Preface

Introduction

Some of the most formidable operating system software to understand—and to write—is the I/O subsystem. Device drivers are essential components of the I/O subsystem. They control the peripherals required for a multipurpose, flexible computer system.

In some systems, device drivers are closely entwined with the operating system, requiring that you have an extensive knowledge of the implementation of the operating system to write a driver.

Writing a driver for the NEXTSTEP™system doesn't demand such difficult prerequisites. You can write a NEXTSTEP device driver in a very modular fashion, without knowing a great deal about NEXTSTEP. NeXT™has packaged together the software and tools you need to write in a driver into the *Driver Kit*™, a part of the NEXTSTEP Developer software. Writing a device driver using the Driver Kit is more like writing an application using the NEXTSTEP Application Kit than like writing a driver for other operating systems.

The Driver Kit provides a framework to help you create device drivers for computers running NEXTSTEP. Although every driver is unique, drivers do have common elements. The Driver Kit generalizes the software required for a driver, removing the hardware-specific details. To create a driver, you essentially fill in the hardware-dependent "blanks" in the Driver Kit software with code that performs the desired operations on your hardware.

By using the structure that the Driver Kit offers, you can greatly reduce the time and effort required to write a driver. The conceptual model of a Driver Kit driver is simpler than that of a driver on other systems. This design simplifies writing a driver and eliminates many of the problems that make debugging drivers difficult.

This document is part 3 of *NEXTSTEP Operating System Software*. Chapter 1, "Driver Kit Architecture," introduces you to the structure of the Driver Kit. You learn about designing a Driver Kit driver in Chapter 2, "Designing a Driver." Chapter 3,

“Support for Specific Devices,” acquaints you with some of the details needed to write specific types of drivers such as network drivers. The fourth, and last, concepts chapter, “Building, Configuring, and Debugging Drivers,” describes these topics. Chapter 5, “Driver Kit Reference,” discusses the classes and other associated tools provided by the Driver Kit.

The Driver Kit is supported on all NEXTSTEP platforms except 680x0-based computers.

Before You Read This Document

This document covers only the parts of driver writing that are specific to the Driver Kit.

To understand this document, however, you need to be familiar with several topics that aren’t covered here. Some of these topics are discussed in other NeXT™documentation.

NeXT Documentation to Read

You need to know the Objective C language, since the Driver Kit is written in this language. Objective C provides a set of simple, object-oriented extensions to ANSI C.

NEXTSTEP systems use the Mach operating system. Writing most drivers requires that you understand such Mach concepts as tasks and threads, and writing many requires familiarity with Mach ports and Mach messages. The Mach Kit contains useful tools such as facilities for locks. Driver Kit drivers are a part of the Mach kernel and are known as *loadable kernel servers*, so you must be familiar with this concept as well. Access to most of the Mach facilities you need is included with the Driver Kit in its set of Mach functions.

The following table shows where you can learn about these topics:

Topic	Where to Read about It
Objective C language	Chapters 1, 2, and 3, *NEXTSTEP Object-Oriented Programming and the Objective C Language*
Mach operating system	Chapter 1, *NEXTSTEP Operating System Software* (read the introduction, “Design Philosophy,” and “The Mach Kernel”)
Mach Kit	Chapter 9, *NEXTSTEP General Reference*

You can get updates to NeXT documentation on archive servers through the NeXTanswers™program. Send e-mail to **nextanswers@next.com** with the two-word subject: **INDEX HELP**. Or if you can't receive NeXT mail, add a third word, **ASCII**. You'll receive the current index of documents and instructions for requesting more information.

Other Reading

It's helpful if you know how to write a device driver on some system other than NEXTSTEP. If you haven't written a driver before, see "Suggested Reading" in the Appendix for a list of books that can help you learn about drivers. If you've never written a driver for a multitasking operating system, you should familiarize yourself with the issues involved. The "Suggested Reading" section also lists books that deal with these issues.

Finally, you should be very familiar with the hardware your driver will control. Besides your device's documentation, you'll also need specifications for the bus your device attaches to. Some sources of bus documentation are listed in "Suggested Reading" in the Appendix.

1

Driver Kit Architecture

The Driver Kit is a tool kit for writing object-oriented device drivers. Part of the NEXTSTEP Developer software (except for 680x0-based computers), it simplifies writing device drivers for NEXTSTEP systems. The Driver Kit provides as much of the software in a device driver as possible without specific information about the device. The Driver Kit developers have already done much of the work of writing a NEXTSTEP device driver for you.

The preface briefly described the Driver Kit and mentioned a few of its advantages. This chapter provides greater detail about what a Driver Kit driver is and how it's structured. It discusses the components of the Driver Kit and what they do. It contrasts developing a Driver Kit driver to developing a typical UNIX™driver—this contrast shows some of the advantages of the Driver Kit approach. It talks about the various Driver Kit classes and how you create a driver with them. The chapter finishes with a discussion of how drivers are integrated into the system at startup time, how interrupts are handled, and how users interface with drivers.

Driver Kit Components

The Driver Kit consists of the following tools:

- Objective C classes and protocols that provide the framework for writing drivers for various types of devices. The first three chapters discuss how to use these classes. The section "Classes" in Chapter 5, "Driver Kit Reference," specifies each class in detail.
- Objective C classes that help user-level programs to configure and communicate with drivers. Configuration is discussed in Chapter 4, "Building, Configuring, and Debugging Drivers." The "The User-Level Interface to Drivers" section in this chapter and "Interfacing with the Driver" in Chapter 2, "Designing a Driver," tell how to communicate with drivers.

- C functions that provide debugging capabilities, kernel services such as memory and time management, and other services. These functions provide most of the operating system services your driver should need. The “Functions” section of Chapter 5, “Driver Kit Reference,” contains specifications for these functions.
- Utility programs that help you load a driver into an already running system and help you test and debug your driver. Chapter 4, “Building, Configuring, and Debugging Drivers,” tells you about these programs.

The rest of this chapter describes the basics of Driver Kit architecture.

Why Objective C?

Why is Objective C the required language for the Driver Kit? Part of the reason is that all other NEXTSTEP Application Program Interfaces (APIs) are object-oriented and use Objective C. But more importantly, drivers benefit in several ways from object-orientation and Objective C:

- **Naturalness**—Object orientation is a natural design method for drivers. Each hardware object can be modeled by a software object, and functionality common to a group of drivers (such as display drivers) can be provided by superclasses.
- **Flexibility**—Objective C provides dynamic typing and binding, which help different objects communicate without having to be compiled together. For example, this lets a SCSI peripheral driver determine at run time which SCSI controller driver it should communicate with. You can simulate dynamism using function lookup tables and type casting in ANSI C, but this results in code that’s harder to understand and maintain.
- **Code reduction**—The Driver Kit provides classes that significantly lessen the amount of code you have to write. For example, the IODirectDevice class greatly simplifies configuration and initialization, and IOFrameBufferDisplay takes care of almost everything that a display driver must do.

See *NEXTSTEP Object-Oriented Programming and the Objective C Language* for more information on Objective C.

Device and Bus Support

The Driver Kit has classes to help you write drivers for several kinds of devices:

- Displays

- Network cards for Ethernet and Token Ring networks
- SCSI controllers and peripherals such as tape drives
- Sound cards

The IOEthernet class, for example, provides much of the functionality required for Ethernet drivers. To write a driver for a new type of Ethernet card, you need to implement only six methods, filling in the details of how your hardware performs the various functions required in an Ethernet driver.

Chapter 3, “Support for Specific Devices,” tells you how to implement a driver for device types the Driver Kit explicitly supports.

You can write drivers for other kinds of devices than those listed above. The devices above are merely those that the Driver Kit specifically supports.

In addition, the Driver Kit has general-purpose classes that support these computer buses:

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

Both ISA and VL-Bus are supported through the EISA bus class.

You indicate the bus type that your driver works with in the configuration file for the driver. See Chapter 4, “Building, Configuring, and Debugging Drivers,” for more information.

Driver Structure

To appreciate the structural simplicity of a Driver Kit driver, first consider how standard UNIX drivers are constructed.

UNIX Driver Architecture

A UNIX driver has a “top-half” that is accessed through the system call interface and runs in the kernel on behalf of a user process. It manages the driver state and initiates data transfers. The “bottom- half” runs at interrupt level since it’s driven by interrupts caused by data transfer completion or other asynchronous events. Interrupts are handled by the driver’s interrupt handler, which may call top-half routines at interrupt

priorities. Indirect devices—devices that are not directly connected to the processor, such as secondary-bus devices or SCSI peripherals—are each handled in an individual fashion—there's no systematic way to treat them.

This design paradigm has several consequences:

- Multiple requests may attempt to access the same hardware or driver data structures at the same time.
- Interrupts may occur at any time, and their handlers may also need to access hardware or data structures.

To coordinate access to these hardware and data resources, the driver must use such tactics as disabling interrupts, changing processor priority, and engaging locks of various types. The resulting code is often complicated: difficult to write, debug, understand, and maintain.

Driver Kit Driver Architecture

You can write a UNIX style driver with the Driver Kit, but that's not the best way to go about it. Driver Kit drivers differ significantly from traditional UNIX or MS-DOS™drivers. Driver Kit drivers have these characteristics:

- Drivers are *objects*. The Driver Kit is written in the Objective C language, which supports object-oriented programming. This programming approach also allows code that's common to all drivers—or a set of drivers such as network drivers—to be written once and inherited by subclasses.

- By default, 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—the I/O kernel task.

- By default, there's one I/O thread for each hardware device. Given any hardware resource, only one thread deals with that resource at a time. Traditional device drivers use locks and disable interrupts to protect access to hardware and data structures. Limiting resource access to only one thread greatly simplifies driver design.

- Interface methods in the driver are invoked from the *user thread*: the thread running in the kernel on behalf of the user. These methods communicate requests to the I/O thread using techniques such as *Mach messaging*, and they enqueue commands for the I/O thread to execute. The I/O thread can then handle one request at a time instead of being subjected to a barrage of requests to access multiple resources at the same time. (Interface methods don't perform I/O requests directly, because only the I/O thread should touch hardware and other critical resources.)

Note: *Mach messages* are not the same as *Objective C messages* that are sent to objects. Mach messaging refers to use of the Mach operating system's message system. See the references on the Mach operating system and the Objective C language in the "Suggested Reading" section of the Appendix.

- The kernel takes all interrupts and notifies the I/O thread via Mach messages. Drivers don't need to run with interrupts disabled. The Driver Kit's thread-based model lets the driver delay responding to interrupts until it's ready to deal with them. The UNIX concept of a direct *interrupt handler*—a section of driver code that executes as soon as an interrupt is detected by the kernel—has been replaced by this Mach messaging mechanism. Interrupt handling is discussed in greater detail in "Servicing Interrupts" in this chapter. You can register your own interrupt handler if that's required, but unless you do, your driver will run at the user or I/O thread level—not at interrupt level.

- Drivers for devices that are connected to the processor indirectly through some secondary bus—such as SCSI peripherals connected to a SCSI bus—have a structured way to communicate with the drivers controlling the secondary bus. For example, SCSI controller objects conform to an Objective C protocol that SCSI peripheral drivers can employ.

- Driver Kit drivers are currently kernel-level drivers, either as loadable kernel servers or as part of the kernel supplied by NeXT. User-level drivers are not yet supported.

Tip: Running drivers at user level would make testing hardware much easier, and it would greatly reduce the likelihood of system panics due to driver bugs. This design goal hasn't been realized yet. However, when you design your driver, you should keep in mind the possibility of it becoming a user-level driver. To make porting drivers from kernel to user level as easy as possible, much of the Driver Kit API is identical at kernel level and at user level. In future releases, the goal is to allow all drivers to run at user level.

Although it's possible to write a UNIX style driver with the Driver Kit, that's not the best way to proceed. You wouldn't be taking full advantage of the capabilities of the Driver Kit, and you would be doing a lot of extra work.

Driver Classes and Instances

You implement a driver by creating a subclass of one of the device type classes in the Driver Kit. *A driver object is an instance of this subclass you've defined.*

Each Driver Kit class has a set of methods, some of which don't actually do anything. These methods—even the ones that do nothing—provide a framework for you to build on. The classes and their methods all ignore hardware-dependent aspects of a driver

to some extent. Of course, every driver must control real hardware, so you must implement or override the methods provided in the Driver Kit so that they perform their intended functions with your hardware. You essentially “fill in the blanks” in the methods to develop much of your driver.

You choose the Driver Kit class for which you’re going to create a subclass based on the device type, such as display, network, sound, and so on.

For example, you can write an Ethernet card driver by creating a subclass of the IOEthernet class. You then override each method in the IOEthernet superclass by writing code that performs that method’s functions—using the software interface to your particular Ethernet card hardware. In other words, you take the generic methods provided by the IOEthernet class and make them specific to your hardware in the subclass that you implement.

Most Driver Kit classes are never instantiated. Instead, they serve as abstract classes that give capabilities to their subclasses. For example, IODisplay is an abstract class that implements functionality common to all displays.

The hierarchy of Driver Kit classes has three main branches, as shown in Figure 1-1.

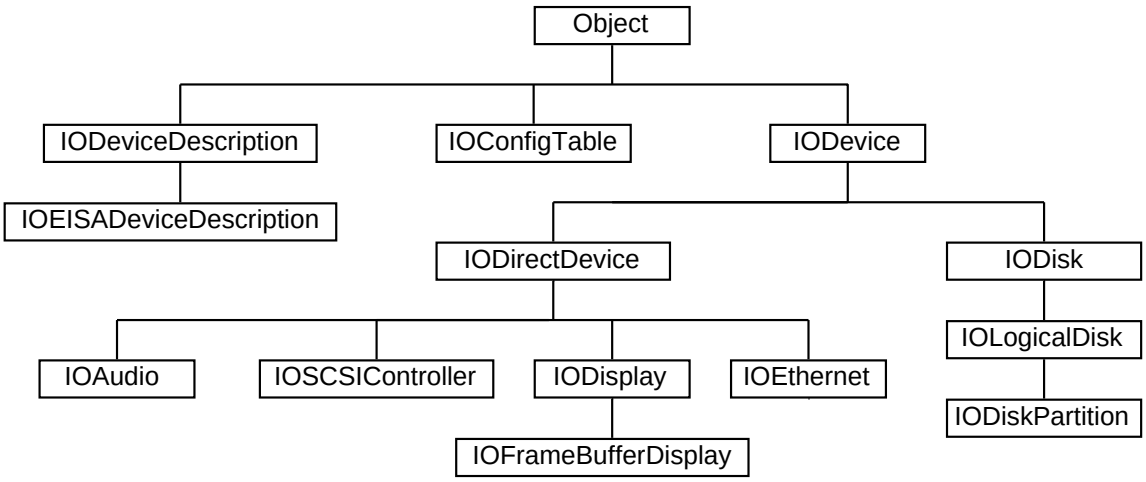

Figure 1-1. Some Core Driver Kit Classes

Note: Classes for developing disk drivers, such as IODisk, aren’t currently documented.

You create a subclass of a class in the IODevice branch to create your driver. All drivers are instances of subclasses of IODevice. These classes provide frameworks for specific types of device drivers.

The other two branches—IODeviceDescription and IOConfigTable—provide information about drivers. IOConfigTable objects get configuration information about

particular devices and the system as a whole from configuration tables, which specify how a driver is to be configured. IODeviceDescription objects encapsulate configuration and other information about the driver and are used for initializing the driver. These classes allow you to configure the driver into the system and allow it to communicate with system hardware.

In summary, the Driver Kit provides a framework for developing a driver for NEXTSTEP systems. It provides many of the pieces you need to create a driver—classes and protocols, methods, functions, and utilities—and puts the pieces together for you. A class hierarchy groups methods logically by function and device type. A thread mechanism, including a default I/O thread, ensures that methods work together, taking advantage of the NEXTSTEP architecture. You still have to implement the methods to fit your hardware, but the basic structure is already there. The paradigm embodied in the Driver Kit fits well with NEXTSTEP, but it's different from the model that standard UNIX drivers use. You can write a driver using a UNIX model, but it would require greater effort.

Direct and Indirect Device Drivers

Some devices, such as displays and network devices, are connected directly to the processor, and their drivers are referred to as *direct device drivers*. Other devices are connected to the processor indirectly through some secondary bus—such as SCSI peripherals connected to a SCSI bus. Drivers for such devices are called *indirect device drivers.* Drivers for direct devices talk to the hardware directly. Indirect device drivers talk to their device hardware indirectly through some direct device. A SCSI disk driver, for instance, communicates with the disk through a SCSI controller driver, which controls the SCSI bus.

Thus drivers talk to hardware either directly or indirectly, or they may not deal with hardware at all. Drivers are thus further classified into these three types:

- Direct device drivers (for example, drivers for SCSI controllers)
- Indirect device drivers (for example, drivers for disks attached to SCSI controllers)
- Pseudo device drivers (drivers that control no hardware)

These classes work differently, are initialized differently, and require different system resources. This manual focuses primarily on direct and indirect drivers, not pseudo device drivers.

Note that the IODevice branch in Figure 1-1 is further split into two branches. On one side is IODirectDevice, from which you would create a subclass for a direct device driver. Indirect device drivers stem from the other branch and are subclasses of IODevice.

Terminology Used in This Document

The term *driver* refers to the implementation of a subclass of one of the Driver Kit device classes—since Driver Kit classes are typically abstract classes. *Instances* of a driver are instances of the subclass. Often an object is referred to as an object of one of its superclasses—for example, as an *IOSCSIController object* or *IODevice object*—to indicate that the object is an instance of any subclass of the superclass. Finally, *device* is sometimes used to refer to any IODevice object.

As Figure 1-1 shows, IOSCSIController, IODisplay, and IOEthernet are subclasses of IODirectDevice. This classification occurs because instances of their subclasses talk directly to the hardware, performing such operations as handling interrupts, mapping memory, and performing DMA operations. IODisk, an indirect device class, is a subclass of IODevice—but not of IODirectDevice. This occurs because IODisk objects don't talk directly to the hardware: They talk indirectly to the hardware by sending request messages to IODirectDevice objects such as IOSCSIControllers.

Figure 1-2 shows how two objects—one an instance of a direct device driver, the other an instance of an indirect device driver—combine to control two pieces of hardware. The indirect driver, an IOSCSIDisk object, uses the direct driver, an IOSCSIController object, to control the hardware.

Note: IOSCSIDisk is a nonpublic subclass of IODisk.

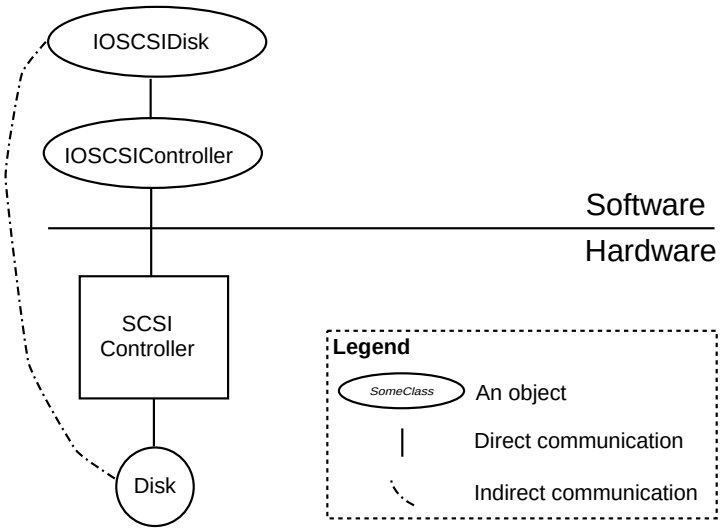

Figure 1-2. How Objects Correspond to Hardware

One Device Driver Object per Hardware Device

There is one device driver object for each hardware device. In Figure 1-3, one IOSCSIController object manages the SCSI controller, and an IOSCSIDisk object manages each disk. Both disks are connected to the same SCSI controller, so both IOSCSIDisk objects communicate with the hardware using the single IOSCSIController object.

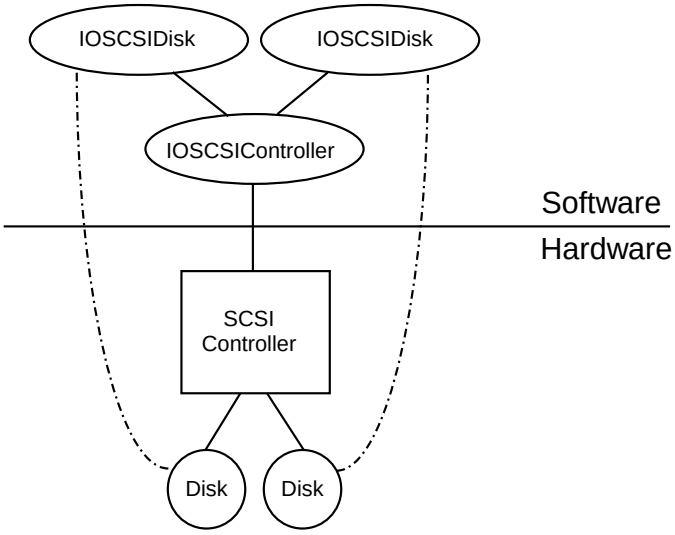

Figure 1-3. One-to-One Correspondence between Driver Objects and Hardware Devices

Key Driver Kit Classes

You typically create a subclass of either IODevice or IODirectDevice (or one of its subclasses) to create a driver.

IODevice: The Generic Device Driver

Every driver is a subclass of IODevice. This class provides a standard programming interface for probing hardware and for creating, initializing, and registering a driver instance.

IODirectDevice: The Class for All Direct Devices

IODirectDevice is the class for drivers that directly control hardware. This class adds data (that is, instance variables) and methods for managing interrupts, DMA channels, address ranges, and other resources. It contains a configuration table, an NXStringTable object of key/value pairs that hold configuration data provided by the system and the user.

The IODirectDevice class has Objective C categories for specific hardware buses:

- IOEISADirectDevice for EISA-, ISA-, and VL-Bus-based systems
- IOPCIDirectDevice for PCI-based systems
- IOPCMCIADirectDevice for PCMCIA-based systems

Display, network, SCSI controller, and sound drivers are all direct drivers that can be implemented as subclasses of IODirectDevice—or its subclasses. IODirectDevice has subclasses for each of these specific device types. For example, you can use the IODisplay class (a subclass of IODirectDevice) to write a display driver.

IODeviceDescription: Device Information

For every IODevice object, there's a device description object—an instance of the IODeviceDescription class—that contains information about the device. Thus every device in a system has a device description that contains information about the device:

- Device address
- System resources (IRQ, DMA channels, and so on) used by the device
- Other information specific to the bus type

Instance variables in IODevice (of which the driver is a subclass) contain the rest of the device information, such as device type. The configuration tables, such as **Default.table** and **Instance*n*.table**, contain the device driver configuration information. These tables can be modified using the Configure application.

Class Components

When you create a subclass, you add instance variables that are appropriate for your hardware, such as variables for memory-mapped registers. A subclass might include the following typical instance variables:

- Pointers to hardware registers
- Device state from volatile or write-only registers
- Driver mode or state
- I/O management variables such as queue heads, locks for critical structures, or

data buffer pointers
- Any per-device private data that normally goes in a UNIX driver's "softc" structure

Your subclass inherits a set of methods from its superclass to perform such actions such as these:

- Initialize the driver object
- Get and set values of instance variables
- Send commands to hardware
- Receive notifications such as interrupts, I/O completions, and timeouts

In your subclass you can override methods from the superclass, and you can also add new ones. You customize these methods to work with your device's hardware.

Suppose, for example, you're implementing a display driver for a display card that can linearly map the entire frame buffer. Create a subclass of the IOFrameBufferDisplay class (a subclass of IODisplay), then override four methods to do the following operations:

- **initFromDeviceDescription:** to invoke **super**'s implementation of **initFromDeviceDescription:**, map the display into the memory, and select the display mode.
- **enterLinearMode** to place the frame buffer device into the linear frame buffer mode selected during device initialization.
- **revertToVGAMode** to set the display to run as a standard VGA device.
- **setBrightness:** to control screen brightness, if the hardware supports this function.

Once you've done this, you've finished much of your driver.

The User-Level Interface to Drivers

You typically don't need to be concerned about interfacing with your driver: 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. For some devices, such as SCSI peripherals, you may need to write an interface program called by user programs or other drivers. This interface program invokes the driver's methods to communicate with the driver.

See "Interfacing with the Driver" in Chapter 2 for more discussion of user-level to driver-level communication.

How IODevice Objects are Created

Drivers are packaged into *driver bundles*. A driver bundle contains its relocatable code and configuration information—everything needed to load and configure the driver. It may also contain help information, programs to be run before and after loading the driver, and a configuration inspector that the Configure application uses to access configuration data. Chapter 4, “Building, Configuring, and Debugging Drivers,” tells you more about bundle contents and how to create a driver bundle.

When the system starts up, it goes through three steps to create each driver object, using the information in the driver bundle:

1. Load the relocatable code for the driver.
2. Create an IODeviceDescription object for the device.
3. Send a **probe:** message to the IODevice class object to instantiate a driver object.

The system goes through two phases of driver creation. In the first phase, it performs these three steps to create all the boot device drivers. Boot drivers are the drivers that must be loaded before the kernel can be active, such as the driver for the boot device. In the second phase, the system creates the active device drivers—drivers for the rest of the devices in the system. The **System.config/Instance0.table** file defines the boot and active devices.

Some driver objects need to know about each other. For instance, an indirect driver controlling a SCSI peripheral needs to communicate with the direct driver that manages the SCSI controller. These drivers get connected with each other during the startup process. See “Connecting the Driver,” in Chapter 2, “Designing a Driver.”

The system is not limited to creating drivers only at system start up time. You can also load a driver after the system has started up with the **driverLoader** command. See “Using the driverLoader Command” in Chapter 4, “Building, Configuring, and Debugging Drivers,” for more information.

Loading Driver Relocatable Code

In the first phase of driver object creation, the kernel loads the driver’s relocatable code (in the file *Driver***_reloc** in the driver bundle, where *Driver* is the driver’s name) if necessary. The driver is already loaded if it’s in the kernel. If there are multiple instances of the driver, the relocatable code is loaded only once.

Creating a Device Description

Next, the kernel creates an IOConfigTable object that provides methods to examine the appropriate configuration file for the driver (either **Default.table** or **Instance*n*.table**). The IOConfigTable object parses the configuration information it gets, which is in configuration key/value pairs in this file. From this information, the kernel instantiates an IODeviceDescription object, which encapsulates information about the driver.

The driver's bus type is indicated in the configuration table as the value associated with the "Bus Type" configuration key (see "Configuration Keys" in the Appendix). The kernel creates the appropriate IODeviceDescription object for the bus:

Bus Type	IODevice Description Subclass
EISA, ISA, VL-Bus	IOEISADeviceDescription
PCI	IOPCIDeviceDescription
PCMCIA	IOPCMCIADeviceDescription

IOPCIDeviceDescription and IOPCMCIADeviceDescription are subclasses of IOEISADeviceDescription, which is a subclass of IODeviceDescription.

After instantiating the IODeviceDescription object, the kernel may do further initialization, using methods in IODeviceDescription to get configuration information. For example, for a PCI-bus device, the kernel might check whether the location of the object on the bus is correct, and if it isn't, the kernel doesn't initialize that device.

If the system supports automatic detection of devices, it automatically scans all system buses to determine which devices are present and to obtain additional configuration information. For more information, see "Auto Detection of Devices" in "Other Features" of Chapter 5, "Reference." Some EISA- and PCI-based systems support this feature.

For more information on configuration tables, see Chapter 4.

Instantiating Drivers

The kernel invokes **probe:**, a class method in the IODevice class, to instantiate a driver. You must override this method in your driver.

The receiver of a **probe:** message determines whether to create a new instance of itself, with the help of information passed as the **probe:** message's argument—the IODeviceDescription object created in the previous step. The IODeviceDescription object contains information about the device's logical location in the system, and the device can query this object for additional information about the way it is configured. From this information, **probe:** can determine whether the device exists. If the device

is present, **probe:** instantiates and initializes the driver. Your **probe:** method should invoke the **initFromDeviceDescription:** method, which initializes the driver.

Note: Use the **alloc** and **initFromDeviceDescription:** methods to instantiate and initialize the driver, not the **new** method.

If **probe:** creates a driver instance, it returns YES. Otherwise, it returns NO.

Note: Declare your **probe:** method to return BOOL—not **id**.

I/O and Interrupt Requests

Everything a driver does—whether or not it's a Driver Kit driver—is the result of one of two types of requests:

- I/O requests (from a user-level program, the kernel, or another driver)
- Interrupt requests (from the hardware)

Interrupt requests include "soft interrupts," such as timeout notifications. The Driver Kit thread-based design allows you to manage I/O requests and interrupts one at a time.

Scheduling Hardware Access with I/O Threads

Different drivers have different requirements for ordering their accesses to the hardware. Driver Kit display drivers are very simple in this respect: they don't have to queue requests because the Window Server is the only process that makes requests, and it sends them one at a time. Display drivers may be particularly simple because on many systems, display hardware doesn't generate interrupts.

Other drivers have to be more careful. These drivers use an I/O thread—a single thread of execution that handles all access to a single hardware device. Some of the device classes, such as those for SCSI controllers, network, and sound devices, start up the default I/O thread for you.

Typically, each driver instance has exactly one I/O thread. However, some drivers use a single I/O thread for more than one instance. What matters is that only one thread at a time has access to any particular hardware resource.

Note: Some hardware devices can handle more than one request at once. For example, some SCSI controllers can queue multiple commands.

At any given time, the I/O thread should be doing exactly one of two things:

- Waiting for an I/O request (from a user, the kernel, or another driver) or an interrupt message
- Executing (dealing with the hardware)

Processes can use a variety of mechanisms to communicate I/O requests to the I/O thread. One of these mechanisms—Mach messages—is the same way the kernel informs the I/O thread that an interrupt has occurred. In this scheme, the kernel enqueues Mach messages for the I/O thread. When the I/O thread isn't executing a request, it dequeues the message and invokes an appropriate driver method in response. (You can also write a custom I/O thread to take whatever action you want in response to messages.) "Synchronizing with the I/O Thread" in Chapter 2 provides more details.

The I/O thread model greatly simplifies driver development and lessens the time needed for debugging the driver. Only one thread deals with any hardware resource at a time, so it's not necessary to use locks and disable interrupts to protect access to hardware and data structures. The user thread communicates requests to the I/O thread, and commands can be enqueued for the I/O thread to execute. The driver can handle one request at a time—instead of many requests to access multiple resources at the same time.

Servicing Interrupts

The Driver Kit has a simple scheme for servicing interrupts: The kernel notifies drivers of interrupts by sending them Mach messages. Each driver can receive these messages whenever it chooses, typically when it isn't executing any other requests.

The advantages of this scheme become clear when you consider an alternative—the traditional UNIX method of handling interrupts. Traditional UNIX drivers handle interrupts as soon as they happen—even if the driver is already executing an I/O request. Each driver registers an interrupt handling function that's called whenever the device interrupts. Some systems can't tell exactly which device interrupted, so they call several drivers' interrupt handlers until one accepts the interrupt. While an interrupt is being handled, nothing else in the system (except higher priority interrupt handlers) can execute.

Under the traditional UNIX scheme, drivers can't control when interrupts occur. All they can do is control when interrupts *don't* occur by disabling interrupts. Drivers disable interrupts to protect critical sections of code, such as those that access hardware or access data structures that are also used by interrupt handlers. However, disabling interrupts has disadvantages:

- If a driver disables interrupts for too long, the consequences can be anything from reduced performance to system crashes or hangs.

- If a driver disables interrupts and, through some bug, fails to reenable them, the system will hang.
- It’s easy to fail to protect a critical section—especially when you’re changing code that someone else wrote—which can result in bugs that are hard to track down.

The Driver Kit scheme of interrupt handling lets you choose when to handle interrupts, so you don’t have to protect critical sections from interrupt handlers. This scheme works well with most hardware devices.

IODirectDevice provides a default I/O thread that intercepts Mach interrupt messages and notifies drivers of them with Objective C messages. Driver objects are notified of interrupts with the **interruptOccurred** or **interruptOccurredAt:** message. See the sections “Interfacing with the Driver” and “Handling Interrupts” in Chapter 2 and the IODirectDevice class specification in Chapter 5 for more information.

A few devices require that interrupts be handled immediately. For example, a device might have a register that must be read within 50 microseconds of the interrupt occurring. On some devices data overruns occur if interrupts aren’t handled quickly enough. In these cases, a kernel-level driver might need to register a direct interrupt handler—a function that’s called as soon as the interrupt is detected. This function should perform any time-critical operations and, if necessary, send a Mach message so that the driver can further process the interrupt. The section “Custom Interrupt Handlers” in Chapter 2 describes how this interrupt handling function should work.

2

Designing a Driver

The previous chapter covered basic Driver Kit concepts. This chapter discusses details of how to design Driver Kit drivers:

- How to create and initialize a driver
- How tasks and threads work in drivers and how to communicate with the I/O thread
- How to handle interrupts
- How to connect a driver with other drivers it needs to communicate with

Information about specific kinds of drivers—for example, how to write a SCSI controller driver—is in Chapter 3, “Support for Specific Devices.”

Driver Writing Guidelines

Here are guidelines to follow in designing and writing a device driver:

- Read the specifications for the hardware you’re working with.
- Read the first four chapters of this manual.
- Read the IODevice and IODirectDevice class descriptions.
- Decide which class your driver will be a subclass of. Read this class description and the descriptions of any protocols the class conforms to. Read the class specification for any other related classes. If you’re writing a network driver, for instance, look at IONetwork.
- Look at examples of drivers for the type you’re writing. Examples are located in **/NextDeveloper/Examples/DriverKit** and **/NextLibrary/Documentation/NextDev/Examples/DriverKit**
- Create a subclass. Add the appropriate instance variables and methods to your driver subclass.
- Override or write methods in your subclass and any protocols it conforms to. Implement the methods to perform their functions with your hardware.

Creating and Initializing Drivers

You must override the **probe:** class method of IODevice in your subclass. This important method looks for the hardware and instantiates and initializes a device driver. The IODeviceDescription object passed as the parameter to **probe:** provides information about the driver object, including configuration information.

Warning: You should use the **alloc** and **initFromDeviceDescription:** methods to instantiate and initialize a driver—not the **new** method.

For direct device drivers, the IODeviceDescription parameter contains architecture-specific information about a device, such as its DMA channels and interrupts. Your driver subclass should determine whether the device is really present. If so, it should create an instance of itself, using the information in the IODeviceDescription. IODeviceDescription and its subclasses provide access to the device information.

After **probe:** instantiates the driver, it should invoke the **initFromDeviceDescription:** method to initialize the instance (with help from some other methods, too). You typically override this method, although you should incorporate the superclass's implementation by invoking this message on super prior to performing the rest of your initialization.

```
[super initFromDeviceDescription:aDeviceDescription];
```

Look at this method's description in your driver's superclass to see what functions it provides for you. For example, the IODirectDevice class's **initFromDeviceDescription:** reserves address ranges, DMA channels, and IRQs (interrupt numbers) for a driver.

The initialization sequence must also include registering the driver with **registerDevice** so the rest of the system knows about the driver.

For direct device drivers, attach interrupts using **attachInterruptPort** or some other method that invokes **attachInterruptPort**. IODirectDevice's **startIOThread** invokes it, for example.

