the Current Kernel Thread

The following code shows how the `el_autosense_thread()` routine sets a
timer for the current kernel thread:

```
wait = 0;
while ((prev_tint == sc->tint) && 1
       (prev_tmo  == sc->xmit_tmo) &&
       (wait++ < 4)) {
   assert_wait((vm_offset_t)&wait_flag, TRUE);
   thread_set_timeout(1*hz); 2
   thread_block();
}
```

1   Waits until the transmit makes it out, a timeout occurs, or 4 seconds
pass.

2   Sets the timer and puts the current thread to sleep. To use a timer,
`thread_set_timeout()` must be called between an `assert_wait()`
and a `thread_block()`.

5.3.13 Testing for Loss of Carrier

The following code shows how the `el_autosense_thread()` routine tests
for loss of carrier:

```
link_beat = 0; 1
switch (sc->lm_media) {
 case LAN_MEDIA_UTP:
    s = splimp();
    simple_lock(&sc->el_softc_lock);
    WRITE_CMD(sc, CMD_WINDOW4);
    i = READ_MD(sc);
    if ((i & MD_VLB) != 0)
      link_beat=1;
    WRITE_CMD(sc, CMD_WINDOW1);
    simple_unlock(&sc->el_softc_lock);
    splx(s);
 case LAN_MEDIA_BNC:
 case LAN_MEDIA_AUI:
    s = splimp();
    simple_lock(&sc->el_softc_lock);
    WRITE_CMD(sc, CMD_WINDOW6);
    WRITE_CMD(sc, CMD_STATSDIS);
    i = READ_BUS_D8(sc->basereg);
    if (i != 0) {
        wait = 100;
        if (sc->debug)
          printf("el%d: autosense: %s carrier loss\n",
                 ifp->if_unit,
                 lan_media_strings_10[sc->lm_media]);
    }
    WRITE_CMD(sc, CMD_STATSENA);
```

```
        WRITE_CMD(sc, CMD_WINDOW1);
        simple_unlock(&sc->el_softc_lock);
        splx(s);
        break;
    default:
        break;
    }
```

[1] Tests for loss of carrier errors. Most network adapters give carrier
     errors if no cable is plugged in or if transceivers are not present.

5.3.14 Determining Whether Packets Were Transmitted Successfully

The following code shows how the `el_autosense_thread()` routine
determines whether packets were successfully transmitted:

```
if ((prev_err == ifp->if_oerrors) &&
    (prev_tmo == sc->xmit_tmo) &&
    (wait < 5)) { [1]
    good_xmits++;
    if (sc->debug)
      printf("el%d: autosense: %s packet sent OK (%d)\n",
             ifp->if_unit, lan_media_strings_10[sc->lm_media],
             good_xmits);
} else {
    good_xmits = 0;
```

[1] Determines whether traffic went out successfully.

5.3.15 Printing Debug Information

The following code shows how the `el_autosense_thread()` routine prints
debug information:

```
if (sc->debug) { [1]
    if (prev_err != ifp->if_oerrors)
      printf("el%d: autosense: %s transmit error\n",
             ifp->if_unit,
             lan_media_strings_10[sc->lm_media]);
    if (prev_tmo != sc->xmit_tmo)
      printf("el%d: autosense: %s driver transmit timeout\n",
             ifp->if_unit,
             lan_media_strings_10[sc->lm_media]);
    if ((wait >= 5) && (wait < 100))
      printf("el%d: autosense: %s transmit timeout\n",
             ifp->if_unit,
             lan_media_strings_10[sc->lm_media]);
}
```

[1] Prints debugging information (if requested).

5.3.16 Setting Up New Media

The following code shows how the `el_autosense_thread()` routine
selects new media to try if the transmit operation failed:

```
    switch (sc->lm_media) { [1]
      case LAN_MEDIA_AUI:
```

```
        if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
           sc->lm_media = LAN_MEDIA_UTP;
        break;
      case LAN_MEDIA_BNC:
        if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
           sc->lm_media = LAN_MEDIA_AUI;
        break;
      case LAN_MEDIA_UTP:
      default:
        if (sc->lm_media_mode == LAN_MODE_AUTOSENSE)
            sc->lm_media = LAN_MEDIA_BNC;
        break;
   }
   el_reset(ifp->if_unit); 2
 }
}
```

1 Selects new media.

2 Calls the el_reset() routine to reset the hardware. This reset will
   establish the next media to try.

5.3.17 Establishing the Media

The following code shows how the el_autosense_thread() routine
establishes the new media:

```
    }
    if (sc->debug) {
        if ((sc->lm_media == LAN_MEDIA_UTP) && !link_beat &&
            (passes <= EL_AUTOSENSE_PASSES))
    printf("el%d: No Link Beat signal\n", ifp->if_unit);
    }
    sc->lm_media_state = LAN_MEDIA_STATE_DETERMINED; 1
    printf("el%d: Autosense selected %s media\n", ifp->if_unit,
        lan_media_strings_10[sc->lm_media]);
    s = splimp(); 2
    simple_lock(&sc->el_softc_lock); 3
    WRITE_CMD(sc, CMD_STATSDIS); 4
    simple_unlock(&sc->el_softc_lock); 5
    splx(s); 6
  }
}
```

1 Sets the lm_media_state member of the softc data structure to
   LAN_MEDIA_STATE_DETERMINED. This indicates that the driver has
   successfully selected a media mode.

2 Calls the splimp() routine to mask all LAN hardware interrupts.
   Upon successful completion, splimp() stores an integer value in the
   s variable. This value represents the CPU priority level that existed
   before the call to splimp().

3 Calls the simple_lock() routine to assert a lock with exclusive
   access for the resource that is associated with the el_softc_lock
   data structure. This means that no other kernel thread can gain access
   to the locked resource until you call simple_unlock() to release it.

Because simple locks are spin locks, `simple_lock()` does not return until the lock has been obtained.

The `el_softc_lock` member of the `el_softc` data structure points to a simple lock data structure. The `if_el` device driver declares this data structure by calling the `decl_simple_lock_data()` routine.

4 Calls the `WRITE_CMD` macro to write data to the command port register. In this call, `el_autosense_thread()` passes the `if_el` driver's `el_softc` data structure pointer. The data to be written is the statistics disable command (`CMD_STATDIS`).

5 Releases the simple lock and resets the IPL.

6 Calls the `splx()` routine to reset the CPU priority to the level that is stored in the `s` variable.

6

Implementing the Autoconfiguration Support Section (attach)

The autoconfiguration support section implements a network device driver's `attach` interface. A network device driver's `attach` interface establishes communication with the device. The interface initializes the pointer to the `ifnet` data structure and attaches the network interface to the packet filter. The bus configuration code calls the driver's `attach` interface.

The `if_el` device driver implements an `attach()` routine called `el_attach()`. The `el_attach()` routine performs the following tasks:

- Initializes the media address and media header lengths (Section 6.2)

- Sets up the media (Section 6.3)

- Initializes simple lock information (Section 6.4)

- Prints a success message (Section 6.5)

- Specifies the network driver interfaces (Section 6.6)

- Sets the baud rate (Section 6.7)

- Attaches to the packet filter and the network layer (Section 6.8)

- Sets network attributes and registers the adapter (Section 6.9)

- Handles reinsertion operations (Section 6.10)

- Enables the interrupt handler (Section 6.11)

- Starts the polling process (Section 6.12)

6.1 Setting Up the el_attach Routine

The following code shows how to set up the `el_attach()` routine:

```
static int el_attach(struct controller *ctlr) 1
{
    register int unit = ctlr->ctlr_num; 2
    register struct el_softc *sc = el_softc[unit]; 3
    register struct ifnet *ifp = &sc->is_if; 4
    register struct sockaddr_in *sin; 5
```

1 Declares as an argument a pointer to a `controller` data structure for this controller. This data structure contains such information as the controller type, the controller name, and the current status

of the controller. The bus configuration code passes this initialized `controller` data structure to the driver's `probe` and `attach` interfaces.

2 Declares a `unit` variable and initializes it to the controller number for this controller. This controller number identifies the specific 3Com 3C5x9 controller that is being attached. The controller number is contained in the `ctlr_num` member of the `controller` data structure for this device.

3 Declares a pointer to the `el_softc` data structure called `sc` and initializes it to the `el_softc` data structure for this device. The controller number (which is stored in the `unit` variable) is used as an index into the array of `el_softc` data structures to determine which `el_softc` data structure is associated with this device.

4 Declares a pointer to an `ifnet` data structure called `ifp` and initializes it to the address of the `ifnet` data structure for this device.

5 Declares a pointer to a `sockaddr_in` data structure called `sin`.

6.2 Initializing the Media Address and Media Header Lengths

The `el_attach()` routine sets up the media's address length and header length, as follows:

```
if (!sc->reprobe) {  1
    ctlr->alive |= ALV_STATIC;  2
    ifp->if_addrlen = 6;  3
    ifp->if_hdrlen =  4
        sizeof(struct ether_header) + 8;
```

1 Examines the value of the `reprobe` member of the driver's `softc` data structure to determine whether the user has reinserted the PCMCIA card. If the card has been reinserted, the driver skips to the code in Section 6.10.

2 Because the `if_el` device driver must always be linked into the kernel, sets the ALV_STATIC bit. If your driver can be dynamically loaded, set the ALV_NOSIZER bit instead.

3 Sets the `if_addrlen` member of the `ifnet` data structure for this device to the media address length, which in this case is 6 bytes (the IEEE standard 48-bit address).

4 Sets the `if_hdrlen` member of the `ifnet` data structure for this device to the media header length. The `el_attach()` routine uses the `sizeof` operator to return the size of the data structure because it can differ from one network type to another. In this example, the media

header length is the size of the `ether_header` data structure plus 8 (the size of the maximum LLC header).

The media headers are represented by the following data structures:

`ether_header`	The media header structure for Ethernet-related media. The `if_ether.h` file defines the `ether_header` structure.
`fddi_header`	The media header structure for FDDI-related media. The `if_fddi.h` file defines the `fddi_header` structure.
`trn_header`	The media header structure for Token Ring-related media. The `if_trn.h` file defines the `trn_header` structure.

6.3 Setting Up the Media

The following code shows how the `el_attach()` routine sets up media-related information:

```
sc->is_ac.ac_bcastaddr = (u_char *)etherbroadcastaddr;  1
sc->is_ac.ac_arphrd = ARPHRD_ETHER;  2
ifp->if_mtu = ETHERMTU;  3
ifp->if_mediamtu = ETHERMTU;  4
ifp->if_type = IFT_ETHER;  5
((struct arpcom *)ifp)->ac_flag = 0;  6

sin = (struct sockaddr_in *)&ifp->if_addr;  7
sin->sin_family = AF_INET;  8
```

1 Sets the `ac_bcastaddr` member of the `softc` data structure for this device to the Ethernet broadcast address. The system stores the Ethernet broadcast address in the `etherbroadcastaddr` character array. Tru64 UNIX defines the `etherbroadcastaddr` character array in the `if_ether.h` file.

 The name `is_ac` is an alternate name for the `ess_ac` member of the `ether_driver` data structure. The `ess_ac` member is referred to as the Ethernet common part and is actually an instance of the `arpcom` data structure.

2 Sets the `ac_arphrd` member of the `softc` data structure for this device to the constant `ARPHRD_ETHER`, which represents the Ethernet hardware address. The `if_arp.h` file defines this constant.

 For the Token Ring interface, set the `ac_arphrd` member to the constant `ARPHRD_802`. The `if_arp.h` file also defines this constant.

 For the FDDI interface, set the `ac_arphrd` member to the constant `ARPHRD_ETHER`, which represents the Ethernet hardware address. See RFC 826 for more details.

3 Sets the `if_mtu` member of the `ifnet` data structure for this device to the maximum transmission unit, which for Ethernet-related media is represented by the constant `ETHERMTU`.

The following media-specific constants represent the maximum transmission unit:

`ETHERMTU`	The maximum transmission unit for Ethernet media. The `if_ether.h` file defines the `ETHERMTU` constant.
`FDDIMTU`	The maximum transmission unit for FDDI media. The `if_fddi.h` file defines the `FDDIMTU` constant.
`TRN4_RFC1042_IP_MTU`	The maximum transmission unit for the 4 megabit-per-second Token Ring media. The `if_trn.h` file defines the `TRN4_RFC1042_IP_MTU` constant.
`TRN16_RFC1042_IP_MTU`	The maximum transmission unit for the 16 megabit-per-second Token Ring media. The `if_trn.h` file defines the `TRN16_RFC1042_IP_MTU` constant.

4 Sets the `if_mediamtu` member of the `ifnet` data structure for this device to the maximum transmission unit for the media, which for Ethernet-related media is represented by the constant `ETHERMTU`. Typically, you set this member to the same constant that is used for the `if_mtu` member.

5 Sets the `if_type` member of the `ifnet` data structure for this device to the type of network interface, which is represented by the constant `IFT_ETHER` (Ethernet I or II interface).

The following describes some of the valid interface types that are defined in the `if_types.h` file:

`IFT_ETHER`	Ethernet I or II interface
`IFT_FDDI`	FDDI interface
`IFT_ISO88025`	Token Ring interface

6 Sets the `ac_flag` member of the `arpcom` data structure for this device to the value 0 (zero). This indicates that an IP address is currently not configured for this interface.

7 Sets the `sockaddr_in` data structure pointer to the address of the network interface. The address of the network interface is referenced through the `if_addr` member of the `ifnet` data structure for this device.

8 Sets the `sin_family` member of the `sockaddr_in` data structure to the address family, which in this case is represented by the constant `AF_INET`. The `socket.h` file defines this and other address family constants.

