

*OpenStep Journal*, Spring 1995 (Volume 1, Issue 1).  
Copyright ©1995 by NeXT Computer, Inc. All Rights Reserved.

# Writing Device Drivers in an Object-Oriented World

*Written by*     **Gary Staas**

*The Device Driver Kit™ in NEXTSTEP 3.3 provides a set of Objective C classes and functions for creating device driver objects. It contains much of the framework for creating various types of drivers in NEXTSTEP, and it simplifies driver development and debugging.*

## HARD TO HANDLE

Writing a device driver is difficult. Yet device drivers are essential components of an I/O subsystem: They make up the software that directly controls the peripherals, including such vital elements as disk drives. Of all operating system software, I/O software is some of the most difficult to understand and write because it must coordinate such inherently asynchronous events as interrupts and device requests. It must handle these requests very carefully to avoid race conditions in which results depend on precisely when requests and interrupts occur, instead of being well-defined. Timing requirements for various hardware devices demand extreme care. And every device driver is unique since each hardware device is idiosyncratic.

Fortunately for NEXTSTEP developers, using the Driver Kit to write a driver can ameliorate

or even eliminate these difficulties. The Driver Kit—part of NEXTSTEP Developer Release 3.3—provides a framework for developing device drivers under NEXTSTEP. The intent of the Driver Kit designers was to make writing a driver with the Driver Kit very similar to developing an application with the Application Kit.

In the Driver Kit paradigm, drivers are *objects*. The Driver Kit provides classes for particular kinds of drivers, such as display device drivers. These classes generalize common device driver elements and provide as much of a device driver's software as possible without specifying a particular hardware device, furnishing these general capabilities in a form suitable for NEXTSTEP and its underlying Mach operating system. In other words, the driver writer fills in the hardware-dependent “blanks” to complete much of the driver. The Driver Kit developers have already done much of the work of writing a NEXTSTEP device driver for you.

Furthermore, the Driver Kit provides the framework for a driver to fit into NEXTSTEP. You typically don't have to explicitly interface your driver with the system—the operating system does this for you automatically. The end result is that writing a driver with the Driver Kit is easier than writing the same type of driver under UNIX.

## DRIVERS AS OBJECTS

Like other NEXTSTEP kits and frameworks, the Driver Kit is written in the Objective C language, which supports object-oriented programming. This programming approach allows code that's common to all drivers—or to a set of drivers such as SCSI bus drivers—to be written once and inherited by subclasses.

The Driver Kit includes a set of device classes for various kinds of drivers. You implement a driver by creating a subclass of the appropriate device class. A Driver Kit driver is an object—an instance of this subclass you've defined.

Each Driver Kit device class has a set of methods. These methods (even those that do nothing) provide a framework for you to build on. Classes and their methods all ignore hardware-dependent aspects of a driver. Of course, most drivers must control real hardware, so you must implement or override the Driver Kit methods to perform their intended functions with your hardware. You essentially <sup>a</sup>fill in the blanks<sup>o</sup> in the methods to develop much of your driver.

You subclass the appropriate Driver Kit device class based on the device type, such as display, network, sound, and so on. For example, the predefined Driver Kit device class IOEthernet supplies the capabilities of a general driver for an Ethernet card. IOEthernet has a set of methods that are useful to Ethernet drivers. You write an Ethernet card driver by creating a subclass of IOEthernet. You then override certain methods in the IOEthernet superclass by writing code that performs that method's functions, using the software interface to your particular Ethernet card. In other words, you take the generic methods your subclass inherits from IOEthernet and implement them specifically for your hardware.

## **DRIVER KIT PARADIGM SIMPLIFIES WRITING DRIVERS**

To appreciate why writing a device driver is difficult, consider how standard UNIX drivers are constructed.

### **UNIX Drivers**

A UNIX driver typically has a <sup>a</sup>top half<sup>o</sup> that's accessed through a system call interface. This portion of the driver runs in the kernel on behalf of a user process. A user process makes an I/O request by communicating with the top half, which initiates data transfers and manages the driver state for the duration of the operation. The <sup>a</sup>bottom half<sup>o</sup> of the driver handles interrupts caused by data transfer completion or other asynchronous events and must run at the interrupt level. Interrupts are handled by the driver's interrupt handler, which may call top-

half routines at interrupt priorities.