Here's a skeleton of the **probe:** method for a direct device driver of the class MyClass. Italicized text delineated in angle brackets, that is << >>, is to 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, e.g. checking to
make
       sure the I/O port range is large enough.  Make sure the
       hardware is really there. Return NO if anything is wrong.
>>

    return [instance initFromDeviceDescription:devDesc] != nil;
}
```

If your driver subclass that receives the **probe:** message is an indirect device driver, the IODeviceDescription specifies an IODevice instance (typically for a direct device) that the indirect device driver might want to work with to communicate with its hardware. For example, if the indirect device driver controls SCSI disks, then the IODeviceDescriptions it receives specify instances of IOSCSIController, a direct device driver. Your driver should determine whether it needs to use the hardware controlled by the specified IODevice instance (for example, whether the SCSI controller has disks attached). If so, your driver subclass should create instances of itself. Here's an outline of **probe:** for this case:

```
+ (BOOL)probe:deviceDescription
{
    MyIndirectDevice *instance = nil;
    /*Get IODirectDevice object this indirect device is connected
to*/
    id controller = [deviceDescription directDevice];
    BOOL rtn = NO;

    for (<< each possible device attached to the direct device >>)
{
        if (instance == nil)
            instance = [MyIndirectDevice alloc];

        if (<< we can't reserve this device
           (implying that another driver controls it) >>) {
            continue;
        }

        << Check whether the device really exists and is a device
we
           can control.  If so, initialize an instance of this
driver
           with a driver-specific version of init.  For example:
             initRtn = [instance initWithController:controller];
>>
```

```
            if (<< the instance was successfully initialized >>) {
                [instance registerDevice];
                /* Do any other driver-specific initialization. */
                instance = nil;
                rtn = YES;
                break;
            }
            else
            << Release our reservation for this device >>
        } /* end of for loop */

        if(instance) {
            /* Free up any leftover indirect devices. */
            [instance free];
        }

        return rtn;
    }
```

Besides the information specific to direct or indirect devices, the IODeviceDescription's IOConfigTable contains miscellaneous configuration information. A beep driver's configuration table, for example, might specify that the driver is a sound-related device and specify the frequency of beeps. The IOConfigTable can be retrieved from the IODeviceDescription using the **configTable** method. The **probe:** method or methods that it invokes may do further initialization using this information.

Connecting a Driver to Other Drivers

The **driverLoader** program loads your driver's code into the kernel, either because you invoke it or as a result of the driver being specified in the system configuration. The **driverLoader** program uses the loadable kernel server mechanism and is described in Chapter 4. Once loaded, the driver needs to be connected with the appropriate direct and indirect device drivers that are already in the kernel.

For example, suppose you load a new indirect device driver that controls a SCSI scanner. The SCSI scanner driver works in combination with one or more SCSI controller drivers, so the SCSI scanner driver needs to find each IOSCSIController object in the system.

For another example, consider a direct device driver that manages a SCSI controller. Once the driver is loaded and initialized, you want to give all of the SCSI indirect devices (such as disks and scanners) a chance to connect to this controller. Each SCSI disk that's attached to the controller needs a new IODisk instance that's connected to an instance of the IOSCSIController.

Terminology: Protocols

A *protocol* is a list of method declarations, unattached to a class definition. Any class, and perhaps many classes, can implement a particular protocol.

Protocols are discussed in Chapter 3 of *NEXTSTEP Object-Oriented Programming and the Objective C Language*.

Discovering Other Objects

When any IODevice subclass is instantiated and initialized, it's automatically connected with any IODevices in the system that need to work with it. Here's how this happens:

- All IODevices to which an indirect device can be connected must declare their exported interface as an Objective C protocol. For example, the IOSCSIController class declares its exported methods (the messages that indirect devices can send it) in the IOSCSIControllerExported protocol.
- All IODevices that are indirect device drivers must implement the **requiredProtocols** class method. This method returns a list of protocols the driver's direct devices must conform to.
- Each IODevice must implement the **deviceStyle** class method, which identifies the driver as a direct, indirect, or pseudo device driver.
- Each IODevice instance must invoke **registerDevice** when it's initialized (usually in its implementation of **initFromDeviceDescription:**). This method tells the rest of the system that the driver exists and also probes all indirect IODevices that require this object's protocols, giving them a chance to connect to this object.

When driver code is loaded into the kernel, the kernel probes the newly added class and possibly other classes in the system. The result is that each class is probed exactly once per object that it might need to connect to. The kernel probes classes with the **probe:** method as described below.

If the newly loaded class is an indirect device driver (the system determines this using the **deviceStyle** class method), the kernel does the following:

```
For each IODevice object (not just IODirectDevices)
   If the object supports all protocols needed by the new class
      The kernel creates an IODeviceDescription that has this object as the direct
```

device
    The kernel probes the new class with the IODeviceDescription as its parameter

If the newly loaded class is a direct device or pseudo device driver, the kernel simply probes the new class, without trying to connect it yet.

Whenever a device of any style invokes **registerDevice**—which should happen whenever a driver object is initialized—the following happens:

For each indirect device class
  If the newly registered object supports all protocols needed by the indirect driver
    The kernel creates an IODeviceDescription that has this object as the direct device
    The kernel probes the indirect device class, giving it the IODeviceDescription

In this way, every indirect driver is probed with the device description for every possible direct driver object it could feasibly be connected to. When the indirect driver's **probe:** method examines the direct device description, it instantiates itself only when the indirect device it supports is physically connected to the direct device, that is, when the hardware is really present.

Interfacing with the Driver

Drivers export a set of methods that the kernel or programs can use to communicate with the driver. These *exported* or *interface* methods communicate requests to the I/O thread.

You don't need to be concerned about the interface to your driver in most cases. The kernel will find your driver and use its exported methods automatically—you don't have to do anything. Most display, network, SCSI controller, and sound drivers are integrated into the system this way.

For some drivers, such as SCSI peripherals, you may need to provide an interface that user-level programs or other drivers can access. This interface program then invokes the driver's exported methods.

The ideal interface between user-level programs and drivers would be Objective C messages. Currently, this direct interface isn't possible for these reasons:

- User-level drivers aren't supported.
- The Distributed Objects system (which enables Objective C messages to be sent between objects in separate tasks) doesn't work in the kernel.

You can make your driver's user level to kernel level API more object-oriented by providing user-level classes that cover your driver's interface. For example, Sound Kit objects such as NXSoundOut hide the sound driver's private Mach message interface.

This section discusses ways you can communicate with the driver if you need to.

Entry Points

If you need to provide an interface, you may want to provide a set of entry points for common driver requests, such as read, write, and so on. Your driver may have UNIX-style or Mach message-based entry points.

UNIX-style Entry Points

You can add a set of UNIX-style entry point functions, such as **open**(2) and **read**(2), to the **cdevsw** table for character drivers by invoking the IODevice class method **addToCdevswFromDescription:open:close:read:write:ioctl:stop:reset:select:mmap:getc:putc:**. A similar method adds entry points to the **bdevsw** table for block drivers. These methods search for free locations in these tables. The entry point functions added can then communicate with your driver by sending it Objective C messages or Mach messages. See Chapter 2, "Using Mach Messages" in *NEXTSTEP Operating System Software*.

Note: *Mach* messages are not the same kind of messages as *Objective C* messages sent to objects. See the references on the Mach operating system and Objective C language in the "Suggested Reading" section of the Appendix.

Your driver can retrieve or set the driver's character major device number with **characterMajor** or **setCharacterMajor**. Similarly, **blockMajor** or **setBlockMajor** retrieves or sets the driver's block major device number.

UNIX entry points are documented in books about UNIX device drivers. See "Suggested Reading" in the Appendix for more information about UNIX device drivers.

Entry Points via Mach Messages

You can develop a message-based driver interface based on Mach messages. You can create a loadable kernel server and communicate with it using Mach messages. Use the Mach Interface Generator (MiG) to create this message interface. (MiG generates remote procedure calls that handle the Mach messaging for you.) The loadable kernel server can then send Objective C or Mach messages to the driver,

just as UNIX entry point routines can do.

For more information, refer to *NEXTSTEP Operating System Software*, Chapter 2, "Using Mach Messages" and Part 2, "Writing Loadable Kernel Servers."

Other Communication Methods

You can provide other ways to interface with your driver besides entry points.

Using IODeviceMaster

An IODeviceMaster object can get the object number of a device driver using one of the **lookUp...** methods such as **lookUpByDeviceName:objectNumber:deviceKind:**. Then it can get or set parameters via methods such as **getCharValues:forParameter:objectNumber:count:** or **setCharValues:forParameter:objectNumber:count:**. Manipulating parameters enables applications to control the driver. It also allows telling preloaded programs which major device numbers are used.

You can also send driver-specific commands and send and receive small amounts of data. Since IODeviceMaster's buffers are small, the performance overhead would be prohibitive to handle large amounts of data. Although any process can use IODeviceMaster to get information from a driver, IODeviceMaster allows only the superuser to send information to a driver. This mechanism replaces the UNIX **ioctl()** interface.

Using IODevice Methods

If the amount of data you need to transfer to and from your driver is relatively small, you can use the **getIntValues**/**setIntValues** or the **getCharValues**/**setCharValues** methods in IODevice to communicate with user-level applications. Using those methods is easier than using Mach messages.

Threads in Kernel-Level Drivers

In a user-level driver, every thread the driver creates executes in the driver's own task, as shown in Figure 2-1. There's no way for any driver code to execute in any other task; neither the kernel nor any task besides the driver's own task ever executes the driver's code. Kernel-level drivers aren't so simple, however—and the

Driver Kit currently supports only kernel-level drivers.

All kernel-level device drivers run in the kernel's memory address space, but unlike user-level drivers, their threads aren't all in the same task. A loaded kernel driver might run in a thread in the kernel task created especially for the driver. (A *kernel task* is a task that shares the kernel's address space but *not* the kernel's IPC space.) Additional threads created by kernel-level drivers execute as part of another kernel task, the *kernel I/O task*. Figure 2-1 shows the relationship between kernel-level driver threads and the kernel I/O task.

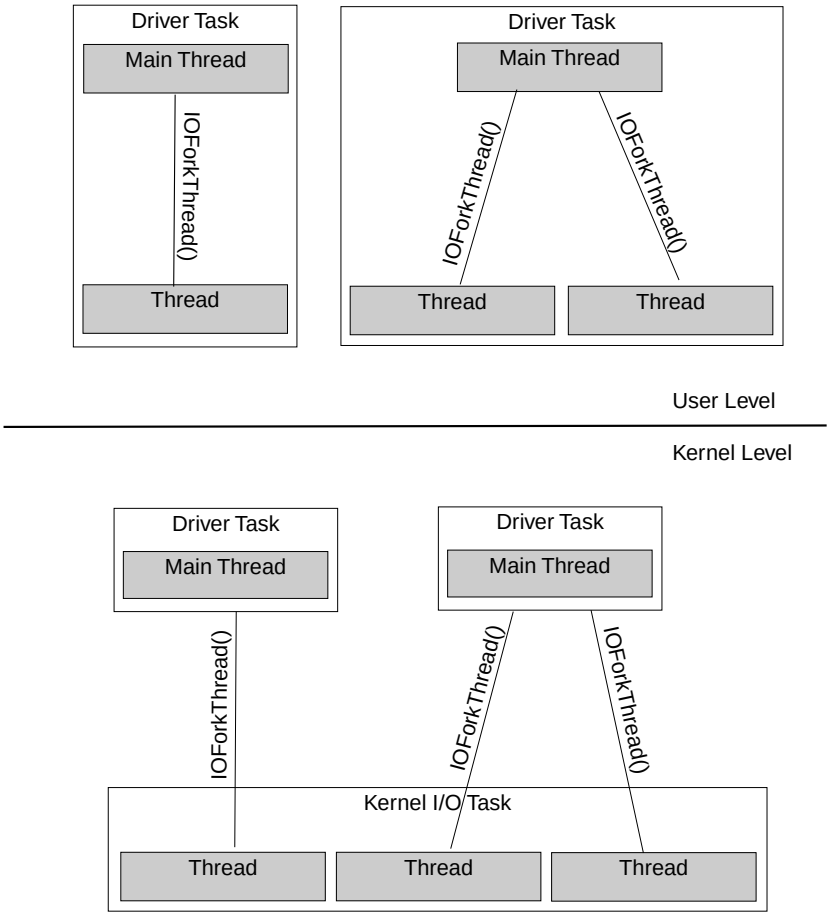

Figure 2-1. Threads in User-Level and Kernel-Level Drivers

A complication for kernel-level drivers is that their code can execute in threads that don't belong to the driver. For example, the kernel invokes a network driver's **outputPacket:address:** method whenever the driver should transmit a packet. This method executes in whatever context the invoker of the method is in, *not* in the context of any of the driver's threads. Another example of executing in a nondriver thread is that drivers with UNIX entry points operate in the calling user process's context.

In general, if a method or function isn't always called directly by an I/O thread (or by functions or methods that are called directly by the I/O thread) and the documentation doesn't say that the method is called in the context of the kernel I/O task, you should assume that the method or function has been called by an unknown thread in an unknown task.

Synchronizing Driver Requests with the I/O Thread

A device driver receives requests to perform operations from various sources external to the driver via its exported methods. Both the user's kernel thread and the I/O thread may invoke the driver's exported methods against the driver. As the previous section "Threads in Kernel-Level Drivers" noted, a driver can run in three places: The user's kernel thread (the thread that synchronously receives user commands), in another kernel thread (a timeout function, for example), or in the I/O thread. This section discusses how to coordinate these activities in different threads.

You may not need to be concerned about synchronizing these requests with your driver. Display drivers don't use an I/O thread. For other devices, the default I/O thread (which is started automatically by the network, SCSI controller, and sound device classes) handles this coordination for you. The driver's methods are invoked from the appropriate threads, and so on. Most display, network, SCSI controller, and sound drivers require no further integration.

For some devices, such as SCSI peripherals, you may need to coordinate these requests and services between the various threads. If you had to provide your own driver interface, for instance, you need to pay attention to these issues.

In keeping with the Driver Kit paradigm, exported methods should generally not perform I/O requests directly but send requests to the I/O thread. Only the I/O thread touches hardware and other critical resources. This way, no exported methods manipulate hardware or other critical resources—the I/O thread does all of the work. This structure eliminates the need to use the UNIX **spl...** functions to change priority, to disable interrupts, or to employ other mechanisms to prevent multiple threads from accessing the hardware and interfering with each other. The I/O thread can perform operations in a straightforward sequence as it chooses, without interference from other threads. The benefit is that your code will be simpler and more reliable, your design will be more comprehensible, and you'll eliminate deadlocks and race conditions.

Starting the I/O Thread

To start the default I/O thread, invoke IODirectDevice's **startIOThread** method. It forks the thread and invokes **attachInterruptPort**, which creates an *interrupt port* for the thread. The thread receives Mach messages on this port. A Mach message could be from the user's kernel thread requesting it to execute an I/O operation, or it could be from the kernel notifying the I/O thread that an interrupt occurred. Some of the device classes, such as those for SCSI controllers, network, and sound devices, start up the default I/O thread automatically.

Note: Even though it is called an interrupt port, the I/O thread receives all its Mach messages on this port—not just interrupt messages.

To start a custom I/O thread, call the function **IOForkThread()**. Its argument is a function, which consists of a while loop that waits for and executes commands from the rest of the driver. This function runs in the kernel's I/O task. Like the default I/O thread, only this function should touch the hardware.

Synchronizing with the I/O Thread

A device driver's exported methods execute in response to some action initiated by a user program. A method may have two flavors of communication with the I/O thread. In some cases, an exported method needs to do *synchronous* communication with the I/O thread—that is, the exported method sends some work to the I/O thread and waits until that work is done. In other cases, an exported method does *asynchronous* I/O—it just sends some work to the I/O thread and continues executing, without waiting for the work to be done.

In either case, the I/O thread may not be ready to perform the requested hardware operation when the user thread requests it. Therefore, there must be a way to synchronize the interface functions with the I/O thread. This synchronization is essentially automatic if you use the default I/O thread, because the thread takes requests only when it's ready to handle them.

Coordination between the driver's user-level exported methods and the I/O thread can occur in two ways:

- Using Mach messages, but it's recommended that they be used only with the default I/O thread. See "Synchronizing Using Mach Messages" later in this section.
- Using a type of lock known as a *condition lock.* See "Synchronizing Using Condition Locks" later in this section. They're fast and easy to use. NXConditionLock is documented in the Mach Kit in *NEXTSTEP General Reference*.

Sometimes, for performance or other reasons, a driver might have its exported methods perform some I/O directly without going through the I/O thread. An Ethernet

driver might be an example of this. The method that's called when a client wants to send a packet out to the network might perform no I/O—it might just add a DMA frame to the device's DMA queue. The exported method could do this directly without waking up the I/O thread. The Ethernet I/O thread would basically just service interrupts and dispatch incoming packets. A lock in the driver would protect access to the hardware in the case where the output method has to start up an idle DMA channel.

Synchronizing Using Mach Messages

A user-level process typically doesn't communicate directly with the driver. The user-level process communicates with a set of UNIX entry points or with a loadable kernel server, as indicated in "Interfacing with the Driver." These entry points or loadable kernel server can then communicate with the I/O thread via Objective C messages (through the driver's exported methods) or Mach messages. Both synchronous and asynchronous I/O requests can be performed using Mach messages between the exported methods and the I/O thread.

A way of communicating with the I/O thread is supported by the default I/O thread provided by IODirectDevice. In this scheme, each request is sent to the IODirectDevice's interrupt port, using a message ID. The file **/NextDeveloper/Headers/driverkit/interruptMsg.h** defines a set of messages. The only information in a message is its ID. Command buffers or other data, for instance, are not part of the message. The default I/O thread invokes one of the following methods, based on the message ID received:

Message ID	Method Invoked
IO_TIMEOUT_MSG	timeoutOccurred
IO_COMMAND_MSG	commandRequestOccurred
IO_DEVICE_INTERRUPT_MSG	interruptOccurred
IO_DEVICE_INTERRUPT_MSG_FIRST to IO_DEVICE_INTERRUPT_MSG_LAST	interruptOccurredAt:
(anything else)	otherOccurred:

You implement these methods to respond appropriately to the condition.

Interrupt messages are sent automatically by the kernel. If you want to use the other types of Mach messages, your driver or some other module it works with must explicitly send them. An advantage of using Mach messages to notify the I/O thread of requests is that the thread can service incoming I/O requests while waiting for interrupt messages.

You can also devise your own Mach messages and invoke whatever I/O thread methods you choose in response to them. You would implement the **receiveMsg**

method in IODirectDevice to dequeue the next Mach message from the interrupt port.

The IOSCSIController class is an example of this. The SCSI bus is capable of performing overlapped I/O requests, in which one I/O request can be started while another is in progress and is disconnected from the bus. In this case, the IOSCSIController I/O thread receives I/O requests through Mach messages.

IOSCSIController itself doesn't manage, allocate, or use any Mach ports at all. It depends on **startIOThread** to set up one port, the standard interrupt port. Everything else is done by subclasses of IOSCSIController. IOSCSIController subclasses currently use the interrupt port for all Mach interprocess communication, including command messages and timeout messages. The messages are distinguished by their message ID, not the port to which they are sent.

The example SCSI driver in **/NextDeveloper/Examples/DriverKit/Adaptec1542B** is a good illustration of these techniques.

An older technique that created a custom Mach message that included the command buffer is no longer used. It's been replaced by the mechanism of enqueuing a command buffer on some well-known location (such as an instance variable) and sending a command message to the interrupt port. This results in **commandRequestOccurred** being invoked by the I/O thread, as noted above.

Synchronizing Using Condition Locks

Condition locks are provided by the Mach Kit's NXConditionLock class, which works at both user and kernel level. For information about NXConditionLock beyond what's given here, see *NEXTSTEP General Reference*.

Using Mach messages and condition locks for synchronization aren't necessarily mutually exclusive. For instance, you could use a condition lock on a buffer as illustrated in "Using a Command Buffer" below and have the I/O thread wait for Mach messages on its interrupt port. However, the following two synchronization techniques *are* mutually exclusive:

- I/O thread waiting for messages on its interrupt port
- I/O thread waiting for work using a condition lock (as shown in the example below)

A general technique for passing I/O information from a driver's exported methods to its I/O thread using condition locks is shown below and illustrated with an example.

Using a Command Buffer

Some known location, perhaps an instance variable in the driver object, can be used to pass commands from the exported driver methods to the I/O thread. This variable

may contain a structure (called **cmdBuf_t** in the following example) that serves as a *command buffer*, the fundamental unit of communication between exported methods and the I/O thread. You would define the command buffer differently for each driver—it must contain all the information needed by the I/O thread to perform a single I/O request. For example, a command buffer for a disk driver might contain a disk address, a virtual address, a byte count, and a read/write command flag. The command buffer might also contain fields by which the I/O thread can indicate completion status—for example, a device-specific status field and a field indicating the number of bytes transferred.

The command buffer contains a variable for a token that indicates which hardware operation the I/O thread should perform. This variable may be the value of an **enum**, for instance.

The command buffer also contains an NXConditionLock (called **cmdBufLock** in the example below), which manages access to the command buffer. An exported method (a write routine, for example) sets the lock unconditionally when it wants the I/O thread to execute a command. It sets up the command buffer for the operation it wants to perform and releases the lock with the condition NOT_COMPLETE. It then waits on the lock until its state is COMPLETE, which results in the user thread sleeping until the I/O thread sets the lock condition to COMPLETE. Meanwhile, the I/O thread is waiting on the lock until its state is NOT_COMPLETE and it has a command to execute. When those conditions are satisfied, the I/O thread then sets the lock. When it finishes executing the command, it releases the lock and sets its state to COMPLETE, which is the cue for the user thread to wake up.

Managing Multiple Requests

You can also queue multiple requests with condition locks. This lock works independently of the lock indicating a command completion.

Declare an instance variable (which may be in the driver object) that’s the head of a queue of command buffers. Command buffers are added to the queue by exported methods and removed from the queue by the I/O thread.

Declare an instance variable that’s an NXConditionLock (this variable is called **ioQueueLock** in the following example). This lock protects the queue and provides a way for the I/O thread to sleep until it has work to do. This lock has two states, QUEUE_EMPTY and QUEUE_NOT_EMPTY. Note that each command buffer has its own condition lock (**cmdBufLock** in the example below) to control completion of the I/O request specified in that particular buffer.

Example

Here’s an example of an exported method that communicates with the I/O thread synchronously. This example shows how locks can be used to synchronize with a

custom I/O thread in lieu of command messages to the interrupt port. It also shows how to queue multiple requests. Italicized text delineated in angle brackets, that is << >>, is to be filled in with device-specific code.

```
- (IOReturn)makeIORequest:(int)anArgument
{
    cmdBuf_t cmdBuf;

    /* Initialize lock */
    [cmdBuf.cmdBufLock lock];

    << Fill in cmdBuf fields appropriate for this I/O. >>
    /* Unlock and set cmdBufLock to condition NOT_COMPLETE. */
    [cmdBuf.cmdBufLock unlockWith:NOT_COMPLETE];

    /*
     * Enqueue this command buffer and let the I/O thread
     * know that it has work to do.
     */
    [ioQueueLock lock];
    << Enqueue cmdBuf on ioQueue. >>
    [ioQueueLock unlockWith:QUEUE_NOT_EMPTY];

    /*
     * Wait for I/O thread to process the command buffer and
signal
     * completion.
     *
     * NOTE: The following is necessary only for synchronous I/O.
     */
    [cmdBuf.cmdBufLock lockWhen:COMPLETE]; //ONLY FOR SYNCHRONOUS
    [cmdBuf.cmdBufLock unlock];

    /*
     * I/O is complete.
     */
    << Free necessary data from cmdBuf. >>
    << Return I/O result. >>
}
```

The I/O thread invokes the following method while waiting for work from the exported methods:

```
- (cmdBuf_t *)waitForWork
{
    cmdBuf_t *cmdBuf;

    [ioQueueLock lockWhen:QUEUE_NOT_EMPTY];
    << Dequeue head of ioQueue, save in cmdBuf. >>
    if(<< ioQueue is empty >>)
        [ioQueueLock unlockWith:QUEUE_EMPTY];
    else
        [ioQueueLock unlockWith:QUEUE_NOT_EMPTY];
    return cmdBuf;
}
```

The I/O thread executes the request and wakes up the user thread as follows:

```
- (void)performIO:(cmdBuf_t *)cmdBuf
```

```
{
    << Execute I/O request >>
    [cmdBuf->cmdBufLock lock];
    [cmdBuf->cmdBufLock unlockWith:COMPLETE];
}
```

Sending Messages Outside the I/O Task

When a driver executes outside the I/O task, it no longer has send rights to ports that it has in the I/O task. A workaround for this problem is to use the **msg_send_from_kernel()** function instead of **msg_send()** to send the message to the port. The port must first be converted to a form that's valid in the kernel's IPC space, using **IOConvertPort()**. An example of using **msg_send_from_kernel()** is in the IOSCSIController class specification.

Handling Interrupts

Most kernel-level drivers don't handle interrupts directly. Instead, the kernel notifies the driver of an interrupt by sending a Mach message to the interrupt port. An interrupt port is allocated when a direct driver object is initialized by the **attachInterruptPort** method of IODirectDevice. Figure 2-2 shows how interrupts are handled by the kernel and the I/O thread of a direct device driver.

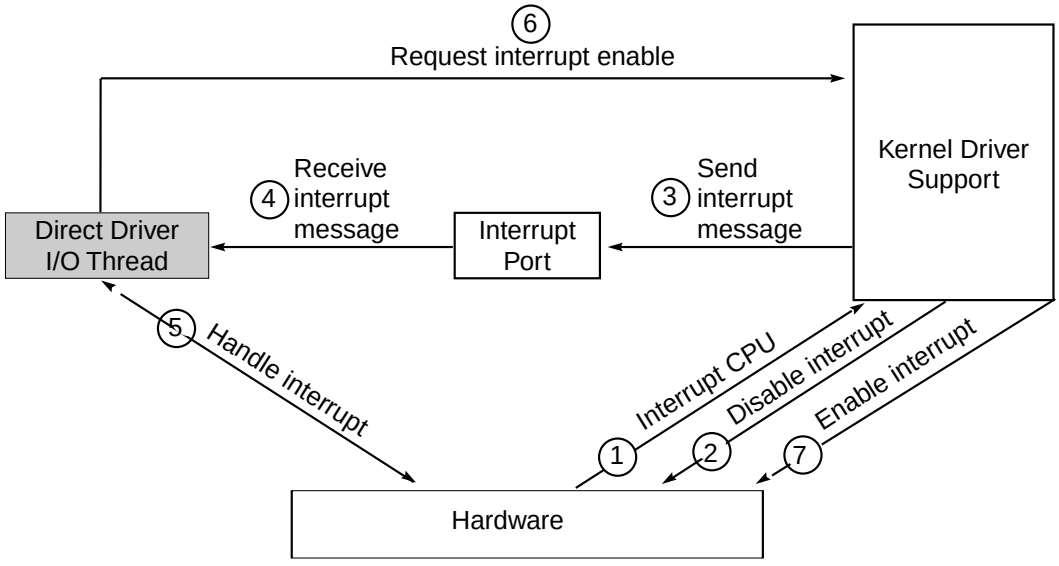

Figure 2-2. Driver Kit Interrupt Handling

As Figure 2-2 shows, when an interrupt occurs (1), the kernel masks off further occurrences of that particular interrupt (2) and sends a message to the appropriate interrupt port (3). It then returns from the interrupt. The interrupt message contains no information except for a message ID in its header that identifies this message as an interrupt message.

When the driver receives an interrupt message (4), it should examine the hardware to determine the cause of the interrupt and perform whatever action is necessary for continuing the I/O transfer in progress (5). It should then request that the kernel reenable interrupt notification for the device (6).

No further interrupt messages are sent to the driver until the kernel enables interrupts (7). If interrupts are shared between devices, the kernel reenables interrupts. If interrupts are not shared, the kernel resumes sending interrupt messages. See the section “Shared Interrupts” in this chapter.

When a device interrupts while a message is queued on the corresponding interrupt port, the kernel returns immediately without sending an interrupt message. After **msg_receive()** returns (which dequeues the message), the kernel regains the ability to send interrupt messages (but not until the device interrupts again). The memory for messages is fixed since the kernel can’t allocate more memory at the interrupt level. The message buffers accommodate only one interrupt message, so any interrupts that arrive while an interrupt message is already queued are lost.

The I/O thread automatically calls **msg_receive()** to get messages on its interrupt port. The default I/O thread also invokes the **interruptOccurred** or **interruptOccurredAt:** method in response to interrupt messages. Most of the device-specific classes in the Driver Kit do this for you.

One Device, One Thread

A driver is responsible for maintaining and dealing with three kinds of resources—hardware, the driver’s private data, and client I/O requests. In a multiprocessor system, or in a system in which driver code contains interrupt handlers, a great deal of care must be taken to protect access to all three of these resources. Almost every function must use locks and disable interrupts. Even in the most well-thought out design, the presence of locks and interrupt disabling makes code hard to read and tends to lead to bugs. The problem is most apparent in code that manipulates the hardware directly.

The Driver Kit’s solution to this problem is this:

> Given any hardware resource, one and only one thread can deal with that resource at a time. Interrupt handlers have no direct access to the resource.

Consider a SCSI controller chip, for example. If exactly one thread in the system has

access to the chip, there's no need for locking or for disabling interrupts to protect the code that manipulates the chip.

Another way of looking at this is that for a given piece of hardware, only one operation at a time can happen. At point A, a driver might be setting up a chip to start I/O. At point B, the driver might be waiting for an interrupt from the chip. At point C, the driver might be responding to an interrupt and interrogating registers to see what caused the interrupt. A driver is never setting up a chip to start an I/O at the same time it's interrogating registers to see what caused an interrupt. In UNIX drivers, a combination of locks, interrupt disabling, and an interrupt-driven state machine assure that the driver attempts only one hardware operation at a time. In the Driver Kit, the one-at-a-time sequence of operations is enforced by having a single thread (the I/O thread) perform all hardware operations.

Another reason for this model is the desire to have drivers run in user space. There's no practical way for user-level drivers to run interrupt handlers with interrupts disabled; only kernel software can do this.

Some drivers in exceptional cases may choose to have multiple threads with access to one piece of hardware. The "one device, one thread" model is not an absolute. It's merely a design goal that has proved to be a viable basis for writing Driver Kit drivers.

Example: Floppy Disk

Let's look at a simple piece of hardware, a floppy disk controller chip. Floppy disk I/O consists of a predictable sequence of operations—starting an I/O request, waiting for an interrupt, and manipulating some registers. A feasible template for a floppy disk I/O thread looks like this:

```
floppyThread()
{
    << Initialize local data structures. >>
    << Initialize hardware. >>
    while(1) {
        << Wait for an I/O request from a client. >>
        << Set up the controller chip to start the I/O. >>
        << Wait for interrupt. >>
        << Manipulate controller registers to finish the I/O. >>
        << Notify client of I/O completion. >>
    }
}
```

Not all devices are this simple, but this illustrates how a single thread suffices to manipulate a hardware resource.

Traditional UNIX Interrupt Handling

Compare the Driver Kit's interrupt handling to the UNIX approach.

The traditional UNIX driver design involves a conceptual *top-half*, which is code called from higher layers in the kernel to initiate an I/O, and a *bottom-half*, which consists of various interrupt handlers and I/O completion logic. A simple example follows:

1. High-level kernel code calls the driver's **strategy()** or **write()** or **read()** routine (in the driver's top-half) to start an I/O.

2. The driver's top-half enqueues the I/O on a queue that is private to the driver, perhaps after translating the incoming data into a driver-specific format.

3. If the bottom half of the driver is idle, the top-half calls a **start()** routine to initiate a hardware operation.

4. The bottom-half takes over from here. When an interrupt occurs, the driver's interrupt handler runs and decides either that the hardware needs some more attention before completing the I/O (in which case a state machine is advanced and the driver awaits another interrupt) or that the I/O is complete (in which case higher-level code in the kernel is notified of this fact).

Things can actually get much more complicated than this. For instance, a certain section of code may sometimes run as the result of an interrupt and run the rest of the time for some other reason. Because an interrupt might occur while the code is already running, the code must protect itself during critical sections by disabling interrupts. One example of code that must be protected is a function that starts I/O. In the example given previously in this section, the **start()** function doesn't run as the result of an interrupt. However, if more work remains at I/O completion time, the **start()** function is called from the interrupt handler. The section of code that starts the I/O must be protected from interrupts so that it can complete its work correctly.

Sometimes interrupts are disabled for hundreds of microseconds or more. Such long periods without interrupts seriously hamper system throughput and cripple the ability of the system to respond to real-time events such as the arrival of serial data.

Another problem with running some subset of a driver's code at interrupt level is that locking shared data structures (even if they are shared only between the files constituting one driver) is difficult on a multiprocessor system. To access a critical data structure on a multiprocessor system—when the data can be accessed at interrupt level by all processors—noninterrupt code must first disable interrupts on all processors and then acquire a lock.

Custom Interrupt Handlers

You may need to write your own interrupt handler in some cases. A driver for a device with high data rates that depends on programmed I/O would be a good candidate for a custom interrupt handler, for instance. The IODirectDevice **getHandler:level:argument:forInterrupt:** method has been provided to support such handlers. It specifies an interrupt handler function for the driver.

Warning: Use interrupt level IPLDEVICE (defined in **/NextDeveloper/Headers/kernserv/i386/spl.h**) unless a higher interrupt level is absolutely necessary and you're fully aware of the possible consequences of using it.

If you want the I/O thread to take some action, the interrupt handler can call the **IOSendInterrupt()** function, which sends a Mach message to the I/O thread with the specified message ID.

Warning: Your driver must not send Objective C messages in an interrupt handler, since sending a message can result in memory allocation. Allocating memory can lead to sleeping, and interrupt handlers must not sleep, as described in *NEXTSTEP Operating System Software*.

Read "Designing a Loadable Kernel Server" in *NEXTSTEP Operating System Software* for more information on executing as the result of an interrupt.

Shared Interrupts

Devices may share the same interrupt. Since there are only 15 IRQs available on Intel-based computers, sharing interrupts may be necessary for some configurations.

Each time an interrupt occurs for a shared IRQ number, every driver that shares the interrupt gets an interrupt message. If the driver has its own interrupt handler, it is called.

At the end of your interrupt handling method or function, you must reenable the interrupt—whether or not the interrupt was intended for your device. You accomplish this by invoking **enableAllInterrupts**:

```
[self enableAllInterrupts];
```

If you are using a special interrupt handler, reenable interrupts by calling **IOEnableInterrupt()** in the handler. You should only reenable the interrupt after removing the source of the interrupt—by clearing the interrupt status register on the device, for example, or by using whatever mechanism is necessary for the hardware your driver controls.

The shared interrupt is masked each time an interrupt occurs. It is only unmasked

after all drivers that are sharing the interrupt reenable their own interrupts.

IODisableInterrupt() allows handlers of non-shared interrupts to indicate that the interrupt should be left disabled on return from the interrupt handler.

Note: **IOEnableInterrupt()** and **IODisableInterrupt()** must be called only inside a special interrupt handler function, that is, at interrupt level. (The special interrupt handler is the one you specified in **getHandler:level:argument:forInterrupt:**.) These functions can't be called from any other context. You shouldn't call them from **interruptOccurred**, for example.

Enable shared interrupts for your system by setting the "Share IRQ Levels" key in your driver's **Default.table**:

```
"Share IRQ Levels" = "Yes";
```

Note: Currently, shared interrupts imply level-triggered interrupts on EISA and PCI bus machines. Shared interrupts are not supported on ISA bus machines.

3

Support for Specific Devices

Earlier chapters considered generic issues for all drivers. This chapter concentrates on the essentials of writing drivers for the following specific types of devices:

- Display
- Network
- SCSI (both controllers and devices attached to controllers)
- Sound

The section for each device type lists the development hardware needed. It indicates the basic operations required for such a device driver and provides some implementation suggestions.

Figure 3-1 shows the IODevice classes that you can use to write specific drivers.

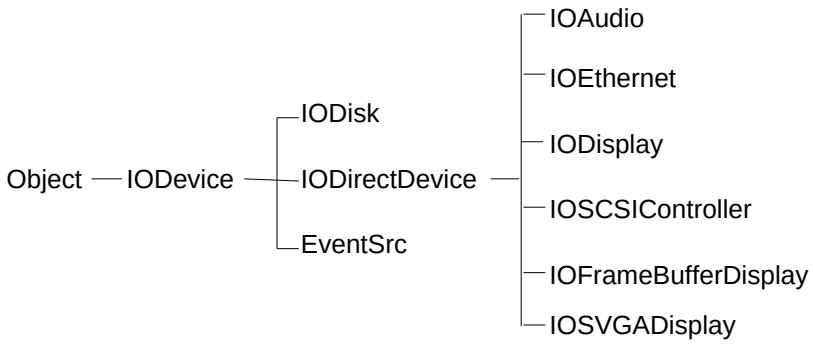

Figure 3-1. Public IODevice Classes

The Driver Kit has special support for these devices although you can also write other kinds of drivers with the Driver Kit.

In addition to device-specific classes, some kinds of drivers use non-IODevice classes that work with the IODevices. For example, network drivers typically use the IONetwork and IONetbufQueue classes.

Examples of each driver type are located in **/NextDeveloper/Examples/DriverKit**. See also the examples located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit**.

Development Support

You need support from these sources during driver development:

- Hardware vendors. You may need new hardware or firmware. You also want advance notice of changes and information about new products. Support for drivers is an ongoing process.
- Accurate and complete specifications for the hardware you're working with. Ancillary documentation such as support notes, technical notes, and sample code is also very useful.

Warnings

Driver development is hazardous to the health of your system. You will corrupt your kernel and hang your system during development, so be prepared to recover from these incidents. Furthermore, you'll most likely corrupt your system disk, so take these precautions:

- Keep code and other critical resources off your development systems.
- Have a plan for backup and rapid restoration of your disk's contents.

Display Devices

A driver for a display card is a subclass of one of these two classes:

- IOFrameBufferDisplay, for cards that can linearly map the entire frame buffer
- IOSVGADisplay, for other display cards

Figure 3-1 shows display device classes' position in the Driver Kit class hierarchy.

IOFrameBufferDisplay supports the following modes:

- 2-bit grayscale
- 8-bit grayscale

- 8-bit color
- 16-bit color (4 or 5 bits each for red, green, and blue, but only 4096 colors in either case)
- 24-bit color (8 bits each for red, green, and blue).

IOSVGADisplay supports only 2-bit grayscale.

Note: All display cards with VGA support work with NEXTSTEP. Without special drivers, however, they have a small display area (640×480) and are 2-bit grayscale.

Both classes support EISA and VL-Bus display cards. A limited number of ISA display cards are supported for performance reasons. PCI display cards are supported, but not PCMCIA display cards.

Driver Kit display drivers are simpler than their DOS or Windows™counterparts because they perform no graphics operations—the Window Server handles all graphics.

See the IODisplay, IOFrameBufferDisplay, and IOSVGADisplay class specifications for additional information about how to implement a driver.

Directories in **/NextDeveloper/Examples/DriverKit** with examples of video drivers include **ATI**, **CirrusLogicGD542X**, **QVision**, **S3**, and **TsengLabsET4000**.

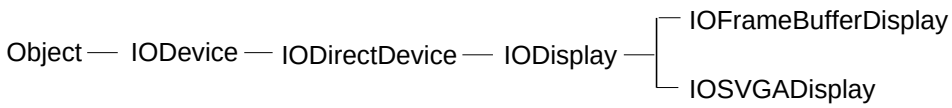

Figure 3-2. Classes for Display Drivers

Development Requirements

The following hardware is required or recommended for development and support efforts:

- A workstation with NEXTSTEP User and Developer software
- A second NEXTSTEP workstation for the target system (optional, but recommended)
- Adapter hardware
- Multisync monitor
- Frequency counter (optional, but recommended)
- Oscilloscope (optional)

Setting the Frame Buffer Address Range

If you implement an IOFrameBufferDisplay driver, you must supply the frame buffer memory range as the first range in the memory range list. This is normally done by placing this range as the first range of the "Memory Maps" key in **Default.table**. (You can also set this list with the **setMemoryRangeList:num:** method in IODeviceDescription.) The value should be the physical address memory byte range of the frame buffer. This range should be high in memory—above 2 GB, for example—to avoid conflicting with physical memory.

On PCI-based systems, the BIOS attempts to allocate the frame buffer address range for you. The BIOS places this address range in the PCI configuration data but not in the device description, so you need to update the device description with this range. Furthermore, the BIOS doesn't always succeed in determining a valid frame buffer address, so you need to check the address. Follow these steps to check and set the frame buffer address range for PCI-based systems:

1. Get the memory ranges from the device description by invoking IODeviceDescription's **memoryRangeList** method. The frame buffer address range is the first one in the list—this is the range value provided in the **Default.table**.

2. Get the PCI configuration space's frame address range, which was determined by the BIOS. Read the PCI configuration space by using the **getPCIConfigData:atRegister:withDeviceDescription:** method. Consult your device's hardware specifications to determine which PCI register holds the frame buffer address.

3. Check that the range's starting address is greater than or equal to 4 MB and correctly aligned for your hardware.

4. If the address is invalid, don't update the device description with this range. Instead, update the PCI configuration space with the range from the device description. Take the device description's address range you determined in the first step and write it to the PCI configuration space using the **setPCIConfigData:atRegister:withDeviceDescription:** method.

5. If the address is valid, update the device description. Replace the first range in the list you obtained in step 1 with the range you got from the PCI configuration. Set the ranges with the **setMemoryRangeList:num:** method in IODeviceDescription.

You should go through these steps in your **probe:** method, prior to invoking **initFromDeviceDescription:**.

Basic Operations

A display driver must perform the following basic operations:

- Instantiating and initializing a driver object
- Selecting the display mode
- Reconfiguring display hardware for the selected display mode
- Reverting to VGA display mode
- Adjusting display brightness

Instantiating and Initializing a Driver Object

Override the **probe:** method in IODevice. Your **probe:** method should find and characterize the hardware. It must verify the presence and operation of the graphics controller (CRTC) and determine its revision. The **probe:** method should also determine the DAC type, the memory size, and the clock chip type, if necessary. For PCI-based drivers, **probe:** should check and set the frame buffer range address, as indicated in “Setting the Frame Buffer Address Range.” It should create a driver instance of IOFrameBufferDisplay or IOSVGADisplay. If invalid values are found during verification, the method shouldn’t create a driver instance but should send an appropriate diagnostic message and return NO.

Note: Instead of using **probe:**, the current display driver examples use **initFromDeviceDescription:** to perform all this initialization, because they were written before the API was fully developed. The **probe:** method is preferred.

Selecting a Display Mode

IOFrameBufferDisplay’s method **selectMode:count:valid:** selects the display mode for you. To use it, you need to declare an IODisplayInfo array with one element per mode and initialize it, as in this example:

```
const IODisplayInfo QVisionModeTable[] = {
  /* 0: QVision 1024 x 768 x 8 (Mode 0x38) @ 60Hz. */
  {
        1024, 768, 1024, 1024, 60, 0,
        IO_8BitsPerPixel, IO_OneIsWhiteColorSpace, "WWWWWWWW",
        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, "WWWWWWWW",
        0, (void *)&Mode_38_66Hz,
  },
}
```

Declare an array of boolean values with one element per display mode and fill it. In

the following example, italicized text delineated in angle brackets, that is << >>, is to 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;
}
```

Finally, send a message to select a mode and handle the result, as this code section 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;
}
```

Reconfiguring Display Hardware for the Selected Display Mode

Using the appropriate commands for your display hardware, reconfigure it for the selected mode with these operations, the order of which is hardware-dependent:

- 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

Reverting to VGA Display Mode

Return the adapter to the state it would be in after a hard reset, and, in the typical case, set VGA mode to 3.

Adjusting Display Brightness

If the hardware supports changing the brightness of the display, implement the

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 then download the transfer table to the DAC. Look at an example of the **setGammaTable** method in one of the display driver examples in **/NextDeveloper/Examples/DriverKit**. Finally, indicate that you've implemented a transfer table by setting a flag:

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

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

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

Network Devices

Two classes, IONetwork and IONetbufQueue, support all drivers that directly control networking hardware.

The Driver Kit contains special support for Ethernet and Token Ring drivers in two IODirectDevice subclasses—IOEthernet and IOTokenRing—from which you create a subclass to build your network driver. IOEthernet and IOTokenRing implement the hardware-independent code needed to control Ethernet and Token Ring cards.

Figure 3-1 shows the network device classes relative to their superclasses.

See the IOEthernet and IOTokenRing class descriptions for additional information on writing Ethernet and Token Ring drivers.

/NextDeveloper/Examples/DriverKit/SMC16 contains a network driver example.

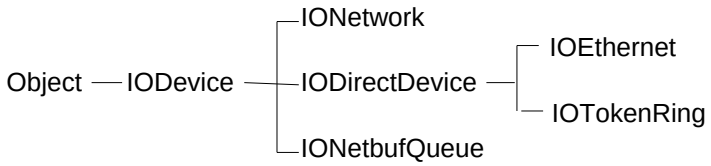

Figure 3-3. Classes for Network Drivers

Development Requirements

The following hardware is required or recommended for development and support efforts:

- Two workstations with NEXTSTEP User software (mandatory—these serve as debug master and slave)
- NEXTSTEP Developer software on one of the workstations
- For Ethernet drivers, two supported Ethernet adapters, one of which supports NEXTSTEP kernel debugging (contact NeXT to get a list of qualifying adapters)
- Target adapter hardware
- Networking hardware (cables, tees, terminators, transceivers, and hub) to link the two workstations
- Network analyzer (optional, but highly recommended)

Basic Operations

A network driver needs to support these operations:

- Instantiating and initializing a driver object
- Handling interrupts and timeouts
- Cold initialization
- Warm initialization
- Transmitting
- Receiving

Instantiating and Initializing a Driver Object

Override IODevice's **probe:** method. Your **probe:** method should find the hardware based upon a user-configured parameter such as an ID sequence or signature. This method must validate the device description passed to **probe:**, failing with a diagnostic message if any values are invalid. The **probe:** method should allocate an instance of IOEthernet or IOTokenRing, if necessary, and invoke **initFromDeviceDescription:** to initialize the instance. If it finds anything invalid in the hardware or device description, it shouldn't create a driver instance and should return NO.

Handling Interrupts and Timeouts

Implement **interruptOccurred** and **timeoutOccurred**. The kernel invokes **interruptOccurred** from the I/O thread whenever the hardware interrupts and invokes **timeoutOccurred** when a timeout occurs.

Cold Initialization

Cold initialization should perform any one-time initialization actions, such as reading the hardware address from ROM or allocating system memory for DMA buffers.

Warm Initialization

Implement the **resetAndEnable:** method to prepare the hardware and software for network activity. This method should do the following:

- Disable interrupts
- Clear pending timeouts
- Initialize hardware settings and software data structures
- Cache physical addresses
- Enable running by invoking **setRunning:**
- Reenable interrupts if the enable parameter is YES

Transmitting

Depending on what your hardware supports, choose between using a single frame or a transmit queue.

To transmit a frame, implement the **transmit:** method to follow these steps:

1. Queue the frame if it can’t be processed immediately.
2. Perform a software loopback if necessary using **performLoopback:**.
3. Transfer the frame to the hardware.
4. Free the frame’s network buffer; you may need to do this in an interrupt handler.
5. Set a timeout.
6. Handle the transmit interrupt or timeout.
7. Increment statistics such as number of frames sent, number of timeouts, and so on by invoking methods such as **incrementOutputPackets** in IONetwork.

Warning: Never attempt to retransmit at the driver level.

Receiving

To receive a packet, follow these steps:

1. Handle the receive interrupt, which indicates that a packet has been received. Incoming frames must be in a network buffer. You can allocate network buffers with **nb_alloc()** or use **nb_alloc_wrapper()** to wrap already allocated memory as a network buffer. Note that these functions can't be called at the interrupt level.

2. Check that the network buffer size is correct. You can shrink it with **nb_shrink_bot()** if needed.

3. Filter unwanted packets with **isUnwantedMulticastPacket:** in IOEthernet if the hardware doesn't provide filtering based on individual multicast addresses.

4. Hand off the packet to the kernel by invoking **handleInputPacket:extra:** in IONetwork. This automatically invokes **incrementInputPackets** to increment that count.

5. Update statistics appropriately using methods such as **incrementInputErrors** in IONetwork.

SCSI Controllers and Peripherals

You can write drivers for both SCSI controllers and SCSI peripherals with the Driver Kit.

Drivers for SCSI controllers should generally be implemented as subclasses of IOSCSIController. Drivers for SCSI devices are indirect drivers that are typically implemented as subclasses of IODevice. These indirect drivers use the IOSCSIControllerExported protocol to communicate with the SCSI controller driver object, which must conform to the IOSCSIControllerExported protocol. (Required protocols and the role they play in connecting drivers are discussed in Chapter 2.)

Figure 3-1 illustrates the position of SCSI driver classes in the Driver Kit class hierarchy.

For more information on writing a SCSI driver, see the IOSCSIController and IODevice class descriptions.

An example of a SCSI controller driver is located in **/NextDeveloper/Examples/DriverKit/Adaptec1542B**. For an example of a SCSI tape drive controller, see **/NextDeveloper/Examples/DriverKit/SCSITape**.

Object — IODevice — IODirectDevice — IOSCSIController

Figure 3-4. Classes for SCSI Controllers

Development Requirements

The following hardware is required or recommended for development and support efforts:

- A workstation with NEXTSTEP User and Developer software
- A second NEXTSTEP workstation with NEXTSTEP User software. This is *strongly* recommended: It's virtually guaranteed that you'll corrupt your disk. It's essentially mandatory if you're developing a boot driver. Furthermore, the second station allows you to debug the loaded driver at source level. Set up a procedure to quickly recover the contents of your disk.
- SCSI Host adapter
- Peripherals for testing the adapter: hard disk, CD-ROM, tape drive
- SCSI analyzer

Basic SCSI Controller Driver Operations

The basic operations needed for a SCSI driver are the following:

- Instantiating and initializing a driver object
- Initiating commands
- Handling interrupts and command completion
- Handling timeouts

Instantiating and Initializing a Driver Object

Override IODevice's **probe:** and **initFromDeviceDescription:** methods.

Implement **probe:** to test for system resources such as I/O ports and to verify the presence of hardware. If the hardware is present, create a driver instance and return YES. If invalid values are found during verification, **probe:** shouldn't create an instance; it should instead send an appropriate diagnostic message and return NO.

Your **initFromDeviceDescription:** method must invoke super's implementation:

```
[super initFromDeviceDescription:deviceDescription];
```

IOSCSIController's **initFromDeviceDescription:** method starts up the default I/O thread provided by IODevice and initializes its instance variables. Your **initFromDeviceDescription:** method should initialize the hardware state and software structures such as queues and locks.

Initiating Commands

Implement **resetSCSIBus** (in the IOSCSIControllerExported protocol) to reset the SCSI bus for your hardware.

Implement **executeRequest:buffer:client:** (also in the IOSCSIControllerExported protocol). This exported method should convert the command and data (in the IOSCSIRequest **struct** passed to it) into the format for the specific hardware and place it in a command buffer. Enqueue the buffer in some well-known location—a queuing instance variable you define in your subclass, for example. Send a Mach message with the ID IO_COMMAND_MSG to the I/O thread's interrupt port to notify the I/O thread that it should execute a command that's been placed in global data. Wait for the command to complete; you can synchronize this with the I/O thread by using an NXConditionLock object in the command buffer. (For example, you set the lock to a CMD_READY state and then do a **lockWhen:**CMD_COMPLETE. The I/O thread sets the lock state to CMD_COMPLETE when it's done. See the example in Chapter 2.) Return SCSI and driver status.

The **commandRequestOccurred** method is invoked by the I/O thread when it receives a Mach message with the ID IO_COMMAND_MSG. Implement this method to dequeue all commands that have been queued for execution. Send them to the host adapter, using the private methods and functions that you implement for your hardware. If the host adapter isn't able to accept all the enqueued commands, wait until an interrupt message arrives indicating that the host adapter has completed commands previously sent to it and may now accept more commands.

Handling Interrupts and Command Completion

When the I/O thread receives a message with the ID IO_INTERRUPT_MSG, it invokes the **interruptOccurred** method against the driver instance. Your implementation of this method should find all commands that the host adapter has completed, mark their respective command buffers complete, and dequeue them. It should reinvoke the **commandRequestOccurred** method to process any remaining enqueued commands.

Handling Timeouts

Just before the I/O thread tells the hardware to execute a command, it should call the **IOScheduleFunc()** function to arrange for a specified timeout function to be called at a certain time in the future. If the timeout function is called, it sends a Mach message with the ID IO_TIMEOUT_MSG to the I/O thread.

The **timeoutOccurred** method is invoked by the I/O thread if it receives a message

with the ID IO_TIMEOUT_MSG. Your implementation of this method should abort pending commands and reset the SCSI bus.

Other Considerations

You need to consider a few other issues in implementing a SCSI driver.

Sending Messages to the I/O Thread

During initialization, get the I/O thread's interrupt port:

```
port = [self interruptPort];
```

Also get the port's name in the kernel's IPC (inter process communication) name space:

```
ioTaskPort = IOConvertPort(port, IO_KernelIOTask, IO_Kernel);
```

Use the **msg_send_from_kernel()** function to actually send a message from the timeout function or from **executeRequest:buffer:client:** to the I/O thread. You can't use **msg_send()** because when a driver executes outside the I/O task, it no longer has send rights to ports that it had in the I/O task. The same applies to any method or function that you specified in a call to **IOScheduleFunc()**.

Alignment

To specify the buffer allocation alignment restrictions that apply to your driver, all you need to do is implement the IOSCSIControllerExported protocol's method **getDMAAlignment**, which returns the DMA alignment requirements for the current architecture. This method must fill in all four fields of an IODMAAlignment structure that indicates buffer starting points and total length for reading and writing.

Client drivers can use **getDMAAlignment** to obtain alignment requirements. They can then use the **IOAlign()** macro to determine how much memory they really need to allocate. These drivers should do the allocation with **allocateBufferofLength:actualStart:actualLength:** that allocates well-aligned memory, which is required for calls to **executeRequest:buffer:client:**.

Mapping Virtual Memory

This is generally not a concern unless the driver itself must touch data, such as in programmed I/O. In these cases, use **IOPhysicalFromVirtual()** to get the physical address of the desired data. Of course, there's no guarantee that you can access every

physical address—you only get a valid physical address if the memory is wired down.

Use **IOMapPhysicalIntoIOTask()** to create a virtual address in the IOTask's virtual address space. Deallocate this virtual memory by calling **IOUnmapPhysicalFromIOTask()**.

Maximum Data Transfer

If you implement the method **maxTransfer**, it may simplify your design. Upper layers can use the value returned by this method to determine the maximum data transfer size your driver can handle. They won't try to send commands that attempt to transfer more data than the driver can handle.

Statistics

A suite of methods such as **maxQueueLength** are available to return statistics used by **iostat** and other commands. The example located in **/NextDeveloper/Examples/DriverKit/Adaptec1542B** illustrates gathering these statistics.

SCSI Peripheral Drivers

To write a SCSI peripheral device driver, create a subclass of IODevice. Use the methods in the IOSCSIControllerExported protocol to allow the SCSI peripheral driver object to talk to the SCSI controller object. Some of this protocol's key methods include:

- **executeRequest:buffer:client:**, which sends SCSI commands to a peripheral device.
- **getDMAAlignment:**, which returns DMA alignment requirements.
- **allocateBufferOfLength:actualStart:actualLength:**, which allocates and returns a pointer to well-aligned memory, required for invoking **executeRequest:buffer:client:**. It's used with other alignment functions such as **IOAlign()** and **getDMAAlignment:**.
- **reserveTarget:lun:forOwner:** and **releaseTarget:lun:forOwner:**, which respectively reserve and release a specified target/lun pair.
- **resetSCSIBus**, which resets the SCSI bus.

Implement the **probe:** method to get the **id** of the SCSI controller object from the IODeviceDescription object that's handed to **probe:** as its parameter. In addition, **probe:** may send a SCSI INQUIRY command to each target/lun pair on its controller to see if a peripheral supported by the driver is connected to the SCSI bus. For every

peripheral it finds, **probe:** should instantiate a SCSI peripheral driver object.

For an example of a SCSI tape drive controller, see **/NextDeveloper/Examples/DriverKit/SCSITape**.

Sound Devices

To write a driver for a sound device, create a subclass of IOAudio. See the IOAudio Class description for additional information on writing a driver.

Directories in **/NextDeveloper/Examples/DriverKit** with examples of sound drivers include **ProAudioSpectrum16** and **SoundBlaster8**.

Development Requirements

The following hardware is required or recommended for development and support efforts:

- At least one workstation with NEXTSTEP User and Developer software
- A second NEXTSTEP workstation (optional, but recommended. One can serve as debug master, the other slave. This allows source debugging of the loaded driver.)
- Sound card
- Microphone, headphones or amplifier, and speakers that are all known to work with the sound card
- Logic analyzer

Basic Operations

Here are the basic operations needed for an audio driver:

- Instantiating and initializing a driver object
- Starting and stopping data transfers
- Handling interrupts
- Determining supported features
- Changing hardware settings such as volume

Instantiating and Initializing a Driver Object

Override **probe:** to allocate an instance of the driver and initialize it by invoking

IOAudio's **initFromDeviceDescription:** method.

Override **initFromDeviceDescription:** method and invoke super's implementation. IOAudio's **initFromDeviceDescription:** method invokes the **reset** method, which you must implement to check whether hardware is present. If hardware is present, the method should set it to a known state. It should also configure the host DMA channel to auto initialize mode if the sound card supports it. (Otherwise, you'll have to restart the DMA transfer every time you handle an interrupt.) It should return **nil** if the hardware isn't present.

If **initFromDeviceDescription:** returns **nil**, **probe:** shouldn't allocate a driver instance and should return NO.

Starting and Stopping Data Transfers

Override IOAudio's **startDMAForchannel:read:buffer:bufferSizeForInterrupts:** method in your driver. Your method should do the following:

- Configure your audio hardware to use the selected sample rate, data encoding, and channel count.
- Set audio hardware to auto initialize mode, if possible.
- Start the audio hardware's data transfer engine.
- Enable interrupts and start the host master DMA.
- Invoke IODirectDevice's **startDMAForBuffer:channel:** (part of the kernel), which you've configured to start the DMA on a selected channel.

Note: **startDMAForchannel:read:buffer:bufferSizeForInterrupts:** must be called only from the I/O thread.

Override IOAudio's **stopDMAForChannel:read:** method in your subclass to perform these operations:

- Disable interrupts.
- Turn off the DMA channel.
- Stop any data transfer from the audio hardware.

Handling Interrupts

The Driver Kit already implements an interrupt handler for sound. You must implement the method **interruptOccurredForInput:forOutput:** to take these actions:

- Determine which, if any, channel interrupted and perform the necessary actions to acknowledge the interrupt.
- Return BOOL values in each of the method's two BOOL parameters: YES if there is data in the corresponding channel and NO otherwise.

Note: The **interruptOccurredForInput:forOutput:** method must be called only from the I/O thread.

Write a function that clears audio hardware interrupts and implement **interruptClearFunc** to return the address of this function. This function is called by the interrupt handler when there's an audio interrupt, so it can't block.

Determining Supported Features

Implement the following methods to provide the following feature information:

- **acceptsContinuousSamplingRates** to return whether continuous sampling rates is supported
- **channelCountLimit** to return 1 for mono or 2 for stereo
- **getDataEncodings:count:** to return an array of supported data encodings
- **getInputChannelBuffer:size:** to return the input channel's buffer address and size
- **getOuputChannelBuffer:size:** to return the output channel's buffer address and size
- **getSamplingRates:count:** to return supported sampling rates in an array and a count of the number of rates supported
- **getSamplingRatesLow:high:** to return the lowest and highest sampling rates supported

Setting Hardware State

The user can set various audio parameters. IOAudio has a set of methods that return the values set by the user. You implement an accompanying set of methods to convert these user values to values your hardware understands by scaling the values appropriately and updating the hardware state to the scaled values. Implement the methods if the audio hardware supports the corresponding features. IOAudio provides the following methods to get the user value and update the associated hardware state:

- **inputGainLeft** and **updateInputGainLeft**
- **inputGainRight** and **updateInputGainRight**
- **isLoudnessEnhanced** and **updateLoudnessEnhanced**
- **isOutputMuted** and **updateOutputMute**
- **outputAttenuationLeft** and **updateOutputAttenuationLeft**
- **outputAttenuationRight** and **updateOutputAttenuationRight**

Input gain runs from 0 (no sound) to 32767 (maximum); attenuation goes from –84 (no sound) to 0 (maximum).

Note: IOAudio invokes all the **update...** methods from the I/O thread.

Caveat

IOAudio's support for audio drivers has the following limitation you should know about:

> You can't override the methods (**dataEncoding** and **getDataEncodings:count:**, for example) that interpret NXSoundParameterTags passed from user-level programs. Consequently, you have to use some other way to provide support for device-specific features such as on-board compression.

Suggestions for Development

If the audio hardware supports a superset of a well-known interface, consider developing it first. It's even better if a template is available. Then add features specific to your audio hardware.

When you start debugging, first try to get an interrupt. When you do, you know data transfers are occurring.

As a debugging aid, consider writing a user-level program to use IODeviceMaster to read and write ports.

4

Building, Configuring, and Debugging Drivers

This chapter tells you how to integrate your Driver Kit driver with the rest of the system. It first describes building the driver using Project Builder. It tells how to set up the initial configuration files and set the configuration parameters with the Configure application. Finally, it highlights some of the debugging aids available for finding driver bugs and tracing your driver's execution. Consult the other sources mentioned for in-depth information about the tools.

Also see Chapter 9, "Building, Loading, and Debugging Loadable Kernel Servers" in *NEXTSTEP Operating System Software* for details on that topic. Look at **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** for an example of building, loading, and debugging a driver.

Driver Bundles

To load your driver into the kernel—even if only for testing—you need to create a *driver bundle* for it with Project Builder. A driver bundle contains all the files needed to load and configure a driver: Its relocatable code and configuration information. A bundle may also contain help information and a configuration inspector for Configure to access configuration data. A driver bundle is also called a *config bundle* because it contains configuration information for the driver and typically has the name *Driver***.config**, where *Driver* is the driver's name.

The driver name should be of the form

<vendor><model><type>Driver

The driver name *Adaptec1542SCSIDriver* follows this form.

Bundle Locations

Driver bundles for each system device—like the mouse, display, network card, SCSI devices, and so on—reside in a special directory called **/NextLibrary/Devices**. The bundles for each type of device are called *Driver***.config**, where *Driver* is a type of device or a device name. In addition, every system has a bundle called **System.config** that configures the whole system.

An average system's directory **/NextLibrary/Devices** might contain the following directories, each of which is a bundle for a specific device:

ATI.config
Adaptec1542B.config
Beep.config
BusMouse.config
CirrusLogicGD542X.config
CompaqAudio.config
DPT2012.config
EtherExpress16.config
EtherLink3.config
Floppy.config
IDE.config
IntelGXProAudio.config
JAWS.config
MSWSoundSystem.config
PS2Keyboard.config
PS2Mouse.config
ParallelPort.config
ProAudioSpectrum.config
QVision.config
S3.config
SCSITape.config
SMC16.config
SerialMouse.config
SerialPorts.config
System.config
TokenExpress.config
TsengLabsET4000.config
VGA.config
Wingine.config

/NextLibrary/Devices is a link to the **/private/Devices** directory, which is a link to the driver directory for the current architecture (for example, **/private/Drivers/i386**). This link is always valid.

What's in a Bundle

Each driver bundle (including **System.config**) can contain the following files and directories:

Default.table
Instance*n***.table** (created by Configure)
*x***.table**
Display.modes
*x***.modes**
CustomInspector (optional binary)

Language.**lproj/**
CustomInspector.**nib** (optional)
Localizable.strings
Help/ (replaces **Info.rtf**)
*Driver*_**reloc** (omitted for NeXT drivers that are compiled into the kernel)
Pre-Load
Post-Load

Default.table is a commented, read-only file that gives the default configuration settings for a generic device. Configure uses **Default.table** to build **Instance***n*.**table** files, which contain specific configuration information for each device you have. There may be other *x*.**table** files, each expressing a different possible instance of the driver.

Each **.table** file is the ASCII representation of an NXStringTable object. Drivers and nondrivers can get access to these tables by using the IOConfigTable class. In addition, Driver Kit classes automatically interpret and use some of the standard keys in these tables.

Direct drivers have one **Instance***n*.**table** for each device. For example, if you have two of the same card, Configure makes two files called **Instance0.table** and **Instance1.table** in the card's bundle. Indirect drivers and the system bundle have only one file, called **Instance0.table**.

Note: Because Configure's default device inspector has no way of knowing whether a device is direct or indirect, it can create more than one **Instance***n*.**table** for an indirect driver. The consequence is that the driver's **probe:** method gets invoked more than once for each direct driver it might want to attach to. To get around this, you should either write your own device inspector or ensure that your driver's **probe:** method can handle more than one probe per direct driver.

The **Display.mode** and *x*.**mode** files hold display mode information. Default information is in **Display.mode**, and *x*.**mode** holds the information for other instances of the driver (just as *x*.**table** expresses configuration information for other driver instances).

For each language, **Localizable.strings** contains the text strings that applications display about the device. For example, it includes the name of the device as it appears in the list of devices in Configure. The **Help/** directory contains files to inform the user about the driver and help them use it.

The *Driver*_**reloc** file is the relocatable object file of the device driver. The *CustomInspector* binary is the executable file for the Inspector panel; its name is the same as the bundle name (without the **.config** suffix). *CustomInspector*.**nib** is the nib file for the Inspector panel.

The bundle may contain *Pre-Load* and/or *Post-Load* programs that are run before

and/or after the driver is loaded.

Configuration Tables

Files with a **.table** suffix contain strings of key/value pairs that describe a configuration. See “Configuration Keys” in the Appendix for information on what these tables should contain.

You can use the **Default.table** of an existing driver as a starting point for a configuration. Later, you should let the Configure application (with your custom inspector, if any) create the **Instance*n*.table** files.

Here’s a sample **Instance*n*.table** for a parallel port driver:

```
"Driver Name"  = "IOParallelPort";
"Title"   = "System Parallel";
"Location"   = "System Baseboard";
"Family"   = "Parallel";
"Version"   = "1.0";
"Server Name"  = "ParallelPort";
"Path 0"   = "/dev/pp0";
"Post-Load"   = "InstallPPDev";
"Memory Maps"  = "";
"Pre-Load"   = "RemovePPDev";
"DMA Channels"  = "";
"Minor Device Number" = "0";
"Valid IRQ Levels"  = "7";
"I/O Ports"   = "0x378-0x37f";
"Instance"   = "0";
"Port Count"   = "1";
"IRQ Levels"   = "7";
```

Warning: C-style comment delimiters (that is, /* */) aren’t recognized in configuration tables, such as **Default.table** or **Instance0.table**. Anything inside the delimiters will be parsed along with the rest of the file. This means that, for example, if you are testing a driver under development, you can’t remove a key-value pair by simply commenting it out.

Other Configuration Tables

A bundle may also contain other configuration tables of the form *x***.table**, where *x* is a prefix such as “PCI”. Each of these is a table like **default.table** but expresses a possible instance of the driver with a slightly different “personality” than **default.table**. For example, **PCI.table** might be identical to **Default.table** except that it contains a line specifying a PCI-compliant driver:

```
"Bus Type" = "PCI";
```

By convention, **Default.table** specifies an ISA or VL-bus compliant driver—the simplest case. The prefix *x* in *x***.table** usually designates the bus type.

These configuration table files should contain all information appropriate for the bus type. PCI-compliant drivers, for instance, contain a line specifying the auto detect IDs, such as this:

```
"Auto Detect IDs" = "0x71789004 0x0e111234";
```

Custom Device Inspector Files

For initial testing, you probably don't need a custom inspector. Instead, you can put the appropriate values directly into your test **Default.table** or **Instance*n*.table** files.

If you create a custom inspector, you should put the executable file and nib file in the places described in "What's in a Bundle," earlier in this chapter. Project Builder does this for you automatically. See "Writing a Custom Inspector" later in this chapter for information on creating custom inspectors.

Note: Project Builder creates an Inspector Panel executable file in the bundle and gives it the same name as the bundle (without the **.config** suffix). This executable loads the default inspector.

Localizable Strings File

This file should contain any strings you add to your Configure inspector's user interface, plus the following strings:

```
"Driver" = "UltimateTech XYZ-12";
  "Long Name" = "Ultimate Technologies XYZ-12 Transmogrifier";
```

where *Driver* is the name of the bundle (minus the **.config** suffix). Configure uses the string associated with the *Driver* key ("UltimateTech XYZ-12") whenever space is tight. When Configure has more space to display the driver's name, it uses the string associated with the "Long Name" key.

Display Mode Tables

If your driver is a display driver that supports multiple display modes, you need to specify which modes the user can choose. This information is supplied in the **Display.modes** file. Here's a sample file:

```
"Height: 600 Width: 800 Refresh: 60Hz ColorSpace: RGB:555/16";
"Height: 600 Width: 800 Refresh: 72Hz ColorSpace: RGB:555/16";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 66Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 76Hz ColorSpace: RGB:256/8";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 66Hz ColorSpace: BW:8";
```

```
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 76Hz ColorSpace: BW:8";
"Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:555/16";
"Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:555/16";
"Height:1024 Width:1280 Refresh: 68Hz ColorSpace: RGB:256/8";
"Height:1024 Width:1280 Refresh: 68Hz ColorSpace: BW:8";
"Height: 400 Width: 640 Refresh: 60Hz ColorSpace: RGB:888/32";
"Height: 400 Width: 640 Refresh: 70Hz ColorSpace: RGB:888/32";
"Height: 480 Width: 640 Refresh: 60Hz ColorSpace: RGB:888/32";
```

If your driver has more than one “personality,” specify alternate display information in *x***.modes** files where *x* is the appropriate prefix such as “PCI”.

See the specification for the IODisplayInspector, IOFrameBufferDisplay, and IOSVGADisplay classes for more information on display modes.

Help Directory

This directory contains the help files supported by the NeXT help facility. You add this directory to your project with Project Builder’s Add Help Directory command. For more information on adding help to your driver, see “Attaching Help to Objects” in Chapter 3, “The Interface Builder Application” of *NEXTSTEP Development Tools and Techniques*.

The **Help** directory replaces the **Info.rtf** file, formerly used to provide information about the driver.

Driver Relocatable Code

This file contains the driver’s relocatable code. An example of building a driver relocatable object file is located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver**.

Pre- and Post-Load Programs

Your driver may require some action to be taken before and/or after it is loaded. For instance, you may want to run a program after the driver is loaded to look up its major device number and create a device node for the driver. Use the “Pre-Load” configuration key to specify a program that will run prior to your driver being loaded; use the “Post-Load” key to specify a program that runs after the driver is loaded.

The System Configuration Bundle

The **System.config** bundle is special in several ways. Its **Instance0.table** has default

configuration information for the system as a whole. For example, it specifies which device drivers to load at boot time (“Boot Drivers”) and which to load later (“Active Drivers”). Here’s a sample **Default. table** from a **System.config** bundle:

```
"Version" = "2.0";
"Boot Drivers" = "PS2Keyboard PS2Mouse BusMouse Adaptec1542B
DPT2012 IDE Floppy VGA";
"Active Drivers" = "SerialPorts SerialMouse ParallelPort";
"Kernel" = "mach_kernel";
"Kernel Flags" = "";
"Boot Graphics" = "No";
```

For writers of Driver Kit drivers, “Active Drivers” and “Boot Drivers” are the most important keywords. They specify which drivers are automatically loaded into the system the next time it’s started. When someone uses Configure to add a device that has a loadable driver, the driver is added to one of these two lists. See the “Boot Drivers” and “Active Drivers” keys in the “Configuration Keys” section of the Appendix to see how to specify which list a driver should be in. This section also lists the other keywords for the system configuration table.

Note: Changes to system configuration information don’t take effect until the system is restarted. However, you can load a driver without rebooting by using the **d** option of **driverLoader** (documented in “Loading a Driver with driverLoader” later in this chapter).

Creating a Driver Bundle

Create a project for your driver with Project Builder, and give the project the name you want your driver to have. Copy your driver files into the project by dragging them into the appropriate suitcase (header files to the Header suitcase and so on) or by using the Add command in the Files menu. Switch to the Builder view in the project window and select “bundle” as the Target. Click the Build button. Project Builder builds the driver and puts it in a driver bundle called *Driver***.config** where *Driver* is the name you chose for the driver. Now you can configure and load the driver.

See *NEXTSTEP Development Tools and Techniques* for more information about using Project Builder. The example in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** shows building a bundle with Project Builder.

Configuring Drivers

After you have built your driver, you need to configure it with the Configure application.

Configure Application

You can configure devices and add drivers with the Configure application. When you select a device, Configure loads the device's inspector, which provides a user interface for manipulating the device configuration (choosing its DMA channels, for example). If you don't supply a device inspector for your driver, Configure uses a default device inspector. See the IODeviceInspector class and IOConfiguration protocol specifications for more information on device inspectors.

The Configure application reads the key/value pairs from a driver bundle's **Default.table** and displays them in a Configuration Inspector Panel. The user interface allows the user to change the displayed parameters and warns of possible value conflicts. When the user finishes modifying the configuration, Configure writes the updated configuration to the indicated **Instance*n*.table** and configures the driver based on the information in the configuration and kernel tables.

When the system starts up, the kernel uses an IOConfigTable object to parse the configuration information in the **Instance*n*.table**. From this information, the kernel instantiates an IODeviceDescription object, which encapsulates information about the driver. The kernel passes the IODeviceDescription object as the parameter to the **probe:** method, which instantiates the driver object based on this information.

There's a list of standard key/value configuration pairs in the "Configuration Keys" section in the Appendix.

How Configuring Kernel-Level Driver Kit Drivers Differs from Configuring Other Loadable Kernel Servers

The configuration of Driver Kit kernel-level drivers differs from that of other Loadable Kernel Servers (LKSs) in the following ways:

- Each Driver Kit driver has its own configuration directory under **/NextLibrary/Devices**. Other LKSs have no standard way of getting configuration information.
- With the Configure application, users can add Driver Kit drivers to the system, as well as specify configuration information for each driver. Other LKSs are generally added to the system by adding a line to **/etc/kern_loader.conf**.
- Driver Kit drivers are allocated and loaded with the **driverLoader** command, which uses the information in the driver's configuration directory. You can load an LKS with the kernel-server utility, **kl_util**, but it doesn't cause the driver to be probed.

- Driver Kit drivers can't currently be unloaded, unlike other LKSs. For example, if you want to change a driver that's already running, you must restart the system to be able to load the new driver.

Writing a Custom Inspector

The Configure application uses inspectors to configure a driver. With the default inspector in Configure, you can configure values that belong to the standard set of keys with no further implementation effort. If you've added custom parameters, however, you need to implement a custom inspector to view and modify them.

You have two choices in implementing a custom inspector:

- Add an accessory view to the inspector, with an 80-pixel height limit.
- Replace the standard inspector completely. You're still limited to a 640×480 view. However, you can use a button to display a panel if you run out of space.

You implement an inspector by creating a subclass of IODeviceInspector. For example, you can create a subclass of IODisplayInspector (a subclass of IODeviceInspector) to implement a display inspector. For an example, study the inspector in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/DriverInspector**.

Other classes relevant to creating an inspector include IOAddressRanger, IODeviceDescription, IODeviceMaster, and IOEISADeviceDescription. Some of these classes adopt the IOConfigurationInspector protocol.

Creating an Inspector

Override the following methods in the IODeviceInspector class and the IOConfigurationInspector protocol:

- **init**. Find and load the nib file that contains the accessory view using the bundle for your inspector. Initialize the user interface and find your driver.
- **inspectionView**. Override this if you're replacing the standard inspector.
- **setTable:**. Invoke the superclass's implementation:

  ```
  [super setTable:]
  ```

  Invoke **setAccessoryView:** to specify and initialize the accessory View. Initialize

the user interface settings from the table being inspected.

- **resourcesChanged:**. Update the user interface in response to resources being chosen or dropped in the inspector.

Modifying Custom Parameters

Implement a set of target/action methods to change the custom parameters. The user interface elements of the inspector invokes these methods. Convert the new parameter state to an appropriate string value for display, and insert it into the inspected table with **insertKey:value:**. The key must be a unique string, and you can use the **NXUniqueString()** function to generate a unique key based on the string argument. The value should be a copy—use **NXCopyStringBuffer()** to copy it:

```
[table insertKey:key value:NXCopyStringBuffer(value)];
```

Changing Driver Parameters with IODeviceMaster

Besides Configure, another way to change parameters associated with a driver is through the IODeviceMaster class, which provides access to a driver instance. First, find your driver using the **lookUpByDeviceName:objectNumber:deviceKind:** method. Then manipulate parameters associated with that instance with these methods:

- **getCharValues:forParameter:objectNumber:count:**
- **setCharValues:forParameter:objectNumber:count:**
- **getIntValues:forParameter:objectNumber:count:**
- **setIntValues:forParameter:objectNumber:count:**

Active driver values should be displayed in the user interface—even if they differ from the current configuration table values. If you want the values you change to persist beyond the time the system is powered off or restarted, you must write them to the configuration table.

Loading a Driver with driverLoader

You can load your driver into an already running system. The **driverLoader** command loads or configures a driver after startup time. You initiate the command as follows (as superuser):

/usr/etc/driverLoader *option* [**v**] [*instance*]

Specifying **v** results in more verbose output from **driverLoader**. The *instance* argument can be used only with the **d** option, as described below.

The *option* is one of the following:

a	Configure all devices. This option is used when **driverLoader** is run during system boot (by **/etc/rc**).
i	Interactive mode. With this option, you can look at all active and boot drivers in the system configuration. Note that if you add a driver to the system, the driver isn't recognized as "active" until you reboot.
d=*deviceName*	Configure one device interactively. This is how you load drivers that aren't specified in the system configuration. This is usually used for testing purposes. You can specify *instance* to use a specific **Instance*n*.table** file. For example, if you specify *instance* as 1, the driver is probed using the information in its **Instance1.table** file.

Here's an example of using the **d** option:

```
# /usr/etc/driverLoader d=myDriver
```

Here's an example of using the **d** option and specifying *instance*:

```
# /usr/etc/driverLoader d=fooDriver 1
```

For another example of using **driverLoader**, see **/NextLibrary/Documentation/NextDev/Examples/DriverKit**.

Recovering from a Bad Configuration

If you can't restart your system because of a bad configuration or because of bugs in your driver, try restarting with a default configuration. To do this, type the following at the **boot:** prompt when the system starts:

```
boot: config=Default
```

This causes the boot program to use **Default.table** in **System.config** as the system configuration, which usually works. Once you've started up, log in as **me** or **root** and use Configure to fix the rest of the configuration.

If you still can't start the system, try starting in single-user mode and editing the bundles by hand. This is risky since the configuring process has many "rules of

thumb," and you might not know all the effects of a change. To restart in single-user mode, type the following at the **boot:** prompt after you restart:

```
boot: mach_kernel -s config=Default
```

You can then use a single-user mode editor (such as **vi** or **emacs**) to edit the configuration bundles.

Debugging a Driver

You have two choices for creating debugging messages: the **IOLog()** function and the Driver Debugging Module (DDM). Most drivers just use **IOLog()** until a need arises for the more powerful and complex DDM functions.

Another debugging tool, **gdb**, is described in *NEXTSTEP Development Tools and Techniques.* You can run the driver with **gdb** from Project Builder—the example located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit/TestDriver** shows how to do this. *NEXTSTEP Development Tools and Techniques* also describes Project Builder.

Using the IOLog Function

Using **IOLog()** is similar to using **printf()** to print error or debug messages. You can output strings and parameters, just as for **printf()**. One difference is that output is placed in the **/usr/adm/messages** file instead of the console window. Place a call to **IOLog()** anywhere in your driver where you want to get information about the driver state—or to indicate that the driver reached that point during execution.

IOLog() is useful both for status messages and as a basic debugging tool. Although **IOLog()** is useful for debugging, it can affect the timing of the driver. When timing is important, you should use DDM instead.

See "Functions" in Chapter 5, "Driver Kit Reference", for more information about **IOLog()**.

Using the Driver Debugging Module (DDM)

The Driver Debugging Module (DDM) provides support for viewing debugging information without disturbing the timing of the kernel. By using the DDMViewer

application (in **/NextDeveloper/Demos**), you can specify which information should be stored in the event buffer and display debugging information from this buffer.

The core of DDM is a circular event buffer that stores the debugging information sent to it by drivers. Each entry in the buffer is timestamped (to the microsecond) and consists of a **printf**-style format string and up to five arguments associated with the format string. A call to the function that timestamps and stores one entry takes about 10 microseconds.

Gathering DDM Events

The function **IOAddDDMEntry()** adds an event to the DDM buffer. An event consists of a character string and several integer values. The **IODEBUG()** macro is provided to call **IOAddDDMEntry()**: A driver typically doesn't call **IOAddDDMEntry()** directly. Instead, the driver should define its own macros using the **IODEBUG()** macro, as in this example:

```
#define ddm_exp(x, a, b, c, d, e)      \
   IODEBUG(A7770_DDM_INDEX, DDM_EXPORTED, x, a, b, c, d, e)
#define ddm_him(x, a, b, c, d, e)      \
    IODEBUG(A7770_DDM_INDEX, DDM_HIM, x, a, b, c, d, e)
```

These macros can then be called like this:

```
ddm_him("abort_channel chan %d\n", channel, 2,3,4,5);

ddm_him("scb_int_preempt: scb 0x%x index %d haStat %s\n",
   scb_ptr, scb_index,
   IOFindNameForValue(compstat, scbHaStatValues),
   4,5);
```

A word of mask bits controls the collection of DDM entries. All calls to **IODEBUG()** don't add data to DDM's circular buffer—only those events whose mask bits are enabled are added. The mask bits are enabled and disabled by a user-level tool like DDMViewer. A driver isn't (and shouldn't be) concerned about which mask bits are enabled. Typically you turn on one or two bits of the mask word to study the trace information for a particular module.

See the SCSI example driver in **/NextDeveloper/Examples/DriverKit/Adaptec1542B**, which illustrates all aspects of using DDM.

Viewing DDM Events with DDMViewer

You can examine DDM traces at the user-level with the DDMViewer application, which is located in **/NextDeveloper/Demos**. You can also specify DDM mask bits with this application. DDMViewer can be run on any computer running NEXTSTEP, not just the machine being tested.

The DDMViewer window contains the following controls:

- **Device Name field**. Enter the name of the target to which you want to attach. The name is determined by the driver.
- **Host Name field**. Enter the name of the host on which the target is running. Leave it empty if you are debugging a driver or kernel on the current machine.
- **List button**. Click this button to start and stop the display of DDM entries. Entries are displayed starting from the last event in time and scrolling backward.
- **Set Mask button**. Click this button to send the mask defined in the Mask window (see below) to the target.
- **Disable button**. Click this button to freeze the state of the DDM buffer at the target. Click again to reenable.
- **Clear Window button**. Click this button to clear the display area.
- **Clear Buffer button**. Click this button to clear the target's circular DDM buffer.

You can specify the value of the DDM mask bits by name if you open a **.ddm** file that specifies the names of the mask bits. You create **.ddm** files with an editor such as Edit. Here's an example of a **.ddm** file:

```
#
#  DDMViewer data file for kernel devices.
#
Index : 0 : "Kernel Devices"
#
#  Common fields.
#
0x0001 : "Device Object"
0x0002 : "Disk Object"
0x0004 : "Net"
0x0020 : "DMA"
#
#  SCSI.
#
0x0100 : "SCSI Control"
0x0400 : "SCSI Disk"
```

Comments start with "#". The line that starts with "Index" defines which DDM Mask word is being defined (there are currently four mask words). The Index line also defines the name of the window associated with this set of mask bits. All other lines define one bit in the mask word, specifying the value of the bit and an ASCII name equivalent. The SCSI example driver in **/NextDeveloper/Examples/DriverKit/Adaptec1542B** has a sample **.ddm** file.

5

Driver Kit Reference

Library: Configure application API has no library
Other user-level API is in **libDriver.a**
Kernel-level API has no library

Header File Directory: /NextDeveloper/Headers/driverkit

This chapter documents the Driver Kit's API—public classes, protocols, functions, and types and constants. The "Other Features" section describes such features as device auto detection.

Warning: You should avoid using an undocumented API, since it's subject to change. For example, if a method is in a class header file but not in the class documentation, the method is likely to change or disappear in future releases.

Functions

This section describes three types of functions and macros:

- General-purpose functions—to perform basic tasks
- Driver Debugging Module (DDM) functions—to help all drivers keep debugging information
- Miscellaneous functions—such as DMA alignment macros, functions that work only in the kernel, and functions specific to a particular machine architecture.

Unless noted otherwise, all of the functions described in this section work in both user-level and kernel-level drivers.

Other Functions Available to Drivers

Almost all Mach functions are available to kernel-level device drivers. If you don't find the appropriate functionality in a method or function, you can use a Mach function. For example, **port_allocate()** and **msg_send()** are used by many drivers.

Note: Instead of including the header file **mach/mach.h**, you must include **mach/mach_user_internal.h** and **mach/mach_interface.h**.

The **host_priv_self()** Mach function does *not* work in the kernel. You should use **IOHostPrivSelf()** instead.

General-Purpose Functions

The general-purpose functions, defined in the header file **driverkit/generalFuncs.h**, provide a consistent interface for device drivers that may have to run in kernel space at one time (or in one configuration) and in user space at another time. Using these functions minimizes the work or porting between the two environments. All the Driver Kit classes, as well all NeXT kernel-level drivers that use the Driver Kit, were written using these functions so that they have one set of source files with minimal kernel and user mode differences.

Warning: Before using any of the general-purpose functions, each user-level driver must call

IOInitGeneralFuncs(). (Kernel-level drivers don't need to call it.)

Thread Functions

These functions provide the functionality of the C-thread functions in a uniform way in both user and kernel space.

IOForkThread()
IOSuspendThread()
IOResumeThread()
IOExitThread()

Timer Functions

IOSleep()
IODelay()
IOScheduleFunc()
IOUnscheduleFunc()
IOGetTimestamp()

Memory Allocation and Copying Functions

IOCopyMemory()
IOMalloc()
IOFree()

Miscellaneous General-Purpose Functions

IOInitGeneralFuncs()
IOFindNameForValue()
IOFindValueForName()
IOLog()
IOPanic()

Driver Debugging Module (DDM) Functions

See the "Adding Debugging Code" section in Chapter 2 for information on using the DDM.

IOAddDDMEntry()
IOClearDDM()

IOCopyString()
IODEBUG()
IOGetDDMEntry()
IOGetDDMMask()
IOInitDDM()
IONsTimeFromDDMMsg()
IOSetDDMMask()

Miscellaneous Functions

Kernel-Only Functions

The function **IOConvertPort()** is necessary for some kernel-level drivers—and not for user-level drivers—because kernel-level drivers can execute in more than one task. The first thread of a kernel-level driver executes in the loadable kernel server's task, any threads that the driver creates execute in the kernel I/O task, and network drivers and drivers with UNIX entry points (at some stage) can execute in the context of an unknown task.

IOGetObjectForDeviceName() provides to kernel-level drivers some of the functionality provided to user-level programs by IODeviceMaster. Similarly, **IOHostPrivSelf()** is used by some kernel-level drivers that need the information normally returned by **host_priv_self()** (which is one of the few Mach functions that doesn't work in the kernel).

The function **IOVmTaskSelf()** supplies a **vm_task_t** for Mach function calls that expect one for the kernel; this is necessary because **vm_task_t** and **task_t** aren't the same in the kernel (as they are at user level). **IOVmTaskCurrent()** supplies a **vm_task_t** that's needed by some UNIX-style drivers. Finally, **IOVmTaskForBuf()** supplies a **vm_task_t** for the unknown task that is requesting UNIX-style I/O.

IOConvertPort()
IOGetObjectForDeviceName()
IOHostPrivSelf()
IOPhysicalFromVirtual()
IOSetUNIXError()
IOVmTaskCurrent()
IOVmTaskForBuf()
IOVmTaskSelf()

DMA Alignment Macros

IOAlign()
IOIsAligned()

Architecture-Specific Functions

The following functions are used by some Intel drivers to read and write I/O ports:

inb()
inw()
inl()
outb()
outw()
outl()

Some Intel drivers use the following function to help handle interrupts:

IODisableInterrupt()
IOEnableInterrupt()
IOSendInterrupt()

Some Intel devices require memory in the low 16 MB:

IOMallocLow()

Intel display drivers often use the following functions to read and write VGA registers:

IOReadRegister()
IOReadModifyWriteRegister()
IOWriteRegister()

inb(), inw(), inl(), outb(), outw(), outl()

SUMMARY

Read or write data to an I/O port

DECLARED IN

driverkit/i386/ioPorts.h

SYNOPSIS

unsigned char **inb(**unsigned int *address***)**
unsigned short **inw(**unsigned int *address***)**
unsigned long **inl(**unsigned int *address***)**
void **outb(**unsigned int *address*, unsigned char *data***)**
void **outw(**unsigned int *address*, unsigned short *data***)**
void **outl(**unsigned int *address*, unsigned long *data***)**

DESCRIPTION

These inline functions let drivers read and write I/O ports on Intel-based computers. Use **inb()** to read a byte at the I/O port *address*. Use **inw()** to read the two bytes at *address* and *address* + 1, and **inl()** to read four bytes starting at *address*. To write a byte, use **outb()**; to write two bytes (to *address* and *address* + 1), use **outw()**; to write four bytes, use **outl()**.

These functions have nothing to do with main memory; they work only for the 64 kilobytes of I/O address space on an Intel-based computer. These functions use the special machine instructions that are necessary for reading and writing data from and to the I/O space.

Note: These functions work only at kernel level and only on Intel-based computers.

EXAMPLE

```
temp_cr = inb(base+CR); /* get current CR value */
```

IOAddDDMEntry()

SUMMARY

Add one entry to the Driver Debugging Module

DECLARED IN

driverkit/debugging.h

SYNOPSIS

void **IOAddDDMEntry(**char ***format*, int *arg1*, int *arg2*, int *arg3*, int *arg4*, int *arg5***)**

DESCRIPTION

This is the exported function that is used to add events to the DDM's circular buffer. However, drivers typically don't use this directly; instead, they should use macros that call **IOAddDDMEntry()** conditionally based on the current state of debugging flags. See the description of **IODEBUG()** for examples.

Note: The last 5 arguments to this function are typed above as **int**, but they are really untyped and could be any 32-bit quantity. They are stored in the debugging log as **int** but are eventually evaluated as arguments to **sprintf()**, so they could be **int**, **char**, **short**, or pointers to a string. See **IOCopyString()**, later in this section, for information on passing string pointers to **IOAddDDMEntry()**.

SEE ALSO

IODEBUG()

IOAddToBdevsw(), IOAddToCdevsw(), IOAddToVfssw()

SUMMARY

Add UNIX-style entry points to a device switch table

DECLARED IN

driverkit/devsw.h

SYNOPSIS

int **IOAddToBdevsw(**IOSwitchFunc *openFunc*, IOSwitchFunc *closeFunc*, IOSwitchFunc *strategyFunc*, IOSwitchFunc *dumpFunc*, IOSwitchFunc *psizeFunc*, BOOL *isTape***)**

int **IOAddToCdevsw(**IOSwitchFunc *openFunc*, IOSwitchFunc *closeFunc*,

IOSwitchFunc *readFunc*, IOSwitchFunc *writeFunc*, IOSwitchFunc *ioctlFunc*, IOSwitchFunc *stopFunc*, IOSwitchFunc *resetFunc*, IOSwitchFunc *selectFunc*, IOSwitchFunc *mmapFunc*, IOSwitchFunc *getcFunc*, IOSwitchFunc *putcFunc***)**

int **IOAddToVfssw(**const char **vfsswName*, const struct vfsops **vfsswOps***)**

DESCRIPTION

These functions find a free row in a device switch table and add the specified entry points. Each function returns the major number (equivalent to the row number) for the device, or -1 if the device couldn't be added to the table.

Note: You should use IODevice's **addToBdevsw...** and **addToCdevsw...** methods instead of **IOAddToBdevsw()** and **IOAddToCdevsw()**, whenever possible.

SEE ALSO

IORemoveFromBdevsw(), **IORemoveFromCdevsw()**, **IORemoveFromVfssw()**

IOAlign()

SUMMARY

Truncate an address so that it's aligned to a buffer size

DECLARED IN

driverkit/align.h

SYNOPSIS

type **IOAlign(***type, address, bufferSize***)**

DESCRIPTION

This macro truncates *address* to a multiple of *bufferSize*.

SEE ALSO

IOIsAligned()

IOClearDDM()

SUMMARY

Clear the Driver Debugging Module's entries

DECLARED IN

driverkit/debugging.h

SYNOPSIS

void **IOClearDDM()**

DESCRIPTION

This function empties the DDM's circular buffer.

IOConvertPort()

SUMMARY

Convert a port name from one IPC space to another

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

port_t **IOConvertPort(**port_t *port*, IOIPCSpace *from*, IOIPCSpace *to***)**

DESCRIPTION

This function lets a kernel driver convert a port name (*port*) so that the port can be used in a different IPC space. Three types of conversion are supported:

- From the current task's IPC space to the kernel I/O task's space
- From the kernel's IPC space to the kernel I/O task's space
- From the kernel I/O task's IPC space to kernel's IPC space

The arguments *from* and *to* should each be specified as one of the following: IO_Kernel, IO_KernelIOTask, or IO_CurrentTask. For example, the following code converts a port name from the current task's name to the name used by the kernel I/O task.

```
ioTaskPort = IOConvertPort(aPort, IO_CurrentTask, IO_KernelIOTask);
```

Note: This function works only in kernel-level drivers.

RETURN

Returns the port's name in the *to* space. Specifying an invalid conversion results in a return value of PORT_NULL.

IOCopyMemory()

SUMMARY

Copy memory using the specified transfer width

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOCopyMemory(**void ***from*, void ***to*, unsigned int *numBytes*, unsigned int *bytesPerTransfer***)**

DESCRIPTION

Copies memory 1, 2, or 4 bytes at a time (as specified by *bytesPerTransfer*) until *numBytes* bytes starting at *from* have been copied to *to*. The *from* and *to* buffers must not overlap.

This function is useful when devices have mapped memory that can be accessed in only 8-bit or 16-bit quantities. In these situations, **bcopy()** isn't appropriate, since it assumes 32-bit access to all memory involved.

If *from* is not aligned on a *bytesPerTransfer* boundary, **IOCopyMemory()** performs 8-bit transfers until it has reached a *bytesPerTransfer* boundary. Similarly, if the end of the *from* buffer extends past a *bytesPerTransfer* boundary, the remaining memory is copied 8 bits at a time.

IOCopyString()

SUMMARY

Return a copy of the specified string

DECLARED IN

driverkit/debugging.h

SYNOPSIS

const char ***IOCopyString(**const char **instring***)**

DESCRIPTION

This function is required when you want to use a pointer to a string whose existence is transitory as an argument. The reason for this is that the string won't be read until the Driver Debugging Module's buffer is examined, which could be a long time (minutes or more) after the call to **IOAddDDMEntry()**. By then, the string pointer passed to **IOAddDDMEntry()** no longer might no longer point to a useful string.

Warning: The string returned by this function is created with **IOMalloc()** and is never freed. Use this function with discretion.

IODEBUG()

SUMMARY

Conditionally add one entry to the Driver Debugging Module

DECLARED IN

driverkit/debugging.h

SYNOPSIS

void **IODEBUG(**int *index*, int *mask*, char **format*, int *arg1*, int *arg2*, int *arg3*, int *arg4*, int *arg5***)**

DESCRIPTION

This macro is used to add entries to the DDM's circular buffer. The entry is added only if both of the following are true:

- The C preprocessor flag DDM_DEBUG is defined.
- A bitwise and operation performed on *mask* and **IODDMMasks[***index***]** results in a nonzero result.

IODEBUG() is typically used to define other macros specific to a driver, as shown in the following example.

EXAMPLE

```
#define MY_INDEX      0

#define MY_INPUT       0x00000001    //
#define MY_OUTPUT      0x00000002    //
#define MY_OTHER       0x00000004    //

#define logInput(x, a, b, c, d, e) \
    IODEBUG(MY_INDEX, MY_INPUT, x, a, b, c, d, e)

#define logOutput(x, a, b, c, d, e) \
    IODEBUG(MY_INDEX, MY_OUTPUT, x, a, b, c, d, e)

#define logOther(x, a, b, c, d, e) \
    IODEBUG(MY_INDEX, MY_OTHER, x, a, b, c, d, e)

. . .
IODDMMasks[MY_INDEX] = MY_INPUT | MY_OUTPUT;
. . .
logInput("Input error %d: %s\n", error, IOFindNameForValue(error,
    &errorList));
```

IODelay()

SUMMARY

Wait (without blocking) for the indicated number of microseconds

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IODelay(**unsigned int *microseconds***)**

DESCRIPTION

This is a quick, nonblocking version of **IOSleep()**.

Note: This function guarantees a *minimum* “spin” delay in the user-level version; due to thread scheduling, the call to **IODelay()** could take much longer than the indicated time. This should not be a problem with properly designed user-level drivers as this is a common real-time constraint on all user-level code.

IODisableInterrupt()

SUMMARY

Prevent interrupt messages from being sent

DECLARED IN

driverkit/IODirectDevice.h

SYNOPSIS

void **IODisableInterrupt(**void *_identity_**)**

DESCRIPTION

This function allows handlers of non-shared interrupts to indicate that the interrupt should be left disabled on return from the interrupt handler.

The *identity* argument should be set to the value that the interrupt handler received in its own arguments.

Note: IODisableInterrupt() must be called inside a special interrupt handler function. It can't be called from any other context.

SEE ALSO

IOEnableInterrupt(), IOSendInterrupt()

IOEnableInterrupt()

SUMMARY

Allow interrupt messages to be sent

DECLARED IN

driverkit/IODirectDevice.h

SYNOPSIS

void **IOEnableInterrupt(**void *_identity_**)**

DESCRIPTION

This function allows interrupt handlers to indicate that the interrupt should be reenabled on return from the interrupt handler. You should only re-enable the interrupt after removing the source of the interrupt—by clearing the interrupt status register on the device, or by using whatever mechanism is necessary for the hardware

your driver controls.

The *identity* argument should be set to the value that the interrupt handler received in its own arguments.

Note: **IOEnableInterrupt()** must be called inside a special interrupt handler function. It can't be called from any other context.

SEE ALSO

IODisableInterrupt(), IOSendInterrupt()

IOExitThread()

SUMMARY

Terminate the execution of the current thread

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

volatile void **IOExitThread()**

DESCRIPTION

This function terminates the execution of the current (calling) thread. Note that there's no way for one thread to kill another thread other than by sending some kind of message to the soon-to-be-terminated thread instructing it to kill itself.

Note: In the user-level implementation, the main C thread (the first thread in the task) doesn't exit until all other C threads in the task have exited.

IOFindNameForValue(), IOFindValueForName()

SUMMARY

Convert an integer to a string, or vice versa, using an **IONamedValues** array

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

const char *__IOFindNameForValue(__int *value*, const IONamedValues **array*__)__
IOReturn __IOFindValueForName(__const char **string*, const IONamedValue **array*, int **value*__)__

DESCRIPTION

These functions are the primary use of the **IONamedValues** data type, which maps integer values to strings. **IOFindNameForValue()** maps a given integer value to a string, given a pointer to an array of **IONamedValues**. **IOFindValueForName()** maps a given string into an integer, returning the integer in *value*.

One typical use for **IOFindNameForValue()** is to map integer return values into error strings. IODevice's **IOStringFromReturn:** method performs this function. A subclass that defines additional IOReturn values should override this method and call **[super IOReturnToString:]** if the specified value does not match one of the class-specific IOReturns.

RETURN

IOFindNameForValue() returns the string corresponding to *value*, or a string indicating that *value* is undefined if the integer wasn't found.
IOFindValueForName() returns IO_R_SUCCESS if it finds the specified string; otherwise, it returns IO_R_INVALIDARG.

IOForkThread()

SUMMARY

Start a new thread

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

IOThread __IOForkThread(__IOThreadFunc *function*, void **arg*__)__

DESCRIPTION

This function causes a new thread to be started up. For kernel-level drivers, the new thread is in the IOTask's address space; for user-level drivers, the thread is in the

current task. The thread begins execution at *function*, which is passed *arg* as its argument.

IOFree()

SUMMARY

Free memory allocated by **IOMalloc()**

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOFree(**void ***var*, int *numBytes***)**

DESCRIPTION

This function frees memory allocated by **IOMalloc()**.

Note: You must use the same value for *numBytes* as you used for the call to **IOMalloc()** that allocated the memory you're now freeing.

IOFreeLow()

SUMMARY

Free memory allocated by **IOMallocLow()**

DECLARED IN

driverkit/i386/kernelDriver.h

SYNOPSIS

void **IOFreeLow(**void ***var*, int *numBytes***)**

DESCRIPTION

This function frees memory allocated by **IOMallocLow()**.

Note: This function works only in kernel-level drivers.

IOGetDDMEntry()

SUMMARY

Obtain an entry from the Driver Debugging Module

DECLARED IN

driverkit/debugging.h

SYNOPSIS

int **IOGetDDMEntry(**int *entry*, int *outStringSize*, char **outString*, ns_time_t **timestamp*, int **cpuNumber***)**

DESCRIPTION

Returns in *outString* an entry from the DDM. The *entry* argument should indicate which entry to return, counting backwards from the most recent entry. The *timestamp* argument is set to a value indicating the time at which the entry was logged. The *cpuNumber* argument is set to the number of the CPU that the retrieved entry is associated with.

RETURN

Returns a nonzero value if the specified entry doesn’t exist. Otherwise, returns zero.

IOGetDDMMask()

SUMMARY

Returns the specified bitmask word

DECLARED IN

driverkit/debugging.h

SYNOPSIS

unsigned **IOGetDDMMask(**int *index***)**

DESCRIPTION

This is typically not used by drivers; it provides a procedural means of obtaining a

specified bitmask value. For performance reasons, the macros that filter and call **IOAddDDMEntry()** typically read the index words directly (the **IODDMMasks** array is a global variable).

IOGetObjectForDeviceName()

SUMMARY

Obtain the **id** of a kernel device, given its name

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

IOReturn **IOGetObjectForDeviceName(**IOString *deviceName*, id **deviceId***)*

DESCRIPTION

This function provides a simple mapping of device names to objects. Since this is valid only at kernel level, no security mechanism is provided; any kernel code can get the **id** of any kernel IODevice.

Note: This function works only in kernel-level drivers.

RETURN

Returns IO_DR_NOT_ATTACHED if *deviceName* isn't found; otherwise returns IO_R_SUCCESS.

IOGetTimestamp()

SUMMARY

Obtains a microsecond-accurate current timestamp

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOGetTimestamp(**ns_time_t **nsp***)*

DESCRIPTION

This function obtains a quick, microsecond-accurate, system-wide timestamp.

IOHostPrivSelf()

SUMMARY

Returns the kernel I/O task's version of the privileged host port

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

port_t **IOHostPrivSelf()**

DESCRIPTION

This function is necessary because the Mach function **host_priv_self()** doesn't work at kernel level.

Note: This function works only in kernel-level drivers. In user-level drivers, use **host_priv_self()** instead.

IOInitDDM()

SUMMARY

Initialize the Driver Debugging Module

DECLARED IN

driverkit/debugging.h

SYNOPSIS

Kernel level: void **IOInitDDM(**int *numBufs***)**
User level: void **IOInitDDM(**int *numBufs*, char **serverPortName*)

DESCRIPTION

This function must be called once by your driver before calling any other DDM functions.

IOInitGeneralFuncs()

SUMMARY

Initialize the general-purpose functions

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOInitGeneralFuncs()**

DESCRIPTION

Each user-level driver must call **IOInitGeneralFuncs()** once before calling any other functions declared in the **driverkit/generalFuncs.h** header file.

Note: Kernel-level drivers don't need to call this function, because it's automatically called by the kernel.

IOIsAligned()

SUMMARY

Determine whether an address is aligned

DECLARED IN

driverkit/align.h

SYNOPSIS

unsigned int **IOIsAligned(***address*, *bufferSize***)**

DESCRIPTION

This macro returns a nonzero value if *address* is a multiple of *bufferSize*; otherwise, it

returns 0.

IOLog()

SUMMARY

Adds a string to the system log

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOLog(**const char ***format*, ...**)**

DESCRIPTION

This is the Driver Kit's substitute for **printf()**; its implementation is similar to **syslog()**. **IOLog()** logs the string to **/usr/adm/messages** by default; you can specify another destination in the configuration file **/etc/syslog.conf**. The arguments are stdargs, just as for **printf()**. This function doesn't block on single-processor systems. It runs at level LOG_ERR and its facility is kern.

SEE ALSO

printf(3) UNIX manual page, **syslog**(3) UNIX manual page

IOMalloc()

SUMMARY

Standard memory allocator

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void ***IOMalloc(**int *numBytes***)**

DESCRIPTION

This function causes *numBytes* bytes of memory to be allocated; a pointer to the memory is returned. No guarantees exist as to the alignment or the physical contiguity of the allocated memory, but when **IOMalloc()** is called at kernel-level, the allocated memory is guaranteed to be wired down. Memory allocated with **IOMalloc()** should be freed with **IOFree()**.

Warning: If no memory is available, **IOMalloc()** blocks until it can obtain memory. For this reason, you shouldn't call **IOMalloc()** from a direct interrupt handler.

Drivers that can control (directly or indirectly) disks, network cards, or other devices used by a file system can run into a deadlock situation if they use **IOMalloc()** during I/O. This deadlock can occur when the pageout daemon attempts to free memory by moving pages out to disk. When the pageout daemon requests this I/O and the driver uses **IOMalloc()** to request more memory than is available, **IOMalloc()** blocks. The result is deadlock: the driver can't perform the I/O until memory is freed, and the memory can't be freed by the pageout daemon until the I/O happens. In general, a driver can avoid this deadlock by not allocating large amounts of memory during I/O. For example, allocating less than 100 bytes is safe, but allocating 8K bytes is very unsafe.

IOMallocLow()

SUMMARY

Allocates memory in the low 16MB of the computer's memory range

DECLARED IN

driverkit/i386/kernelDriver.h

SYNOPSIS

void ***IOMallocLow(**int *numBytes***)**

DESCRIPTION

This function acts like **IOMalloc()**, except that the allocated range of memory is guaranteed to be in the low 16MB of system memory and to be physically contiguous. This function is provided because some cards for Intel-based computers must be mapped to low memory. Memory allocated with **IOMallocLow()** should be freed with **IOFreeLow()**.

Note: This function works only in kernel-level drivers running on Intel-based computers.

IOMapPhysicalIntoIOTask

SUMMARY

Map a physical address range into your IOTask's address space

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

IOReturn **IOMapPhysicalIntoIOTask**(unsigned *physicalAddress*,
unsigned *length*,
vm_address_t **virtualAddress*)

DESCRIPTION

This function maps a range of physical memory into your IOTask. It returns the virtual address at which the range is mapped in the virtualAddress argument.

Note: This function works only in kernel-level drivers.

RETURN

Returns an error if the specified physical range could not be mapped; otherwise, returns IO_R_SUCCESS.

SEE ALSO

IOUnmapPhysicalFromIOTask()

IONsTimeFromDDMMsg()

SUMMARY

Extracts the time from a Driver Debugging Module message

DECLARED IN

driverkit/debuggingMsg.h

SYNOPSIS

ns_time_t **IONsTimeFromDDMMsg(**IODDMMsg **msg***)**

DESCRIPTION

This inline function combines the **timestampHighInt** and **timestampLowInt** fields from *msg* and returns the result.

IOPanic()

SUMMARY

Panic or dump memory after logging a string to the console

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOPanic(**const char ***reason***)**

DESCRIPTION

The *reason* argument is logged to the console, after which either a kernel panic (if in kernel space) or a memory dump (if in user space) occurs.

Note: Use of this function is an extreme measure. Use **IOPanic()** only when continued execution may cause system corruption.

IOPhysicalFromVirtual()

SUMMARY

Find the physical address corresponding to a virtual address

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

IOReturn **IOPhysicalFromVirtual(**vm_task_t *task*,

vm_address_t *virtualAddress*, unsigned int **physicalAddress***)**

DESCRIPTION

This function gets the physical address (if any) that corresponds to *virtualAddress*. It returns IO_R_INVALID_ARG if no physical address corresponds to *virtualAddress*. On success, it returns IO_R_SUCCESS. If *virtualAddress* is in the current task, then the *task* argument should be set to **IOVmTaskSelf()**. This function will never block. Use this function only to find the physical address of wired down memory since the physical address of unwired down memory might change over time.

Note: This function is available only at kernel level. This function shouldn't be used in a custom interrupt handler—it can't run at the interrupt level.

IOReadRegister(), IOWriteRegister(), IOReadModifyWriteRegister()

SUMMARY

Read or write values of display registers

DECLARED IN

driverkit/i386/displayRegisters.h

SYNOPSIS

unsigned char **IOReadRegister(**

IOEISAPortAddress *port*,

unsigned char *index***)**

void **IOWriteRegister(** IOEISAPortAddress *port*, unsigned char *index*, unsigned char *value***)**

void **IOReadModifyWriteRegister(** IOEISAPortAddress *port*, unsigned char *index*, unsigned char *protect*, unsigned char *value***)**

DESCRIPTION

These inline functions perform operations commonly used to read or write display registers. **IOReadRegister** reads and returns the value of the register specified by *port* and *index*. **IOWriteRegister()** writes *value* to the register specified by *port* and *index*. **IOReadModifyWriteRegister()** reads the specified register, zeroes every bit that isn't set in the *protect* mask, sets every bit that's set in *value*, and sets the register to the new value. When the *protect* mask is zero, the effect is to set the register to *value*.

Note: These functions are supported only on Intel-based computers.

IORemoveFromBdevsw(), IORemoveFromCdevsw(), IORemoveFromVfssw()

SUMMARY

Remove UNIX-style entry points from a device switch table

DECLARED IN

driverkit/devsw.h

SYNOPSIS

void **IORemoveFromBdevsw(**int *bdevswNumber***)**
void **IORemoveFromCdevsw(**int *cdevswNumber***)**
void **IORemoveFromVfssw(**int *vfsswNumber***)**

DESCRIPTION

These functions remove a device from a device switch table, replacing it with a null entry.

Note: You should use IODevice's **removeFromBdevsw** and **removeFromCdevsw** methods instead of **IORemoveFromBdevsw()** and **IORemoveFromCdevsw()**, whenever possible.

SEE ALSO

IOAddToBdevsw(), **IOAddToCdevsw()**, **IOAddToVfssw()**

IOResumeThread()

SUMMARY

Resume the execution of a thread suspended with **IOSuspendThread()**

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOResumeThread(**IOThread *thread***)**

DESCRIPTION

This function causes the execution of a suspended thread to continue.

IOScheduleFunc()

SUMMARY

Arrange for the specified function to be called at a certain time in the future

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOScheduleFunc(**IOThreadFunc *function*, void **arg*, int *seconds***)**

DESCRIPTION

This function causes *function* to be called in *seconds* seconds, with *arg* as *function*'s argument. The call to *function* occurs in the context of the caller's task, but in a thread that is unique to the Driver Kit. The call to *function* can be cancelled with **IOUnscheduleFunc()**.

Note: The kernel version of **IOScheduleFunc()** performs the callback in the kernel task's context, not the I/O Task context. One consequence is that *function* can't send Mach messages with **msg_send()**; it needs to use **msg_send_from_kernel()** instead, as described in Chapter 2.

IOSendInterrupt()

SUMMARY

Arrange for an interrupt message to be sent

DECLARED IN

driverkit/IODirectDevice.h

SYNOPSIS

void **IOSendInterrupt(**void ***identity*, void ***state*, unsigned int *msgId***)**

DESCRIPTION

This function is useful if you need to handle interrupts directly—for example, because of a timing constraint in the hardware—but don't wish to give up the advantages of interrupt notification by messages. To handle interrupts directly, you must implement the **getHandler:level:argument:forInterrupt:** message of IODirectDevice.

The *msgId* argument specifies the message ID of the interrupt message that will be sent. This should be IO_DEVICE_INTERRUPT_MSG unless the driver's documentation specifies otherwise. The *identify* and *state* arguments should be set to the values that the interrupt handler received in its own arguments. For example (italicized text delineated in angle brackets, that is << >>, is to be filled in with device-specific code):