6.4 Initializing Simple Lock Information

The following code shows how the `el_attach()` routine sets up simple lock information:

```
ifp->if_affinity = NETALLCPU; 1
ifp->lk_softc = &sc->el_softc_lock; 2
simple_lock_setup(&sc->el_softc_lock, el_lock_info); 3
```

1 Sets the `if_affinity` member of the `ifnet` data structure for this device to the constant `NETALLCPU`. The `if_affinity` member specifies which CPU to run on. You can set this member to one of the following constants defined in `if.h`:

NETMASTERCPU	Specifies that you want to funnel the network device driver because you have not made it symmetric multiprocessor (SMP) safe. This means that the network driver is forced to execute on a single (the master) CPU. This setting is *not* recommended. You are encouraged to make your driver SMP safe.
NETALLCPU	Specifies that you do not want to funnel the network device driver because you have made it SMP safe. This means that the network driver can execute on multiple CPUs. You make a network device driver SMP safe by using the simple or complex lock mechanism in all critical sections of the driver.

The `if_el` driver uses the simple lock mechanism and is, therefore, SMP safe.

2 Sets the `lk_softc` member of the `ifnet` data structure for this device to the address of the `el_softc_lock`. Both the `if_el` driver and the network software above the driver use this lock whenever modifications are made to the shared members of the `ifnet` data structure. Make sure to supply a lock for the shared portion of the `ifnet` structure also.

3 Calls the `simple_lock_init()` routine to initialize the simple lock structure called `el_softc_lock`. You need to initialize the simple lock structure only once.

6.5 Printing a Success Message

The following code shows how the `el_attach()` routine prints a success
message:

```
printf("el%d: %s, hardware address: %s\n", unit,
        ifp->if_version, ether_sprintf(sc->is_addr)); 1
```

1 Calls the `printf()` routine to display the following information
    message on the console terminal:

   • The controller number that is stored in the `unit` variable.

   • The version of the network interface that is stored in the
     `if_version` member of the `ifnet` data structure pointer.

   • The hardware address that is accessed through the `is_addr`
     member of the `el_softc` data structure for this device. The `if_el`
     device driver maps the `ac_enaddr` member of the `arpcom` data
     structure to the alternate name `is_addr`.

   The argument list that is passed to `printf()` contains a call to the
   `ether_sprintf()` routine. The `ether_sprintf()` routine converts
   an Ethernet address to a printable ASCII string representation.

Make sure that your driver prints a similar message during its `attach()`
routine.

6.6 Specifying the Network Driver Interfaces

The following code shows how the `el_attach()` routine specifies the
network driver interfaces for the `if_el` driver:

```
ifp->if_ioctl = el_ioctl; 1
ifp->if_watchdog = el_watch; 2
ifp->if_start = (int (*)())el_start; 3

mb();
ifp->if_output = ether_output; 4
mb();
ifp->if_flags = IFF_BROADCAST|IFF_MULTICAST|
                IFF_NOTRAILERS|IFF_SIMPLEX; 5

ifp->if_timer = 0; 6
ifp->if_sysid_type = 0; 7

ifp->if_version = "3Com EtherLink III"; 8
```

1 Sets the `if_ioctl` member of the `ifnet` data structure for this device
    to `el_ioctl`, which is the `if_el` device driver's `ioctl` interface.

2 Sets the `if_watchdog` member of the `ifnet` data structure for this
    device to `el_watch`, which is the `if_el` device driver's `watchdog`
    interface.

3 Sets the `if_start` member of the `ifnet` data structure for this device to `el_start`, which is the `if_el` device driver's start transmit for output interface.

4 Sets the `if_output` member of the `ifnet` data structure for this device to `ether_output`, which is the `if_el` device driver's output interface. Tru64 UNIX provides this kernel routine. All network device drivers, including Token Ring and FDDI drivers, must set `if_output` to `ether_output`, rather than implementing a driver-specific output interface.

An `mb()` (memory barrier) preceeds the setting of the `if_output` member. Members of the `ifnet` structure *must be* initialized in the order shown. The `mb()` ensures that all other function pointers are set before the `if_output` function pointer is set. This order is necessary because the `if_el` device can be unattached and later attached again.

5 Sets the `if_flags` member of the `ifnet` data structure for this device to the bitwise inclusive OR of the following status bits that are defined in the `if.h` file:

`IFF_BROADCAST`	Signifies that the network interface supports broadcasting and that the associated broadcast address is valid.
`IFF_MULTICAST`	Signifies that the network interface supports multicast.
`IFF_NOTRAILERS`	Signifies that the transmission avoids the use of trailers. The term trailers refers to the IP trailer encapsulation protocol, which is obsolete.
`IFF_SIMPLEX`	Signifies that the interface cannot identify its own transmissions.

An `mb()` (memory barrier) precedes the setting of the `if_flags` member. All the function pointers *must be* initialized before the `if_flags` field is set, in case the `if_el` device has been unattached and then attached again.

6 Sets the `if_timer` member of the `ifnet` data structure for this device to the value 0 (zero). This is the number of seconds to wait until the driver's `watchdog` interface is called. Setting the `if_timer` member to 0 (zero) disables the timer.

7 Sets the `if_sysid_type` member of the `ifnet` data structure for this device to the value 0 (zero). This optional member specifies a unique number that identifies the bus adapter hardware to the network management software. This unique number is referred to as the MOP system ID device code.

8 Sets the `if_version` member of the `ifnet` data structure for this device to the string `3Com EtherLink III`.

6.7 Setting the Baud Rate

The following code shows how the `el_attach()` routine sets the baud rate:

```
ifp->if_baudrate = ETHER_BANDWIDTH_10MB; 1
```

1 Sets the `if_baudrate` member of the `ifnet` data structure for this device to the constant `ETHER_BANDWIDTH_10MB`. The `if_baudrate` member specifies the line speed.

You can use the following media-specific constants:

`ETHER_BANDWIDTH_10MB`

> Ethernet line speed is 10 megabits per second. The `if_ether.h` file defines the `ETHER_BAND-WIDTH_10MB` constant.

`ETHER_BANDWIDTH_100MB`

> Fast Ethernet line speed is 100 megabits per second. The `if_ether.h` file defines the `ETHER_BAND-WIDTH_100MB` constant.

`FDDI_BANDWIDTH_100MB`

> FDDI line speed is 100 megabits per second. The `if_fddi.h` file defines the `FDDI_BANDWIDTH_100MB` constant.

`TRN_BANDWIDTH_4MB`

> Token Ring line speed is 4 megabits per second. The `if_trn.h` file defines the `TRN_BANDWIDTH_4MB` constant.

`TRN_BANDWIDTH_16MB`

> Token Ring line speed is 16 megabits per second. The `if_trn.h` file defines the `TRN_BANDWIDTH_16MB` constant.

6.8 Attaching to the Packet Filter and the Network Layer

The following code shows how the `el_attach()` routine attaches to the packet filter and the network layer:

```
attachpfilter(&(sc->is_ed)); 1

if_attach(ifp); 2
el_configured ++; 3
```

1 Calls the `attachpfilter()` routine to inform the packet filter driver about this network driver. The `attachpfilter()` routine is passed a pointer to the `ether_driver` data structure for this network device driver.

2 Calls the `if_attach()` routine to attach an interface to the list of active interfaces. The argument to the `if_attach()` routine is a pointer to the `ifnet` data structure for with this device.

3 If the `probe` and `attach` operations were successful, increments the number of successfully configured `el` devices. You must do this if you are using `lan_configure()`.

6.9 Setting Network Attributes and Registering the Adapter

The following code shows how the `if_attach()` routine sets the known nonzero network attributes for the enhanced hardware management (EHM) facility and registers the adapter:

```
lan_set_common_attributes(sc->ehm.current_val, &sc->is_ed); 1
lan_set_attribute(sc->ehm.current_val, NET_METHOD_NDX,
                  net_method_automatic);
lan_register_adapter(&sc->ehm, ctlr); 2
```

1 Sets any known nonzero network attributes for the enhanced hardware management facility. Make sure that this function call is made only after the call to `if_attach()`.

2 Registers the adapter with EHM.

6.10 Handling the Reinsert Operation

If the user has reinserted the PCMCIA card, the `if_el` device driver does not need to initialize the media address and media length. It does not need to set up the media, specify the network driver interfaces, set the baud rate, or initialize simple lock information. These tasks are done during the first `attach` operation. The `el_attach()` routine needs only to initialize the device, as follows:

```
} else {
    printf("el%d: %s, reloaded -- current lan address: %s\n", unit,
           ifp->if_version, ether_sprintf(sc->is_addr)); 1

    if (ifp->if_flags & IFF_RUNNING) 2
      el_init(unit);
}
```

1 If the adapter was reinserted, calls the `printf()` routine to display the following information on the console terminal:

- The controller number (which is stored in the `unit` variable).

- The version of the network interface (which is stored in the `if_version` member of the `ifnet` data structure).

- The hardware address of the device.

2 Calls the driver's `el_init()` routine if the resources that are associated with the network interface were previously allocated.

6.11 Enabling the Interrupt Handler

The following code shows how the el_attach() routine enables the interrupt handler:

```
handler_enable(sc->hid);  1
```

1 Calls the handler_enable() routine to enable a previously registered interrupt handler. The el_probe() routine calls handler_add to register the interrupt handler and it stores the handler ID in the hid member of the el_softc data structure for this device.

6.12 Starting the Polling Process

The following code shows how the el_attach() routine starts the polling process:

```
if (el_polling && !sc->polling_flag) {  1
    sc->polling_flag = 1;
    timeout((void *)el_intr, (void *)unit, (1*hz)/el_pollint);
} else
    sc->polling_flag = 0;  2

return(0);
}
```

1 Starts the polling process if the el_polling attribute specifies that polling is to be done.

To start the polling process, el_attach() sets the polling_flag member to 1 (true), then calls the timeout() routine to schedule the interrupt handler to run at some point in the future. timeout() is called with the following arguments:

- A pointer to the el_intr() routine, which is the if_el device driver's interrupt handler.

- The unit variable, which contains the controller number associated with this device. This argument is passed to the el_intr() routine.

- The el_pollint variable, which specifies the amount of time to delay before calling the el_intr() routine.

2 If the user requests that polling be disabled, el_attach() sets the polling_flag member to 0 (false).

7

Implementing the unattach Routine

The `el_unattach()` routine is called to stop the device and to free memory and other resources prior to unloading the driver or powering off the bus to which the device is attached. The `el_unattach()` routine undoes everything that was performed by the `el_probe()` and `el_attach()` routines.

Note

The PCMCIA bus does not support the `el_unattach()` routine.

The `el_unattach()` routine performs the following tasks:

- Verifies that the interface has shut down (Section 7.2)
- Obtains and releases the simple lock (Section 7.3)
- Disables the interrupt handler (Section 7.4)
- Terminates the `autosense` thread (Section 7.5)
- Unregisters the PCMCIA event callback routine (Section 7.6)
- Stops the polling process (Section 7.7)
- Unregisters the shutdown interface (Section 7.8)
- Terminates the simple lock (Section 7.9)
- Unregisters the card from the hardware management database (Section 7.10)
- Frees data structures and resources used by the adapter (Section 7.11)

7.1 Setting Up the el_unattach Routine

The following code shows how to set up the `el_unattach()` routine:

```
static int el_unattach(struct bus *bus, 1
                       struct controller *ctlr)
{
    int unit = ctlr->ctlr_num; 2
    int s, status;
```

```
struct el_softc *sc = el_softc[unit];
struct ifnet *ifp = &sc->is_if;
```

1 Declares as an argument a pointer to a `bus` data structure and a
   `controller` data structure for this controller. The `controller` data
   structure contains such information as the controller type, the controller
   name, and the current status of the controller. This completely identifies
   the adapter that is being unattached.

2 Declares a `unit` variable and initializes it to the controller number for
   this controller. This controller number identifies the specific 3Com
   3C5x9 controller that is being unattached. The controller number is
   contained in the `ctlr_num` member of the `controller` data structure
   for this device.

7.2 Verifying That the Interface Has Shut Down

The following code verifies that the interface is down. Make sure that other
errors returned by `if_detach` do not stop interface shutdown.

```
status = if_detach(ifp);  1
if (status == EBUSY)  2
    return(status);
else if (status == ESUCCESS)  3
    detachpfilter(sc->is_ed);
ifp->if_flags &= ~IFF_RUNNING;  4
```

1 Calls `if_detach` to remove this interface from the list of active
   interfaces.

2 If the interface is still in use, it cannot be detached, so failure is
   returned.

3 If the interface is not in use, detaches it from the list of those that the
   packet filter monitors.

4 Marks the interface as no longer running.

7.3 Obtaining the Simple Lock and Shutting Down the Device

The following code shows how the `el_unattach()` routine obtains the
simple lock, shuts down the device, and releases the simple lock:

```
s = splimp();  1
simple_lock(&sc->el_softc_lock);  2

el_shutdown(sc);  3

simple_unlock(&sc->el_softc_lock);  4
splx(s);  5
```

1 Calls the `splimp()` routine to mask all LAN hardware interrupts.
   Upon successful completion, `splimp()` stores an integer value in the

s variable. This value represents the CPU priority level that existed before the call to `splimp()`.

[2] Calls the `simple_lock()` routine to assert a lock with exclusive access for the resource that is associated with the `el_softc_lock` data structure. This means that no other kernel thread can gain access to the locked resource until you call `simple_unlock()` to release it. Because simple locks are spin locks, `simple_lock()` does not return until the lock has been obtained.

[3] Stops the device and puts it in a reset state.

[4] Calls the `simple_unlock()` routine to release the simple lock.

[5] Calls the `splx()` routine to reset the CPU priority to the level that is stored in the s variable.

7.4 Disabling the Interrupt Handler

The following code shows how the `el_unattach()` routine disables and deletes the interrupt handler:

```
if (sc->hid) { 1
    handler_disable(sc->hid);
    handler_del(sc->hid);
    sc->hid = NULL;
}
```

[1] Disables and deletes the interrupt handler. The argument that is supplied to each function is the handler ID that was returned by `handler_add` in the `el_probe()` routine.