ΥΝΙΕΔρυπερΗαλπες4.επσ ←

**Figure 1:** *UNIX driver structure*

In this model, the driver has no control of key events—I/O requests and interrupts. Since I/O requests can occur at any time, multiple requests may attempt to access the same hardware or data structures at the same time. If the driver writer isn't extremely careful, the driver may be prone to race conditions—in which results depend on the order in which requests occur, instead of being independent of each other.

Suppose, for example, the driver changes the hardware state to perform an I/O request when the hardware is already executing another I/O request. In the case of a disk driver, the driver might write a hardware register to initiate a disk controller seek operation when the disk is already transferring data to memory from a previous disk read. The results are hardware-dependent: The current transfer may be aborted, the seek request may be ignored, or both operations may fail. In any case, the desired disk operations won't occur. All sorts of scenarios like this are possible. Similarly, the driver might change a table for one I/O request in the midst of another I/O request's alterations, throwing the table into disarray.

Handling an interrupt requires changing the state of hardware and data structures, and this can interfere with I/O requests in a fashion similar to the above scenarios. Interrupts for one I/O request are obviously not synchronized with the events of other I/O requests. UNIX drivers

can't control when interrupts occur; they can only control when interrupts *don't* occur by disabling them.

A driver must carefully coordinate changes to the hardware and data structure state. To avoid I/O requests stepping on each other, the driver must employ techniques such as disabling interrupts, changing processor priority, and using locks or semaphores. For instance, the

driver could disable interrupts whenever the top half modifies a data structure that an interrupt handler might also change—to prevent the interrupt handler from modifying the structure in the middle of the top half's modifications. Otherwise, the table may be placed in an inconsistent state—the result of the two different driver halves' changes muddling or nullifying each other.

Although these techniques protect critical resources, they present other problems. For instance, disabling interrupts has two disadvantages. For one, if a driver disables interrupts for too long, performance may be reduced or the system may crash. For another, if a driver disables interrupts and fails to reenables them, the system hangs. There are many ways in which the top half and the bottom half of the driver can interfere with themselves and each other. Failure to take all of these scenarios into consideration can result in very obscure bugs. However, handling all these cases can produce very complex code—difficult to write, debug, understand, and maintain.

Let's see how Driver Kit drivers deal with or avoid these difficulties.

## Driver Kit Drivers

Driver Kit drivers avoid the resource contention problems of conventional UNIX drivers in several ways. First, each driver uses only one thread—the *I/O thread*—to access its hardware device. All I/O threads reside in a separate kernel task, called the I/O kernel task. By default, there's only one I/O thread for each hardware device. Only one I/O thread deals with any hardware resource at a time. There's no need to use locks or disable interrupts to protect access to hardware and data structures. Limiting resource access to only one thread simplifies driver design.

Second, a driver gets I/O requests from the user thread, the driver thread that's running in the kernel on behalf of the user. Interface methods in the driver are invoked from the user thread. These methods communicate requests to the I/O thread by sending it Mach

messages or by other techniques, and they enqueue commands for the I/O thread to execute. In this way, the I/O thread can handle one request at a time, instead of being subjected to a barrage of requests to access several resources at once. Interface methods don't perform I/O requests directly, because only the I/O thread touches hardware and other critical resources.

Third, the operating system kernel takes all interrupts and notifies the I/O thread via Mach messages. The I/O thread intercepts a Mach interrupt message and notifies the driver with an **interruptOccurred** or **interruptOccurredAt:** Objective C message. The driver can delay responding to interrupts until it's ready to deal with them. The driver may have no interrupt handler at all (although you can register your own interrupt handler if that's required). Drivers run at the user or I/O thread level—not at interrupt level. A driver doesn't need to run with interrupts disabled since it controls when it processes interrupts.

## DRIVER KIT SCOPE

You can write drivers with the Driver Kit's predefined classes for many kinds of devices, including displays, network cards for Ethernet and Token Ring networks, SCSI controllers and peripherals, and sound cards.

The Driver Kit supports drivers for various buses, independently of device type:

- ISA (Industry Standard Architecture)
- EISA (Extended Industry Standard Architecture, a superset of ISA)
- PCI (Peripheral Component Interconnect)
- PCMCIA (Personal Computer Memory Card International Association)
- VL-Bus (VESA Local Bus, where VESA stands for Video Electronics Standards)

Association)

A driver may have multiple personalities: The same driver may support several kinds of buses, depending on configuration parameters.