```
static void myInterruptHandler(void *identity, void *state,
    unsigned int arg)
{
    << handle the interrupt >>
    IOSendInterrupt(identity, state, IO_DEVICE_INTERRUPT_MSG);
}
```

SEE ALSO

IODisableInterrupt(), IOEnableInterrupt()

IOSetDDMMask()

SUMMARY

Set specified bitmask word to specified value

DECLARED IN

driverkit/debugging.h

SYNOPSIS

void **IOSetDDMMask(**int *index*, unsigned int *bitmask*)

DESCRIPTION

This is typically used by individual user-level drivers at initialization time, if then. Subsequently, it is usually used only by the Driver Debugging Module's server thread to change the current bitmask value.

The *index* argument is an index into **IODDMMasks**, which is an array of **unsigned int**. Each entry of the array contains 32 mask bits.

IOSetUNIXError()

SUMMARY

Explicitly return an error value from a UNIX-style driver

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

void **IOSetUNIXError(**int *errno***)**

DESCRIPTION

Most UNIX-style drivers don't need to use this function. However, those that explicitly set the caller's errno can use this function to do so. This function is used when the caller executes as a result of a UNIX-style entry point.

Note: This function works only in kernel-level drivers.

IOSleep()

SUMMARY

Sleep for indicated number of milliseconds

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOSleep(**unsigned int *milliseconds***)**

DESCRIPTION

This function causes the caller to block for the indicated number of milliseconds.

IOSuspendThread()

SUMMARY

Suspend the execution of a thread started with **IOForkThread()**

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOSuspendThread(**IOThread *thread***)**

DESCRIPTION

This function causes the execution of a running thread to pause. The thread can be resumed with **IOResumeThread()**.

IOUnmapPhysicalFromIOTask

SUMMARY

Unmap a physical address range from your IOTask's address space

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

IOReturn **IOUnmapPhysicalFromIOTask**(vm_address_t *virtualAddress*, unsigned *length*)

DESCRIPTION

This function unmaps a range of memory that was mapped with **IOMapPhysicalIntoIOTask()**. You should use this to destroy a mapping when you no longer need to use it.

Note: This function works only in kernel-level drivers.

RETURN

Returns an error if the specified virtual range was not mapped by **IOMapPhysicalIntoIOTask()**; otherwise, returns IO_R_SUCCESS.

SEE ALSO

IOMapPhysicalIntoIOTask()

IOUnscheduleFunc()

SUMMARY

Cancel a request made with **IOScheduleFunc()**

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

void **IOUnscheduleFunc(**IOThreadFunc *function*, void *_*arg***)**

DESCRIPTION

This function removes a request made using **IOScheduleFunc()** from the current list of pending requests. An error will be logged to the console if the specified *function*/*arg* pair is not currently registered.

IOVmTaskCurrent()

SUMMARY

Returns the **vm_task_t** of the current task

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

vm_task_t **IOVmTaskCurrent()**

DESCRIPTION

Returns the **vm_task_t** for the current task. The only reason to use this function is to perform DMA to user space memory transfers in a UNIX-style driver.

Note: This function works only in kernel-level drivers.

SEE ALSO

IOVmTaskSelf()

IOVmTaskForBuf()

SUMMARY

Returns the **vm_task_t** associated with a **buf** structure

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

vm_task_t **IOVmTaskForBuf(**struct buf **buffer***)**

DESCRIPTION

Block drivers use this function to determine the task for which they're doing I/O. The value returned by this function is used in calls to **IOPhysicalFromVirtual()**, which returns an address that's used in IODirectDevice's **createDMABufferFor:...** method.

Note: This function works only in kernel-level drivers.

IOVmTaskSelf()

SUMMARY

Obtain the **vm_task_t** of the kernel

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

vm_task_t **IOVmTaskSelf()**

DESCRIPTION

This function is used to obtain the kernel's **vm_task_t**, which is the **vm_task_t** for memory allocated with **IOMalloc()**. This function is required because the type definition of **vm_task_t** at kernel level is different from that of **vm_task_t** at user level.

Note: This function works only in kernel-level drivers.

Other Features

Auto Detection of Devices

EISA- and PCI-compliant systems can support automatic detection of devices, referred to as the *auto detect* feature. When auto detect is supported, the system can determine which devices are connected to the bus and the location of the devices. Devices are easier to configure because less information is required in the **.table** files for the driver. Auto detect is nondestructive: It doesn't change the state of any device.

Auto detect determines which devices are connected to a bus and their bus location. Each device identifies itself with an *auto detect ID* and indicates its location with an *anchor*.

Auto Detect IDs and Anchors

Each device driver identifies itself by an auto detect ID string, which is a list of identifiers that can be used to detect the devices that can be controlled by the driver. The meaning of the identifiers is different for each bus type.

For the EISA, PCI and Plug and Play bus types, this ID is expressed as a 32-bit hexadecimal number containing the vendor ID and the device ID for the device. For the EISA bus and for Plug and Play, the device ID is in the lower 16 bits, and the vendor ID is in the upper 16 bits. For the PCI bus, the vendor ID is in the lower 16 bits, and the device ID is in the upper 16 bits. The "Auto Detect IDs" key should contain IDs for all the devices that can be controlled by the device driver. There is currently no provision for matching "don't care" bits in the ID, although that may be added in the future.

When your driver is configured in the system, the configuration software will scan the bus for devices that match your auto detect IDs. When it finds a device, it will create a device description for your driver with a value for the "Location" key that allows you to locate your device on the bus. This key is known as the "anchor" for your device and is different for each bus type.

For the EISA bus, the anchor is a slot number between 0 and 15. The value of the "Location" key is "Slot <n>", where <n> is your slot number.

For the PCI bus, the anchor is a three-part identifier containing the bus number, the device number, and the function number for your device. The bus number can be between 0 and 255, the device number can be between 0 and 31, and the function number can be between 0 and 7. The syntax of the "Location" key is "Dev:<d>

Func:<f> Bus:<b>", where <b> is the bus number, <d> is the device number, and <f> is the function number.

Plug and Play support does not currently define an anchor for the card. Instead, the resources assigned in your configuration table, such as base I/O address, IRQ level, and DMA channels, are programmed into your device using the Plug and Play control registers. In the future, an anchor will be assigned so you can use new Driver Kit methods to control resources in more detail.

Auto Detect Process

The driver bundle's **Default.table** has two key/value pairs of interest for auto detection: "Bus Type" and "Auto Detect IDs". The first tells which bus the driver supports. The second lists the auto detect IDs of all the supported devices for this driver, expressed in the 32-bit hexadecimal number format.

Auto detection is used at two times: During installation and when you use the Configure application.

During initial installation, the auto detect software scans each bus and obtains from each device its auto detect ID and its anchor in the form that the bus uses. It adds the "Location" key to your driver's device description in memory.

Note: A computer may have more than one bus, and the buses may be different types.

When you use Configure to add a driver to your system, it looks at every file with a **.table** extension (with the exception of **Instance*n*.table** files) in each configuration bundle, trying to match bus types and auto detect IDs. It first examines the "Bus Type" and then the "Auto Detect IDs" key/value pairs and generates a candidate list of drivers for each device found. There may be more than one candidate driver for a device. In that case, the user is presented with a list of drivers for the device and asked to pick one. After the user chooses, the **.table** file is copied to an **Instance*n*.table** with this line appended:

```
"Location" = "anchor"
```

where *anchor* is the anchor in the format appropriate for the bus.

There are cases where "Location" is blank. Each bus-specific category of IODirectDevice (IOEISADirectDevice, IOPCIDirectDevice, and so on) and IODeviceDescription subclass (IOPCMCIADeviceDescription and so on) provide methods for extracting this information, such as **getPCIDevice:function:bus** and **getEISASlotNumber**.

Protocols

IOConfigurationInspector

Adopted By: IODeviceInspector class
IODisplayInspector class

Declared In: driverkit/IODeviceInspector.h

Protocol Description

The IOConfigurationInspector protocol is adopted by inspectors that are loaded into the Configure application. Each inspector lets the user inspect and set information about a device, such as a specific brand of Ethernet card. The inspector stores this information in an NXStringTable that is specified to the inspector with the **setTable:** method.

The default, customizable inspector implemented by the IODeviceInspector class is sufficient for many devices. However, if IODeviceInspector doesn't suit your configuration needs, you should implement your own inspector class that adopts the IOConfigurationInspector protocol. An example of adopting this protocol is under **/NextLibrary/Documentation/NextDev/Examples/DriverKit** in the **DriverInspector** directory.

Method Types

Get the inspector's View – inspectionView

Notify that resources have changed – resourcesChanged

Set the description table – setTable

Instance Methods

inspectionView

– (View *)**inspectionView**

Returns the View of the inspector.

resourcesChanged:

– **resourcesChanged:**(IOResources *)*resources*

The Configure application sends this message to all inspectors whenever an interrupt, DMA channel, I/O port, or memory range is chosen or dropped in any inspector. This method should check for conflicts and update the UI.

This message is sent as often as you might need it, including immediately after a **setTable:** and after your own changes. You are guaranteed to be deactivated before your current table is freed, but you will not receive a **setTable:nil**, so don't count on accessing or modifying the table except in response to a user action.

setTable:

– **setTable:**(NXStringTable *)*anObject*

Sets the NXStringTable describing the inspector's device to *anObject*. You should update the UI when **setTable:** gives you a table to inspect. Your object should keep a handle to the table. When the user makes changes, immediately update the table; do not use OK/Revert buttons.

IOEventThread

Adopted By: The event system

Declared In: driverkit/eventProtocols.h

Protocol Description

The IOEventThread protocol provides access to the event system's I/O thread. You can obtain an IOEventThread-compliant object from IOEventSource's **owner** method.

Method Types

Sending messages
- sendIOThreadAsyncMsg:to:with:
- sendIOThreadMsg:to:with:

Instance Methods

sendIOThreadAsyncMsg:to:with:

– (IOReturn)**sendIOThreadAsyncMsg:**(id)*instance*
to:(SEL)*selector*
with:(id)*data*

From the event system's I/O thread, sends the message specified by *selector* to *instance*, with the argument *data*. This method doesn't wait for the *selector* method to be called, and doesn't detect whether *selector* is a valid method of *instance*. Returns IO_R_IPC_FAILURE if an error occurred; otherwise, returns IO_R_SUCCESS.

See also: – **sendIOThreadMsg:to:with:**

sendIOThreadMsg:to:with:

– (IOReturn)**sendIOThreadMsg:**(id)*instance*
to:(SEL)*selector*
with:(id)*data*

From the event system's I/O thread, sends the message specified by *selector* to

instance, with the argument *data*. This method waits until the *selector* method has returned. Returns IO_R_IPC_FAILURE if the message couldn't be sent; otherwise, returns IO_R_SUCCESS.

See also: – **sendIOThreadAsyncMsg:to:with:**

IONetworkDeviceMethods

Adopted By: IOEthernet
IOTokenRing

Declared In: driverkit/IONetwork.h

Protocol Description

This protocol must be implemented by network direct device drivers that use IONetwork to tie into the kernel network system. These methods are invoked by IONetwork objects in response to events in the network system.

Note: Network drivers must run at kernel level.

Method Types

Creating netbufs	– allocateNetbuf
Initializing the hardware	– finishInitialization
Sending out a packet	– outputPacket:address:
Performing control commands	– performCommand:data:

Instance Methods

allocateNetbuf

– (netbuf_t)**allocateNetbuf**

This method creates and returns a netbuf to be used for an impending output.

This method doesn't always have to return a buffer. For example, you might want to limit the number of buffers your driver instance can allocate (say, 200 kilobytes worth) so that it won't use too much wired-down kernel memory. When this method fails to return a buffer, it should return NULL.

Here's an example of implementing **allocateNetbuf**.

```
#define my_HDR_SIZE    14
```

```
#define my_MTU        1500
#define my_MAX_PACKET  (my_HDR_SIZE + my_MTU)

- netbuf_t allocateNetbuf
{
    if (_numbufs == _maxNumbufs)
        return(NULL);
    else {
        _numbufs++;
        return(nb_alloc(my_MAX_PACKET));
    }
}
```

See also: **nb_alloc()** (*NEXTSTEP Operating System Software)*

finishInitialization

– (int)**finishInitialization**

This method should perform any initialization that hasn't already been done. For example, it should make sure its hardware is ready to run. You can specify what the integer return value (if any) should be.

If you implement this method, you need to check that [self isRunning] == YES.

outputPacket:address:

– (int)**outputPacket:**(netbuf_t)*packet* **address:**(void *)*address*

This method should deliver the specified packet to the given address. Its return value should be zero if no error occurred; otherwise, return an error number from the header file **sys/errno.h**.

If you implement this method, you need to check that [self isRunning] == YES. If so, insert the necessary hardware addresses into the packet and check it for minimum length requirements.

performCommand:data:

– (int)**performCommand:**(const char *)*command* **data:**(void *)*data*

This method performs arbitrary control operations; the character string *command* is used to select between these operations. Although you don't have to implement any operations, there are five standard operations. You can also define your own operations.

The standard commands are listed in the following table. The constant strings listed below are declared in the header file **net/netif.h** (under the **bsd** directory of

/NextDeveloper/Headers).

Command	Operation
IFCONTROL_SETFLAGS	Request to have interface flags turned on or off. The *data* argument for this command is of type **union ifr_ifru** (which is declared in the header file **net/if.h**).
IFCONTROL_SETADDR	Set the address of the interface.
IFCONTROL_GETADDR	Get the address of the interface.
IFCONTROL_AUTOADDR	Automatically set the address of the interface.
IFCONTROL_UNIXIOCTL	Perform a UNIX **ioctl()** command. This is only for compatibility; **ioctl()** isn't a recommended interface for network drivers. The argument is of type **if_ioctl_t ***, where the **if_ioctl_t** structure contains the UNIX ioctl request (for example, SIOCSIFADDR) in the **ioctl_command** field and the ioctl data in the **ioctl_data** field.

An example of implementing **performCommand:data:** follows.

```
- (int)performCommand:(const char *)command data:(void *)data
{
    int error = 0;

    if (strcmp(command, IFCONTROL_SETFLAGS) == 0)
        /* do nothing */;
    else
    if (strcmp(command, IFCONTROL_GETADDR) == 0)
        bcopy(&my_address, data, sizeof (my_address));
    else
        error = EINVAL;

    return (error);
}
```

IOSCSIControllerExported

Adopted By: IOSCSIController class

Declared In: driverkit/scsiTypes.h

Protocol Description

Indirect device drivers for devices attached to SCSI controllers use the methods in this protocol to communicate with IOSCSIController.

Method Types

Allocating well-aligned buffers	– allocateBufferOfLength:actualStart:actualLength: – getDMAAlignment:
Requesting I/O	– executeRequest:buffer:client: – maxTransfer
Reserving SCSI targets	– reserveTarget:lun:forOwner: – releaseTarget:lun:forOwner:
Resetting the SCSI bus	– resetSCSIBus
Getting the IOReturn equivalent of a **sc_status_t** value	– returnFromScStatus:

Instance Methods

allocateBufferOfLength:actualStart:actualLength:

– (void *)**allocateBufferOfLength:**(unsigned)*length*
actualStart:(void **)*actualStart*
actualLength:(unsigned *)*actualLength*

Allocates and returns a pointer to some well-aligned memory. Well-aligned memory is necessary for calls to **executeRequest:buffer:client:**. You should use *actualStart* and *actualLength* when freeing the memory, as follows (italicized text delineated in

angle brackets, that is << >>, is to be filled in with device-specific code):

```
dataBuffer = [_controller allocateBufferOfLength:block_size
        actualStart:&freePtr, actualLength:&freeLength];
<< Use the buffer... >>
IOFree(freePtr, freeLength);
```

Here's a typical use of this method:

```
IODMAAlignment    dmaAlign;
unsigned int      alignment, alignedLength, freeLength;
void             *alignedPtr = NULL;
unsigned int      maxLength; /* Max length of the current transfer
*/
/* . . . */
[_controller getDMAAlignment:&dmaAlign];
if(<< we're doing a write >>)
    alignment = dmaAlign.writeLength;
else
    alignment = dmaAlign.readLength;

if(alignment > 1)
    alignedLength = IOAlign(unsigned int, maxLength, alignment);
else
    alignedLength = maxLength;

alignedPtr = [_controller allocateBufferOfLength:alignedLength
        actualStart:&freePtr
        actualLength:&freeLength];

<< If we're going to do a write, copy the data to alignedPtr.
   Set up the request and submit it, as described in the
   executeRequest:buffer:client: description. >>
<< Do any post-I/O processing that's necessary. >>

IOFree(freePtr, freeLength);
```

See also: – **getDMAAlignment:**

executeRequest:buffer:client:

– (sc_status_t)**executeRequest:**(IOSCSIRequest *)*scsiRequest*
buffer:(void *)*buffer*
client:(vm_task_t)*client*

Executes the specified request. Indirect devices invoke this method whenever they need the IOSCSIController to perform I/O.

Subclasses of IOSCSIController must implement this method. A typical implementation of this method consists of the following:

- Using **IOScheduleFunc()** to schedule a timeout function to be called after *scsiRequest*->**timeoutLength** time has elapsed without I/O completion
- Sending the command descriptor block (CDB) specified in *scsiRequest* to the controller

- When the I/O has completed, unscheduling the timeout function

This method should return *scsiRequest*->**driverStatus**, which should be set by the part of the driver that detected I/O completion or timeout.

Indirect devices use this method as shown below (italicized text delineated in angle brackets, that is << >>, is to be filled in with device-specific code):

```
void             *alignedPtr = NULL;
unsigned int      alignedLength;
IOSCSIRequest     request;
cdb_t             cdb;

/* . . . */
if (<< we're going to be doing DMA >>) {
    << Ensure we have a well-aligned buffer that starts at
alignedPtr
       and continues for alignedLength bytes.  See the
       allocateBuffer: description for one way of doing this. >>
} else {
    alignedLength = 0;
    alignedPtr = 0;
}

bzero(&request, sizeof(request));
request.target = [self target];
request.lun    = [self lun];
request.read = << YES if this is a read; NO otherwise >>;
request.maxTransfer = alignedLength;
request.timeoutLength = << some timeout length, in seconds >>;
request.disconnect = << 1 if allowed to disconnect; otherwise 0 >>;
request.cdb = cdb;
<< Set up the cdb (command descriptor block) field. The type of
this
   field, cdb_t, is defined and described in the header file
   bsd/dev/scsireg.h. >>

rtn = [_controller executeRequest:&request
        buffer:alignedPtr
        client:IOVmTaskSelf()];
```

getDMAAlignment:

– (void)**getDMAAlignment:**(IODMAAlignment *)*alignment*

Returns the DMA alignment requirements for the current architecture. IOSCSIController subclasses can override this method to specify any device-specific alignment requirements. See the description of **allocateBufferOfLength:actualStart:actualLength:** for an example of using this method.

See also: – **allocateBufferOfLength:actualStart:actualLength:**

maxTransfer

– (unsigned)**maxTransfer**

Returns the maximum number of bytes per DMA transfer. This is the maximum transfer that can be requested in a call to **executeRequest:buffer:client:**.

releaseTarget:lun:forOwner:

– (void)**releaseTarget:**(unsigned char)*target*
lun:(unsigned char)*lun*
forOwner:*owner*

Releases the specified target/lun pair. If *owner* hasn't reserved the pair, this method uses IOLog to print an error message.

See also: – **reserveTarget:lun:forOwner:**

reserveTarget:lun:forOwner:

– (int)**reserveTarget:**(unsigned char)*target*
lun:(unsigned char)*lun*
forOwner:*owner*

Reserves the specified target/lun pair, if it isn't already reserved. This method is invoked by a client (for example, a SCSIDisk instance) to mark a particular target/lun as being in use by that client. Usually, this happens at **probe:** time; however, the SCSIGeneric driver uses this method at other times.

This method returns a nonzero value if the target/lun pair is already reserved. Otherwise, it returns zero.

See also: – **releaseTarget:lun:forOwner:**

resetSCSIBus

– (sc_status_t)**resetSCSIBus**

Resets the SCSI bus. Subclasses of IOSCSIController must implement this method so that it resets the SCSI bus. The **sc_status_t** enumerated type is defined and described in the header file **bsd/dev/scsireg.h**.

returnFromScStatus:

– (IOReturn)**returnFromScStatus:**(sc_status_t)*sc_status*

Returns the IOReturn value corresponding to the specified **sc_status_t** value. The **sc_status_t** enumerated type is defined and described in the header file **bsd/dev/scsireg.h**.

IOScreenEvents

Adopted By: IODisplay

Declared In: driverkit/eventProtocols.h

Protocol Description

The methods in this protocol are invoked by the event system, at the request of the Window Server or of pointer management software.

Method Types

Manipulating the cursor	– hideCursor: – moveCursor:frame:token: – showCursor:frame:token:
Get the device port	– devicePort
Set screen brightness	– setBrightness:token:

Instance Methods

devicePort

– (port_t)**devicePort**

Returns the device port, which should be obtained from this instance's IODeviceDescription.

hideCursor:

– **hideCursor:**(int)*token*

Removes the cursor from the screen.

moveCursor:frame:token:

– **moveCursor:**(Point *)*cursorLoc*
frame:(int)*frame*
token:(int)*token*

Removes the cursor from the screen, moves it, and displays the cursor in its new position.

setBrightness:token:

– **setBrightness:**(int)*level* **token:**(int)*token*

Sets the brightness of the screen. Many devices (and thus many drivers) don’t permit this operation.

See also: – **setBrightness:token:** (IOFrameBufferDisplay class)

showCursor:frame:token:

– **showCursor:**(Point *)*cursorLocation*
frame:(int)*frame*
token:(int)*token*

Displays the cursor at *cursorLocation*.

IOScreenRegistration

Adopted By: The event system

Declared In: driverkit/eventProtocols.h

Protocol Description

Display drivers use the messages in the IOScreenRegistration protocol to register and unregister themselves with the event system. These methods are called by IODisplay in response to an **getIntValues:forParameter:count:** call that specifies the "IO_Framebuffer_Register" parameter.

You shouldn't need to invoke the methods in this protocol, because they're already invoked automatically by IOFrameBufferDisplay and IOSVGADisplay.

Instance Methods

registerScreen:bounds:shmem:size:

– (int)**registerScreen:**(id)*instance*
bounds:(Bounds *)*bounds*
shmem:(void **)*address*
size:(int *)*num*

Registers *instance* as a display driver. Returns a token that's used to refer to the display in other calls to the event system.

unregisterScreen:

– (void)**unregisterScreen:**(int)*token*

Unregisters the instance associated with *token* as a display driver.

Types and Constants

Defined Types

IOAddressRange

DECLARED IN

driverkit/IODeviceInspector.h

SYNOPSIS

```
typedef struct IOAddressRange {
        unsigned        start;
        unsigned        length;
} IOAddressRange
```

DESCRIPTION

Used to describe address ranges.

IOCache

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

```
typedef enum {
        IO_CacheOff,

        IO_WriteThrough,
        IO_CopyBack
} IOCache
```

Used <<*where*?>> to specify caching. IO_CacheOff inhibits the cache.

IOChannelCommand

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned int **IOChannelCommand**

DESCRIPTION

IOChannelDequeueOption

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned int **IOChannelDequeueOption**

DESCRIPTION

IOChannelEnqueueOption

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned int **IOChannelEnqueueOption**

DESCRIPTION

IOCharParameter

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef char **IOCharParameter**[IO_MAX_PARAMETER_ARRAY_LENGTH]

DESCRIPTION

Standard type for a character parameter value, used by the get/set parameter functionality provided by IODevice and IODeviceMaster.

IODDMMsg

DECLARED IN

driverkit/debuggingMsg.h

SYNOPSIS

```
typedef struct {
        msg_header_t header;
        msg_type_t argType;
        unsigned index;
        unsigned maskValue;
        unsigned status;
        unsigned timestampHighInt;
        unsigned timestampLowInt;
        int cpuNumber;
        msg_type_t stringType;
        char string[IO_DDM_STRING_LENGTH];
} IODDMMsg
```

DESCRIPTION

The message format understood by the Driver Debugging Module. You don’t usually have to use this message, as long as DDMViewer <<*check*>> is sufficient for your needs.

IODescriptorCommand

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned char **IODescriptorCommand**

DESCRIPTION

IODeviceNumber

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned int **IODeviceNumber**

DESCRIPTION

IODeviceStyle

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef enum {
    IO_DirectDevice,
    IO_IndirectDevice,
    IO_PseudoDevice
} **IODeviceStyle**

DESCRIPTION

Returned by the **deviceStyle** method to specify whether the driver is a direct device driver (one that directly controls hardware), an indirect device driver (one that controls hardware using a direct device driver), or a pseudodriver (one that controls no hardware). The driver style determines how it's configured into the system, as described <<*somewhere*>>.

IODisplayInfo

DECLARED IN

bsd/dev/i386/displayDefs.h <<*to be moved to driverkit/*>>

SYNOPSIS

```
typedef struct{
        int width;
        int height;
        int totalWidth;
        int rowBytes;
        int refreshRate;
        void *frameBuffer;
        IOBitsPerPixel bitsPerPixel;
        IOColorSpace colorSpace;
        unsigned int flags;
        void *parameters;
} IODisplayInfo;
```

DESCRIPTION

This structure describes a video display. Each linear mode supported by an IOFrameBufferDisplay has a corresponding IODisplayInfo. <<*Tell when it's used.*>> The structure's fields are

width	Width, in pixels
height	Height, in pixels
totalWidth	Width including undisplayed pixels
rowBytes	The number of bytes to get from one scanline to next. To determine this value, determine how many 8-bit bytes each pixel occupies (rounding up to an integer) and multiply this by the value of **totalWidth**. For example, a color display mode that uses 15 bits per pixel and has a **totalWidth** of 1024 has a **rowBytes** value of 2048.
refreshRate	Monitor refresh setting, in Hz <<*how do you decide this?*>>
frameBuffer	Pointer to origin of screen; untyped to force actual screen writes to be dependent on bitsPerPixel. The driver's **initFromDeviceDescription:** method should set this field to the value returned by **mapFrameBufferAtPhysicalAddress:length:**.
bitsPerPixel	The memory space occupied by one pixel. 8-bit black and

	white display modes use the value IO_8BitsPerPixel, and “16-bit” color display modes that use 5 bits each for red, green, and blue use the value IO_15BitsPerPixel. See the documentation of the IOBitsPerPixel type for other values.
colorSpace	Specifies the sample-encoding format.<<*what does that mean?*>> Typically, this value is either IO_DISPLAY_ONEISWHITECOLORSPACE (for monochrome modes) or IO_DISPLAY_RGBCOLORSPACE (for color modes). See the documentation of the IOColorSpace type for other values.
flags	Flags used to indicate special requirements or conditions to DPS. Currently, this should always be zero. <<*true? Or is it ignored?*>>
parameters	Driver-specific parameters.

Here’s an array of IODisplayInfo structures for a driver that supports several monochrome and color modes:

```
static const IODisplayInfo MyModes[MY_NUM_MODES] = {
  { 1024, 768, 1024, 1024, 66, 0,
    IO_8BitsPerPixel, IO_DISPLAY_ONEISWHITECOLORSPACE, 0, 0 },
  { 1280, 1024, 2048, 2048, 68, 0,
    IO_8BitsPerPixel, IO_DISPLAY_ONEISWHITECOLORSPACE, 0, 0 },
  { 800, 600, 800, 1600, 72, 0,
    IO_15BitsPerPixel, IO_DISPLAY_RGBCOLORSPACE, 0, 0 },
  { 1024, 768, 1024, 2048, 72, 0,
    IO_15BitsPerPixel, IO_DISPLAY_RGBCOLORSPACE, 0, 0 }
};
```

These structures correspond to the display modes specified in the device configuration bundle’s **Localizable.strings** file:

```
"DisplayModes" = "Height:768 Width:1024 Refresh:66Hz ColorSpace:
BW:8;
    Height:1024 Width:1280 Refresh: 68Hz ColorSpace: BW:8;
    Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:555/16;
    Height: 600 Width: 800 Refresh: 72Hz ColorSpace: RGB:555/16";
```

IODMAAlignment

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef struct {

unsigned **readStart**;
unsigned **writeStart**;
unsigned **readLength**;
unsigned **writeLength**;
} **IODMAAlignment**

DESCRIPTION

Used <<*by whom?*>> to specify DMA alignment. A field value of 0 means that alignment isn't restricted for values corresponding to the field.

IODMABuffer

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef void ***IODMABuffer**

DESCRIPTION

Used as a machine-independent type for a machine-dependent DMA buffer.

SEE ALSO

IOEISADMABuffer

IODMADirection

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef enum {
IO_DMARead,
IO_DMAWrite
} **IODMADirection**

DESCRIPTION

Used <<*where?*>> to specify the direction of DMA. IO_DMARead indicates a

transfer from the device into system memory; IO_DMAWrite indicates a transfer from system memory to the device.

IODMAStatus

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef enum {
    IO_None,
    IO_Complete,
    IO_Running,
    IO_Underrun,
    IO_BusError,
    IO_BufferError,
} **IODMAStatus**

DESCRIPTION

Used <<*where?*>> to specify machine-independent DMA channel status.

IO_None	No appropriate status
IO_Complete	DMA channel idle
IO_Running	DMA channel running
IO_Underrun	Underrun or overrun
IO_BusError	Bus error
IO_BufferError	DMA buffer error

IODMATransferMode

DECLARED IN

driverkit/i386/directDevice.h

SYNOPSIS

typedef enum {
    IO_Demand,
    IO_Single,
    IO_Block,
    IO_Cascade,
} **IODMATransferMode**

DESCRIPTION

Used only in the **setTransferMode:forChannel:** method of the EISA/ISA category of IODirectDevice.

IOEISADMABuffer

DECLARED IN

driverkit/i386/driverTypes.h

SYNOPSIS

typedef void ***IOEISADMABuffer**

DESCRIPTION

Used as a machine-dependent type for a DMA buffer.

IOEISADMATiming

DECLARED IN

driverkit/i386/directDevice.h

SYNOPSIS

typedef enum {
    IO_Compatible,
    IO_TypeA,
    IO_TypeB,
    IO_Burst,
} **IOEISADMATiming**

DESCRIPTION

Used only in the **setDMATiming:forChannel:** method of the EISA/ISA category of IODirectDevice.

IOEISADMATransferWidth

DECLARED IN

driverkit/i386/directDevice.h

SYNOPSIS

typedef enum {
    IO_8Bit,
    IO_16BitWordCount,
    IO_16BitByteCount,
    IO_32Bit,
} **IOEISADMATransferWidth**

DESCRIPTION

Used only in the **setDMATransferWidth:forChannel:** method of the EISA/ISA category of IODirectDevice.

IOEISAInterruptHandler

DECLARED IN

driverkit/i386/driverTypes.h

SYNOPSIS

typedef void (*__IOEISAInterruptHandler__)
    (void *_identity_,

    void *_state_,
    unsigned int _arg_)

DESCRIPTION

IOEISAPortAddress

DECLARED IN

driverkit/i386/driverTypes.h

SYNOPSIS

typedef unsigned short **IOEISAPortAddress**

DESCRIPTION

IOEISAStopRegisterMode

DECLARED IN

driverkit/i386/directDevice.h

SYNOPSIS

typedef enum {
    IO_StopRegisterEnable,
    IO_StopRegisterDisable,
} **IOEISAStopRegisterMode**

DESCRIPTION

Used only in the **setStopRegisterMode:forChannel:** method of the EISA/ISA category of IODirectDevice.

IOIncrementMode

DECLARED IN

driverkit/i386/directDevice.h

SYNOPSIS

typedef enum {
    IO_Increment,
    IO_Decrement,
} **IOIncrementMode**

DESCRIPTION

Used only in the **setIncrementMode:forChannel:** method of the EISA/ISA category of IODirectDevice.

IOInterruptHandler

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef void (*__IOInterruptHandler__)
(void *_identity_,

void *_state_,
unsigned int _arg_)

DESCRIPTION

IOInterruptMsg

DECLARED IN

driverkit/interruptMsg.h

SYNOPSIS

typedef struct {
msg_header_t
header;
} **IOInterruptMsg**

DESCRIPTION

The format of the message sent by the kernel to a driver's interrupt handler.

IOIntParameter

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef int **IOIntParameter**[IO_MAX_PARAMETER_ARRAY_LENGTH]

DESCRIPTION

Standard type for an integer parameter value, used by the get/set parameter functionality provided by IODevice and IODeviceMaster.

IOIPCSpace

DECLARED IN

driverkit/kernelDriver.h

SYNOPSIS

typedef enum {
    IO_Kernel,
    IO_KernelIOTask,
    IO_CurrentTask
} **IOIPCSpace**

DESCRIPTION

Used only by the **IOConvertPort()** function to specify which space to convert the port from and to.

IONamedValue

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef struct {
    int **value**;
    const char ***name**;
} **IONamedValue**

DESCRIPTION

Map between constants or enumerations and text description.

IOObjectNumber

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef unsigned int **IOObjectNumber**

DESCRIPTION

IOParameterName

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef char **IOParameterName**[IO_MAX_PARAMETER_NAME_LENGTH]

DESCRIPTION

Standard type for a parameter name, used by the get/set parameter functionality provided by IODevice and IODeviceMaster.

IORange

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

```
typedef struct range {
        unsigned int start;
        unsigned int size;
} IORange
```

DESCRIPTION

Indicates a range of values. Used for memory regions, port regions, and so on.

IOReturn

DECLARED IN

driverkit/return.h

SYNOPSIS

typedef int **IOReturn**

DESCRIPTION

IOReturn values are returned by many Driver Kit classes.

IOSCSIRequest

DECLARED IN

driverkit/scsiTypes.h

SYNOPSIS

```
typedef struct {
        unsigned char target;
        unsigned char lun;
        cdb_t cdb;
        BOOL read;
        int maxTransfer;
        int timeoutLength;
        unsigned disconnect:1;
        unsigned pad:31;
        sc_status_t driverStatus;
        unsigned char scsiStatus;
        int bytesTransferred;
        ns_time_t totalTime;
        ns_time_t latentTime;
        esense_reply_t senseData;
} IOSCSIRequest
```

DESCRIPTION

Used in the IOSCSIController protocol's **executeRequest:buffer:client:** method.

IOString

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

typedef char **IOString**[IO_STRING_LENGTH]

DESCRIPTION

Standard type for an ASCII name, such as a device's name or type.

IOSwitchFunc

DECLARED IN

driverkit/devsw.h

SYNOPSIS

typedef int (***IOSwitchFunc**)()

DESCRIPTION

Used by **IOAddToBdevsw()** and **IOAddToCdevsw()** to specify UNIX-style entry points into a driver.

IOThread

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

typedef void ***IOThread**

DESCRIPTION

An opaque type used by the general-purpose functions to represent a thread.

IOThreadFunc

DECLARED IN

driverkit/generalFuncs.h

SYNOPSIS

typedef void (***IOThreadFunc**)
(void **arg*)

DESCRIPTION

Used by the general-purpose functions to specify the function that a thread should execute.

Symbolic Constants

Length Constants

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

IO_STRING_LENGTH
IO_MAX_PARAMETER_NAME_LENGTH
IO_MAX_PARAMETER_ARRAY_LENGTH

DESCRIPTION

These constants are used to determine the maximum length of the following types:

IO_STRING_LENGTH	IOString
IO_MAX_PARAMETER_NAME_LENGTH	IOParameterName
IO_MAX_PARAMETER_ARRAY_LENGTH	IOIntParameter IOCharParameter

Debugging String Length

DECLARED IN

driverkit/debuggingMsg.h

SYNOPSIS

IO_DDM_STRING_LENGTH

DESCRIPTION

The length of the **string** field in an IODebuggingMsg.

Debugging Messages

DECLARED IN

driverkit/debuggingMsg.h

SYNOPSIS

Constant	Meaning
IO_DDM_MSG_BASE	The lowest ID an IODebuggingMsg can have
IO_LOCK_DDM_MSG	Lock the Driver Debugging Module (DDM)
IO_UNLOCK_DDM_MSG	Unlock the DDM
IO_GET_DDM_ENTRY_MSG	Get an entry from the DDM
IO_SET_DDM_MASK_MSG	Set the debugging mask for the DDM
IO_CLEAR_DDM_MSG	Clear all entries from the DDM

DESCRIPTION

Values for the **header.msg_id** field of an IODebuggingMsg. See the discussion of the DDM in Chapter 2 for more information on these messages. <<*check*>>

Return Values from the DDM

DECLARED IN

driverkit/debuggingMsg.h

SYNOPSIS

Constant	Meaning
IO_DDM_SUCCESS	The message was received and understood
IO_NO_DDM_BUFFER	The DDM has no entry at the specified offset
IO_BAD_DDM_INDEX	The specified index isn't valid

DESCRIPTION

Values for the **status** field of an IODebuggingMsg.

DDM Masks

DECLARED IN

driverkit/debugging.h

SYNOPSIS

IO_NUM_DDM_MASKS

DESCRIPTION

This constant specifies the number of masks used by the Driver Debugging Module.

Interrupt Messages

DECLARED IN

driverkit/interruptMsg.h

SYNOPSIS

Constant	Meaning
IO_INTERRUPT_MSG_ID_BASE	The lowest ID an IOInterruptMsg can have
IO_TIMEOUT_MSG	
IO_COMMAND_MSG	
IO_DEVICE_INTERRUPT_MSG	Sent by the kernel when an interrupt occurs
IO_DMA_INTERRUPT_MSG	
IO_FIRST_UNRESERVED_INTERRUPT_MSG	

DESCRIPTION

Values for the **header.msg_id** field of an IOInterruptMsg. See the discussion of interrupts in Chapter 2 for more information on interrupt messages. <<*check. WHO USES everything except IO_DEVICE_INTERRUPT_MSG, and how?*>>

IOReturn Constants

DECLARED IN

driverkit/return.h

SYNOPSIS

Constant	Meaning
IO_R_SUCCESS	No error occurred
IO_R_NO_MEMORY	Couldn’t allocate memory
IO_R_RESOURCE	Resource shortage
IO_R_VM_FAILURE	Miscellaneous virtual memory failure
IO_R_INTERNAL	Internal library error

IO_R_RLD	Error in loading a relocatable file
IO_R_IPC_FAILURE	Error during IPC
IO_R_NO_CHANNELS	No DMA channels are available
IO_R_NO_SPACE	No address space is available for mapping
IO_R_NO_DEVICE	No such device
IO_R_PRIVILEGE	Privilege/access violation
IO_R_INVALID_ARG	Invalid argument
IO_R_BAD_MSG_ID	???
IO_R_UNSUPPORTED	Unsupported function
IO_R_INVALID	Should never be seen
IO_R_LOCKED_READ	Device is read locked
IO_R_LOCKED_WRITE	Device is write locked
IO_R_EXCLUSIVE_ACCESS	Device is exclusive access and is already open
IO_R_CANT_LOCK	Can’t acquire requested lock
IO_R_NOT_OPEN	Device not open
IO_R_OPEN	Device is still open
IO_R_NOT_READABLE	Reading not supported
IO_R_NOT_WRITABLE	Writing not supported
IO_R_IO	General I/O error
IO_R_BUSY	Device is busy
IO_R_NOT_READY	Device isn’t ready
IO_R_OFFLINE	Device is off line
IO_R_ALIGN	DMA alignment error
IO_R_MEDIA	Media error
IO_R_DMA	DMA failure
IO_R_TIMEOUT	I/O timeout
IO_R_NOT_ATTACHED	The device or channel isn’t attached
IO_R_PORT_EXISTS	The device port already exists
IO_R_CANT_WIRE	Can’t wire down physical memory <<*ever used? can you ever wire down physical memory?*>>
IO_R_NO_INTERRUPT	No interrupt port is attached
IO_R_NO_FRAMES	No DMA is enqueued

DESCRIPTION

Values for IOReturns.

IODevice Parameter Names

DECLARED IN

driverkit/IODevice.h

SYNOPSIS

Constant	Meaning
IO_CLASS_NAME	The value returned by + **name**
IO_DEVICE_NAME	The value returned by – **name**
IO_DEVICE_KIND	The value returned by – **deviceKind**
IO_UNIT	The value returned by – **unit**

DESCRIPTION

Null Constants

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

```
#define NULL                0
#define IO_NULL_VM_TASK     ((vm_task_t)0)
```

DESCRIPTION

Standard null values, used in various places.

Unused Constants

DECLARED IN

driverkit/driverTypes.h

IO_CC_START_READ
IO_CC_START_WRITE
IO_CC_ABORT
IO_CC_ENABLE_DEVICE_INTERRUPTS
IO_CC_DISABLE_DEVICE_INTERRUPTS
IO_CC_ENABLE_INTERRUPTS
IO_CC_DISABLE_INTERRUPTS

IO_CC_CONNECT_FRAME_LOOP

IO_CC_DISCONNECT_FRAME_LOOP
IO_CDO_DONE

IO_CDO_ALL
IO_CDO_ENABLE_INTERRUPTS
IO_CDO_ENABLE_INTERRUPTS_IF_EMPTY
IO_CEO_END_OF_RECORD
IO_CEO_DESCRIPTOR_INTERRUPT
IO_CEO_ENABLE_INTERRUPTS
IO_CEO_DESCRIPTOR_COMMAND
IO_CEO_ENABLE_CHANNEL
IO_MAX_BOARD_SIZE
IO_MAX_NRW_SLOT_SIZE
IO_MAX_SLOT_SIZE
IO_NATIVE_SLOT_ID
IO_NO_CHANNEL
IO_NULL_SLOT_ID
IO_NULL_DEVICE_TYPE
IO_NULL_DEVICE_INDEX
IO_NULL_DMA_ID
IO_SLOT_DEVICE_TYPE

DESCRIPTION

These constants aren’t used by drivers for Intel-based computers.

Note:

Global Variables

IODDMMasks

DECLARED IN

driverkit/debugging.h

SYNOPSIS

unsigned int **IODDMMasks**[IO_NUM_DDM_MASKS]

DESCRIPTION

The bitmask used to filter storing of debugging events. See the discussion of the Driver Debugging Module in Chapter 2 for more information.

IODMAStatusStrings

DECLARED IN

driverkit/driverTypes.h

SYNOPSIS

const IONamedValue **IODMAStatusStrings**[]

DECLARED IN

Used as an argument to **IOFindNameForValue()** to convert an IODMAStatus value into an error string.

Classes

The Driver Kit has two main groups of classes—those that user-level nondriver programs can use, and those used by drivers.

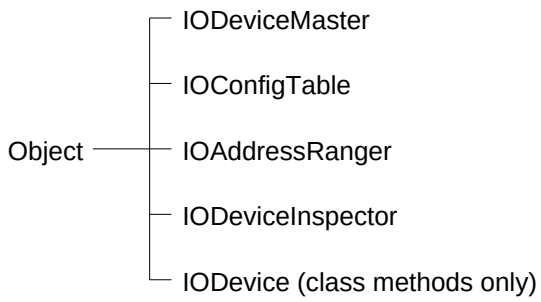

Figure 5-1. Classes Used by User-level Nondriver Programs

The classes used by drivers are further divided into those that are device-independent and those that are only used for specific kinds of devices.

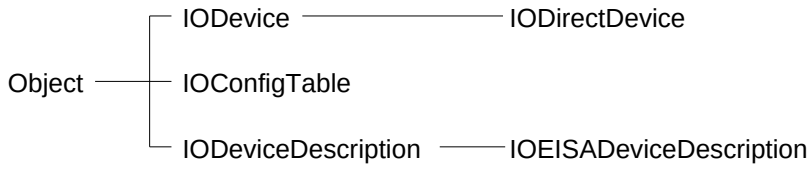

Figure 5-2. Device-independent Classes Used by Drivers

See Chapter 3 for information on the classes used for specific kinds of devices.

Some of the methods in the Driver Kit classes are stubs: they simply return without doing anything. Their method description says that they do nothing. They're typically hardware dependent, so you can implement them based on how your hardware operates and what interface you have available to the hardware. However, these methods provide a framework for you to build your driver on.

Note: The disk driver classes (IODisk, IOLogicalDisk, and IODiskPartition) are

public but haven’t been documented yet.

Other Classes Available to Drivers

Besides the Object class and the classes documented here, four more classes are available for drivers’ use. Three of these classes—NXLock, NXConditionLock, and NXSpinLock—are part of the Mach Kit, and are implemented at both user and kernel level. NXRecursiveLock, also part of the Mach Kit, is *not* available at kernel level. See the “Mach Kit” chapter in *NEXTSTEP General Reference* for more information.

IOAddressRanger

Inherits From: Object

Declared In: driverkit/IOAddressRanger.h

Class Description

The IOAddressRanger class provides user interface and type checking to be used in device inspecting modules for the Configure application. IOAddressRangers are used in IODeviceInspector for choosing values for I/O port ranges and ranges of memory.

An IOAddressRanger limits the range to a constant length that you specify with **setRangeLength:**. The range is also limited to be between the addresses you specify with **setAddressLimits::**. Whenever an address limit or range length is changed or the selected address range is changed, the IOAddressRanger adjusts the address range as follows:

- If the start address is less than the lower address limit, the start address is changed to be equal to the lower limit.
- If the range contains addresses above the higher limit, the start address is adjusted downward so that the range's last address is equal to the higher limit.

Instance Variables

None declared in this class.

Method Types

Checking address ranges	– checkRangesForConflicts:num:
	– checkText:
Setting and getting the start address	
	– setStartAddress:
	– startAddress
Action methods	– minus:
	– plus:

Setting and getting the range length	– rangeLength – setRangeLength:
Limiting the address range	– setAddressLimits::
Assigning a delegate	– setDelegate: – delegate
Delegate methods	– rangeDidChange:

Instance Methods

checkRangesForConflicts:num:

– (BOOL)**checkRangesForConflicts:**(IOAddressRanger *)*ranges*
num:(unsigned int)*numRanges*

A configuration inspector invokes this method to check whether this IOAddressRanger uses any addresses already used by the specified IOAddressRangers. If so, this method changes the color of the text in the text field to gray and sets the status button on. If no conflicts exist, this method sets the status button off and changes the color of the text to black. Returns NO if no conflicts exist and YES if conflicts exist.

checkText:

– **checkText:***sender*

Checks whether *sender*'s string value is an address and, if so, sets the range's start address (adjusted as described in the class description), updates the display, and sends the delegate a **rangeDidChange:** message. If the string isn't an address, the system beeps and updates the range's display. Returns **self** if the string is an address; otherwise, returns **nil**.

delegate

– **delegate**

Returns the IOAddressRanger's delegate, or **nil** if it doesn't have one.

minus:

– **minus:***sender*

This method is the target of the minus button in the IOAddressRanger. It moves the range down by the amount of the range's length (but no lower than the lower limit), updates the display, and sends the delegate a **rangeDidChange:** message. As an example, if the range is currently from 0x000e00 to 0x000eff, this method changes the range to be from 0x000d00 to 0x000dff. Returns **self**.

plus:

– **plus:***sender*

This method is the target of the plus button in the IOAddressRanger. It moves the range higher by the amount of the range's length (but not above the higher limit), updates the display, and sends the delegate a **rangeDidChange:** message. As an example, if the range is currently from 0x000e00 to 0x000eff, this method changes the range to be from 0x000f00 to 0x000fff. Returns **self**.

rangeLength

– (unsigned long)**rangeLength**

Returns the length of the range. This length should be set at initialization using **setRangeLength:**.

setAddressLimits::

– **setAddressLimits:**(unsigned long)*low* **:**(unsigned long)*high*

Limits the address range to values between *low* (inclusive) and *high* (inclusive) and adjusts the start address, as described in the class description. Returns **self**.

setDelegate:

– **setDelegate:***anObject*

Makes *anObject* the IOAddressRanger's delegate, and returns **self**. The delegate is sent a **rangeDidChange:** message whenever the address range changes.

setRangeLength:

– **setRangeLength:**(unsigned long)*length*

Sets the length of the range and returns **self**. The new length is displayed and the start address is adjusted as described in the class description.

setStartAddress:

– **setStartAddress:**(unsigned long)*address*

Sets the start address of the range (adjusted as described in the class description) and returns **self**.

startAddress

– (unsigned long)**startAddress**

Returns the start of the range. This length should be set at initialization using **setStartAddress:**.

Delegate Methods

rangeDidChange:

– **rangeDidChange:***sender*

Informs the delegate that the range changed.

IOAudio

Inherits From: IODirectDevice : IODevice : Object

Declared In: driverkit/IOAudio.h

Class Description

IOAudio is an abstract class for controlling sound cards. It works closely with the Sound Kit, interpreting messages from user-level programs into method invocations in the driver. IOAudio has three threads—one that listens for messages from user-level programs, one that waits for sound-related keyboard events such as Insert (which raises the volume), and one that serves as the I/O thread. Only the I/O thread is used to invoke subclass methods that might need access to the hardware.

Audio drivers have some restrictions. Because they're closely tied to the Window Server, for security reasons, you can't start up an audio driver at just any time. Instead, it's easiest to reboot to load a new version of an audio driver. Because the Sound Kit currently has no way to choose between audio drivers, only one IOAudio driver instance at a time can run.

To play (output) sound data, IOAudio mixes together the data (obtained from NXPlayStreams) into a circular DMA buffer. If a DMA transfer isn't already in progress, IOAudio invokes **startDMAForChannel:read:buffer:bufferSizeForInterrupts:** (a hardware-specific method). After the number of bytes specified by **bufferSizeForInterrupts** has been transferred, the hardware interrupts; IOAudio zeros out the just-transferred part of the buffer and puts more data into it, if possible. In this way, DMA proceeds continuously until no more data is left to be transferred. When no more data is left (all the NXPlayStreams have completed), IOAudio invokes **stopDMAForChannel:read:**.

Note: The word "channel" has two meanings in sound-related API. It can refer to a DMA channel, or to a *sound channel*. A sound channel is a transmission path for sound. IOAudio currently supports either 1 (mono) or 2 (stereo) sound channels for each DMA transfer.

The sample rate, data encoding, and number of sound channels used for a DMA transfer remain the same from the time **startDMA...** is invoked until the time **stopDMA...** is invoked. Their values are taken from the first NXPlayStream associated with the DMA transfer.

Recording sound data is similar to playing it. One DMA buffer exists for playing sound, and one for recording it. The buffers can share a DMA channel, or they can each have their own. Either way, IOAudio currently schedules transfers on only one channel at a time; that is, simultaneous playback and recording isn't allowed. In the future, support may be added for using both channels simultaneously.

Warning: Currently, the DMA buffer size is 64KB for ISA-based systems and 128K for EISA-based systems, and the interrupt interval is 8KB. You should *not* depend on either the size or number of these buffers—they will change in future releases.

Implementing a Subclass

Your subclass of IOAudio must implement the following methods:

- probe: (IODevice class method)
- reset
- startDMAForChannel:read:buffer:bufferSizeForInterrupts:
- stopDMAForChannel:read:
- interruptClearFunc (and its associated function)
- interruptOccurredForInput:forOutput:
- channelCountLimit
- getDataEncodings:count:
- getSamplingRatesLow:high:
- getSamplingRates:count:

Your subclass should implement the following methods if the hardware supports the associated feature. For example, if your hardware supports loudness enhancement, you should implement **updateLoudnessEnhanced**.

- updateLoudnessEnhanced
- updateInputGainLeft
- updateInputGainRight
- updateOutputMute
- updateOutputAttenuationLeft
- updateOutputAttenuationRight

Besides implementing the methods listed above, you might also need to implement the following:

- acceptsContinuousSamplingRates
- timeoutOccurred

Note: In the future, subclasses may be able to override methods that interpret NXSoundParameterTags passed from user-level programs. This mechanism will allow your subclass to interpret device-specific parameters.

Instance Variables

None declared in this class.

Method Types

Creating and freeing instances
- initFromDeviceDescription:
- free
- reset

Starting and stopping DMA
- startDMAForChannel:read:buffer:bufferSizeForInterrupts:
- stopDMAForChannel:read:

Getting DMA buffer address and size
- getInputChannelBuffer:size:
- getOutputChannelBuffer:size:

Handling interrupts
- interruptOccurredForInput:forOutput:
- interruptClearFunc

Getting notification of I/O thread difficulties
- timeoutOccurred

Getting and setting information about sound channels
- channelCountLimit
- isInputActive
- isOutputActive

Getting supported sampling rates
- acceptsContinuousSamplingRates
- getSamplingRates:count:
- getSamplingRatesLow:high:

Getting supported data encodings
- getDataEncodings:count:

Getting device settings
- channelCount
- dataEncoding
- sampleRate

Determining what hardware settings are or should be
- inputGainLeft
- inputGainRight
- isOutputMuted
- isLoudnessEnhanced
- outputAttenuationLeft

	– outputAttenuationRight
Setting hardware state	– updateInputGainLeft – updateInputGainRight – updateOutputMute – updateLoudnessEnhanced – updateOutputAttenuationLeft – updateOutputAttenuationRight

Instance Methods

acceptsContinuousSamplingRates

– (BOOL)**acceptsContinuousSamplingRates**

Returns NO. Drivers that accept continuous sampling rates, as opposed to accepting a few, discrete sampling rates, should implement this method so that it returns YES. For example, if a device has a low rate of 2000 Hz and a high rate of 44100 Hz and supports every sampling rate in between, its implementation of this method should return YES.

See also: – **getSamplingRates:**, – **getSamplingRatesLow:High:**

channelCount

– (unsigned int)**channelCount**

Returns the number of sound channels to be used for the audio data that's about to be played or recorded. This value, which can be either 1 (for mono) or 2 (for stereo), is determined during mixing and is set before **startDMAForChannel:...** is invoked.

Note: The number of sound channels has nothing to do with the number of DMA channels used by the device.

See also: – **dataEncoding**, – **sampleRate**

channelCountLimit

– (unsigned int)**channelCountLimit**

Returns zero. Drivers must implement this method so that it returns either 1 (if only mono is supported) or 2 (if both mono and stereo are supported).

See also: – **channelCount**

dataEncoding

– (NXSoundParameterTag)**dataEncoding**

Returns the data encoding to be used for the audio data that's about to be played or recorded. This value is determined during mixing and is set before **startDMAForChannel:...** is invoked. Possible values (defined in the header file **soundkit/NXSoundParameterTags.h**) are currently NX_SoundStreamDataEncoding_Linear16, NX_SoundStreamDataEncoding_Linear8, NX_SoundStreamDataEncoding_Mulaw8, and NX_SoundStreamDataEncoding_Alaw8.

See also: – **channelCount**, – **sampleRate**

free

– **free**

Frees the instance and returns **nil**.

getDataEncodings:count:

– (void)**getDataEncodings:**(NXSoundParameterTag *)*encodings*
count:(unsigned int *)*numEncodings*

Returns zero in *numEncodings*. Subclasses must override this method to supply an array of supported data encodings. Possible values (defined in the header file **soundkit/NXSoundParameterTags.h**) are currently NX_SoundStreamDataEncoding_Linear16, NX_SoundStreamDataEncoding_Linear8, NX_SoundStreamDataEncoding_Mulaw8, and NX_SoundStreamDataEncoding_Alaw8. Below is an example of implementing this method. Note that you don't have to allocate memory for *encodings*; it already has enough space to hold all possible encodings.

```
- (void)getDataEncodings: (NXSoundParameterTag *)encodings
                   count:(unsigned int *)numEncodings
{
    encodings[0] = NX_SoundStreamDataEncoding_Linear16;
    encodings[1] = NX_SoundStreamDataEncoding_Linear8;
    *numEncodings = 2;
}
```

getInputChannelBuffer:size:

– (void)**getInputChannelBuffer:**(void *)*address* **size:**(unsigned int *)*byteCount*

Gets the starting address and size of the (already allocated) DMA buffer for the input channel. This method allows the driver to access data in the audio buffer directly.

See also: – **getOutputChannelBuffer:size:**

getOutputChannelBuffer:size:

– (void)**getOutputChannelBuffer:**(void *)*address* **size:**(unsigned int *)*byteCount*

Gets the starting address and size of the (already allocated) DMA buffer for the output channel. This method allows the driver to access data in the audio buffer directly.

See also: – **getInputChannelBuffer:size:**

getSamplingRates:count:

– (void)**getSamplingRates:**(int *)*rates* **count:**(unsigned int *)*numRates*

Returns zero in *numRates*. Subclasses must override this method to supply the supported sampling rates in *rates* array, which has room for up to 256 entries. If the driver supports continuous sampling rates, this method should return some common sampling rates, as shown below.

```
- (void)getSamplingRates:(int *)rates
                   count:(unsigned int *)numRates
{
    /* Return a few common rates */
    rates[0] = 2000;
    rates[1] = 8000;
    rates[2] = 11025;
    rates[3] = 16000;
    rates[4] = 22050;
    rates[5] = 32000;
    rates[6] = 44100;
    *numRates = 7;
}
```

See also: – **acceptsContinuousSamplingRates**, – **getSamplingRatesLow:High:**

getSamplingRatesLow:high:

– (void)**getSamplingRatesLow:**(int *)*lowRate* **high:**(int *)*highRate*

Returns zero in *lowRate* and *highRate*. Subclasses must override this method to supply their highest and lowest supported sampling rates. Here’s an example of implementing this method.

```
- (void)getSamplingRatesLow:(int *)lowRate
                       high:(int *)highRate
```

```
{
    *lowRate = 2000;
    *highRate = 44100;
}
```

See also: – **acceptsContinuousSamplingRates**, – **getSamplingRates:**

initFromDeviceDescription:

– **initFromDeviceDescription:***description*

Initializes a newly allocated IOAudio instance. Subclasses don't generally override this method; they merely invoke it in their **probe:** method. Subclasses perform device-specific initialization in their implementation of the **reset** method.

IOAudio's implementation of **initFromDeviceDescription:** invokes **super**'s version of **initFromDeviceDescription:**, invokes **attachInterruptPort**, sets the interrupt port to have a maximum backlog, and then performs the **reset** method. Next, it creates and initializes the private objects that perform much of the driver's work, creates private ports, and forks threads to listen to requests on the ports. Finally, it invokes **registerDevice**. Returns **nil** if initialization was unsuccessful; otherwise, returns the IOAudio instance.

inputGainLeft

– (unsigned int)**inputGainLeft**

Returns the general scaling factor that's applied to the left channel of the incoming sound. This value can be anywhere from 0 to 32768, where 0 is no gain and 32768 is maximum gain. User-level programs specify the gain using the Sound Kit. To support input gain, you must implement **updateInputGainLeft** and **updateInputGainRight**.

See also: – **inputGainRight**

inputGainRight

– (unsigned int)**inputGainRight**

Returns the general scaling factor that's applied to the right channel of the incoming sound. This value can be anywhere from 0 to 32768, where 0 is no gain and 32768 is maximum gain. User-level programs specify the gain using the Sound Kit. To support input gain, you must implement **updateInputGainLeft** and **updateInputGainRight**.

See also: – **inputGainLeft**

interruptClearFunc

– (IOAudioInterruptClearFunc)**interruptClearFunc**

Does nothing and returns zero. Subclasses must implement this method so that it returns the address of a function that clears interrupts on the card. The function is called only when the audio system needs to guarantee that your card has no pending interrupts. If you don't implement this method and function, your card is likely to suffer from poor performance with some applications. The function runs at interrupt level, so it must not block.

Here's an example of implementing this method.

```
static void clearInterrupts(void)
{
    /* Driver-specific code that clears the card's interrupt
     * register(s) goes here. */
}

- (IOAudioInterruptClearFunc) interruptClearFunc
{
    return clearInterrupts;
}
```

interruptOccurredForInput:forOutput:

– (void)**interruptOccurredForInput:**(BOOL *)*serviceInput*
forOutput:(BOOL *)*serviceOutput*

Notifies the instance that an interrupt occurred for its hardware. The IOAudio version of this method generates an error message; each subclass must implement this method.

The subclass implementation of this method should try to determine whether the hardware really has interrupted. If so, this method should clear the card's interrupt state, set *serviceInput* to YES if the interrupt was for input, and set *serviceOutput* to YES if the interrupt was for output. (The values of *serviceInput* and *serviceOutput* are initialized to NO.)

After invoking this method, IOAudio checks whether any more data is available for DMA on the channels that require service. If none is available, **stopDMAForChannel:read:** is invoked. IOAudio always invokes this method from the I/O thread.

isInputActive

– (BOOL)**isInputActive**

Returns YES if data is being read from the hardware using DMA; otherwise, returns NO.

See also: – **isOutputActive**

isLoudnessEnhanced

– (BOOL)**isLoudnessEnhanced**

Returns YES if loudness is enhanced; otherwise, returns NO. Loudness enhancement refers to the ability of some hardware to help compensate for the decreased sensitivity of the human ear by boosting the gain at low and high frequencies as the volume is decreased. User-level programs specify whether to use loudness enhancement with the NX_SoundDeviceOutputLoudness parameter. To support loudness enhancement, you must implement **updateLoudnessEnhanced**.

isOutputActive

– (BOOL)**isOutputActive**

Returns YES if data is being sent to the hardware using DMA; otherwise, returns NO.

See also: – **isInputActive**

isOutputMuted

– (BOOL)**isOutputMuted**

Returns YES if output is muted; otherwise, returns NO. The user can mute audio output by holding down the Command key and pressing the Delete key. User-level programs can mute output using the Sound Kit.

See also: – **updateOutputMute**

outputAttenuationLeft

– (int)**outputAttenuationLeft**

Returns the attenuation setting of the left channel of the device. The user modifies the left and right attenuation simultaneously using the Volume slider in the Preferences application or with the Insert and Delete keys on the keyboard. User-level programs can specify the attenuation using the Sound Kit. The range is -84 decibels (inaudible) to 0 decibels (no attenuation).

See also: – **updateOutputAttenuationLeft**, – **outputAttenuationRight**

outputAttenuationRight

– (int)**outputAttenuationRight**

Returns the attenuation setting of the right channel of the device. The user modifies the left and right attenuation simultaneously using the Volume slider in the Preferences application or with the Insert and Delete keys on the keyboard. User-level programs can specify the attenuation using the Sound Kit. The range is -84 decibels (inaudible) to 0 decibels (no attenuation).

See also: – **updateOutputAttenuationRight**, – **outputAttenuationLeft**

reset

– (BOOL)**reset**

Generates an error message and returns NO. Subclasses must implement this method so that it resets and initializes the hardware. This method is invoked from **initFromDeviceDescription:**, as described above.

This method should initialize basic information by invoking **setName:** and **setDeviceKind:**. It should then check whether its interrupt (IRQ) and DMA channels (all obtained from its IODeviceDescription) have valid values. After initializing the hardware, this method should disable its DMA channels and then set any DMA parameters necessary, such as the transfer width.

This method should return YES on success; otherwise, it should return NO, which will cause **initFromDeviceDescription:** to return **nil**.

See also: – **initFromDeviceDescription:**, – **setName:** (IODevice), – **setDeviceKind:** (IODevice)

sampleRate

– (unsigned int)**sampleRate**

Returns the sample rate to be used for the audio data that's about to be played or recorded. This value is determined during mixing and is set before **startDMAForChannel:...** is invoked.

See also: – **channelCount**, – **dataEncoding**

startDMAForChannel:read:buffer:bufferSizeForInterrupts:

– (BOOL)**startDMAForChannel:**(unsigned int)*localChannel*
read:(BOOL)*isRead*
buffer:(IODMABuffer)*buffer*
bufferSizeForInterrupts:(unsigned int)*bufferSize*

Generates an error message and returns NO. Subclasses must override this method.

This method should perform DMA after configuring the hardware to reflect the values returned by **sampleRate**, **dataEncoding**, and **channelCount**. The DMA should be set up so that it generates an interrupt after every *bufferSize* byte interval. If *isRead* is YES, then the DMA is from the card to memory; otherwise, DMA is from memory to the card. See the example IOAudio driver for an implementation of this method.

IOAudio invokes this method from the I/O thread. You should never invoke this method in an IOAudio subclass implementation.

This method should return YES if it started DMA successfully; otherwise, it should return NO.

See also: – **startDMAForBuffer:channel** (IODirectDevice architecture-specific category), – **enableChannel** (IODirectDevice architecture-specific category), – **enableAllInterrupts** (IODirectDevice architecture-specific category)

stopDMAForChannel:read:

– (void)**stopDMAForChannel:**(unsigned int)*localChannel* **read:**(BOOL)*isRead*

Generates an error message. Subclasses must override this method.

This method should disable the specified DMA channel, disable interrupts, and do anything else necessary to stop the DMA in progress on *localChannel*. See the example IOAudio driver for an implementation of this method.

IOAudio invokes this method from the I/O thread. You should never invoke this method in an IOAudio subclass implementation.

This method is invoked when an interrupt occurs and no more data is available to be transferred. It's also invoked any time that **startDMAForChannel:...** returns NO.

See also: – **startDMAForChannel:read:buffer:bufferSizeForInterrupts:**, – **disableChannel** (IODirectDevice architecture-specific category), – **disableAllInterrupts** (IODirectDevice architecture-specific category)

timeoutOccurred

– (void)**timeoutOccurred**

Notifies the instance that although a DMA transaction is in progress, no interrupts have been detected for a long time (currently one second). The IOAudio version of this method does nothing; each subclass can implement it or not.

The subclass implementation of this method might reset the hardware. IOAudio invokes this method from the I/O thread.

updateInputGainLeft

– (void)**updateInputGainLeft**

Does nothing. Subclasses should implement this method so that it updates the hardware to the value returned by **inputGainLeft**. You generally have to convert the device-independent value returned by **inputGainLeft** to the appropriate value for your device.