7.5 Terminating the Autosense Kernel Thread

The following code shows how the `el_unattach()` routine terminates the autosense kernel thread:

```
if (sc->autosense_thread) { 1
    thread_force_terminate(sc->autosense_thread);
    sc->autosense_thread = NULL;
}
```

[1] Terminates the autosense kernel thread.

7.6 Unregistering the PCMCIA Event Callback Routine

The following code shows how the `el_unattach()` routine unregisters
the PCMCIA event callback routine:

```
if (sc->ispcmcia) 1
  pcmcia_unregister_event_callback(sc->cinfop->socket_vnum,
                                   CARD_REMOVAL_EVENT,
                                   (caddr_t)el_card_remove);
```

1 For PCMCIA versions of the card, directs the bus code not to return
notification if the card has been removed.

7.7 Stopping the Polling Process

The following code shows how the `el_unattach()` routine stops the polling
process:

```
s = splimp();
simple_lock(&sc->el_softc_lock);
if (el_polling && sc->polling_flag) { 1
    untimeout((void *)el_intr, (void *)ifp->if_unit); 2
    sc->polling_flag = 0; 3
}
simple_unlock(&sc->el_softc_lock);
splx(s);
```

1 Stops the polling process if polling had originally been requested by
the user.

2 Removes the scheduled event from the system's timer queue.

3 Sets the `polling_flag` member to 0 (false) to indicate that polling
has stopped.

7.8 Unregistering the Shutdown Routine

The following code shows how the `el_unattach()` routine unregisters
the shutdown routine:

```
drvr_register_shutdown(el_shutdown, (void*)sc, DRVR_UNREGISTER); 1
```

1 Unregisters the shutdown routine, which was registered during the
`probe` operation.

7.9 Terminating the Simple Lock

The following code shows how the `el_unattach()` routine terminates the
`softc` lock:

```
simple_lock_terminate(&sc->el_softc_lock); 1
```

1 Frees up the `softc` lock.

7.10 Unregistering the Card from the Hardware Management Database

The following code shows how the `el_unattach()` routine unregisters the card from the hardware management database:

```
lan_ehm_free(&sc->ehm);   1
```

1 Frees up any memory allocated for enhanced hardware management and unregisters this card from the hardware management database.

7.11 Freeing Resources

The following code shows how the `el_unattach()` routine frees data structures and memory that the adapter uses:

```
FREE(sc, M_DEVBUF);   1
el_softc[unit] = NULL;
el_info[unit] = NULL;
el_configured--;

return (ESUCCESS);
}
```

1 Frees all memory that the adapter uses, and returns ESUCCESS to indicate that the `unattach` operation completed successfully.

8

Implementing the Initialization Section

The initialization section prepares the network interface to transmit and receive data packets. It can also allocate `mbuf` data structures for the receive ring.

The `if_el` device driver implements the following routines in its initialization section:

- `el_init` (Section 8.1)
- `el_init_locked` (Section 8.2)

8.1 Implementing the el_init Routine

The `el_init()` routine is a jacket routine that performs the following tasks:

- Determines whether the PCMCIA card is in the slot (Section 8.1.2)
- Sets the IPL and obtains the simple lock (Section 8.1.3)
- Calls the `el_init_locked()` routine to perform the initialization (Section 8.1.4)
- Releases the simple lock and resets the IPL (Section 8.1.5)
- Returns the status from `el_init_locked()` (Section 8.1.6)

8.1.1 Setting Up the el_init Routine

The following code shows how to set up the `el_init()` routine:

```
static int el_init(int unit) 1
{
   register struct el_softc *sc = el_softc[unit];  2
   register struct ifnet *ifp = &sc->is_if;  3
   int i, s; 4
```

1. Specifies the unit number of the network interface as the only argument to `el_init()`.

2. Declares a pointer to the `el_softc` data structure called `sc` and initializes it to the `el_softc` data structure for this device. The controller number (which is stored in the `unit` variable) is used as an index into the array of `el_softc` data structures to determine which `el_softc` data structure is for this device.

3 Declares a pointer to an `ifnet` data structure called `ifp` and initializes it to the address of the `ifnet` data structure for this device. The `ifnet` data structure is referenced through the `is_if` member of the `el_softc` data structure pointer. The `is_if` name is an alternate name for the `ac_if` member of the `arpcom` data structure. The `ac_if` member is referred to as the network-visible interface.

4 Declares the `i` and `s` variables. The `i` variable stores the value that `el_init_locked()` returns. The `s` variable stores the value that `splimp()` returns.

8.1.2 Determining Whether the PCMCIA Card Is Present

The following code shows how the `el_init()` routine determines whether the PCMCIA card is still present in the system.

```
if (sc->cardout) return(EIO);  1
```

1 If the user has removed the PCMCIA card from the slot, returns the error code `EIO` to the `el_attach()` routine. The `el_card_remove()` routine sets the `cardout` member.

8.1.3 Setting the IPL and Obtaining the Simple Lock

All network device drivers must set the interrupt priority level (IPL) to mask all LAN hardware interrupts. Raising the IPL protects the driver from interrupts on the same CPU. Only network device drivers that operate on multiple CPUs need to obtain a simple lock. The simple lock mechanism protects resources in a symmetric multiprocessing environment.

The following code shows how the `el_init()` routine sets the IPL and obtains the simple lock:

```
s = splimp();  1
simple_lock(&sc->el_softc_lock);  2
```

1 Calls the `splimp()` routine to mask all LAN hardware interrupts. Upon successful completion, `splimp()` stores an integer value in the `s` variable. This value represents the CPU priority level that existed before the call to `splimp()`.

2 Calls the `simple_lock()` routine to assert a lock with exclusive access for the resource that is associated with the `el_softc_lock` data structure. This means that no other kernel thread can gain access to the locked resource until you call `simple_unlock()` to release it. Because simple locks are spin locks, `simple_lock()` does not return until the lock has been obtained.

The `el_softc_lock` member of the `el_softc` data structure points to a simple lock data structure. The `if_el` device driver declares this data structure by calling the `decl_simple_lock_data()` routine.

8.1.4 Calling the el_init_locked Routine

The following code shows how the `el_init()` routine calls the `el_init_locked()` routine, which performs the actual initialization tasks:

```
i = el_init_locked(sc, ifp, unit);
```

8.1.5 Releasing the Simple Lock and Resetting the IPL

The following code shows how the `el_init()` routine releases the simple lock and resets the IPL. All network device drivers that do not use the simple lock mechanism must reset the IPL. All network device drivers that use the simple lock mechanism must reset the IPL after releasing the simple lock.

```
simple_unlock(&sc->el_softc_lock); 1
splx(s); 2
```

1 Calls the `simple_unlock()` routine to release the simple lock.

2 Calls the `splx()` routine to reset the CPU priority to the level that is stored in the `s` variable.

8.1.6 Returning the Status from the el_init_locked Routine

The following code shows how the `el_init()` routine returns status from `el_init_locked()`:

```
   return(i); 1
}
```

1 Exits and returns the status from `el_init_locked()`.

8.2 Implementing the el_init_locked Routine

The `el_init_locked()` routine initializes the network interface. It is called by the `if_el` device driver's `el_init()` and `el_reset_locked()` routines.

The `el_init_locked()` routine performs the following tasks:

- Resets the transmitter and receiver (Section 8.2.1)
- Clears interrupts (Section 8.2.2)
- Starts the device (Section 8.2.3)
- Ensures that the 10Base2 transceiver is off (Section 8.2.4)
- Sets the LAN media (Section 8.2.5)

- Sets the LAN media type attribute (Section 8.2.6)
- Selects memory mapping (Section 8.2.7)
- Resets the transmitter and receiver a second time (Section 8.2.8)
- Sets the LAN address (Section 8.2.9)
- Processes special flags (Section 8.2.10)
- Sets the debug flag (Section 8.2.11)
- Enables TX and RX (Section 8.2.12)
- Enables interrupts (Section 8.2.13)
- Sets the operational window (Section 8.2.14)
- Marks the device as running (Section 8.2.15)
- Starts the autosense kernel thread (Section 8.2.16)
- Starts transmitting pending packets (Section 8.2.17)

8.2.1 Resetting the Transmitter and Receiver

The following code shows how the `el_init_locked()` routine resets the transmitter and receiver. This task is specific to the 3Com 3C5x9 device. Make sure that you perform similar initialization tasks for the hardware device that your network driver controls.

```
static int el_init_locked(struct el_softc *sc,
    struct ifnet *ifp,
    int unit)
{
    register struct controller *ctlr = el_info[unit];
    int i;

    WRITE_CMD(sc, CMD_TXRESET);  1
    WRITE_CMD(sc, CMD_RXRESET);  2
```

1 Calls the `WRITE_CMD` macro to write data to the command port register. In this call, `el_init_locked()` passes the `if_el` driver's `el_softc` data structure pointer. The data to be written is the transmit (TX) reset command (`CMD_TXRESET`).

2 Calls the `WRITE_CMD` macro a second time to write data to the command port register. In this call, the data to be written to the command port register is the receive (RX) reset command (`CMD_RXRESET`).

8.2.2 Clearing Interrupts

The following code shows how the `el_init_locked()` routine clears interrupts.

This task is specific to the 3Com 3C5x9 device. Make sure that you perform similar initialization tasks for the hardware device that your network driver controls.

```
WRITE_CMD(sc, CMD_ACKINT+0xff);  1
```

1 Calls the `WRITE_CMD` macro to write data to the command port register. The data written to the command port register is the acknowledge interrupt command (`CMD_ACKINT`) plus a mask that specifies that all interrupts are to be acknowledged.

8.2.3 Starting the Device

The following code shows how the `el_init_locked()` routine starts the device. This task is specific to the 3Com 3C5x9 device. Make sure that you perform similar initialization tasks for the hardware device that your network driver controls.

```
WRITE_CMD(sc, CMD_WINDOW0);
i = READ_CCR(sc);
WRITE_CCR(sc, CCR_ENA | i);  1
WRITE_RCR(sc,
        (sc->irq << 12) | RCR_RSV);  2
```

1 Calls the `WRITE_CCR` macro to write data to the 3Com 3C5x9 device's configuration control register. The data to be written consists of the original register contents but with the enable adapter bit (CCR_ENA) set.

2 Calls the `WRITE_RCR` macro to write data to the 3Com 3C5x9 device's resource configuration register. The data to be written is the bitwise inclusive OR of the interrupt request (IRQ) stored in the `irq` member of the `el_softc` data structure and the reserved bit for the resource configuration register (`RCR_RSV`).

8.2.4 Ensuring That the 10Base2 Transceiver Is Off

The following code shows how the `el_init_locked()` routine ensures that the 10Base2 transceiver is off. This task is specific to the 3Com 3C5x9 device. You may want to perform similar initialization tasks for the hardware device that your network driver controls.

```
WRITE_CMD(sc, CMD_STOP2);  1
DELAY(800);  2
```

1 Calls the `WRITE_CMD` macro to write data to the command port register. The data to be written is the stop 10Base2 command bit (`CMD_STOP2`).

2 Calls the `DELAY` macro to wait 800 microseconds before continuing execution.

8.2.5 Setting the LAN Media

The following code shows how the `el_init_locked()` routine sets the LAN media. This task is specific to the 3Com 3C5x9 device. You may want to perform similar initialization tasks for the hardware device that your network driver controls.

```
i = READ_ACR(sc);  1
i &= ~(ACR_BASE|ACR_10B2);  2
switch (sc->lm_media) {  3
 case LAN_MEDIA_BNC:  4
    WRITE_ACR(sc,
              i | ACR_10B2 | sc->iobase);  5
    WRITE_CMD(sc, CMD_START2);  6
    DELAY(800);  7
    break;
 case LAN_MEDIA_AUI:  8
    WRITE_ACR(sc,
              i | ACR_10B5 | sc->iobase);  9
    break;
 default:  10
    sc->lm_media = LAN_MEDIA_UTP;
 case LAN_MEDIA_UTP:  11
    WRITE_ACR(sc,
              i | ACR_10BT | sc->iobase);  12
    WRITE_CMD(sc, CMD_WINDOW4);  13
    i = READ_MD(sc);
    WRITE_MD(sc, i | (MD_LBE | MD_JABE));  14
    break;
}
```

1 Calls the `READ_ACR` macro to read the data from the address control register.

2 Clears the `ACR_BASE` (the I/O base address) and the `ACR_10B2` (Ethernet thin coaxial cable) bits.

3 Evaluates the value that is stored in the `lm_media` member of the `el_softc` data structure for this device.

4 Determines whether `lm_media` evaluates to `LAN_MEDIA_BNC` (media mode is thin wire).

5 Calls the `WRITE_ACR` macro to write data to the address control register. The data to be written establishes the Ethernet thin coaxial cable as the media.

6 Calls the `WRITE_CMD` macro a second time to write data to the command port register. In this call, the data that is written to the command port register is `CMD_START2` (the start 10Base2 command bit).

7 Calls the `DELAY` macro to wait 800 microseconds.

8 Determines whether `lm_media` evaluates to `LAN_MEDIA_AUI` (media mode is the Attachment Unit Interface).

9 Calls `WRITE_ACR` to write to the address control register. The data to be written establishes the Ethernet thick coaxial cable as the media.

[10] For the default case, sets the `lm_media` member to `LAN_MEDIA_UTP`
(media mode is unshielded twisted pair cable).

[11] Determines whether `lm_media` evaluates to `LAN_MEDIA_UTP`.

[12] Calls `WRITE_ACR` to write to the address control register. The data to
be written establishes the Ethernet unshielded twisted-pair cable as
the media.

[13] Calls `WRITE_CMD` to write to the command port register. The data to be
written is the window 4 diagnostic command bit (`CMD_WINDOW4`).