*See NEXTSTEP In Focus, Summer/Fall 1994, for more information about various PC bus architectures.*

## DRIVER KIT COMPONENTS

The Driver Kit consists of a set of Objective C classes and protocols, C functions, and utilities. Objective C classes and protocols provide the framework for writing drivers. Device type classes provide the basic capabilities to write drivers for particular kinds of devices. For instance, the IOEthernet class provides methods to write Ethernet card drivers. Other Objective C classes help user-level programs configure and communicate with drivers. IODeviceDescription objects, for example, encapsulate information about devices and are used for communicating this information throughout the operating system. C functions provide kernel services, such as memory and time management. These functions provide the operating system services your driver needs. Utility programs and functions allow you to load a driver into an already running system and help you test and debug your driver.

### Driver Kit Classes

Figure 2 shows the various Driver Kit classes. Note that there are three main branches in this hierarchy.

DriverKitClasses2.eps ↪

**Figure 2:** *Driver Kit Classes*

IODeviceDescription and IOConfigTable objects provide information about a device for the

operating system kernel and drivers. An IOConfigTable object gets configuration information from configuration tables, such as **Default.table** or **Instance0.table**. These tables specify the driver's configuration—what bus it uses, for instance. For every device in the system, there's an IODeviceDescription object, which encapsulates device configuration information (using IOConfigTable objects) and other information. The kernel initializes a driver using its associated IODeviceDescription object.

The device classes—the ones you subclass to create a driver—are all subclasses of IODevice, the generic device driver class. IODirectDevice classes are used to create *direct* device drivers—drivers that actually manipulate hardware, such as display cards or SCSI bus controllers. *Indirect* device drivers communicate with their associated device via a direct driver. For example, the SCSI Tape and SCSI Disk drivers are indirect drivers.

For example, a SCSI disk driver communicates with the disk drive through a SCSI controller driver, which controls the SCSI bus, as illustrated in Figure 3.

ΔριωερΧοννεχτιονσ.επσ ←

**Figure 3:** *Communication and correspondence of drivers and devices, directly and indirectly*

Figure 3 also shows the one-to-one correspondence between driver objects and hardware devices. There's one IO SCSIController driver object for the SCSI controller, and one IO SCSI Disk driver object for each disk attached to the controller. Note the lines of communication: The IO SCSI Disk drivers talk to the disks through the IO SCSIController driver.

Finally, some ancillary classes provide services for drivers. The IO Network class, for instance, provides capabilities, such as tallying packet statistics, to all network drivers, whether they're Ethernet or Token Ring drivers.



## Class Components

A large part of the effort of writing a Driver Kit driver goes into augmenting the instance variables and methods your driver subclass inherits from its superclass. You add instance variables to your subclass to fit the needs of the device. You might create such variables as pointers to memory-mapped hardware registers; device state from volatile or write-only registers; driver mode or state; I/O management variables, such as queue heads or data buffer pointers; or any per-device private data that normally goes in a UNIX driver's `softc` structure.

Your subclass inherits methods from its superclass to perform such actions as instantiating and initializing the driver object, getting and setting values of instance variables, sending commands to hardware, and receiving notifications such as interrupts, I/O completions, and timeouts (through the **interruptOccurred** method, for instance). You can override these methods to customize them for your hardware, and you can add new methods, too.

For example, `IODevice` contains the **probe:** method, which all drivers must override. The **probe:** method gets passed an `IODeviceDescription` object as its parameter. A **probe:** implementation queries the hardware to determine whether it's present and functioning. If the device verifications pass, **probe:** creates a driver instance, invokes `IODirectDevice's initFromDeviceDescription:` method to initialize the driver instance, and returns YES. Otherwise, **probe:** doesn't create an instance and returns NO.

Here's a skeleton of a **probe:** implementation for a direct device driver of the class `MyClass`. Italicized text in angle brackets (`<< >>`) would be filled in with device-specific code.

```
+ (BOOL)probe:devDesc
{
    MyClass *instance = [self alloc];
    IOEISADeviceDescription
        *deviceDescription = (IOEISADeviceDescription *)devDesc;
```

```

if (instance == nil)
    return NO;

/* Check the device description to see that we have some
 * I/O ports, mapped memory, and interrupts assigned. */
if ([deviceDescription numPortRanges] < 1
    || [deviceDescription numMemoryRanges] < 1
    || [deviceDescription numInterrupts] < 1) {
    [instance free];
    return NO;
}

<< Perform more device-specific validation, for example, checking to make
sure the I/O port range is large enough. Make sure the hardware
is really there. Free the instance and return NO if anything is wrong. >>

return [instance initWithDeviceDescription:devDesc] != nil;
}

```

## Driver Interface

You typically don't need to do anything to interface your driver with the operating system: The kernel automatically finds the driver and uses its methods to communicate with the driver. Most display, network, SCSI controller, and sound drivers are integrated into the system this way.