```
- (void) updateInputGainLeft
{
    /* Convert gain (0 - 32768) into attenuation (0 - 31). */
    unsigned int gain = [self inputGainLeft] / 1057;

    setInputAttenuation(MICROPHONE, LEFT_CHANNEL,
                        (unsigned char) gain);
    setInputAttenuation(EXTERNAL_LINE_IN, LEFT_CHANNEL,
                        (unsigned char) gain);
}
```

IOAudio invokes this method from the I/O thread.

See also: – **updateInputGainRight**

updateInputGainRight

– (void)**updateInputGainRight**

Does nothing. Subclasses should implement this method so that it updates the hardware to match the value returned by **inputGainRight**. You generally have to convert the device-independent value returned by **inputGainRight** to the appropriate value for your device. IOAudio invokes this method from the I/O thread.

See also: – **updateInputGainLeft**

updateLoudnessEnhanced

– (void)**updateLoudnessEnhanced**

Does nothing. Subclasses that support loudness enhancement should implement this method so that it updates the hardware to match the value returned by **isLoudnessEnhanced**. IOAudio invokes this method from the I/O thread.

updateOutputAttenuationLeft

– (void)**updateOutputAttenuationLeft**

Does nothing. Subclasses should implement this method so that it updates the

hardware to match the value returned by **outputAttenuationLeft**. You generally have to convert the device-independent value returned by **outputAttenuationLeft** to the appropriate value for your device. Here's an example of implementing this method.

```
- (void) updateOutputAttenuationLeft
{
    /* Get the software value and convert it from the software
range
     * (0 - -84) to the device range (0 - 31, for this card). */
    unsigned int attenuation = [self outputAttenuationLeft] + 84;
    attenuation = ((attenuation * 10)/27);

    /* Device-specific code sets the output attention of the left
     * sound channel to the value of the attenuation variable. */
}
```

IOAudio invokes this method from the I/O thread.

See also: – **updateOutputAttenuationRight**

updateOutputAttenuationRight

– (void)**updateOutputAttenuationRight**

Does nothing. Subclasses should implement this method so that it updates the hardware to match the value returned by **outputAttenuationRight**. You generally have to convert the device-independent value returned by **outputAttenuationRight** to the appropriate value for your device. IOAudio invokes this method from the I/O thread.

See also: – **updateOutputAttenuationLeft**

updateOutputMute

– (void)**updateOutputMute**

Does nothing. Subclasses should implement this method so that it mutes the output if **isOutputMuted** returns YES and unmutes the output if **isOutputMuted** returns NO. IOAudio invokes this method from the I/O thread.

IOConfigTable

Inherits From: Object

Declared In: driverkit/IOConfigTable.h

Class Description

IOConfigTable is used at both kernel and user level to get configuration information about particular devices and the system as a whole. Which IOConfigTables a software module can obtain, as well as the way it obtains them, depends on whether the module is a driver and whether it's at user or kernel level.

IODevices inside and outside the kernel can get their own IOConfigTables using IODeviceDescription's **configTable** method. User-level programs can use the methods **newForDriver:unit:**, **newDefaultTableForDriver:**, **tablesForBootDrivers**, and **tablesForInstalledDrivers** to get IOConfigTables for specific drivers. Both user-level and kernel-level modules can use **newFromSystemConfig** to get an IOConfigTable that describes the system-wide configuration.

Each IOConfigTable describes one hardware device. Thus, each IOConfigTable (except the system-wide one) corresponds to one IODevice object, which may or may not exist at the time the IOConfigTable is created. At least one IOConfigTable can be created for each driver that's listed in the system-wide IOConfigTable as an active or boot driver. Specifically, each IOConfigTable corresponds to an **Instance*n*.table** file in the driver's bundle. See Chapter 4, "Configuring Drivers," for more information on driver bundles.

Note: From an IODevice's viewpoint, its IOConfigTable doesn't change. The IOConfigTable keeps the values that were in the corresponding **Instance*n*.table** when the driver was loaded. To see changes in **Instance*n*.table**, the driver must be reloaded. However, the user-level version of driver IOConfigTables can be synchronized with the corresponding **Instance*n*.table** at any time. This means that a user-level program might see different values in a driver's IOConfigTable than the driver sees.

Instance Variables

None declared in this class.

Method Types

Creating and freeing instances	+ newForDriver:unit: + newDefaultTableForDriver: + newFromSystemConfig + tablesForBootDrivers + tablesForInstalledDrivers – free
Getting information	– valueForStringKey:
Getting the driver bundle	– driverBundle

Class Methods

newDefaultTableForDriver:unit:

+ **newDefaultTableForDriver:**(const char *)*driverName*

Creates, if necessary, and returns the default IOConfigTable for the specified driver. The *driverName* corresponds to the value returned by IODevice's **name** class method.

Note: This method can be used only by user-level programs.

newForDriver:unit:

+ **newForDriver:**(const char *)*driverName* **unit:**(int)*unit*

Creates, if necessary, and returns the IOConfigTable for the specified driver and unit. The *driverName* and *unit* values correspond to the values returned by IODevice's **name** class method and **unit** instance method, respectively.

Note: This method can be used only by user-level programs.

newFromSystemConfig

+ **newFromSystemConfig**

Creates, if necessary, and returns the IOConfigTable describing the system-wide configuration. This IOConfigTable's values are initialized at boot time, and don't change until the system is rebooted.

tablesForBootDrivers

+ (List *)**tablesForBootDrivers**

Creates, if necessary, and returns IOConfigTables, one for each device that was loaded into the system at boot time. This method might return some IOConfigTables that aren't returned by **tablesForInstalledDrivers**, since this method detects drivers that were loaded due to user action at boot time.

Note: This method can be used only by user-level programs.

tablesForInstalledDrivers

+ (List *)**tablesForInstalledDrivers**

Creates, if necessary, and returns IOConfigTables, one for each device that has been loaded into the system. This method knows only about those devices that are specified with the system configuration table's "Active Drivers" and "Boot Drivers" keys. It does *not* detect drivers that were loaded due to user action at boot time. To get the IOConfigTables for those drivers, you can use **tablesForBootDrivers**.

Note: This method can be used only by user-level programs.

Instance Methods

driverBundle

– (NXBundle *)**driverBundle**

Creates, if necessary, and returns the NXBundle corresponding to the driver this IOConfigTable describes. The bundle corresponds to the driver's **.config** directory under **/usr/Devices**.

Note: This method can be used only by user-level programs.

free

– **free**

Frees the object and returns **nil**.

valueForStringKey:

– (const char *)**valueForStringKey:**(const char *)*key*

Returns the string value associated with the specified key. Kernel-level drivers should free this string when it's no longer needed, using **freeString:**. User-level programs

should *not* free this string.

IODevice

Inherits From: Object

Declared In: driverkit/IODevice.h

Class Description

IODevice is an abstract class that is the superclass of all device driver classes. Functionality provided by IODevice includes:

- Standard driver startup and connection to driver objects
- Standard ways of getting and setting driver parameters
- Getting and setting standard information such as the instance's unit number
- Mapping IOReturn values to strings and to UNIX error numbers
- Adding and removing drivers from UNIX device switch tables
- Getting the Driver Kit version that the IODevice was compiled under

Getting and Setting Parameters

The IODevice methods **getCharValues:forParameter:count:**, **getIntValues:forParameter:count:**, **setIntValues:forParameter:count:**, and **setCharValues:forParameter:count:** provide a general, extensible means for user-level programs to get and set device-specific parameters for drivers that reside in the kernel. The general scheme is as follows:

- A parameter's value is either an array of **int**s or an array of **char**s. The maximum number of elements in a parameter array is IO_MAX_PARAMETER_ARRAY_LENGTH, a system constant (currently 512 though you shouldn't count on this value).
- Parameters are specified with human-readable strings. For example, the parameter for getting an IODevice's unit number is named "IOUnit" and defined as the constant IO_UNIT.
- Any subclass of IODevice can define any parameters it wishes. Any class that does so must implement the appropriate methods by which the parameters can be accessed (**getIntValues:...**, for example). If such a method is invoked with a parameter name that the class does not recognize, the method invocation should be passed up to **super**. If no classes recognize the parameter name, IODevice returns IO_R_UNSUPPORTED.

- By sending messages to an IODeviceMaster object, a user program can find the desired instance of a device driver and get or set device-specific parameters.

Implementing a Subclass

Subclasses of IODevice that are indirect or direct device drivers must implement the following methods:

+ **deviceStyle**
+ **probe:**
– **initFromDeviceDescription:**

Indirect device drivers also need to implement the **requiredProtocols** class method.

Note: If your class's direct superclass isn't IODevice, check the documentation for the superclass—it may implement some or all of these methods for you.

During initialization, indirect and direct drivers must invoke the following methods:

– **registerDevice**
– **setDeviceKind:**
– **setLocation:**
– **setName:**

The **registerDevice** method should be invoked at the end of initialization. Generally, indirect and direct drivers also invoke **setUnit:**.

Instance Variables

None declared in this class.

Method Types

Creating, initializing, and freeing instances	+ probe: – init – initFromDeviceDescription: – free
Registering the class	+ deviceStyle + registerClass: + unregisterClass: + requiredProtocols

Registering the instance
– registerDevice
– unregisterDevice

Getting and setting standard information
– setDeviceKind:
– deviceKind
– setLocation:
– location
– setName:
– name
– setUnit:
– unit

Converting an IOReturn value
+ stringFromReturn:
– stringFromReturn:
– errnoFromReturn:

Adding and removing the driver from UNIX device switch tables
+ addToBdevswFromDescription:open:close:strategy: dump:psize:isTape:
+ addToCdevswFromDescription:open:close:read:write: ioctl:stop:reset:select:mmap:getc:putc:
+ blockMajor
+ characterMajor
+ removeFromBdevsw
+ removeFromCdevsw
+ setBlockMajor:
+ setCharacterMajor:

Getting the Driver Kit version of the IODevice
+ driverKitVersion
+ driverKitVersionForDriverNamed:

Getting and setting parameter values
– setCharValues:forParameter:count:
– getCharValues:forParameter:count:
– setIntValues:forParameter:count:
– getIntValues:forParameter:count:

Class Methods

addToBdevswFromDescription:open:close:strategy:dump:psize:isTape:

+ (BOOL)**addToBdevswFromDescription:**(id)*deviceDescription*
open:(IOSwitchFunc)*openFunc*
close:(IOSwitchFunc)*closeFunc*
strategy:(IOSwitchFunc)*strategyFunc*
dump:(IOSwitchFunc)*dumpFunc*
psize:(IOSwitchFunc)*psizeFunc*
isTape:(BOOL)*isTape*

Adds the specified values to the **bdevsw** table. Drivers that have UNIX block entry points should use this method during initialization.

The major number to use is taken from the value of the "Block Major" key in the class's configuration table. If "Block Major" isn't specified, the first available major number is used. If the entry is successfully added, this method invokes **setBlockMajor:**.

If the entry was successfully added, this method returns YES; otherwise, it logs an error message and returns NO.

See also: + **blockMajor**, + **addToCdevswFromDescription:open:close:read:write: ioctl:stop:reset:select:mmap:getc:putc:**

addToCdevswFromDescription:open:close:read:write:ioctl:stop:reset:select:mmap:getc:putc:

+ (BOOL)**addToCdevswFromDescription:**(id)*deviceDescription*
open:(IOSwitchFunc)*openFunc*
close:(IOSwitchFunc)*closeFunc*
read:(IOSwitchFunc)*readFunc*
write:(IOSwitchFunc)*writeFunc*
ioctl:(IOSwitchFunc)*ioctlFunc*
stop:(IOSwitchFunc)*stopFunc*
reset:(IOSwitchFunc)*resetFunc*
select:(IOSwitchFunc)*selectFunc*
mmap:(IOSwitchFunc)*mmapFunc*
getc:(IOSwitchFunc)*getcFunc*
putc:(IOSwitchFunc)*putcFunc*

Adds the specified values to the cdevsw table. Drivers that have UNIX character entry points should use this method during initialization.

The major number to use is taken from the value of the "Character Major" key in the class's configuration table. If "Character Major" isn't specified, the first available major number is used. If the entry is successfully added, this method invokes **setCharacterMajor:**.

If the entry was successfully added, this method returns YES; otherwise, it logs an error message and returns NO.

See also: + **characterMajor**, + **addToBdevswFromDescription:open:close:strategy: dump:psize:isTape:**

blockMajor

+ (int)**blockMajor**

Returns the block major number associated with this driver, or −1 if this driver has no block major number. The block major number is set using **setBlockMajor:**, which is invoked by **addToBdevswFromDescription...**.

characterMajor

+ (int)**characterMajor**

Returns the character major number associated with this driver, or −1 if this driver has no character major number. The character major number is set using **setCharacterMajor:**, which is invoked by **addToCdevswFromDescription...**.

deviceStyle

+ (IODeviceStyle)**deviceStyle**

Implemented by subclasses to return the basic style of driver (IO_DirectDevice, IO_IndirectDevice, or IO_PseudoDevice). The meaning of direct, indirect, and pseudo device drivers is discussed in Chapters 1 and 2.

See also: + **deviceStyle** (IODirectDevice)

driverKitVersion

+ (int)**driverKitVersion**

Returns the version of the currently running DriverKit objects. The Driver Kit compares this value to the value returned by **driverKitVersionForDriverNamed:** to determine whether the driver is compatible with the driver environment.

driverKitVersionForDriverNamed:

+ (int)**driverKitVersionForDriverNamed:**(char *)*driverName*

Returns the version of the Driver Kit that the specified driver was compiled for. The Driver Kit compares this value to the value returned by **driverKitVersion** to determine whether the driver is compatible with the driver environment.

probe:

+ (BOOL)**probe:**(id)*deviceDescription*

Does nothing and returns NO. This method is invoked by the kernel (in the context of the kernel I/O task) to conditionally instantiate an instance of an IODevice subclass.

This method should be implemented by every direct and indirect driver. It should determine whether it needs to instantiate itself, examining the hardware if appropriate. It should then allocate and initialize all the necessary instances for the specified *deviceDescription*. Should return YES if any IODevice objects were created; otherwise, this method should return NO.

See Chapter 1 for information on when **probe:** is invoked.

See also: – **initFromDeviceDescription:**

registerClass:

+ (void)**registerClass:***aClass*

Adds the specified class to the kernel list of device driver classes.

See also: + **unregisterClass:**

removeFromBdevsw

+ (BOOL)**removeFromBdevsw**

Removes the driver's entry from the bdevsw table. This method finds the driver's entry in the table by invoking **blockMajor**. If **blockMajor** is −1, this method does nothing and returns NO. Otherwise, this method sets the block major number to −1 (using **setBlockMajor:**) and returns YES.

removeFromCdevsw

+ (BOOL)**removeFromCdevsw**

Removes the driver's entry from the cdevsw table. This method finds the driver's entry in the table by invoking **characterMajor**. If **characterMajor** is −1, this method does nothing and returns NO. Otherwise, this method sets the character major number to −1 (using **setCharacterMajor:**) and returns YES.

requiredProtocols

+ (Protocol **)**requiredProtocols**

Returns NULL. Indirect device drivers should implement this method to return a NULL-terminated list of the protocols to which associated drivers must conform. Kernel-level indirect devices must implement this.

setBlockMajor:

+ (void)**setBlockMajor:**(int)*bmajor*

Sets the driver's block major number. You usually don't have to invoke this, since it's invoked by **addToBdevswFromDescription:...**.

setCharacterMajor:

+ (void)**setCharacterMajor:**(int)*cmajor*

Sets the driver's character major number. You usually don't have to invoke this, since it's invoked by **addToCdevswFromDescription:...**.

stringFromReturn:

+ (const char *)**stringFromReturn:**(IOReturn)*returnValue*

Returns a text string that describes the specified IOReturn value.

See also: – **stringFromReturn:**

unregisterClass:

+ (void)**unregisterClass:***classId*

Removes the specified class from the kernel list of device driver classes. This method is invoked when a class is being removed from the address space of a program such as the kernel.

See also: + **registerClass:**

Instance Methods

deviceKind

– (const char *)**deviceKind**

Returns a string that identifies the object in general terms. For example, IOSCSIDisk objects return “SCSIDisk”. See the description of **setDeviceKind:** for more information.

See also: – **setDeviceKind:**

errnoFromReturn:

– (int)**errnoFromReturn:**(IOReturn)*returnValue*

Returns a UNIX error number that corresponds to the specified IOReturn value. Subclasses that add additional IOReturn values should override this method and send an **errnoFromReturn:** to **super** for IOReturn values that the subclass doesn’t handle.

free

– **free**

Frees resources used by the IODevice and returns **nil**.

getCharValues:forParameter:count:

– (IOReturn)**getCharValues:**(unsigned char *)*array*
forParameter:(IOParameterName)*parameter*
count:(unsigned int *)*count*

Gets the array of character values associated with *parameter*. IODevice accepts the following character parameters: IO_CLASS_NAME (which returns [[self class] name]), IO_DEVICE_NAME (which returns [self name]), and IO_DEVICE_KIND (which returns [self deviceKind]).

Subclasses should override this method if they support parameters not understood by the superclass. Here’s an example of overriding this method:

```
- (IOReturn)getCharValues   : (unsigned char *)parameterArray
               forParameter : (IOParameterName)parameterName
                      count : (unsigned int *)count
{
    const char  *param;
    unsigned int length;
    unsigned int maxCount = *count;

    if(strcmp(parameterName, my_PARAMETER_NAME) == 0){
        param = _myParameter;  /* _myParameter is an instance var
*/
        length = strlen(param);
        if(length >= maxCount) {
```

```
            length = maxCount - 1;
        }
        *count = length + 1;
        strncpy(parameterArray, param, length);
        parameterArray[length] = '\0';
        return IO_R_SUCCESS;
    }
    else {
    /* Pass parameters we don't recognize to our superclass. */
        return [super getCharValues:parameterArray
            forParameter:parameterName count:count];
    }
}
```

Returns IO_R_SUCCESS if *parameter* is a valid parameter with character values that can be read; otherwise, returns IO_R_UNSUPPORTED.

See also: – **getIntValues:forParameter:count:**, – **setCharValues:forParameter:count:**

getIntValues:forParameter:count:

– (IOReturn)**getIntValues:**(unsigned int *)*array*
forParameter:(IOParameterName)*parameter*
count:(unsigned int *)*count*

Returns IO_R_UNSUPPORTED. Subclasses should implement this method if necessary to return (in *array*) the array of integer values associated with *parameter*. See **getCharValues:forParameter:count:** for an example of implementing this kind of method. This method should return IO_R_SUCCESS if *parameter* is a valid parameter with integer values that can be read; otherwise, it should return IO_R_UNSUPPORTED.

See also: – **getCharValues:forParameter:count:**, – **setIntValues:forParameter:count:**

init

– **init**

Initializes and returns a newly allocated IODevice. Returns **self** if successful; otherwise, returns **nil**.

Note: Direct and indirect drivers should use **initFromDeviceDescription:** instead of this method.

initFromDeviceDescription:

– **initFromDeviceDescription:***deviceDescription*

Does nothing and returns **self**. Subclasses that implement this method should have it initialize and return a newly allocated instance of the subclass, using the information from *deviceDescription*. This method should return **nil** on error.

See also: – **initFromDeviceDescription:** (IODirectDevice)

location

– (const char *)**location**

Returns the device-specific location of the IODevice—for example, "0xf7f04000". See the description of **setLocation:** for information on how this location is used.

See also: – **setLocation:**

name

– (const char *)**name**

Returns the device-specific name of the IODevice—for example, "sd0a". See the description of **setName:** for information on how the name is used.

See also: – **setName:**

registerDevice

– **registerDevice**

Registers the IODevice in the current name space and adds a string to the system log that announces the device's registration. The IODevice must be ready to perform I/O, its name must have been set already using **setName:**, and its location (set with **setLocation:**) must be either valid or NULL.

This method also probes all indirect IODevices that require this object's protocols, giving them a chance to connect to this object.

Each IODevice should invoke this method at the end of its initialization. Returns **self**.

Note: I/O can begin before this method returns.

See also: – **unregisterDevice**

setCharValues:forParameter:count:

– (IOReturn)**setCharValues:**(unsigned char *)*array*
forParameter:(IOParameterName)*parameter*

count:(unsigned int)*count*

Returns IO_R_UNSUPPORTED. Subclasses should implement this method if necessary to set (from *array*) the array of character values associated with *parameter*. See **getCharValues:forParameter:count:** for an example of implementing this kind of method. This method should return IO_R_SUCCESS if *parameter* is a valid parameter with character values that can be written; otherwise, it should return IO_R_UNSUPPORTED.

See also: – **setIntValues:forParameter:count:**, – **getCharValues:forParameter:count:**

setDeviceKind:

– (void)**setDeviceKind:**(const char *)*type*

Sets a string that identifies the object in general terms. For example, IOFrameBufferDisplay objects have a device kind of "Linear Framebuffer". The string should be no longer than IO_STRING_LENGTH – 1 characters. The standard parameter name IO_DEVICE_KIND refers to this string.

See also: – **deviceKind**

setIntValues:forParameter:count:

– (IOReturn)**setIntValues:**(unsigned int *)*array*
forParameter:(IOParameterName)*parameter*
count:(unsigned int)*count*

Returns IO_R_UNSUPPORTED. Subclasses should implement this method if necessary to set (from *array*) the array of character values associated with *parameter*. See **getCharValues:forParameter:count:** for an example of implementing this kind of method. This method should return IO_R_SUCCESS if *parameter* is a valid parameter with integer values that can be written; otherwise, it should return IO_R_UNSUPPORTED.

See also: – **setCharValues:forParameter:count:**, – **getIntValues:forParameter:count:**

setLocation:

– (void)**setLocation:**(const char *)*location*

Sets the device-specific location of the IODevice—for example, "0xf7f04000". If the location is irrelevant, its value should be set to NULL. The location is used in the system log when this object is registered and unregistered.

See also: – **location**

setName:

– (void)**setName:**(const char *)*name*

Sets the device-specific name of the IODevice—for example, “sd0a”. The name should be no longer than IO_STRING_LENGTH – 1 characters.

The specified name is used to identify this instance. For example, it’s used in the system log when this object is registered and unregistered, and it’s used by the UNIX command **iostat**. The name is also used by user-level programs to find this object, using the IODeviceMaster method **lookUpByDeviceName:objectNumber:deviceKind:**. The standard parameter name IO_DEVICE_NAME refers to this string.

See also: – **name**

setUnit:

– (void)**setUnit:**(unsigned int)*unit*

Sets the IODevice’s unit number, a device-specific number that can be used like a UNIX minor number.

See also: – **unit**

stringFromReturn:

– (const char *)**stringFromReturn:**(IOReturn)*returnValue*

Returns the text string that corresponds to the specified IOReturn value. Subclasses that add additional IOReturn values should override this method and invoke **stringFromReturn:** against the superclass for IOReturn values that the subclass doesn’t handle.

See also: + **stringFromReturn:**

unit

– (unsigned int)**unit**

Returns the IODevice’s unit number, a device-specific number that can be used like a UNIX minor number.

See also: – **setUnit:**

unregisterDevice

– (void)**unregisterDevice**

Removes the IODevice from the current name space.

See also: – **registerDevice**

IODeviceDescription

Inherits From: Object

Declared In: driverkit/IODeviceDescription.h

Class Description

IODeviceDescription objects are used to encapsulate information about an IODevice object. Usually, you need only to pass around IODeviceDescription objects, without creating them, subclassing them, or sending messages to them. The main purpose of an IODeviceDescription object is to describe an IODirectDevice. However, IODeviceDescriptions are also used at **probe:** time to describe indirect drivers (specifically, to specify the IODirectDevice that the indirect driver might want to work with).

Each architecture has its own subclass of IODeviceDescription that contains architecture-specific information:

Architecture	IODeviceDescription Subclass
ISA and EISA Intel-based	IOEISADeviceDescription
PCI	IOPCIDeviceDescription
PCMCIA	IOPCMCIADeviceDescription

Instance Variables

None declared in this class.

Method Types

Getting and setting the list of interrupts

– interrupt
– interruptList
– numInterrupts
– setInterruptList:num:

Getting and setting the list of memory ranges

– memoryRangeList
– numMemoryRanges
– setMemoryRangeList:num:

Getting and setting the port
– devicePort
– setDevicePort:

Getting and setting the direct device
– directDevice
– setDirectDevice:

Getting and setting the configuration table
– configTable
– setConfigTable:

Instance Methods

configTable

– (IOConfigTable *)**configTable**

Returns the table of configuration information for this driver instance.

See also: – **setConfigTable:**

devicePort

– (port_t)**devicePort**

Returns the device port associated with the device. This port is used by the Driver Kit. You shouldn't need to invoke this method if your driver uses only supported Driver Kit API.

Holding send rights to the device port gives a task rights to access a device's registers, to program its DMA channel, and receive interrupt notification. The kernel responds to requests sent on this port to provide these services to the requesting task. Device ports are created early in system initialization and passed out to the appropriate device drivers at configuration time.

See also: – **setDevicePort:**

directDevice

– **directDevice**

If the driver instance described by IODeviceDescription is an indirect device driver,

this method returns the IODevice object to which this driver instance is connected. Usually, the returned object is an IODirectDevice; however, this isn't required. If this IODeviceDescription's object is a direct or pseudo device driver, this method returns **nil**.

See also: – **setDirectDevice:**

interrupt

– (unsigned int)**interrupt**

Returns the first interrupt (IRQ) associated with this device. The return value is undefined if this device has no interrupts associated with it.

See also: – **interruptList**, – **numInterrupts**, – **setInterruptList:num:**

interruptList

– (unsigned int *)**interruptList**

Returns all the interrupts (IRQs) associated with this device. You can get the number of items in the returned array by invoking **numInterrupts**. You should never free the data returned by this method.

See also: – **interrupt**, – **numInterrupts**, – **setInterruptList:num:**

memoryRangeList

– (IORange *)**memoryRangeList**

Returns all the memory ranges associated with this device. You can get the number of items in the returned array by invoking **numMemoryRanges**. You should never free the data returned by this method.

See also: – **numMemoryRanges**, – **setMemoryRangeList:num:**

numInterrupts

– (unsigned int)**numInterrupts**

Returns the total number of interrupts (IRQs) associated with this device.

See also: – **interrupt**, – **interruptList**, – **setInterruptList:num:**

numMemoryRanges

– (unsigned int)**numMemoryRanges**

Returns the total number of memory ranges associated with this device.

See also: – **memoryRangeList**, – **setMemoryRangeList:num:**

setConfigTable:

– (void)**setConfigTable:**(IOConfigTable *)*configTable*

Sets the table of configuration information for this driver instance. In normal use of the Driver Kit, you should never invoke this method.

See also: – **configTable**

setDevicePort:

– (void)**setDevicePort:**(port_t)*devicePort*

Sets the device port for this driver instance. In normal use of the Driver Kit, you should never invoke this method.

See also: – **devicePort**

setDirectDevice:

– (void)**setDirectDevice:***directDevice*

Records *directDevice* as the IODevice object that is connected to the driver instance that this IODeviceDescription describes. In normal use of the Driver Kit, you should never invoke this method.

See also: – **directDevice**

setInterruptList:num:

– (IOReturn)**setInterruptList:**(unsigned int *)*aList* **num:**(unsigned int)*numInterrupts*

Sets the array and number of interrupts (IRQs) associated with this device. You shouldn't normally invoke this method, since it overrides the normal configuration scheme (which is documented in Chapter 4).

See also: – **interrupt**, – **interruptList**, – **numInterrupts**

setMemoryRangeList:num:

– (IOReturn)**setMemoryRangeList:**(IORange *)*aList* **num:**(unsigned int)*numMemoryRanges*

Sets the array and number of memory ranges associated with this device. You shouldn't normally invoke this method, since it overrides the normal configuration scheme (which is documented in Chapter 4).

See also: **– memoryRangeList**, **– numMemoryRanges**

IODeviceInspector

Inherits From: Object

Conforms To: IOConfigurationInspector

Declared In: driverkit/IODeviceInspector.h

Class Description

This class provides the default Configure inspector used for devices. IODeviceInspector lets the user select which resources—DMA channels, interrupts, I/O ports, and memory ranges—a device should use. IODeviceInspector also provides an accessory View, in which you can put additional controls.

You shouldn't need to use this class unless you're providing an accessory View. To provide an accessory View, you should first create the View in Interface Builder and then subclass IODeviceInspector so that it displays the View.

Note: When creating an accessory View, try to keep it no more than 80 pixels high. Configure's window is already about 400 pixels high; adding more than 80 pixels to it means that the window won't fit on the smallest supported screens (which are 640 pixels wide by 480 high).

Implementing a Subclass

To provide an accessory View, you should create an IODeviceInspector subclass that does the following:

- Overrides Object's **init** method so that it loads the nib file that contains the accessory View by invoking **loadMainNibFile** and initializing (but not displaying) the interrupt and DMA matrices.
- Implement the **setTable:** method so that it invokes **[super setTable:]**, invokes **setAccessoryView:** to specify its accessory View, and initializes the accessory View
- Modifies the configuration table as necessary, in response to the user's actions in the accessory View. For example, you might need to insert a key in the configuration table.

Here's an example of changing the configuration table when the user operates a

control. In this case, the control sends a **connectorChanged:** message to its target (which is the IODeviceInspector subclass). The **table** instance variable is the NXStringTable corresponding to the configuration table.

```
- connectorChanged:sender
{
    [table insertKey:CONNECTOR
            value:connectorType[sender selectedTag]];
    return self;
}
```

If you have localizable strings displayed in your accessory View, be careful to use the strings from the driver's configuration bundle, not from the Configure application's bundle. Here's an example taken from an IODeviceInspector subclass's **init** method.

```
#define LOCAL_CONNECTOR_STRING(bundle)
NXLocalStringFromTableInBundle(NULL, bundle, "Connector", NULL,
"The interface connector on the EtherExpress16 adaptor which will
be used to access the network.")
.
.
.
char      buffer[MAXPATHLEN];
NXBundle *myBundle = [NXBundle bundleForClass:[self class]];

[super init];

if (![myBundle getPath:buffer forResource:MYNAME ofType:NIB_TYPE])
{
    [self free];
    return nil;
}
if (![NXApp loadNibFile:buffer owner:self withNames:NO]) {
    [self free];
    return nil;
}
[connectorBox setTitle:LOCAL_CONNECTOR_STRING(myBundle)];
```

Instance Variables

id **accessoryHolder**;

id **statusTitle**;

id **origWindow**;

id **dmaBox**;

id **dmaMatrix**;

id **irqBox**;

id **irqMatrix**;

id **memoryBox**;

id **memoryController**;

id **portsBox**;

id **portsController**;
id **inspectionView**;
id **infoButton**;
id **infoPanel**;
id **infoText**;
NXStringTable ***table**;
int **numInterrupts**;
int **numChannels**;
int **portRangeLength**;
int **memoryRangeLength**;
BOOL **infoTextLoaded**;
BOOL **knowsDetails**;
IOConfigurationConflict **portConflict**;
IOConfigurationConflict **memoryConflict**;
IOConfigurationConflict **totalConflict**;

accessoryHolder	View where the accessory View is placed
statusTitle	At top of window
origWindow	For getting contentView
dmaMatrix	Buttons for DMA channels
dmaBox	In case no DMA channels
irqMatrix	Buttons for IRQ levels
irqBox	In case no IRQ levels
memoryController	Handles ranges
memoryBox	In case no mapped memory
portsController	Handles ranges
portsBox	In case no port addresses
inspectionView	The inspection View
infoButton	Brings up device info panel
infoPanel	Contains text about the device
infoText	Text object for info file
table	The NXStringTable we’re editing
numInterrupts	How many IRQs to configure
numChannels	How many DMA channels to configure
portRangeLength	Number of I/O ports in the range

memoryRangeLength	Length of the memory map
infoTextLoaded	YES if the info panel has been loaded
knowsDetails	YES if we already know the device's requirements
portConflict	I/O port conflict state
memoryConflict	Memory range conflict state
totalConflict	Overall conflict state

Adopted Protocols

IOConfigurationInspector	– inspectionView – resourcesChanged: – setTable:

Method Types

Displaying the IODeviceInspector	– loadMainNibFile – showInfo:
Setting initial resource values	– setNumInterrupts:numChannels:portRangeLength:memoryRangeLength:
Notification of resource changes	– channelOrInterruptPicked: – rangeDidChange:
Customizing the IODeviceInspector	– setAccessoryView:

Instance Methods

channelOrInterruptPicked:

– **channelOrInterruptPicked:***sender*

Notifies the receiver that a DMA channel or interrupt has been picked. IODeviceInspector changes the appearance the associated button and updates the configuration table, if appropriate. Returns **self**.

loadMainNibFile

– **loadMainNibFile**

Loads the nib file for the IODeviceInspector. This method should be invoked by **init**. Returns **nil** on failure; otherwise, returns **self**.

rangeDidChange:

– **rangeDidChange:***sender*

Notifies the receiver that a range of I/O ports or memory has been changed. This method updates the configuration table. Returns **self**.

setAccessoryView:

– **setAccessoryView:***aView*

Adds *aView* to the IODeviceInspector's View hierarchy. The inspector is automatically resized to accommodate *aView*. Returns **self**.

setNumInterrupts:numChannels:portRangeLength: memoryRangeLength:

– **setNumInterrupts:**(int)*numInterrupts*
numChannels:(int)*numChannels*
portRangeLength:(int)*numPorts*
memoryRangeLength:(int)*numMaps*

Invoked once by **setTable:** to initialize the number of each kind of resource that the device uses.

showInfo:

– **showInfo:***sender*

Brings up a panel containing information about the device.

IODeviceMaster

Inherits From: Object

Declared In: driverkit/IODeviceMaster.h

Class Description

IODeviceMaster is a class used by user-level programs to gain access to device driver instances. To use IODeviceMaster, the program uses the **alloc** and **init** methods to obtain and initialize an IODeviceMaster instance. It then attempts to get the object number of the device driver instance using one of the **lookUp...** methods. If successful, it can use this object number to get and set parameters associated with the driver instance.

Programs don't need superuser privileges to get information from IODeviceMaster. However, they do need superuser privileges to be able to set device information (with the **setCharValues:...** and **setIntValues:** methods).

Here's an example of using IODeviceMaster. It's taken from the **DriverInspector** directory located in **/NextLibrary/Documentation/NextDev/Examples/DriverKit**.