[14] Calls the `WRITE_MD` macro to write data to the media type and status
register. The data to be written consists of the original data from that
register but with the link beat enabled (`MD_LBE`) and the jabber enabled
(`MD_JABE`) bits set.

8.2.6 Setting a LAN Attribute

The following code shows how the `el_init_locked()` routine sets the LAN
media type attribute for enhanced hardware management (EHM) support:

```
lan_set_attribute(sc->ehm.current_val, NET_MEDIA_NDX,
                  lan_media_strings[sc->lm_media]); [1]
```

[1] Sets the LAN media type attribute for EHM support.

8.2.7 Selecting Memory Mapping

The following code shows how the `el_init_locked()` routine selects
memory mapping. This task is specific to the 3Com 3C5x9 device.

```
if (ctlr->bus_hd->bus_type == BUS_PCMCIA) { [1]
    WRITE_CMD(sc, CMD_WINDOW0);
    i = READ_CCR(sc);
    if ((i & 0xc000) == 0x8000) {
        WRITE_CMD(sc, CMD_WINDOW3);
        i = sc->eeprom.icw & ~(ASI_RS|ASI_RS|ASI_RSIZE8|ASI_RSIZE32|
                               ASI_PAR_35|ASI_PAR_13|ASI_PAR_11);
        i |= (ASI_PAR_11 | ASI_RSIZE32);
        WRITE_DATA(sc, i);
    }
}
```

[1] If the `if_el` device driver operates on the PCMCIA bus, performs a
read operation and a number of write operations to select the memory
mapping.

8.2.8 Resetting the Transmitter and Receiver Again

The following code shows how the `el_init_locked()` routine resets the
transmitter and receiver a second time. This task is specific to the 3Com
3C5x9 device. Make sure that you perform similar initialization tasks for
the hardware device that your network driver controls.

```
WRITE_CMD(sc, CMD_TXRESET);  1
WRITE_CMD(sc, CMD_RXRESET);  2
```

1 Calls the `WRITE_CMD` macro to write data to the command port
register. The data to be written is the transmit (TX) reset command
(`CMD_TXRESET`).

2 Calls the `WRITE_CMD` macro to write data to the command port register.
In this call, the data to be written is the receive (RX) reset command
(`CMD_RXRESET`).

8.2.9 Setting the LAN Address

The following code shows how the `el_init_locked()` routine sets the
LAN address. This task is specific to the 3Com 3C5x9 device. You may want
to perform similar initialization tasks for the hardware device that your
network driver controls.

```
WRITE_CMD(sc, CMD_WINDOW2);  1
i = (sc->is_addr[1] << 8) + sc->is_addr[0];
WRITE_AD1(sc, i);
i = (sc->is_addr[3] << 8) + sc->is_addr[2];
WRITE_AD2(sc, i);
i = (sc->is_addr[5] << 8) + sc->is_addr[4];
WRITE_AD3(sc, i);

lan_set_attribute(sc->ehm.current_val, NET_MAC_NDX,
                  ether_sprintf(sc->is_addr));  2
```

1 Performs several write operations to set the LAN address.

2 Sets the LAN MAC address attribute for EHM support.

8.2.10 Processing Special Flags

The following code shows how the `el_init_locked()` routine processes
special flags. This task is specific to the 3Com 3C5x9 device. Make sure
that you perform similar initialization tasks for the hardware device that
your network driver controls.

```
if (ifp->if_flags & IFF_LOOPBACK) {  1
    WRITE_CMD(sc, CMD_WINDOW4);
    i = READ_ND(sc);
    WRITE_ND(sc, ND_LOOP | i);
    lan_set_attribute(sc->ehm.current_val, NET_LOOP_NDX, (void *)1);  2
}
else {
    lan_set_attribute(sc->ehm.current_val, NET_LOOP_NDX, (void *)0);
}
i = RF_IND | RF_BRD;                 3
if ((ifp->if_flags & IFF_ALLMULTI) || (sc->is_multi.lan_nmulti)) {  4
    i |= RF_GRP;
}
if (ifp->if_flags & IFF_PROMISC) {  5
    i |= RF_PRM;
    lan_set_attribute(sc->ehm.current_val, NET_PROMISC_NDX, (void *)1);  6
}
```

```
else {
    lan_set_attribute(sc->ehm.current_val, NET_PROMISC_NDX, (void *)0);
}
WRITE_CMD(sc, CMD_FILTER+i);  7
```

1 If loopback mode is requested, enables it.

2 Sets the LAN loopback attribute for EHM support.

3 Selects to receive frames that are sent to both the local address and the
   broadcast address.

4 If the network device receives all multicast packets, selects all group
   addresses.

5 If the network device receives all packets destined to all stations, selects
   promiscuous mode.

6 Sets the LAN promiscuous mode attribute for EHM support.

7 Calls the WRITE_CMD macro to write data to the command port register.
   In this call, the data to be written is the set receive (RX) filter command
   (CMD_FILTER) with the appropriate flags set.

8.2.11 Setting the Debug Flag

The following code shows how the el_init_locked() routine sets the
debug flag for turning on debugging on a running system. This task is
optional.

```
if (ifp->if_flags & IFF_DEBUG)  1
    sc->debug++;
else
    sc->debug = 0;

if (sc->debug) {  2
    WRITE_CMD(sc, CMD_WINDOW3);
    i = READ_TXF(sc);
    printf("el%d: Transmit FIFO size == %d\n", unit, i);
    i = READ_RXF(sc);
    WRITE_CMD(sc, CMD_WINDOW1);
    printf("el%d: Receive FIFO size == %d\n", unit, i);
}
```

1 Sets debug mode if the IFF_DEBUG bit is set.

2 If debugging mode is set, prints the transmit and receive first-in/first-out
   (FIFO) sizes.

8.2.12 Enabling TX and RX

The following code shows how the el_init_locked() routine enables
transmit (TX) and receive (RX). Make sure that you perform similar
initialization tasks for the hardware device that your network driver
controls.

```
WRITE_CMD(sc, CMD_RXENA);  1
WRITE_CMD(sc, CMD_TXENA);  2
```

1 Calls the WRITE_CMD macro to write data to the command port register.
   The data to be written is the receive (RX) enable command (CMD_RXENA).

2 Calls the WRITE_CMD macro to write data to the command port register.
   In this call, the data to be written is the transmit (TX) enable command
   (CMD_TXENA).

8.2.13 Enabling Interrupts

The following code shows how the el_init_locked() routine enables
interrupts. Make sure that you perform similar initialization tasks for the
hardware device that your network driver controls.

LAN device drivers typically do not perform polling operations. However,
this example shows how polling operations can be done on the 3Com 3C5x9
device.

```
if (!el_polling) {  1
    WRITE_CMD(sc, CMD_ZINTMASK+0xfe);
    WRITE_CMD(sc, CMD_SINTMASK+(S_AF|S_TC|S_RC));

} else {  2
    WRITE_CMD(sc, CMD_ZINTMASK+0xfe);
    WRITE_CMD(sc, CMD_SINTMASK+0);
}
```

1 If the device is not polling (the el_polling flag is not set), calls
   the WRITE_CMD macro to set the interrupt mask and enable adapter
   failure (S_AF), transmit complete (S_TC), and receive complete (S_RC)
   interrupts.

2 If the device is polling, calls the WRITE_CMD macro to clear the interrupt
   mask and disable all interrupts.

8.2.14 Setting the Operational Window

The following code shows how the el_init_locked() routine sets the
operational window. This task is specific to the 3Com 3C5x9 device.

```
WRITE_CMD(sc, CMD_WINDOW1);  1
sc->txfree = READ_TXF(sc);
```

1 Calls the WRITE_CMD macro to set the operational window register.

8.2.15 Marking the Device as Running

The following code shows how the el_init_locked() routine marks the
device as running. All network device drivers perform this task.

```
ifp->if_flags |= IFF_RUNNING;  1
ifp->if_flags &=  ~ IFF_OACTIVE;  2
```

1 Sets the `IFF_RUNNING` flag to mark the device as running.

2 Clears the `IFF_OACTIVE` flag to indicate that there is no output
   outstanding.

8.2.16 Starting the Autosense Kernel Thread

The following code shows how the `el_init_locked()` routine starts the
autosense kernel thread. Only network device drivers that implement an
autosense kernel thread perform this task.

```
if (sc->lm_media_mode == LAN_MODE_AUTOSENSE) {  1
    sc->lm_media_state = LAN_MEDIA_STATE_SENSING;
    thread_wakeup_one((vm_offset_t)&sc->autosense_flag);
}
```

1 If in autosense mode, starts the autosense kernel thread.

8.2.17 Starting the Transmit of Pending Packets

The following code shows how the `el_init_locked()` routine starts
transmitting pending packets. Because `el_init_locked()` may have
been called as a result of an error or a reset operation, it needs to examine
its transmit queue for any pending transmit requests. If there are any, it
starts transmitting them.

```
if (ifp->if_snd.ifq_head)  1
    el_start_locked(sc, ifp);

return ESUCCESS;  2
}
```

1 If there are any pending packets, starts transmitting them by calling
   the `el_start_locked()` routine.

2 Returns `ESUCCESS` to the calling routine.

9

Implementing the Start Section

The start section of a network device driver transmits data packets across the network. When the network protocol has a data packet to transmit, it prepares the packet, then calls the `start` interface for the appropriate network device driver. The `start` interface transmits the packet. When the transmission is complete, it frees up the buffers that are associated with the packet.

The `if_el` device driver implements the following routines in its start section:

- `el_start()` (Section 9.1)
- `el_start_locked()` (Section 9.2)

9.1 Implementing the el_start Routine

The `el_start()` routine is a jacket routine that performs the following tasks:

- Sets the IPL and obtains the simple lock (Section 9.1.1)
- Calls the `el_start_locked()` routine (Section 9.1.2)
- Releases the simple lock and resets the IPL (Section 9.1.3)

9.1.1 Setting the IPL and Obtaining the Simple Lock

The following code shows how the `el_start()` routine sets the IPL and acquires the simple lock.

```
static void el_start(struct ifnet *ifp)
{
   register int unit = ifp->if_unit, s;
   register struct el_softc *sc = el_softc[unit];

   s = splimp();  1
   if (!simple_lock_try(&sc->el_softc_lock)) {  2
      splx(s);  3
      return;
   }
```

1 Calls the `splimp()` routine to mask all LAN hardware interrupts. On successful completion, `splimp()` stores an integer value in the `s` variable. This integer value represents the CPU priority level that existed before the call to `splimp()`.

2 Calls the `simple_lock_try()` routine to try to assert a lock with read and write access for the resource that is associated with the specified simple lock. The `el_start()` routine calls `simple_lock_try()` rather than `simple_lock()` because `simple_lock_try()` returns immediately if the resource is already locked; `simple_lock()` spins until the lock has been obtained. Make sure that you call `simple_lock_try()` when you need a simple lock but the code cannot spin until the lock is obtained.

In this example, `simple_lock_try()` was used as an optimization. If the simple lock is already held, then another thread is executing somewhere in the driver and is either currently servicing the transmit request queue or will service it soon. Therefore, the transmit request that was put on the send queue prior to calling the `start` interface will be handled shortly. In this case, the code does not need to wait for the lock (because someone else will do the transmit) and can return to the caller.

The argument to `simple_lock_try()` is a pointer to a simple lock data structure. The `if_el` device driver declares the simple lock data structure by calling the `decl_simple_lock_data()` routine, and it stores a pointer to this data structure in the `el_softc` data structure.

3 If the `simple_lock_try()` routine fails to assert the simple lock, calls the `splx()` routine to reset the CPU priority to the level that the `s` variable specifies, then returns. Otherwise, the simple lock was obtained.

9.1.2 Calling the el_start_locked Routine

The following code shows how the `el_start()` routine calls the `el_start_locked()` routine, which starts the transmit operation:

```
el_start_locked(sc, ifp);  1
```

1 Calls the `el_start_locked()` routine, which performs the tasks that are related to the start operation.

9.1.3 Releasing the Simple Lock and Resetting the IPL

The following code shows how the `el_start()` routine releases the simple lock and resets the IPL.

```
simple_unlock(&sc->el_softc_lock);  1
splx(s);  2
}
```

1 Calls the `simple_unlock()` routine to release a simple lock for the resource that is associated with the specified simple lock data structure.

This simple lock was previously asserted by calling the `simple_lock()` or `simple_lock_try()` routine.

[2] Calls the `splx()` routine to reset the CPU priority to the level that the `s` variable specifies.

9.2 Implementing the el_start_locked Routine

The `el_start_locked()` routine performs the start operation. It is called by the `if_el` device driver's `el_init_locked()`, `el_start()`, `el_intr()`, and `el_autosense_thread()` routines.

The `el_start_locked()` routine performs the following tasks:

- Discards all transmits if the user has removed the PCMCIA card (Section 9.2.1)

- Removes packets from the pending queue and prepares the transmit buffer (Section 9.2.2)

- Transmits the packets (Section 9.2.3)

- Accounts for the outgoing bytes (Section 9.2.4)

- Updates counters, frees the transmit buffer, and marks the output process as active (Section 9.2.5)

- Indicates when to start the `watchdog` interface (Section 9.2.6)

_____ **Note** _____

If you decide not to implement your start section as a jacket routine, then some of the tasks listed in this section would be performed by your start section.

_____

9.2.1 Discarding All Transmits After the User Removes the PCMCIA Card

The following code shows how the `el_start_locked()` routine discards all pending transmits after the user has removed the card from the system.

```
static void el_start_locked(struct el_softc *sc,
        struct ifnet *ifp)
{
   struct mbuf *m, *ms, *mp, *mn;
   int len, i, j, val;
   unsigned char *dat;
   struct ether_header *eh; 1
   if (sc->cardout) { 2
```

```
        IF_DEQUEUE(&ifp->if_snd, m); 3
        while (m) { 4
            m_freem(m);
            IF_DEQUEUE(&ifp->if_snd, m);
        }
        return;
    }
```

1 Declares a pointer to an `ether_header` data structure called `eh`. The
   `ether_header` data structure contains information that is associated
   with a 10 Mb/s and 100 Mb/s Ethernet header.