## A DISPLAY DRIVER EXAMPLE

To give a taste of device driver development with the Driver Kit, this section illustrates some of the basics of writing a display driver. The directory **/NextDeveloper/Examples/DriverKit** contains examples of display, network, SCSI, and audio devices. Video driver examples in this directory are in the **ATI**, **CirrusLogicGD542X**, **QVision**, **S3**, and **TsengLabsET4000**

directories.

## Display Class Capabilities

The Driver Kit has two classes for writing display drivers: *IOFramebufferDisplay* for cards that can linearly map the entire display frame buffer, and *IOSVGADisplay* for all other display cards. Implementation-specific details, of course, depend on the video display hardware. A driver writer needs to be intimately familiar with the hardware specification for any hardware device they're writing a driver for.

The *IOFramebufferDisplay* class supports several display modes. These include 2- and 8-bit grayscale and 8-bit color. Sixteen-bit color is also supported, which uses 4 or 5 bits each for red, green, and blue, but offers only 4096 colors in either case. In addition, 24-bit color is supported, with 8 bits each for red, green, and blue. The *IOSVGADisplay* class supports only 2-bit grayscale display mode. Both classes support EISA, VL-Bus, PCI, and a limited number of ISA display cards.

## Defining the Subclass

You pick the display subclass depending on the capability of the display card. Then you add instance variables for data unique to your device.

The ATI example in **/NextDeveloper/Examples/DriverKit/ATI** defines an *IOFramebufferDisplay* subclass in this way:

```
@interface ATI:IOFramebufferDisplay
{
/* The flavor of ATI chipset that we have. */
    ATIFlavor ati_flavor;

    /* Setup parameters. */
```

```

const ATI_CRTCSetup *CRTControllerSetup;

/* The information for the selected display mode. */
const IODisplayInfo *displayMode;

unsigned int _ATI_reserved[8];
}

± (void)enterLinearMode;
± (void)revertToVGAMode;
± initWithDeviceDescription: deviceDescription;
± setTransferTable:(const unsigned int *)table count:(int)count;
@end

```

## Display Driver Basic Operations

The NEXTSTEP Window Server handles all graphics, which simplifies writing a Driver Kit display driver. A display driver performs the following basic operations by overriding these methods:

- Instantiating and initializing a driver object with **probe:** and **initWithDeviceDescription:**
- Selecting the display mode with **selectMode:count:valid:**
- Reconfiguring display hardware for the selected display mode with **enterLinearMode**
- Reverting to VGA display mode with **revertToVGAMode**
- Adjusting display brightness with **setBrightness:** if the display card supports this feature

### Instantiating and initializing a driver object

Override the **probe:** method in IODevice. Your implementation should check that the display hardware it expects is present and characterize it. In particular, **probe:** should check for the presence of the graphics controller (CRTC) and determine its version. It should also

determine

the DAC type, the memory size, and the clock chip type, if necessary. PCI-based drivers' **probe:** method should also check and set the frame buffer range address. If the hardware checks pass, **probe:** should create an instance of IOFrameBufferDisplay or IOSVGADisplay, initialize it with **initFromDeviceDescription:**, and return YES. Otherwise, **probe:** shouldn't create an instance but instead send an appropriate diagnostic message and return NO.

### Selecting a display mode

IOFrameBufferDisplay's **selectMode:count:valid:** method selects the display mode. This method requires that you declare an IODisplayInfo array with one element per mode and initialize it with display mode information, as in this example for a Compaq QVision Video Adapter driver:

```
const IODisplayInfo QVisionModeTable[] = {
    /* 0: QVision 1024 x 768 x 8 (Mode 0x38) @ 60Hz. */{
        1024, 768, 1024, 1024, 60, 0,
        IO_8BitsPerPixel, IO_OneIsWhiteColorSpace, "WWWWWWWWW",
        0, (void *)&Mode_38_60Hz,
    },

    /* 1: QVision 1024 x 768 x 8 (Mode 0x38) @ 66Hz. */
    {
        1024, 768, 1024, 1024, 66, 0,
        IO_8BitsPerPixel, IO_OneIsWhiteColorSpace, "WWWWWWWWW",
        0, (void *)&Mode_38_66Hz,
    },
}
```

*The example display drivers that come with NEXTSTEP Developer 3.3 don't invoke **probe:** because they were written before this method was available. Instead they use **initFromDeviceDescription:** to perform all initialization tasks.*

To indicate the valid display modes, declare an array of Boolean values with one element per display mode and fill it appropriately. In the following example, italicized text delineated in angle brackets should be filled in with driver-specific code:

```
BOOL validModes[QVisionModeTableCount];

for (k = 0; k < QVisionModeTableCount; k++) {
    if (<< current hardware supports this mode >>)
        validModes[k] = YES;
    else
        validModes[k] = NO;
}
```

During initialization, you would use this method to select a mode and handle the result, as this code fragment illustrates:

```
mode = [self selectMode:QVisionModeTable count:QVisionModeTableCount
    valid:validModes];

if (mode < 0) {
    IOLog("%s: Sorry, cannot use requested display mode.\n",
        [self name]);

    /* Pick a reasonable default */
    mode = DEFAULT_QVISION_MODE;
}
```

**IOLog()** is a logging function the Driver Kit supplies for debugging.

### **Reconfiguring display hardware for the selected display mode**

Using the appropriate commands for your display hardware, reconfigure it for the selected mode with the following operations (whose order is hardware-dependent, determined by the display specification):

- Turn off the CRTC.
- Configure the CRTC.
- Configure the DAC.
- Configure the clock chip.
- Configure memory, if necessary.
- Restart the CRTC.
- Enable linear frame buffer mode, if applicable.

### Reverting to VGA display mode

Override the **revertToVGAMode** method to return the adapter to the state it's in after a hard reset. You would typically set VGA mode to 3. Here's an example of implementing this method for an ATI Graphics Ultra Pro Video Adapter display card, from the ATIDriver Kit example in **/NextDeveloper/Examples/DriverKit/ATI/ATI\_reloc.tproj/ATI.m**:

```
± (void)revertToVGAMode
{
    /* Select VGA setup, re-enabling the VGA CRT controller. */
    SelectShadowSet(0); /* Select VGA CRT configuration. */
    reset_DAC(); /* Restore DAC for VGA operation. */
    [super revertToVGAMode]; /* Let superclass do generic VGA stuff. */
}
```

Note that this method invokes **revertToVGAMode** to perform any general functions the superclass's method provides. This driver defines the function **SelectShadowSet()** this way:

```
static void
SelectShadowSet(int set)
{
```

```

unsigned char v;

switch (set) {
    case 0:
        v = 2;
        break;
    case 1:
        v = 3;
        break;
    case 2:
        v = 7;
        break;
    default:
        return;
}
outb(ADVFUNC_CNTL, v);
}

```

**SelectShadowSet()** selects the appropriate hardware value based on the function input. It then writes the value to the memory-mapped address ADVFUNC\_CNTL using the Driver Kit function **outb()**, which writes a byte to an I/O port address. The address ADVFUNC\_CNTL and the value written to it are the values appropriate for an ATI Graphics card.

## Adjusting display brightness

If the video hardware allows changing the display's brightness, implement **setBrightness:token:** and use the **setTransferTable:count:** method to adjust it as desired.

If the DAC supports downloading a color palette, override **setTransferTable:count:** to receive a gamma-corrected transfer table from the Window Server, or declare your own table in a static array. Override **setBrightness:token:** and download the transfer table to the DAC. For an example, look at an implementation of **setGammaTable** in [/NextDeveloper/Examples/DriverKit](#), such as in



**QVision/QVision\_reloc.tproj/QVisionDAC.m.**

Finally, indicate that you've implemented a transfer table by setting a flag in a **struct** you've defined:

```
displayInfo->flags |= IO_DISPLAY_HAS_TRANSFER_TABLE;
```

If the DAC doesn't support downloading a color palette, don't override these methods, but set the flag to indicate there's no transfer table:

```
displayInfo->flags |= IO_DISPLAY_NEEDS_SOFTWARE_GAMMA_CORRECTION;
```

## **BUILDING, CONFIGURING, AND DEBUGGING DRIVERS**

Every Driver Kit driver resides in a configuration bundle that contains all the files needed to load and configure the driver: its relocatable code, configuration information, and other information, such as help files. You can create driver bundles using Project Builder.

The Configure application allows you to add your device and configure its driver. Configure takes information from the configuration tables and displays it in a configuration inspector panel where a user can modify it. If you have standard configuration parameters, you can use the default inspector provided with the Driver Kit. You can load a driver into an already running system using the **driverLoader** command. **driverLoader** provides a variety of options for loading and configuring a driver.