2 If the `cardout` member of the `el_softc` data structure for this device
   is set to 1 (true), the user removed the PCMCIA card from the slot.

3 Calls the `IF_DEQUEUE` macro to remove an entry from the output queue.
   The output queue is referenced through the `if_snd` member of the
   `ifnet` data structure for this device. The memory buffer information
   that `IF_DEQUEUE` manipulates is specified in the instance of the `mbuf`
   data structure called `m`.

4 As long as a transmit request was dequeued from the output queue,
   calls `m_freem()` to free the request and `IF_DEQUEUE` to dequeue the
   next transmit request.

9.2.2 Removing Packets from the Pending Queue and Preparing the Transmit Buffer

The following code shows how the `el_start_locked()` routine removes
packets from the pending queue and prepares the transmit buffer:

```
while(1) { 1
    IF_DEQUEUE(&ifp->if_snd, m); 2
    if ((m) && ((m->m_pkthdr.len+8) < sc->txfree) ) { 3
        ms = m; 4
        while (ms && (ms->m_len == 0)) 5

          ms = ms->m_next; 6
        if (ms == NULL) { 7
            m_freem(m);
            continue;
        }
```

```
mp = ms;  8
mn = mp->m_next;  9
len = mp->m_len;  10
while (mn != NULL) {  11
    if (mn->m_len == 0) {
        mp->m_next = mn->m_next;
        mn->m_next = NULL;
        m_free(mn);
    } else {  12
        len += mn->m_len;
        mp = mn;
    }
    mn = mp->m_next;
}
```

1 While true, removes packets from the pending queue and has the device transmit the packets.

2 Calls the IF_DEQUEUE macro to remove an entry from the output queue. The output queue is referenced through the if_snd member of the ifnet data structure for this device. The memory buffer information is the instance of the mbuf data structure called m.

3 Checks that the total packet length is less than the number of bytes left in the transmit first-in/first-out (FIFO).

4 Eliminates any zero-length segments. The ms mbuf pointer will point to the first buffer segment with data.

5 Skips over any leading zero-length segments.

6 Stores the next memory buffer in the chain of mbuf data structures in the ms mbuf pointer. The m_next member stores the next memory buffer in the chain. Network device drivers typically reference this member through the alternate name m_next, which is defined in the mbuf.h header file.

7 If this is a zero-length transmit, calls the m_freem() routine to free the mbuf buffer chain.

8 Stores the first memory buffer in the chain of mbuf data structures in the mp mbuf pointer.

9 Stores the next memory buffer in the chain of mbuf data structures in the mn mbuf pointer.

10 Stores the amount of data in the mp mbuf in the len variable. The mh_len member of the mbuf data structure pointer stores the amount of data in this mbuf data structure. Network device drivers typically reference this member through the alternate name m_len, which is defined in the mbuf.h header file.

11 While the mn mbuf is not NULL, manipulates the mh_len and mh_next members of the mbuf data structure to eliminate any zero-length buffers

in the middle. The `mfree()` routine is called to free any zero-length memory buffers.

12 Otherwise, adds the length and sets the next memory buffer in the chain to the `mp` mbuf pointer.

9.2.3 Transmitting the Buffer

The following code shows how the `el_start_locked()` routine transmits the buffer:

```
WRITE_DATA(sc, len | TX_INT); 1
dat = mtod(ms, unsigned char *);
len = ms->m_len;
while (ms != NULL) {

    io_blockwrite((vm_offset_t)dat, 2
                sc->data,
                (u_long)(len & ~3),
                HANDLE_LONGWORD);
    dat += (len & ~3);

    ms = ms->m_next;

    i = len % 4; 3
    if (ms == NULL) {
        if (i) {
            val = 0;
            for (j=0; j<i; j++)
              val |= (*dat++ << (8*j));
            WRITE_DATA(sc, val);
        }
    } else {
        if (i) {
            val = 0;

            for (j=0; j<i; j++)
              val |= (*dat++ << (8*j));
            dat = mtod(ms, unsigned char *);

            if (ms->m_len <= (4-i)) {
                for (j=0; j<ms->m_len; j++)
                  val |= (*dat++ << (8*(j+i)));
                ms = NULL;
            } else {
                len = ms->m_len - (4-i);
                for (j=i; j<4; j++)
                  val |= (*dat++ << (8*j));
            }
            WRITE_DATA(sc, val);
        } else {

            dat = mtod(ms, unsigned char *);
            len = ms->m_len;
        }
    }
}
```

1 Requests an interrupt upon completion of the transmit operation.

2 Copies transmit data from memory to the card using 32-bit writes. Only a multiple of 4 bytes can be copied this way.

3 If some number of bytes (fewer than 4) remain in the current memory buffer, the driver either copies those bytes directly to the card (if they were the last bytes for the entire frame), or combines those bytes with bytes from the next memory buffer (if there is more data for this frame).

9.2.4 Accounting for Outgoing Bytes

The following code shows how the `el_start_locked()` routine accounts for the outgoing bytes:

```
sc->txfree -= ((m->m_pkthdr.len + 3) & ~0x3);  1
sc->txfree -= 4;
```

1 Maintains the number of bytes free in the transmit FIFO.

9.2.5 Updating Counters, Freeing the Transmit Buffer, and Marking the Output Process as Active

The following code shows how the `el_start_locked()` routine updates counters, frees the transmit buffer, and marks the output process as active:

```
            ADD_XMIT_PACKET(ifp, sc, m->m_pkthdr.len);  1
            eh = mtod(m, struct ether_header *);
            if (eh->ether_dhost[0] & 0x1) {
                ADD_XMIT_MPACKET(ifp, sc, m->m_pkthdr.len);
            }

            m_freem(m);  2

            ifp->if_flags |= IFF_OACTIVE;

        } else if (m) {  3
            IF_PREPEND(&ifp->if_snd, m);
            break;
        } else
          break;
    }
```

1 Updates the counters using the `ADD_XMIT_PACKET` and possibly the `ADD_XMIT_MPACKET` (for multicast packets) macros. These macros are defined in the `lan_common.h` file. Most network drivers perform this task in the transmit complete interface.

2 Calls the `m_freem()` routine to free the `mbuf` buffer. Network drivers must free the buffer after the transmit operation is complete.

3 If there is no room for this transmit, puts the `mbuf` back on the queue.

9.2.6 Indicating When to Start the Watchdog Routine

The following code shows how the `el_start_locked()` routine indicates the time for starting the driver's `watchdog` interface. Although this task is optional, we recommend that all network drivers perform this task.

```
    ifp->if_timer = 3;  1
}
```

1 Sets the time (in seconds) for starting the `if_el` driver's `watchdog()` routine, called `el_watch()`. After the transmit complete interrupt is received, the interrupt service routine sets `if_timer` back to zero, thereby disabling the `watchdog` timer.

10

Implementing a Watchdog Section

Network device drivers can take advantage of the `watchdog` timer. The network layer implements this mechanism to ensure that the network device is transmitting data. The driver starts the `watchdog` timer when it sends a transmit request to the device. After it receives the transmit completion interrupt, the driver stops the timer. If the interrupt never happens, the timer expires and the driver's `watchdog` interface is called.

The `watchdog` timer is implemented using the `if_timer` member of the device's `ifnet` data structure. The value stored there represents the number of seconds to wait for the transmit to complete. Once per second, the network layer examines this value. If it is 0 (zero), then the timer is disabled. Otherwise, the value is decremented, and if it reaches 0 (zero), the driver's `watchdog` interface is called.

The watchdog section of a network device driver is an optional interface, but we recommend that all network drivers have one.

The `if_el` device driver implements a `watchdog()` routine called `el_watch()`, which performs the following tasks:

- Sets the IPL and obtains the simple lock (Section 10.1)
- Increments the transmit timeout counter and calls the `el_reset_locked()` routine to reset the unit (Section 10.2)
- Releases the simple lock and resets the IPL (Section 10.3)

10.1 Setting the IPL and Obtaining the Simple Lock

The following code shows how to set up the `el_watch()` routine and shows how `el_watch()` sets the IPL and obtains the simple lock.

```
static int el_watch(int unit)
{
   register struct el_softc *sc = el_softc[unit];
   register struct ifnet *ifp = &sc->is_if;
   int s;
   s = splimp();  1
   simple_lock(&sc->el_softc_lock);  2
```

1 Calls the `splimp()` routine to mask all LAN hardware interrupts. On successful completion, `splimp()` stores an integer value in the `s` variable. This integer value represents the CPU priority level that existed prior to the call to `splimp()`.

2   Calls the `simple_lock()` routine to assert a lock with exclusive access for the resource that is associated with the `el_softc_lock` simple lock data structure pointer. This means that no other kernel thread can gain access to the locked resource until you call `simple_unlock()` to release it. Because simple locks are spin locks, `simple_lock()` does not return until the lock has been obtained.

10.2 Incrementing the Transmit Timeout Counter and Resetting the Unit

The following code shows how the `el_watch()` routine counts the number of transmit timeouts, clears the timer, and resets the unit:

```
sc->xmit_tmo++;  1

ifp->if_timer = 0;
el_reset_locked(sc, ifp, unit);  2
```

1   Increments the transmit timeout counter, which stores the number of times transmit timeouts occur.

2   Calls the `el_reset_locked()` routine to reset the device.

10.3 Releasing the Simple Lock and Resetting the IPL

The following code shows how the `el_watch()` routine releases the simple lock and resets the IPL.

```
simple_unlock(&sc->el_softc_lock);
splx(s);
return(0);
}
```

11

Implementing the Reset Section

The reset section of a network device driver contains the code that resets the LAN adapter when there is a network failure and there is a need to restart the device. It resets all of the counters and local variables and can free up and reallocate all of the buffers that the network driver uses.

The `if_el` device driver implements the following routines in its reset section:

- `el_reset()` (Section 11.1)
- `el_reset_locked()` (Section 11.2)

11.1 Implementing the el_reset Routine

The `el_reset()` routine is a jacket routine that performs the following tasks:

- Determines whether the user removes the PCMCIA card from the slot
- Sets the IPL and obtains the simple lock
- Calls the `el_reset_locked()` routine to reset the device
- Releases the simple lock and resets the IPL

The following code shows how this is done:

```
static void el_reset(int unit)
{
    struct el_softc *sc = el_softc[unit];
    struct ifnet *ifp = &sc->is_if;
    int s;

    if (sc->cardout) return; 1
    s = splimp(); 2
    simple_lock(sc->el_softc_lock);

    el_reset_locked(sc, ifp, unit); 3

    simple_unlock(sc->el_softc_lock); 4
    splx(s);
```

1   If the user has removed the PCMCIA card from the slot, returns to the calling routine.

2   Calls the `splimp()` routine to mask all LAN hardware interrupts before obtaining the simple lock for the `el_softc` resource.

3 Calls the `el_reset_locked()` routine, which performs the actual tasks that are associated with resetting the device.

4 Calls the `simple_unlock()` routine to release the simple lock for the `el_softc` data structure and then resets the CPU priority to the level that it was originally at upon entrance to this routine.

11.2 Implementing the el_reset_locked Routine

The following code shows how the `el_reset_locked()` routine resets and restarts the hardware:

```
static void el_reset_locked(struct el_softc *sc,
       struct ifnet *ifp,
                            int unit)
{
   ifp->if_flags &= ~IFF_RUNNING;  1
   el_init_locked(sc, ifp, unit);  2
}
```

1 Indicates that the device is no longer running by clearing the `IFF_RUNNING` bit in the interface flags structure member.

2 Calls the `el_init_locked()` routine. See Section 8.2 for a description of the `el_init_locked()` routine.

12

Implementing the ioctl Section

The ioctl section of a network device driver contains the code that implements a network device driver's `ioctl` interface. The `ioctl` interface performs miscellaneous tasks that have nothing to do with data packet transmission and reception. Typically, it turns specific features of the hardware on or off.

The `el_ioctl()` routine performs the following tasks:

- Determines whether the user has removed the PCMCIA card from the slot (Section 12.2)

- Sets the IPL and obtains the simple lock (Section 12.3)

- Recognizes the `ioctl` command and performs the appropriate operations. Table 12–1 lists the `ioctl` commands that network device drivers must recognize.

- Releases the simple lock and resets the IPL (Section 12.17)

Table 12–1: Network ioctl Commands

ioctl Command	Required	Description	For More Information
SIOCENABLBACK	No	Enables loopback mode.	Section 12.4
SIOCDISABLBACK	No	Disables loopback mode.	Section 12.5
SIOCRPHYSADDR	Yes	Returns the current and default MAC addresses.	Section 12.6
SIOCSPHYSADDR	Yes	Sets the local MAC address.	Section 12.7
SIOCADDMULTI	Yes	Adds the device to a multicast group.	Section 12.8
SIOCDELMULTI	Yes	Removes the device from a multicast group.	Section 12.9
SIOCRDCTRS	Yes	Reads counters.	Section 12.10
SIOCRDZCTRS	Yes	Reads and zeros counters.	Section 12.10
SIOCSIFADDR	Yes	Brings up the device.	Section 12.11

Table 12–1: Network ioctl Commands (cont.)

ioctl Command	Required	Description	For More Information
SIOCSIFFLAGS	Yes	Ensures that the interface is operating correctly according to the interface flags (if_flags).	Section 12.12
SIOCSIPMTU	Yes	Sets the IP maximum transmission unit (MTU).	Section 12.13
SIOCSMACSPEED	Yes	Sets the media speed.	Section 12.14
SIOCIFRESET	No	Resets the device.	Section 12.15
SIOCIFSETCHAR	Yes	Sets network device characteristics, such as full duplex or promiscuous mode.	Section 12.16

12.1 Setting Up the el_ioctl Routine

The following code shows how to set up the el_ioctl() routine:

```
static int el_ioctl(struct ifnet *ifp,  1
                    u_int cmd,  2
                    caddr_t data)  3
{
    register struct el_softc *sc = el_softc[ifp->if_unit];  4
    register unit = ifp->if_unit;  5
    struct ifreq *ifr = (struct ifreq *)data;  6
    struct ifdevea *ifd = (struct ifdevea *)data;  7
    struct ctrreq *ctr = (struct ctrreq *)data;  8
    struct ifchar *ifc = (struct ifchar *)data;  9
    int s, i, j, need_reset, lock_on = 1, status = ESUCCESS;  10
    unsigned short ifmtu, speed;  11
    u_char mclist_buf[NET_SZ_MCLIST];  12
```

1 Specifies a pointer to the ifnet data structure for an if_el device.

2 Specifies the ioctl command.

3 Specifies a pointer to ioctl command-specific data to be passed to or initialized by the device driver.

4 Declares a pointer to the el_softc data structure that is called sc and initializes it to the el_softc data structure for this device.

5 Declares a unit variable and initializes it to the unit number for the device.

6 Casts the data argument to a data structure of type ifreq for use with the SIOCPHYSADDR, SIOCADDMULTI, SIOCDELMULTI, SIOCSIPMTU, and SIOCSMACSPEED ioctl commands.

[7] Casts the data argument to a data structure of type ifdevea for use
with the SIOCRPHYSADDR ioctl command.

[8] Casts the data argument to a data structure of type ctrreq for use
with the SIOCRDCTRS and SIOCRDZCTRS ioctl commands.

[9] Casts the data argument to a data structure of type ifchar for use
with the SIOCIFSETCHAR ioctl command.

[10] Declares a lock_on variable and sets it to the value 1 (true), which
indicates that the simple lock is held. The el_ioctl() routine sets this
variable to the value 0 (false) when the simple lock is no longer in effect.

Declares a status variable and sets it to the constant ESUCCESS.

[11] Declares an ifmtu variable that stores the requested MTU value for
the SIOCIPMTU command.

Declares a speed variable that stores the requested network speed
for the SIOCMACSPEED command.

[12] Declares an mclist_buf buffer, which holds a character string. This
string is a list of all multicast addresses currently in use on the device.

12.2 Determining Whether the User Has Removed the PCMCIA Card from the Slot

The following code shows how the el_ioctl() routine determines whether
the user has removed the PCMCIA card from the slot:

```
if (sc->cardout) return(EIO);  [1]
```

[1] Examines the value of the cardout member of the el_softc data
structure for this device. If it is set to 1 (true), the user has removed
the PCMCIA card from the slot, and the driver returns the EIO error
constant to indicate an I/O error.

12.3 Setting the IPL and Obtaining the Simple Lock

The following code shows how the el_ioctl() routine sets the IPL and
obtains the simple lock:

```
s = splimp();  [1]
simple_lock(&sc->el_softc_lock);  [2]
```

[1] Calls the splimp() routine to mask all LAN hardware interrupts.
On successful completion, splimp() stores an integer value in the s
variable that represents the CPU priority level that existed before the
call to splimp().

[2] Calls the simple_lock() routine to assert a lock with exclusive access
for the resource that is associated with el_softc_lock. This means
that no other kernel thread can gain access to the locked resource until

you call `simple_unlock()` to release it. Because simple locks are spin locks, `simple_lock()` does not return until the lock has been obtained.

12.4 Enabling Loopback Mode (SIOCENABLBACK ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCENABLBACK` ioctl command to enable loopback mode when an application requests it. Support for the `SIOCENABLBACK` command is optional. You can choose whether or not your driver supports it.

```
switch (cmd) {  1

case SIOCENABLBACK:  2
    ifp->if_flags |= IFF_LOOPBACK;  3
    if (ifp->if_flags & IFF_RUNNING)  4
        el_reset_locked(sc, ifp, unit);
    break;
```

1 Evaluates the value passed in through the `cmd` argument to determine which ioctl command the caller has requested.

2 Determines whether the `cmd` argument is `SIOCENABLBACK`.

3 Sets the `IFF_LOOPBACK` bit in the `if_flags` member of the `ifnet` data structure for this device.

4 If the device is running, calls the `el_reset_locked()` routine to restart the network interface in loopback mode.

12.5 Disabling Loopback Mode (SIOCDISABLBACK ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCDISABLBACK` ioctl command to disable loopback mode when an application requests it. Support for the `SIOCDISABLBACK` command is optional. However, if your driver supports `SIOCENABLBACK`, it must support `SIOCDISABLBACK`.

```
case SIOCDISABLBACK:  1
    ifp->if_flags &=  ~IFF_LOOPBACK;  2
    if (ifp->if_flags & IFF_RUNNING)  3
        el_reset_locked(sc, ifp, unit);
    break;
```

1 Determines whether the `cmd` argument is `SIOCDISABLBACK`.

2 Clears the `IFF_LOOPBACK` bit in the `if_flags` member of the `ifnet` data structure for this device.

3 If the device is running, calls the `el_reset_locked()` routine. The `el_reset_locked()` routine calls `el_init_locked()`, which restarts the network interface in normal mode.

12.6 Reading Current and Default MAC Addresses (SIOCRPHYSADDR ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCRPHYSADDR ioctl` command to read the current and default MAC addresses when an application requests them:

```
case SIOCRPHYSADDR: 1
    bcopy(sc->is_addr, ifd->current_pa, 6); 2
    for (i=0; i<3; i++) { 3
        j = sc->eeprom.addr[i];
        ifd->default_pa[(i*2)] = (j>>8) & 0xff;
        ifd->default_pa[(i*2)+1] = (j) & 0xff;
    }
    break;
```

1 Determines whether the `cmd` argument is `SIOCRPHYSADDR`.

2 Copies the current MAC address that is stored in the `el_softc` data structure for this device to the `ifd` data structure, a command-specific data structure of type `ifdevea`.

3 Copies the default MAC address that is stored in the driver's `el_softc` data structure for this device to the `ifd` data structure.

12.7 Setting the Local MAC Address (SIOCSPHYSADDR ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCSPHYSADDR ioctl` command to set the local MAC address:

```
case SIOCSPHYSADDR: 1
    bcopy(ifr->ifr_addr.sa_data, sc->is_addr, 6); 2

    pfilt_newaddress(sc->is_ed.ess_enetunit, sc->is_addr); 3

    if (ifp->if_flags & IFF_RUNNING) { 4
        el_reset_locked(sc, ifp, unit);
    }

    simple_unlock(&sc->el_softc_lock); 5
    splx(s); 6
    lock_on = 0; 7

    if (((struct arpcom *)ifp)->ac_flag & AC_IPUP) { 8
        rearpwhohas((struct arpcom *)ifp);
    }

    if_sphyaddr(ifp, ifr); 9
    break;
```

1 Determines whether the `cmd` argument is `SIOCSPHYSADDR ioctl`.

2 Copies the new MAC address to the `ifnet` data structure.

3 Calls the `pfilt_newaddress()` routine to copy the new address to the packet filter.

4   If the 3Com 3C5x9 device is running, calls the `el_reset_locked()` routine to restart the network interface with the new address.

5   Calls the `simple_unlock()` routine to release the simple lock for the resource that is associated with `el_softc_lock`.

6   Calls the `splx()` routine to reset the CPU priority to the level that the `s` variable specifies.

7   Sets the `lock_on` variable to 0 (false), which indicates that the simple lock is no longer held.

8   If an IP address was configured, broadcasts an ARP packet to notify all hosts that currently have this address in their ARP tables to update their information.

9   Notifies the network layer about a possible change in the `af_link` address.

12.8 Adding the Device to a Multicast Group (SIOCADDMULTI ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCADDMULTI ioctl` command to add a multicast address:

```
case SIOCADDMULTI: 1

    need_reset = 0;
    if (bcmp(ifr->ifr_addr.sa_data, etherbroadcastaddr, 6) == 0) { 2
        sc->is_broadcast++;
    } else {
        i = lan_add_multi(&sc->is_multi,
                          (unsigned char *)ifr->ifr_addr.sa_data);
        switch (i) {
         case LAN_MULTI_CHANGED:
            if (sc->is_multi.lan_nmulti == 1) 3
                need_reset++;
            break;
         case LAN_MULTI_NOT_CHANGED:
            break;
         case LAN_MULTI_FAILED:
         default:
            status = EINVAL;
            break;
        }
    }

    if ((ifp->if_flags & IFF_RUNNING) && (need_reset)) 4
        el_reset_locked(sc, ifp, unit);

    if (sc->debug) {
        j = 0;
        printf("el%d: Dump of multicast table after ADD (%d entries)\n",
               unit, sc->is_multi.lan_nmulti);
        for (i=0; i<sc->is_multi.lan_nmulti; i++) {
            unsigned char *maddr;

            LAN_GET_MULTI(&sc->is_multi, maddr, j);
            printf("  %d  %s (muse==%d)\n", i+1,
```

```
                    ether_sprintf(maddr),
                    sc->is_multi.lan_mtable[j-1].muse);
        }
    }
    lan_build_mclist (mclist_buf, NET_SZ_MCLIST, &sc->is_multi);  5
    lan_set_attribute(sc->ehm.current_val, NET_MCLIST_NDX, mclist_buf);
    break;
```

1 Determines whether the `cmd` argument is `SIOCADDMULTI`.

2 If the address is broadcast, indicates the presence of another broadcast
   user. If the address is multicast, the `el_ioctl()` routine adds the
   address to the table. The EtherLink III family does not support any
   multicast filtering. Either you receive all multicast addresses or you
   do not receive any. The EtherLink III family does special-case the
   broadcast address.

3 If the add succeeds and there are no other multicasts enabled,
   increments a counter that indicates that the device needs to be reset.

4 If the device is running and multicasts and broadcasts have not already
   been enabled, enables them.

5 Builds a text string that lists all currently active multicast addresses,
   and sets this list as an enhanced hardware management (EHM)
   attribute for this network device.

12.9 Deleting the Device from a Multicast Group (SIOCDELMULTI ioctl Command)

The following code shows how the `el_ioctl()` routine implements the
`SIOCDELMULTI ioctl` command to delete a multicast address:

```
case SIOCDELMULTI:  1

    need_reset = 0;
    if (bcmp(ifr->ifr_addr.sa_data, etherbroadcastaddr, 6) == 0) {  2
        sc->is_broadcast--;
    } else {
        i = lan_del_multi(&sc->is_multi,
                          (unsigned char *)ifr->ifr_addr.sa_data);
        switch (i) {
         case LAN_MULTI_CHANGED:
            if (sc->is_multi.lan_nmulti == 0)
                need_reset++;
            break;
         case LAN_MULTI_NOT_CHANGED:
            break;
         case LAN_MULTI_FAILED:
         default:
            status = EINVAL;
            break;
        }
    }

    if ((ifp->if_flags & IFF_RUNNING) && (need_reset))
        el_reset_locked(sc, ifp, unit);
```

```
if (sc->debug) {
    j = 0;
    printf("el%d: Dump of multicast table after DEL (%d entries)\n",
            unit, sc->is_multi.lan_nmulti);
    for (i=0; i<sc->is_multi.lan_nmulti; i++) {
        unsigned char *maddr;

        LAN_GET_MULTI(&sc->is_multi, maddr, j);
        printf("   %d  %s (muse==%d)\n", i+1, ether_sprintf(maddr),
                sc->is_multi.lan_mtable[j-1].muse);
    }
}
lan_build_mclist (mclist_buf, NET_SZ_MCLIST, &sc->is_multi); 3
lan_set_attribute(sc->ehm.current_val, NET_MCLIST_NDX, mclist_buf);
break;
```

1   Determines whether the `cmd` argument is `SIOCDELMULTI`.

2   Examines the type of the multicast address and decrements the appropriate counter. The `el_ioctl()` routine removes the capability from the device only when there are no more active multicast addresses.

3   Builds a text string that lists all currently active multicast addresses, and sets this list as an enhanced hardware management (EHM) attribute for this network device.

12.10 Accessing Network Counters (SIOCRDCTRS and SIOCRDZCTRS ioctl Commands)

The `SIOCRDCTRS` ioctl command returns the values of network counters. The driver's `softc` data structure stores a pointer to the counter information. The driver returns the information to the caller in a `ctrreq` data structure, which is passed into the `ioctl()` routine through the `data` argument.

The `SIOCRDZCTRS` ioctl command also zeroes the network counters.

The following code shows how the `el_ioctl()` routine implements the `SIOCRDCTRS` and `SIOCRDZCTRS` ioctl commands:

```
case SIOCRDCTRS: 1
case SIOCRDZCTRS:

    ctr->ctr_ether = sc->ctrblk; 2
    ctr->ctr_type = CTR_ETHER; 3
    ctr->ctr_ether.est_seconds = (time.tv_sec - sc->ztime) > 0xfffe ?
                                  0xffff : (time.tv_sec - sc->ztime); 4

    if (cmd == SIOCRDZCTRS) { 5
        sc->ztime = time.tv_sec;
        bzero(&sc->ctrblk, sizeof(struct estat));
    }
    break;
```

1   Determines whether the `cmd` argument is `SIOCRDCTRS` or `SIOCRDZCTRS`.

2. Copies the current counters to the `ctrreq` data structure.

3. Indicates that these are Ethernet counters.

4. Returns the number of seconds since the counters were last zeroed.

5. If the user process requested the `SIOCRDZCTRS` command, zeroes the counters and sets the `ztime` member of the `softc` data structure to the current time. This indicates when the counters were zeroed.

For other types of network interfaces, you can specify a different counter type and a different set of counters. Table 12–2 lists the types of counters that the various network interfaces support.