You can also use **gdb** as a source-level debugger on device drivers.

Once you've loaded the driver, the Driver Kit provides two different tools to debug it. The first, the **IOLog()** function, allows you to output strings and parameters similarly to **printf()** and deposits error or debugging messages in a file. You can place a call to **IOLog()** anywhere in

your driver either to get information about the driver state at that point or to indicate that the driver reached that point during execution.

When timing is important, you can use the second tool, the Driver Debugging Module (DDM), which allows you to view debugging information without altering the timing of the kernel. A driver sends messages to DDM, which time stamps them and places them in a circular buffer. With the DDMViewer application (in **/NextDeveloper/Demos**), you can specify which information DDM stores in the event buffer and also display that buffer's information. DDM provides a set of macros you can use in your driver to add debugging entries to the buffer, similar to **IOLog()** calls.

## SUMMARY

To write a device driver, driver writers add hardware-specific details to the Driver Kit framework. The NEXTSTEP Driver Kit treats device drivers as objects. This paradigm lends itself to reusing driver software for specific devices such as network cards. The Driver Kit provides a framework for writing display, network, SCSI, and audio drivers. The main task of the driver writer is to fill in details that depend on a specific hardware device. The developer does this by subclassing the appropriate Driver Kit device class, adding instance variables as needed, and overriding and implementing existing or new methods in the subclass.

Driver Kit drivers avoid complications typical drivers are prone to, such as resource contention, by providing an I/O thread—the only thread to touch the hardware or data structures. Driver requests are handled linearly, which simplifies design and implementation. Similarly, the kernel handles interrupts and sends interrupt messages to the driver, which can process interrupts at a convenient time—and avoid conflicts with other parts of the driver altering the hardware or data state at the same time.

The Driver Kit also provides tools to facilitate building, loading, and debugging drivers. After you've built your driver, you can load it into an already running system and debug it with a

variety of tracing tools.

*Gary Staas is a writer in NeXT's Developer Publications group.*

## References

Closkey, Cynthia, ed. *NEXTSTEP In Focus*, Summer/Fall 1994. Redwood City, CA: NeXT Computer, Inc., 1994. *This issue contains several articles about various PC bus architectures, including PCI and PCMCIA.*

Egan, Janet I., and Thomas J. Teixeira. *Writing a UNIX Device Driver*, second edition. New York: John Wiley and Sons, 1992. *An excellent general introduction to UNIX drivers.*

Ferraro, Richard F. *Programmer's Guide to the EGA and VGA Cards*, second edition. Palo Alto, CA: Addison-Wesley, 1990.

*IBM Token-Ring Network Architecture Technical Reference. (SC30-3374-02.) This definitive and readable manual describes a superset of the 802.5 specification. You can get it from IBM or from IBM dealers.*

*Information Technology-Local and Metropolitan Area Networks. Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications. International Standard ISO/IEC 8802-3; ANSI/IEEE Std. 802.3. This is the specification for 802.3.*

*Information Technology-Local and Metropolitan Area Networks. Part 5: Token Ring Access Method and Physical Layer Specifications. International Standard ISO/IEC 8802-5; ANSI/IEEE Std. 802.5. This is the specification for 802.5.*

Kettle, Peter, and Steve Statler. *Writing Device Drivers for SCO UNIX, A Practical Approach*. Palo Alto, CA: Addison-Wesley, 1993. *This book includes some details of Intel hardware. It also contains a good reference section.*

NeXT Computer, Inc. *Writing Device Drivers with the Driver Kit*. Redwood City, CA: NeXT Computer, Inc., 1994. *Available on-line with NEXTSTEP Developer Release 3.3 in:*  
***/NextLibrary/Documentation/NextDev/OperatingSystem/Part3\_DriverKit***

Shanley, Tom. *EISA System Architecture*, second edition. Richardson, TX: Mindshare Press, 1993.

Shanley, Tom. *PCI System Architecture*, second edition. Richardson, TX: Mindshare Press, 1993. *This book tells how to work with a version 2.0-compliant bus.*

Shanley, Tom. *PCMCIA System Architecture*. Richardson, TX: Mindshare Press, 1994.

Shanley, Tom, and Don Anderson. *ISA System Architecture*, second edition. Richardson, TX: Mindshare Press, 1993.

Tanenbaum, Andrew S. *Computer Networks*. Englewood Cliffs, NJ: Prentice Hall, 1981. *This book contains information on networking in general.*