Table 12–2: Network Interface Counter Types

Network Interface	Counter Types
FDDI	FDDI interface statistics
	Status information
	SMT attributes
	MAC attributes
	Path attributes
	Port attributes
	SMT MIB attributes
	Extended MIB attributes (Compaq proprietary)
Token Ring	Characteristics
	Counters
	MIB counters
	MIB statistics

12.11 Bringing Up the Device (SIOCSIFADDR ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCSIFADDR` ioctl command to bring up the device:

```
case SIOCSIFADDR: 1
    ifp->if_flags |= IFF_UP; 2
    el_reset_locked(sc, ifp, unit);

    if (sc->ztime == 0) sc->ztime = time.tv_sec; 3
    break;
```

1. Determines whether the `cmd` argument is `SIOCSIFADDR`.

2   Marks the interface as up and calls the `el_reset_locked()` routine to start the network interface with the current settings.

3   Sets the counter cleared time (used by DECnet, netstat, clusters, and so forth).

12.12 Using Currently Set Flags (SIOCSIFFLAGS ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCSIFFLAGS ioctl` command to reset the device using currently set flags:

```
case SIOCSIFFLAGS: 1
   if (ifp->if_flags & IFF_RUNNING) 2
      el_reset_locked(sc, ifp, unit);
   break;
```

1   Determines whether the `cmd` argument is `SIOCSIFFLAGS`.

2   If the 3Com 3C5x9 device is running, calls the `el_reset_locked()` routine to restart the network interface with the current flag settings.

12.13 Setting the IP MTU (SIOCSIPMTU ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCSIPMTU ioctl` command to set the IP MTU. You must implement this task in your network driver to accommodate the IP layer.

```
case SIOCSIPMTU: 1
   bcopy(ifr->ifr_data, (u_char *)&ifmtu, sizeof(u_short));
   if (ifmtu > ETHERMTU || ifmtu < IP_MINMTU) 2
     status = EINVAL;
   else {
     ifp->if_mtu = ifmtu;
     lan_set_attribute(sc->ehm.current_val, NET_MTU_NDX, (void *)ifmtu);
   }
   break;
```

1   Determines whether the `cmd` argument is `SIOCSIPMTU`.

2   Compares the passed value to the media's maximum and minimum values. If this value is not within the range allowed, the driver returns an error. Otherwise, it sets the `if_mtu` member of the driver's `ifnet` data structure to the specified IP MTU value. Also, updates the corresponding hardware attribute in the enhanced hardware management (EHM) database.

12.14 Setting the Media Speed (SIOCSMACSPEED ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCSMACSPEED ioctl` command to set the media speed. (The

`SIOCSMACSPEED` **and** `SIOCIFSETCHAR` `ioctl` **commands perform some of the same tasks.)**

```
case SIOCSMACSPEED: 1
    bcopy(ifr->ifr_data, (u_char *)&speed, sizeof(u_short));

    if ((speed != 0) && (speed != 10)) { 2
        status = EINVAL;
        break;
    }
    break;
```

[1] Determines whether the `cmd` argument is `SIOCSMACSPEED`.

[2] If the LAN speed passed is anything other than 10 (0 means no change), fails the request. (The `if_el` device can only operate at 10 Mb per second.)

12.15 Resetting the Device (SIOCIFRESET ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCIFRESET` `ioctl` command to reset the device. Support for the `SIOCIFRESET` command is optional. You can choose whether or not your driver supports it.

```
case SIOCIFRESET: 1
    el_reset_locked(sc, ifp, unit); 2
    break;
```

[1] Determines whether the `cmd` argument is `SIOCIFRESET`.

[2] Calls the `el_reset_locked()` routine to restart the network interface.

12.16 Setting Device Characteristics (SIOCIFSETCHAR ioctl Command)

The following code shows how the `el_ioctl()` routine implements the `SIOCIFSETCHAR` `ioctl` command to set characteristics:

```
case SIOCIFSETCHAR: 1
    need_reset = 0; 2

    if ((ifc->ifc_media_speed != -1) && (ifc->ifc_media_speed != 10)) { 3
        status = EINVAL;
        break;
    }

    if ((ifc->ifc_auto_sense == LAN_AUTOSENSE_ENABLE) && 4
        (ifc->ifc_media_type != -1)) {
        status = EINVAL;
        break;
    }
```

```
        if (ifc->ifc_auto_sense != -1) {  5
            if ((ifc->ifc_auto_sense == LAN_AUTOSENSE_ENABLE) &&
                (sc->lm_media_mode != LAN_MODE_AUTOSENSE)) {
                sc->lm_media_mode = LAN_MODE_AUTOSENSE;
                need_reset++;
            } else if ((ifc->ifc_auto_sense == LAN_AUTOSENSE_DISABLE) &&
                    (sc->lm_media_mode == LAN_MODE_AUTOSENSE)) {
                sc->lm_media_mode = sc->lm_media;  6
                need_reset++;
            }
        }

        if (ifc->ifc_media_type != -1) {  7

            switch (ifc->ifc_media_type) {  8
             case LAN_MEDIA_UTP:
             case LAN_MEDIA_AUI:
             case LAN_MEDIA_BNC:

                if (ifc->ifc_media_type != sc->lm_media)  9
                  need_reset++;
                sc->lm_media_mode = sc->lm_media = ifc->ifc_media_type;
               break;
             default:
                status = EINVAL;

                break;
            }
        }
        if (need_reset && (ifp->if_flags & IFF_RUNNING))  10
          el_reset_locked(sc, ifp, unit);
        break;

    default:  11
        status = EINVAL;
}
```

1 Determines whether the `cmd` argument is `SIOCIFSETCHAR`.

2 Assumes no device reset is necessary.

3 If the LAN speed passed is anything other than 10 (–1 means no change), fails the request.

4 Examines the media mode settings. If the `ioctl` request specifies both autosense enable and an explicit media setting, fails the request.

5 Determines whether autosensing has changed.

6 If autosensing is now disabled, selects the last known media.

7 Determines whether the explicit media type selection has changed.

8 If the requested media value is out of range or not supported by the EtherLink III family, fails the `ioctl` request immediately.

The EtherLink III family supports the usual 802.3 media. The `if_el` driver does not check the card's capability in the registers because it is not useful to do so. The registers always indicate they have all media, regardless of what they really have.

If the user sets media that the card does not have, the interface may not work.

[9] Selects the new mode.

[10] Resets the device to pick up the new mode (if the interface was running).

[11] The default case returns an error that indicates that the caller has issued an invalid `ioctl` command.

12.17 Releasing the Simple Lock and Resetting the IPL

The following code shows how the `el_ioctl()` routine releases the simple lock and resets the IPL:

```
if (lock_on) {  1
    simple_unlock(&sc->el_softc_lock);
    splx(s);
}
return (status);  2
}
```

[1] If the simple lock is still held, calls the `simple_unlock()` routine.

[2] Returns the status of the `ioctl` request.

13

Implementing the Interrupt Section

The interrupt section of a network device driver contains the code that is called whenever the network interface transmits or receives a frame.

The `if_el` device driver implements the following routines in its interrupt section:

- `el_intr()` (Section 13.1)
- `el_rint()` (Section 13.2)
- `el_tint()` (Section 13.3)
- `el_error()` (Section 13.4)

_____ **Note** _____

The `if_el` device driver implements a shared interrupt handler. A shared interrupt handler is a driver routine that is registered to take advantage of the shared interrupt framework that Tru64 UNIX provides for hardware devices that share an interrupt line.

The ISA bus does not currently support shared interrupts.

_____

13.1 Implementing the el_intr Routine

The `if_el` device driver implements an interrupt handler called `el_intr()`, which performs the following tasks:

- Sets the interrupt and priority level (IPL) and obtains the simple lock (Section 13.1.1)
- Rearms the next timeout (Section 13.1.2)
- Reads the interrupt status (Section 13.1.3)
- Processes completed receive and transmit operations (Section 13.1.4)
- Acknowledges the interrupt (Section 13.1.5)
- Transmits pending frames (Section 13.1.6)
- Releases the simple lock and resets the IPL (Section 13.1.7)
- Indicates that the interrupt was serviced (Section 13.1.8)

13.1.1 Setting the IPL and Obtaining the Simple Lock

The following code shows how the `el_intr()` routine sets the CPU's IPL
and obtains the simple lock:

```
static int el_intr(int unit)  1
{
   register u_int s;
   volatile u_int status;
   register struct el_softc *sc = el_softc[unit];
   register struct ifnet *ifp = &sc->is_if;

   if (el_card_out(sc)) return (INTR_NOT_SERVICED);  2

   s = splimp();  3
   simple_lock(&sc->el_softc_lock);  4
```

1 Declares an argument that specifies the unit number of the network
   interface that generated the interrupt.

2 Determines whether the card is still in the socket. If the card is no
   longer in the socket, then returns the constant `INTR_NOT_SERVICED` to
   the kernel interrupt dispatcher.

3 Calls the `splimp()` routine to mask all Ethernet hardware interrupts.

4 Calls the `simple_lock()` routine to assert a lock with exclusive access
   for the resource that is associated with `el_softc_lock`.

13.1.2 Rearming the Next Timeout

The following code shows how the `el_intr()` routine rearms the next
timeout:

```
   if (sc->polling_flag) 1
      timeout((void *)el_intr, (void *)unit, (1*hz)/el_pollint); 2
```

1 Determines whether polling was started by testing the `polling_flag`
   flag member in the `el_softc` data structure for this device.

2 If the polling process was started, calls the `timeout()` routine to
   rearm the next timeout. The `timeout()` routine is called with the
   following arguments:

   • A pointer to the `el_intr()` routine, the `if_el` device driver's
     interrupt handler.

   • The `unit` variable, which contains the controller number for this
     device. This argument is passed to the `el_intr()` routine.

   • The `el_pollint` variable, which specifies the amount of time to
     delay before calling the `el_intr()` routine.

13.1.3 Reading the Interrupt Status

The following code shows how the `el_intr()` routine uses the `READ_STS` macro to read the interrupt status from the I/O status register:

```
status = READ_STS(sc);
```

13.1.4 Processing Completed Receive and Transmit Operations

The following code shows how the `el_intr()` routine processes the receive and transmit rings:

```
if (((status & (S_RC|S_TC|S_AF)) == 0) || sc->cardout) { 1
    simple_unlock(&sc->el_softc_lock);
    splx(s);
    return INTR_NOT_SERVICED;
}
while ((status & (S_RC|S_TC|S_AF)) && (!el_card_out(sc))) { 2
        if (status & S_RC)
            el_rint(sc, ifp);
        if (status & S_TC)
            el_tint(sc, ifp);
        if (status & S_AF)
            el_error(sc, ifp);
        status = READ_STS(sc);
}
```

1 Examines the status that the `READ_STS` macro returns.

   If the `status` variable does not have the receive complete (`S_RC`) bit, the transmit complete (`S_TC`) bit, or the adapter failure (`S_AF`) bit set, or if the PCMCIA card is out of the slot:

   - Calls the `simple_unlock()` routine to release the simple lock for the resource that is associated with `el_softc_lock`.

   - Calls the `splx()` routine to reset the CPU priority to the level that the `s` variable specifies.

   - Returns the constant `INTR_NOT_SERVICED` to the kernel interrupt dispatcher. This constant indicates that this shared interrupt was not for the `if_el` device.

2 While the `status` variable has the receive complete (`S_RC`) bit, the transmit complete (`S_TC`) bit, or the adapter failure (`S_AF`) bit set, and if the card has not been removed from the machine:

   - If the `status` variable has the `S_RC` bit set, calls the `el_rint()` routine to process the receive interrupt.

   - If the `status` variable has the `S_TC` bit set, calls the `el_tint()` routine to process the transmit interrupt.

- If the `status` variable has the `S_AF` bit set, calls the `el_error()` routine to process the error.

- Calls the `READ_STS` macro to read the interrupt status again from the I/O status register.

13.1.5 Acknowledging the Interrupt

The following code shows how the `el_intr()` routine acknowledges the interrupt:

```
WRITE_CMD(sc, CMD_ACKINT+(S_IL)); 1
```

1 Calls the `WRITE_CMD` macro to write data to the command port register. In this call, the `regE` member of the `el_softc` data structure specifies the I/O handle that references the register in bus address space. The acknowledge interrupt (`CMD_ACKINT`) and interrupt latch (`S_IL`) bits specify the data to be written.

13.1.6 Transmitting Pending Frames

The following code shows how the `el_intr()` routine transmits pending frames:

```
if (ifp->if_snd.ifq_head) { 1
    el_start_locked(sc, ifp);
} else {
    ifp->if_timer = 0; 2
}
```

1 Determines whether there are any transmits pending. If so, `el_intr()` calls `el_start_locked()` to start the transmit operation.

2 Otherwise, disables the watchdog timer by setting the `el_timer` member of the `ifnet` data structure to 0 (zero).

13.1.7 Releasing the Simple Lock and Resetting the IPL

The following code shows how the `el_intr()` routine releases the simple lock and resets the IPL:

```
simple_unlock(&sc->el_softc_lock);
splx(s);
```

13.1.8 Indicating That the Interrupt Was Serviced

The following code shows how the `el_intr()` routine indicates that the interrupt was serviced:

```
    return INTR_SERVICED; 1
}
```

1 Returns the `INTR_SERVICED` constant to the kernel interrupt dispatcher to indicate that `el_intr()` serviced the shared interrupt.

13.2 Implementing the el_rint Routine

The `if_el` driver's `el_rint()` routine is the receive interrupt completion routine. It performs the following tasks:

- Counts the receive interrupt and reads the receive status (Section 13.2.1)
- Pulls the packets from the FIFO buffer (Section 13.2.2)
- Examines the first part of the packet (Section 13.2.3)
- Copies the received packet into the `mbuf` (Section 13.2.4)
- Discards a packet (Section 13.2.5)

13.2.1 Counting the Receive Interrupt and Reading the Receive Status

The following code shows how the `el_rint()` routine counts the receive interrupt and reads the receive status:

```
#define RXLOOP ((16*1024)/64) 1

static void el_rint(struct el_softc *sc,
        struct ifnet *ifp)
{
    int len, i, count=RXLOOP;
    volatile short status;
    struct mbuf *m;
    unsigned char *dat;
    unsigned int in;
    struct ether_header eh;

    sc->rint++; 2

    status = READ_RXS(sc); 3
```

1 Defines a constant that represents the maximum number of packets in a 16K receive buffer.

2 Increments the receive interrupt counter.

3 Calls the `READ_RXS` macro to read the receive status.

13.2.2 Pulling the Packets from the FIFO Buffer

The following code shows how the `el_rint()` routine pulls the packets
from the first-in/first-out (FIFO) buffer. This task is specific to the hardware
device that is associated with the `if_el` device driver. If you need to perform
a similar task with your hardware device, use this example as a model.

```
while ((status > 0) && (count-- > 0)) { 1
    len = status & RX_BYTES;
    if ((status & RX_ER) || (len > 1518) || (len < 60)) { 2
        if (status & RX_ER) { 3
            status &= RX_EM;
            if (sc->ctrblk.est_recvfail != 0xffff)
              sc->ctrblk.est_recvfail++;
            switch (status) {
case RX_EOR: 4
                if (sc->ctrblk.est_overrun != 0xffff)
                  sc->ctrblk.est_overrun++;
                if (sc->debug)
                  printf("el%d: Overrun\n", ifp->if_unit);
                break;

            case RX_ERT: 5
            case RX_EOS:
              sc->ctrblk.est_recvfail_bm |= 4;
              if (sc->debug)
                  printf("el%d: Bad Sized packet\n", ifp->if_unit);
              break;
            case RX_ECR: 6
              sc->ctrblk.est_recvfail_bm |= 1;
              if (sc->debug)
                  printf("el%d: CRC\n", ifp->if_unit);
              break;
            case RX_EAL: 7
            default:
              sc->ctrblk.est_recvfail_bm |= 2;
              if (sc->debug)
                  printf("el%d: Alignment\n", ifp->if_unit);
              break;
            }
        } else
          if ((sc->debug) && (len != 0)) 8
              printf("el%d: Received illegal size packet (%d)\n",
                    ifp->if_unit, len);

    } else {
        if (len <= MHLEN-2-4) { 9
          MGETHDR(m, M_DONTWAIT, MT_DATA);
        } else {
          MGETHDR(m, M_DONTWAIT, MT_DATA);
          if (m) {
              MCLGET2(m, M_DONTWAIT);
              if ((m->m_flags & M_EXT) == 0) {
                  m_freem(m);
                  m = (struct mbuf *)NULL;
              }
          }
        }
    }
```

1 Sets up a `while` loop that executes as long as there are complete
packets.

2 Looks for errors.

3 Processes the error.

4 Processes the overrun error case.

5 Processes the runt and oversized error cases.

6 Processes the CRC error case.

7 Processes the alignment error case.

8 Discards the packet if none of the previous cases apply. This indicates a size error.

9 Allocates a buffer for the received data. If the length of the received data is less than a small `mbuf`, allocates a small `mbuf`. Otherwise, a 2K cluster `mbuf` is allocated. This code is an optimization. In most cases, a driver does not know the size of a receive packet when the buffer resource is allocated.

13.2.3 Examining the First Part of the Packet

The following code shows how the `el_rint()` routine examines the first part of the received packet:

```
if (m != NULL) {  1
    m->m_pkthdr.len = m->m_len = len - sizeof(struct ether_header); 2
    m->m_pkthdr.rcvif = ifp;
    m->m_data += 2;  3

    dat = mtod(m, unsigned char *);  4
    len = (len + 3) & ~3;

    if ((ifp->if_flags & (IFF_PROMISC|IFF_ALLMULTI)) == 0) {  5
        io_blockread(sc->data,
                     (vm_offset_t)dat,
                     2UL*4UL,
                     HANDLE_LONGWORD);  6
        len -= (2*4);
        dat += (2*4);
        if (*mtod(m, unsigned char *) & 0x01) {  7
```

```
                 if (bcmp(mtod(m, unsigned char *),
                      etherbroadcastaddr, 6) != 0) {  8
                 int ix;
                 LAN_FIND_MULTI(&sc->is_multi,
                               mtod(m, unsigned char *),
                               ix, i);  9

                 if ( (i != LAN_MULTI_FOUND) ||  10
                      (sc->is_multi.lan_mtable[ix].muse == 0)) {
                 m_freem(m);
                 goto scrap;
                 }
              }
          }
       }
```

1. If an `mbuf` was successfully allocated, copies the packet data into the `mbuf` (receive data are 32-bit aligned).

2. Computes the length of the received data, excluding the size of the MAC header. Records this length in the `mbuf` header. Sets the receiving interface to be the `if_el` device by saving the `if_el` device's `ifnet` data structure address in the `mbuf` header.

3. Aligns the data pointer so that the IP header will be aligned on a 32-bit boundary. Make sure that your network driver does this also.

4. Obtains the pointer to the data and calculates the number of longwords in the FIFO transfer.

5. Because the EtherLink III performs no multicast filtering, if the promiscuous bit and all multicast bits are not set, determines whether any multicast addresses are actually wanted.

6. Reads the first two longwords to determine whether the packet is sent to a multicast address.

7. Determines whether the packet contains either a multicast or a broadcast group address.

8. Because the driver receives all broadcasts, makes sure that the group address is not the broadcast address.

9. Calls the `LAN_FIND_MULTI` macro to find the multicast address.

10. If the multicast is not found, scraps the packet.

13.2.4 Copying the Received Packet into the mbuf

The following code shows how the `el_rint()` routine copies the received packet into the `mbuf`:

```
        io_blockread(sc->data,
                    (vm_offset_t)dat,
                    (u_long)len,
                    HANDLE_LONGWORD);  1
```

```
        eh = *(mtod(m, struct ether_header *));  2
        eh.ether_type = ntohs((unsigned short)eh.ether_type);  3
        m->m_data += sizeof(struct ether_header);  4

        ADD_RECV_PACKET(ifp, sc, m->m_pkthdr.len);  5
        if (eh.ether_dhost[0] & 0x1) {
            ADD_RECV_MPACKET(ifp, sc, m->m_pkthdr.len);
        }

            i = READ_RXS(sc);  6
            if (i &= 0x7ff) {
                if ((i & 0x400) == 0) {
                    m_freem(m);
                    goto scrap;
                }
            }
            ether_input(ifp, &eh, m);  7
        }
    }
```

1 Calls the `io_blockread()` routine to perform the data transfer from the FIFO buffer on the adapter to the `mbuf` in host memory.

2 Makes a copy of the `ether_header` data structure for the `ether_input()` routine.

3 Converts the 2-byte `ether_type` field from network byte order to host byte order and saves it in the `ether_header` data structure.

4 Adjusts the pointer to the received data to point past the MAC header (skips past the destination address, source address, and `ether_type` fields).

5 Calls the `ADD_RECV_PACKET` macro to increment the receive packet (block) count. If this packet was destined for a broadcast or multicast address, calls the `ADD_RECV_MPACKET` macro to increment those statistics as well.

6 Calls the `READ_RXS` macro to read the receive status. If the packet just received was not fully received, scraps the packet.

7 Calls the `ether_input()` routine to process the received Ethernet packet. The packet is in the `mbuf` chain without the `ether_header` data structure, which is provided separately.

13.2.5 Discarding a Packet

The following code shows how the `el_rint()` routine discards a packet. Some receive `interrupt` handlers perform a copy of a few bytes of the packet to determine if the packet is actually destined for the device. Thus, this task is an optional optimization.

```
scrap:
        WRITE_CMD(sc, CMD_RXDTP);  1
        status = READ_RXS(sc);
    }
```

```
    if ((sc->debug) && (count <= 0))
        printf("el%d: Receive in INFINITE loop %04X\n", ifp->if_unit, status);
}
```

1 Calls the `WRITE_CMD` macro to write data to the command port register.
The data to be written is the receive discard top packet command
(`CMD_RXDTP`).

13.3 Implementing the el_tint Routine

The `if_el` device driver's `el_tint()` routine is the transmit interrupt
completion routine. It performs the following tasks:

- Counts the transmit interrupt (Section 13.3.1)
- Reads the transmit status and counts all significant events
  (Section 13.3.2)
- Manages excessive data collisions (Section 13.3.3)
- Writes to the status register to obtain the next value (Section 13.3.4)
- Queues other transmits (Section 13.3.5)

13.3.1 Counting the Transmit Interrupt

The following code shows how the `el_tint()` routine counts the transmit
interrupt:

```
#define TXLOOP ((16*1024)/64)
static void el_tint(struct el_softc *sc,
        struct ifnet *ifp)
{
    int count=TXLOOP;
    volatile unsigned int status;

    sc->tint++; 1
```

1 Increments a counter of the number of the transmit interrupts that
have been processed.

13.3.2 Reading the Transmit Status and Counting All Significant Events

The following code shows how the `el_tint()` routine reads the transmit
status and counts all significant events:

```
    status = READ_TXS(sc); 1
    while ((status & (TX_CM<<8)) && (count-- > 0)) {

        if (status & ((TX_JB|TX_UN)<<8)) { 2
            ifp->if_oerrors++;
            sc->ctrblk.est_sendfail++;
            sc->txreset++;

            WRITE_TXS(sc, status); 3
```

```
        WRITE_CMD(sc, CMD_TXRESET);
        DELAY(10);        4
        WRITE_CMD(sc, CMD_TXENA);
```

1. Calls the READ_TXS macro to read the transmit status from the transmit status register.

2. Examines the status for a jabber or an underrun error. If either of these errors happened, then the transmitter must be reset.

3. Clears the transmit status register and resets the transmitter.

4. Calls the DELAY macro to wait for 10 microseconds before reenabling the transmitter.

13.3.3 Managing Excessive Data Collisions

The following code shows how the el_tint() routine manages excessive data collisions:

```
} else if (status & (TX_MC<<8)) {

    ifp->if_oerrors++;  1
    ifp->if_collisions+=2;
    if (sc->ctrblk.est_sendfail != 0xffff)
      sc->ctrblk.est_sendfail++;
    sc->ctrblk.est_sendfail_bm |= 1;  2
    WRITE_TXS(sc, status);
    WRITE_CMD(sc, CMD_TXENA);
} else {
```

1. Increments the output errors because the excessive data collisions status means that the transmit failed.

2. Indicates excessive collisions.

13.3.4 Writing to the Status Register to Obtain the Next Value

The following code shows how the el_tint() routine writes to the status register to obtain the next value:

```
        WRITE_TXS(sc, status);  1
    }

    status = READ_TXS(sc);
  }
  sc->txfree = READ_TXF(sc);  2

  if (sc->debug)
    if (count <= 0)
      printf("el%d: Transmit in INFINITE loop %04X\n", ifp->if_unit,
             status);
```

1. Writes to the transmit status register to clear the current status in preparation for reading the status for the next transmit completion (if any).

|2| Updates the `softc` data structure with the amount of space that is available in the transmit FIFO.

13.3.5 Queuing Other Transmits

The following code shows how the `el_tint()` routine clears the output active flag to permit other transmits to be queued to the device:

```
    ifp->if_flags &= ~IFF_OACTIVE;
}
```

13.4 Implementing the el_error Routine

The `if_el` driver's `el_error()` routine implements the interface adapter error routine, as follows:

```
static void el_error(struct el_softc *sc,
                     struct ifnet *ifp)
{
    int i;

    WRITE_CMD(sc, CMD_WINDOW4);
    i = READ_FDP(sc);  |1|
    printf("el%d: Adapter Failure - %04X\n", ifp->if_unit, i);
    el_reset_locked(sc, ifp, ifp->if_unit);  |2|
}
```

|1| Reads the FIFO diagnostic port register.

|2| Resets the adapter to clear the failure condition.

14

Network Device Driver Configuration

Device driver configuration incorporates device drivers into the kernel to make them available to system administration and other utilities. The operating system provides two methods for configuring drivers into the kernel: static and dynamic. We recommend that you implement your driver products as a single binary module so that customers can statically or dynamically configure them into the kernel.

The driver's `configure` interface handles all configuration operations either at startup (for static configuration) or at run time (for dynamic configuration). To support configuration, you must provide a `sysconfigtab` file fragment, which contains device special file and bus-specific information. The information in the `sysconfigtab` file fragment is added to the system's `/etc/sysconfigtab` database when the driver is installed. The startup procedure and the `sysconfig` utility use the information that the `/etc/sysconfigtab` database provides to locate the driver module and to set device attributes.

The information that you provide in the `sysconfigtab` file fragment depends on the bus on which the driver operates. The following `sysconfigtab` file fragment entries are bus-specific:

- `PCI_Option`

  The `PCI_Option` entry specifies the option data that is associated with the PCI bus. See *Writing PCI Bus Device Drivers* for a description of the values that you can specify with this entry.

- `VBA_Option`

  The `VBA_Option` entry specifies the option data that is associated with the VMEbus. See *Writing VMEbus Device Drivers* for a description of the values that you can specify with this entry.

For more information on the `sysconfigtab` file fragment, as well as how to build and either statically or dynamically link your driver, see *Writing Kernel Modules*.

Index

Free Manuals Download Website

http://myh66.com

http://usermanuals.us

http://www.somanuals.com

http://www.4manuals.cc

http://www.manual-lib.com

http://www.404manual.com

http://www.luxmanual.com

http://aubethermostatmanual.com

Golf course search by state

http://golfingnear.com

Email search by domain

http://emailbydomain.com

Auto manuals search

http://auto.somanuals.com

TV manuals search

http://tv.somanuals.com