```
IOReturn        ret;
IOObjectNumber  objectNumber;
IOString        kind;
IOCharParameter value;
unsigned int    count = IO_MAX_PARAMETER_ARRAY_LENGTH, unit = 0;
char            name[80];
IODeviceMaster *devMaster;

/* Look up the test driver, using IODeviceMaster. */
devMaster = [IODeviceMaster new];
sprintf(name, "%s%d", my_DEVICE_NAME, unit);
ret = [devMaster lookUpByDeviceName:name objectNumber:&objectNumber
    deviceKind:&kind];
if (ret != IO_R_SUCCESS) { /* handle the error */
    fprintf(stderr, "Lookup failed: %s\n",
        [IODevice stringFromReturn:ret]);
    exit(-1);
} else { /* Successfully got the object number */

    /* Get the value of the test driver's parameter. */
    ret = [devMaster getCharValues:value
        forParameter:my_PARAMETER_NAME objectNumber:objectNumber
        count:&count];
    if (ret != IO_R_SUCCESS) { /* handle the error */
        fprintf(stderr, "Failed to get parameter value: %s\n",
            [IODevice stringFromReturn:ret]);
```

```
    exit(-1);
} else /* Successfully got the parameter value */
    printf("Parameter value: %s; count = %d\n", value, count);
```

Instance Variables

None declared in this class.

Method Types

Creating and freeing instances
+ new
– free

Finding IODevice objects
– lookUpByDeviceName:objectNumber:deviceKind:
– lookUpByObjectNumber:deviceKind:deviceName:

Getting and setting parameter values
– getCharValues:forParameter:objectNumber:count:
– getIntValues:forParameter:objectNumber:count:
– setCharValues:forParameter:objectNumber:count:
– setIntValues:forParameter:objectNumber:count:

Class Methods

new

+ new

Returns an IODeviceMaster object. An application has no more than one IODeviceManager object, so this method either returns the previously created object (if it exists) or creates a new one.

Instead of **new**, use **alloc** and **init** to create and initialize an instance:

```
[[IODeviceMaster alloc] init];
```

Instance Methods

free

– **free**

Does nothing; an IODeviceMaster should never be freed.

getCharValues:forParameter:objectNumber:count:

– (IOReturn)**getCharValues:**(unsigned char *)*array*
forParameter:(IOParameterName)*parameter*
objectNumber:(IOObjectNumber)*objectNumber*
count:(unsigned int *)*count*

Gets the array of values associated with *parameter* for the IODevice object specified by *objectNumber*; returns IO_R_SUCCESS. Unless *count* is specified to be 0, the returned array contains no more than *count* characters. On return, *count* is set to the number of characters in the array. You can obtain values for *objectNumber* using the method **lookUpByDeviceName:objectNumber:deviceKind:**.

If *objectNumber* is larger than the highest existing object number, this method returns IO_R_NO_DEVICE. If *objectNumber* refers to an object number for a device driver that's no longer registered, this method returns IO_R_OFFLINE. If *parameter* is invalid (it isn't recognized by the IODevice instance to have character values that can be read), this method returns IO_R_UNSUPPORTED.

See also: – **getIntValues:forParameter:objectNumber:count:**, – **setCharValues:forParameter:objectNumber:count:**

getIntValues:forParameter:objectNumber:count:

– (IOReturn)**getIntValues:**(unsigned int *)*array*
forParameter:(IOParameterName)*parameter*
objectNumber:(IOObjectNumber)*objectNumber*
count:(unsigned int *)*count*

Gets the array of values associated with *parameter* for the IODevice object specified by *objectNumber*; returns IO_R_SUCCESS. Unless *count* is specified to be 0, the returned array contains no more than *count* characters. On return, *count* is set to the number of characters in the array. You can obtain values for *objectNumber* using the method **lookUpByDeviceName:objectNumber:deviceKind:**.

If *objectNumber* is larger than the highest existing object number, this method returns IO_R_NO_DEVICE. If *objectNumber* refers to an object number for a device driver that's no longer registered, this method returns IO_R_OFFLINE. If *parameter* is invalid (it isn't recognized by the IODevice instance to have integer values that can

be read), this method returns IO_R_UNSUPPORTED.

See also: – **getCharValues:forParameter:objectNumber:count:**, – **setIntValues:forParameter:objectNumber:count:**

lookUpByDeviceName:objectNumber:deviceKind:

– (IOReturn)**lookUpByDeviceName:**(IOString)*deviceName*
objectNumber:(IOObjectNumber *)*objectNumber*
deviceKind:(IOString *)*deviceKind*

Gets the object number and descriptive string associated with the specified device name. The name is device-specific; it's the same as the value the driver sets using **setName:**. Returns IO_R_SUCCESS if the lookup was successful. Otherwise, returns IO_R_NO_DEVICE.

See also: – **lookUpByObjectNumber:deviceKind:deviceName:**

lookUpByObjectNumber:deviceKind:deviceName:

– (IOReturn)**lookUpByObjectNumber:**(IOObjectNumber)*objectNumber*
deviceKind:(IOString *)*deviceKind*
deviceName:(IOString *)*deviceName*

Gets the descriptive strings associated with the IODevice specified by *objectNumber*. Returns IO_R_SUCCESS if the lookup was successful. If *objectNumber* is larger than the highest existing object number, returns IO_R_NO_DEVICE. If *objectNumber* refers to an object number for a device driver that's no longer registered, returns IO_R_OFFLINE.

See also: – **lookUpByDeviceName:objectNumber:deviceKind:**

setCharValues:forParameter:objectNumber:count:

– (IOReturn)**setCharValues:**(unsigned char *)*array*
forParameter:(IOParameterName)*parameter*
objectNumber:(IOObjectNumber)*objectNumber*
count:(unsigned int)*count*

Sets the array of values associated with *parameter* for the IODevice object specified by *objectNumber*; returns IO_R_SUCCESS. The *count* argument specifies the number of elements in the array. You can obtain values for *objectNumber* using the method **lookUpByDeviceName:objectNumber:deviceKind:**.

If *objectNumber* is larger than the highest existing object number, this method returns IO_R_NO_DEVICE. If *objectNumber* refers to an object number for a device driver

that's no longer registered, this method returns IO_R_OFFLINE. If *parameter* is invalid (it isn't recognized by the IODevice instance to have character values that can be written), this method returns IO_R_UNSUPPORTED.

See also: – **setIntValues:forParameter:objectNumber:count:**, – **getCharValues:forParameter:objectNumber:count:**

setIntValues:forParameter:objectNumber:count:

– (IOReturn)**setIntValues:**(unsigned int *)*array*
forParameter:(IOParameterName)*parameter*
objectNumber:(IOObjectNumber)*objectNumber*
count:(unsigned int)*count*

Sets the array of values associated with *parameter* for the IODevice object specified by *objectNumber*; returns IO_R_SUCCESS. The *count* argument specifies the number of elements in the array. You can obtain values for *objectNumber* using the method **lookUpByDeviceName:objectNumber:deviceKind:**.

If *objectNumber* is larger than the highest existing object number, this method returns IO_R_NO_DEVICE. If *objectNumber* refers to an object number for a device driver that's no longer registered, this method returns IO_R_OFFLINE. If *parameter* is invalid (it isn't recognized by the IODevice instance to have integer values that can be written), this method returns IO_R_UNSUPPORTED.

See also: – **setCharValues:forParameter:objectNumber:count:**, – **getIntValues:forParameter:objectNumber:count:**

IODirectDevice

Inherits From: IODevice : Object

Declared In: driverkit/IODirectDevice.h
driverkit/*architecture*/directDevice.h
driverkit/*architecture*/IOPCIDirectDevice.h
driverkit/*architecture*/IOPCMCIADirectDevice.h

Class Description

IODirectDevice is a device-independent abstract class that is the superclass of all direct device driver classes. Most of the functionality of IODirectDevice is provided by device-dependent categories, which are described in detail below. IODirectDevice provides:

- An implementation of the **deviceStyle** IODevice class method, so IODirectDevice subclasses don't have to override it
- Methods for getting and setting IODirectDevice information, such as the interrupt port and the IODeviceDescription
- A default I/O thread that listens for messages to the interrupt port
- An efficient way to receive messages, to be used by drivers that provide their own I/O thread (see the **waitForInterrupt:** method description)

To use the default I/O thread, subclasses invoke one of the **startIOThread...** methods and implement one or more of the following methods:

- **interruptOccurred** or **interruptOccurredAt:**
- **timeoutOccurred**
- **commandRequestOccurred**
- **otherOccurred:**
- **receiveMsg**

Each of these methods is invoked when the I/O thread receives a corresponding Mach message on its interrupt port. For example, when the kernel sends an IO_DEVICE_INTERRUPT_MSG Mach message to the interrupt port, the I/O thread receives it and invokes **interruptOccurred**. The documentation for **startIOThread** describes in detail how the I/O thread listens for Mach messages and which methods it invokes in response to which Mach messages.

Interrupt messages are the only Mach messages that the kernel automatically sends.

If you want to receive other types of Mach messages, your driver or some other module it works with must explicitly send them. For example, if you want your driver's **timeoutOccurred** method to be invoked by the I/O thread, you must ensure that your driver sends an IO_TIMEOUT_MSG at some point. Some classes, such as IOEthernet, have this functionality built in. Others, such as IOSCSIController, don't. See the IOSCSIController class description for an example of how to send a message.

ISA and EISA IODirectDevices

The IOEISADirectDevice category of IODirectDevice defined in the header file **driverkit/i386/directDevice.h** provides the following additional functionality for IODirectDevices that control hardware on ISA or EISA Intel-based computers:

- Reserving and releasing ranges of I/O ports
- Reserving, releasing, enabling, and disabling interrupts (also known as *IRQs*)
- A way of providing an interrupt handler, if interrupt messages aren't sufficient
- Mapping device memory into virtual memory
- Reserving and releasing DMA channels
- Starting DMA and dealing with DMA buffers
- Determining whether the computer has EISA slots

Note: The ISA/EISA category works for all hardware attached to ISA and EISA computers—ISA slots, EISA slots, VL-Bus, and so on. Remember that EISA computers can have ISA slots, but ISA computers don't have EISA slots.

I/O ports, interrupts, device memory ranges, and DMA channels are collectively known as *resources*.

PCI IODirectDevices

The IOPCIDirectDevice category of IODirectDevice defined in the header file **driverkit/i386/IOPCIDirectDevice.h** provides the following additional functionality for IODirectDevices that control hardware on PCI Intel-based computers:

- Indicating whether the PCI bus is enabled or not
- Reading and writing the device's configuration space

The PCI configuration space is memory available for configuation information for each device. A 256-byte portion is available for each device, addressed by the PCI anchor, which consists of three fields:

- Device number between 0 and 31
- Function number between 0 and 7
- Bus number between 0 and 255

Methods can either read or write the entire configuraion space or access individual 32-bit pieces, accessing it by a *register address*—a byte address into the 256-byte portion.

PCMCIA IODirectDevices

The IOPCMCIADirectDevice category of IODirectDevice defined in the header file **driverkit/i386/IOPCMCIADirectDevice.h** provides the following additional functionality for IODirectDevices that control hardware on PCMCIA Intel-based computers:

- Mapping and unmapping attribute memory

Attribute memory resides on the PCMCIA card and contains tuples, i.e., configuration information that's stored on the card. To access attribute memory, you must map the memory using the mapping method; when you've completed your access, you must unmap it with the method provided. If you attempt to map the memory and it's already mapped, the mapping method returns failure status.

Local Equivalents of Resources

The ISA/EISA category refers to resources not by their actual numbers or addresses, but by their *local equivalent*. The local equivalent of a resource is the position (starting at 0) of that resource in the configuration list of all resources of that type.

For example, if a device is configured to have one DMA channel (DMA channel 6, for example), the local equivalent of that channel is 0. If a device is configured to have two DMA channels (specified in order as 4 and 6, for example), then channel 4 has the local equivalent of 0, and channel 6 has the local equivalent of 1.

Similarly, the first range of I/O ports in a device's configuration has the local equivalent of 0, the second range is 1, and so on.

The local equivalent is used in all ISA/EISA methods that refer to DMA channels, specific interrupts, I/O ports, and memory ranges. For example, to enable the first DMA channel in a device's configuration, a driver sends an **enableChannel:** message to **self**, specifying 0 as the channel.

See Chapter 4 and Chapter 5, "Configuation Keys" in "Other Features" for information on configuration files.

Implementing a Subclass

The IODirectDevice methods you must implement in a subclass depend on your driver's capabilities. To start with, you must implement all the methods that IODevice

requires, except for **deviceStyle**, which is implemented by IODirectDevice. You must also implement **initFromDeviceDescription:** to perform any driver- or device-specific initialization.

If your device performs DMA, you must implement **startDMAForBuffer:channel:**.

If your device can interrupt, you generally need to implement either **interruptOccurred** (if your device uses only one interrupt) or **interruptOccurredAt:**. If your driver needs to handle some interrupts directly, instead of receiving interrupt notification by Mach messages, you must implement **getHandler:level:argument:forInterrupt:**.

If your driver uses other Mach messages, you might also need to implement **timeoutOccurred**, **commandRequestOccurred**, **otherOccurred:**, or **receiveMsg**.

Most drivers need an I/O thread, as discussed in Chapter 1. All Driver Kit subclasses of IODirectDevice (such as IOEthernet) provide an I/O thread for you, if necessary. However, if your class is a direct subclass of IODirectDevice, you need to provide your own I/O thread. You can do so by invoking one of the **startIOThread...** methods.

Instance Variables

None declared in this class.

Method Types (Architecture-Independent)

Freeing instances
- free

Registering the class
+ deviceStyle

Getting and setting the interrupt port
- attachInterruptPort
- interruptPort

Handling messages to the interrupt port
- commandRequestOccurred
- interruptOccurred
- interruptOccurredAt:
- receiveMsg
- timeoutOccurred
- waitForInterrupt:

Running an I/O thread
- startIOThread
- startIOThreadWithPriority:

– startIOThreadWithFixedPriority:

Getting and setting the IODeviceDescription
– deviceDescription
– setDeviceDescription:

Method Types (ISA/EISA Architecture)

Initializing instances
– initFromDeviceDescription:

Reserving I/O ports
– reservePortRange:
– releasePortRange:

Dealing with interrupts
– enableAllInterrupts
– disableAllInterrupts
– reserveInterrupt:
– releaseInterrupt:
– enableInterrupt:
– disableInterrupt:
– getHandler:level:argument:forInterrupt:

Mapping memory
– mapMemoryRange:to:findSpace:cache:
– unmapMemoryRange:from:

Dealing with DMA channels
– enableChannel:
– disableChannel:
– reserveChannel:
– releaseChannel:

Dealing with DMA buffers
– startDMAForBuffer:channel:
– createDMABufferFor:length:read:needsLowMemory:limitSize:
– freeDMABuffer:
– abortDMABuffer:

Setting the DMA mode
– setTransferMode:forChannel:
– setAutoinitialize:forChannel:
– setIncrementMode:forChannel:

Using the EISA extended mode register
– setDMATransferWidth:forChannel:
– setDMATiming:forChannel:

– setEOPAsOutput:forChannel:
– setStopRegisterMode:forChannel:

Getting a DMA channel's status
– currentAddressForChannel:
– currentCountForChannel:
– getDMATransferWidth:forChannel:
– isDMADone:

Optional DMA locking
– reserveDMALock
– releaseDMALock

Getting information about EISA slots
– isEISAPresent
– getEISAId:forSlot:

Method Types (PCI Architecture)

Determining if PCI bus support is enabled
+ isPCIPresent
– isPCIPresent

Reading and writing the entire configuration space
+ getPCIConfigSpace:withDeviceDescription:
+ setPCIConfigSpace:withDeviceDescription:
– getPCIConfigSpace:withDeviceDescription:
– setPCIConfigSpace:withDeviceDescription:

Reading and writing the configuration space
+ getPCIConfigData:atRegister:withDeviceDescription:
+ setPCIConfigData:atRegister:withDeviceDescription:
– getPCIConfigData:atRegister:withDeviceDescription:
– setPCIConfigData:atRegister:withDeviceDescription:

Method Types (PCMCIA Architecture)

Managing attribute memory
– mapAttributeMemoryTo:findSpace:

– unmapAttributeMemory:

Class Methods (Architecture-Independent)

deviceStyle

+ (IODeviceStyle)**deviceStyle**

Reports the basic style of driver as IO_DirectDevice. Because IODirectDevice implements this method, its subclasses don't have to.

See also: + **deviceStyle** (IODevice)

Instance Methods (Architecture-Independent)

attachInterruptPort

– (IOReturn)**attachInterruptPort**

Creates the interrupt port, if none exists already, and requests that the interrupt port receive all interrupt messages for the device's reserved interrupts. This method is invoked whenever an interrupt is enabled. Returns IO_R_SUCCESS if successful; otherwise, returns IO_R_NOT_ATTACHED.

See also: – **interruptPort**, – **enableAllInterrupts** ("Instance Methods (ISA/EISA Architecture)")

commandRequestOccurred

– (void)**commandRequestOccurred**

Does nothing; subclasses can implement this method if desired. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it receives a bodyless message with ID IO_COMMAND_MSG. The part of a driver that handles user requests can use this message to notify the I/O thread that it should execute a command that's been placed in global data.

See also: – **startIOThread**

deviceDescription

– **deviceDescription**

Returns the IODeviceDescription associated with this instance.

See also: – **setDeviceDescription:**

free

– **free**

Deallocates the IODirectDevice's memory and its interrupt port, if one exists. Returns **nil**.

interruptOccurred

– (void)**interruptOccurred**

Invokes **interruptOccurredAt:** with an argument of zero. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it receives a bodyless Mach message with the ID IO_DEVICE_INTERRUPT_MSG. Subclasses that support only one interrupt should implement this method so that it processes the hardware interrupt, as described in Chapter 1 and 2.

See also: – **interruptOccurredAt:**, – **startIOThread**

interruptOccurredAt:

– (void)**interruptOccurredAt:**(int)*localInterrupt*

Does nothing; subclasses that need to handle interrupts should implement this method so that it processes the hardware interrupt, as described in Chapter 1. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it receives a bodyless Mach message with an ID between IO_DEVICE_INTERRUPT_MSG_FIRST and IO_DEVICE_INTERRUPT_MSG_LAST (excluding IO_DEVICE_INTERRUPT_MSG).

See also: – **interruptOccurred**, – **startIOThread**

interruptPort

– (port_t)**interruptPort**

Returns the Mach port on which the IODirectDevice should receive interrupt messages. The returned **port_t** is in the context of the kernel I/O task.

See also: – **attachInterruptPort:**

otherOccurred:

– (void)**otherOccurred:**(int)*msgID*

Does nothing; subclasses can implement this method if desired. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it receives a bodyless message with an unrecognized ID. The ID is given in *msgID*.

See also: – **receiveMsg**, – **startIOThread**

receiveMsg

– (void)**receiveMsg**

Dequeues the next Mach message from the interrupt port and throws it away; subclasses can implement this method if desired to handle custom messages. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it tries to receive a message that has a body. To implement this message, you need to call **msg_receive()** on the interrupt port. In this sample implementation, fill in the italicized text between angle brackets, that is << >>, with device-specific code:

```
- (void)receiveMsg
{
    IOReturn        result;
    port_t          inPort;
    MyMsg           myMsg;
    kern_return_t   result;

    inPort = [self interruptPort];
    if (inPort == PORT_NULL) {
        << React to having no interrupt port. >>
        return;
    }

    myMsg.header.msg_size = sizeof (myMsg);
    myMsg.header.msg_local_port = inPort;

    result = msg_receive(&myMsg.header, (msg_option_t)RCV_TIMEOUT,
0);

    if (result != RCV_SUCCESS) {
        IOLog("%s receiveMsg:  msg_receive returns %d\n", result);
        return;
    }
    else {
        switch (myMsg.header.msg_id) {
            case MyMsg1:
                [self handleMsg1];
                break;

            case MyMsg2:
                [self handleMsg2];
                break;
            .
            .
```

```
                .
            }
        }
    }
```

See also: – **otherOccurred:**, – **startIOThread**

setDeviceDescription:

– (void)**setDeviceDescription:***deviceDescription*

Records *deviceDescription* as the IODeviceDescription associated with this instance. ISA/EISA-architecture devices don't need to invoke this method because **initFromDeviceDescription:** already does so.

See also: – **deviceDescription**

startIOThread

– (IOReturn)**startIOThread**

Invokes **attachInterruptPort** and, if attaching the interrupt port was successful, forks a thread to serve as the instance's I/O thread. This thread, which is appropriate for most drivers, sits in an endless loop that does the following:

- Waits for a Mach message on the interrupt port by invoking **waitForInterrupt:**
- If the message couldn't be dequeued because it was too large, invokes **receiveMsg** so that the subclass can dequeue and handle the message itself
- If the message is dequeued successfully, invokes one of five methods, depending on the message ID:

Message ID	Method Invoked
IO_TIMEOUT_MSG	timeoutOccurred
IO_COMMAND_MSG	commandRequestOccurred
IO_DEVICE_INTERRUPT_MSG	interruptOccurred
IO_DEVICE_INTERRUPT_MSG_FIRST to IO_DEVICE_INTERRUPT_MSG_LAST	interruptOccurredAt:
(anything else)	otherOccurred:

Returns the value returned by **attachInterruptPort**.

See also: – **startIOThreadWithFixedPriority:**, – **startIOThreadWithPriority:**

startIOThreadWithFixedPriority:

– (IOReturn)**startIOThreadWithFixedPriority:**(int)*priority*

The same as **startIOThreadWithPriority:**, except that the I/O thread's priority never lessens due to aging. This method lets you do performance tuning by disabling priority aging.

For more information about scheduling policies and priorities, see Chapter 1 of the *NEXTSTEP Operating System Software* manual.

See also: – **startIOThread**, – **startIOThreadWithPriority:**

startIOThreadWithPriority:

– (IOReturn)**startIOThreadWithPriority:**(int)*priority*

The same as **startIOThread**, except that the I/O thread runs at the specified priority. This method lets you do performance tuning by raising or lowering the thread's scheduling priority. By default, kernel I/O threads start with a priority equal to the maximum user priority (currently 18).

For more information about priorities, see Chapter 1 of the *NEXTSTEP Operating System Software* manual.

See also: – **startIOThread**, – **startIOThreadWithFixedPriority:**

timeoutOccurred

– (void)**timeoutOccurred**

Does nothing; subclasses that support timeouts can implement this method. See the IOEthernet class for an example of implementing this method as part of timeout support. This method is invoked by the default I/O thread (implemented by **startIOThread...**) whenever it receives a bodyless Mach message with an ID of IO_TIMEOUT_MSG. See the IOSCSIController class for an example of sending Mach messages.

See also: – **startIOThread**

waitForInterrupt:

– (IOReturn)**waitForInterrupt:**(int *)*msgID*

Listens to the interrupt port until it detects a Mach message; dequeues the message if possible. This method should be invoked by the I/O thread whenever the thread needs to listen to the interrupt port. The default I/O thread provided by IODirectDevice

invokes this message as described under **startIOThread**.

If the interrupt port hasn't been set, this message returns IO_R_NO_INTERRUPT. If the message has a body, this method leaves the message on the queue and returns IO_R_MSG_TOO_LARGE. If the message couldn't be dequeued due to another reason, this method returns IO_R_IPC_FAILURE and logs an error message.

If a message is already on the queue when this method is invoked, this method dequeues the message and then attempts to give up the processor before returning. Without this precaution, a thread with many messages queued could prevent other kernel threads from being executed.

If this method successfully detects and dequeues a message, it sets *msgId* to the message's ID and returns IO_R_SUCCESS.

See also: – **startIOThread**

Instance Methods (ISA/EISA Architecture)

abortDMABuffer:

– (void)**abortDMABuffer:**(IOEISADMABuffer)*buffer*

Frees the memory allocated to *buffer*. If a read transfer is in progress, the data read is lost.

See also: – **freeDMABuffer:**

createDMABufferFor:length:read:needsLowMemory:limitSize:

– (IOEISADMABuffer)**createDMABufferFor:**(unsigned int *)*physicalAddress*
length:(unsigned int)*numBytes*
read:(BOOL)*isRead*
needsLowMemory:(BOOL)*lowerMem*
limitSize:(BOOL)*limitSize*

Returns a DMA buffer for the contents of physical memory starting at *physicalAddress* and continuing for *numBytes* bytes. You should specify YES for *isRead* if the data will be read from the device; if the data will be written to the device, specify NO. *lowerMem* should be YES if the transfer must be from or to the first 16MB of physical memory (as required by some ISA devices); otherwise, it should be NO. To limit the size of the transfer to 64KB, specify *limitSize* as YES; otherwise, *limitSize* should be NO.

This method changes the physical address if necessary to accommodate the ISA bus. When the physical address is changed, the data is copied to the new physical address (if the transfer is a write), and the new physical address is returned in

physicalAddress.

Returns NULL if kernel memory for the buffer couldn't be allocated.

See also: – **freeDMABuffer:**

currentAddressForChannel:

– (unsigned int)**currentAddressForChannel:**(unsigned int)*localChannel*

Returns the physical address currently in the address register of the specified DMA channel. This method can be invoked at any time—even when DMA is in progress. This method is often used along with autoinitialize mode. It's also used to help diagnose errors when a device or channel aborts a DMA transfer.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **currentCountForChannel:**, – **setAutoinitialize:forChannel:**

currentCountForChannel:

– (unsigned int)**currentCountForChannel:**(unsigned int)*localChannel*

Returns the number of bytes remaining to be transferred on the specified channel. The maximum number returned is equal to the length of the DMA buffer currently being handled by the channel. This method is often used along with autoinitialize mode. It's also used to help diagnose errors when a device or channel aborts a DMA transfer.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **currentAddressForChannel:**, – **setAutoinitialize:forChannel:**

disableAllInterrupts

– (void)**disableAllInterrupts**

Disables all interrupts associated with this IODirectDevice, so that no interrupts can be generated by the hardware. Returns IO_R_NO_INTERRUPT if no interrupt port is attached; otherwise, returns IO_R_SUCCESS.

Note: Even after invoking **disableAllInterrupts:** successfully, your driver may still receive interrupt messages for interrupts that occurred before they were disabled.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **enableAllInterrupts**, – **disableInterrupt:**

disableChannel:

– (void)**disableChannel:**(unsigned int)*localChannel*

If the DMA channel corresponding to *localChannel* is reserved by this device, this method disables the channel. You typically disable the channel just before changing its setting. You need to invoke **enableChannel:** once the channel is set up so that transfers can occur.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **enableChannel:**

disableInterrupt:

– (void)**disableInterrupt:**(unsigned int)*localInterrupt*

Disables the interrupt corresponding to *localInterrupt*.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **disableAllInterrupts**, – **enableInterrupt:**

enableAllInterrupts

– (IOReturn)**enableAllInterrupts**

Creates and attaches an interrupt port, if one isn't already attached, and enables all interrupts associated with this IODirectDevice. Returns IO_R_NO_INTERRUPT if the interrupt port couldn't be attached; otherwise, returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **attachInterruptPort**, – **disableAllInterrupts**, – **enableInterrupt:**

enableChannel:

– (IOReturn)**enableChannel:**(unsigned int)*localChannel*

Enables transfers on the DMA channel corresponding to *localChannel*. Returns IO_R_NOT_ATTACHED if *localChannel* doesn't correspond to a DMA channel or if

the DMA channel isn't reserved by this device. Otherwise, returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **disableChannel:**, – **startDMAForBuffer:channel:**

enableInterrupt:

– (IOReturn)**enableInterrupt:**(unsigned int)*localInterrupt*

Invokes **attachInterruptPort** and, if **attachInterruptPort** succeeds, enables the interrupt corresponding to *localInterrupt* and returns IO_R_SUCCESS. If **attachInterruptPort** doesn't succeed, returns IO_R_NOT_ATTACHED.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **disableInterrupt:**, – **enableAllInterrupts**

freeDMABuffer:

– (void)**freeDMABuffer:**(IOEISADMABuffer)*buffer*

Completes the transfer associated with *buffer* and frees the buffer. *buffer* should be a value returned by **createDMABufferFor:...**. If **createDMABufferFor:...** changed the physical address and the transfer is a read, this method moves the data from the new physical address to the old one. In other words, any data that's read appears at the address passed to **createDMABufferFor:...** in the *physicalAddress* argument, not at the address returned in *physicalAddress*.

See also: – **abortDMABuffer:**, – **createDMABufferFor:length:read:needsLowMemory:limitSize:**

getDMATransferWidth:forChannel:

– (IOReturn)**getDMATransferWidth:**(IOEISADMATransferWidth *)*width*
forChannel:(unsigned int)*localChannel*

Returns in *width* the width currently used for DMA transfers on the specified channel. The width can be 8-bit (IO_8Bit), 16-bit (IO_16BitByteCount), or 32-bit (IO_32Bit). On EISA systems, you can set the width using **setDMATransferWidth:forChannel:**.

If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method

returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **setDMATransferWidth:forChannel:**

getEISAId:forSlot:

– (BOOL)**getEISAId:**(unsigned int *)*id* **forSlot:**(int)*slotNumber*

Returns in *id* the EISA id for the specified slot. Returns YES if the slot is a valid EISA slot; otherwise, returns NO. You can use this method to loop through the computer's slots, testing each slot for whether it contains a particular card. For example, the following code is executed in the QVision display driver's **initFromDeviceDescription:** method to determine whether QVision hardware is present in the system.

```
adapter = UnknownAdapter;
for (slot = 1; slot <= 0xF; slot++) {
    if ([self getEISAId:&product_id forSlot:slot] == YES) {
        switch (product_id) {
        case QVISION_EISA_ID:
            adapter = QVisionAdapter;
            break;
        case ORION_EISA_ID:
            adapter = OrionAdapter;
            break;
        case ORION12_EISA_ID:
            adapter = Orion12Adapter;
            break;
        case QVISION_ISA_ID:
        case ORION_ISA_ID:
        case ORION12_ISA_ID:
            IOLog("%s: Sorry, ISA cards are not supported.\n",
                    [self name]);
            break;
        }
        break;
    }
}
```

See also: – **isEISAPresent**

getHandler:level:argument:forInterrupt:

– (BOOL)**getHandler:**(IOInterruptHandler *)*handler*
level:(unsigned int *)*ipl*
argument:(unsigned int *)*arg*
forInterrupt:(unsigned int)*localInterrupt*

Does nothing and returns NO. Subclasses can implement this method to specify a

function to directly handle the interrupt specified by *localInterrupt*. This method is invoked every time an interrupt is enabled.

If this method returns YES, interrupts from the device result directly in a call to *handler*, with the driver-dependent argument *arg*, at interrupt level *ipl*. Otherwise, interrupts result in a Mach message to the instance's interrupt port.

If you implement this method, you should use interrupt level 3 (IPLDEVICE, as defined in **kernserv/i386/spl.h**) unless a higher interrupt level is absolutely necessary. Using interrupt levels greater than 3 requires great care and a good grasp of NeXT kernel internals.

Note: The interrupt level is different from the interrupt number (which is also known as the IRQ). The kernel handles interrupts on each of the 15 IRQs at an interrupt level between 0 and 7; the default is 3. The interrupt level determines which devices can interrupt; specifically, only devices with an interrupt level higher than the current interrupt level can interrupt. For example, a device that interrupts using IRQ 9 might have a direct interrupt handler that runs at interrupt level 3. While this interrupt handler is running, other devices with handlers that run at interrupt level 3 can't interrupt the CPU.

Here's a typical implementation of this method:

```
- (BOOL) getHandler:(IOEISAInterruptHandler *)handler
       level:(unsigned int *) ipl
       argument:(unsigned int *) arg
       forInterrupt:(unsigned int) localInterrupt
{
    *handler = myIntHandler;
    *ipl = IPLDEVICE;
    *arg = 0;
    return YES;
}
```

In the example above, **myIntHandler** is the function that handles the interrupt. It might be implemented as follows (fill in the italicized text between angle brackets, that is << >>, with device-specific code):

```
static void myIntHandler(void *identity, void *state,
                         unsigned int arg)
{
    << . . . Do what we must at interrupt level . . . >>
    if (<< I/O thread doesn't need to know about this interrupt >>)
        return;

    /* Forward this to the I/O thread for further handling. */
    IOSendInterrupt(identity, state, IO_DEVICE_INTERRUPT_MSG);
}
```

See also: IOSendInterrupt()

initFromDeviceDescription:

– **initFromDeviceDescription:***deviceDescription*

Initializes and returns the IODirectDevice instance. Records *deviceDescription* as the IODeviceDescription corresponding to this IODirectDevice. Reserves all the interrupts, DMA channels, and I/O ports specified in *deviceDescription*. If any resources can't be reserved, releases all resources and returns **nil**.

This method must be invoked before any methods that require local equivalents of resources can be used. For example, **mapMemoryRange:...** requires that you specify the local equivalent of a memory range. However, IODirectDevices don't know what memory ranges they can use until **initFromDeviceDescription:** has been invoked. This means, for example, that subclass implementations of **initFromDeviceDescription:** must invoke the superclass's implementation of **initFromDeviceDescription:** before they can map any memory ranges or do anything else that requires access to resources.

isDMADone:

– (BOOL)**isDMADone:**(unsigned int)*localChannel*

Returns YES if DMA has completed on the specified channel; otherwise, returns NO. If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

isEISAPresent

– (BOOL)**isEISAPresent**

Returns YES if the computer conforms to the EISA specification; otherwise, returns NO.

See also: – **getEISAId:forSlot:**

mapMemoryRange:to:findSpace:cache:

– (IOReturn)**mapMemoryRange:**(unsigned int)*localMemoryRange*
to:(vm_address_t *)*destinationAddress*
findSpace:(BOOL)*findSpace*
cache:(IOCache)*caching*

Maps the device memory corresponding to *localMemoryRange* into the calling task's address space. *localMemoryRange* is the local range number in the device description.

If *findSpace* is TRUE, this method ignores the *destinationAddress* and determines where the mapped memory should go, returning the value in *destinationAddress*. If *findSpace* is FALSE, this method truncates *destinationAddress* to the nearest page boundary, maps the memory to the truncated address, and returns the truncated address.

The *caching* argument determines how the memory is cached. Usually, it should be IO_WriteThrough. However, if caching seems to be causing problems, try using IO_CacheOff instead.

If *localMemoryRange* doesn't correspond to one of this device's memory ranges, IO_R_INVALID_ARG is returned. There must also be more than one I/O port range associated with the device (i.e. [deviceDescription numPortRanges] > 1); otherwise IO_R_INVALID_ARG is returned. If the mapping couldn't be performed for another reason, IO_R_NO_SPACE is returned. If the mapping was successful, returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **unmapMemoryRange:from:**

releaseChannel:

– (void)**releaseChannel:**(unsigned int)*localChannel*

Releases the DMA channel corresponding to *localChannel* so that another device can use the channel.

See also: – **reserveChannel:**

releaseDMALock

– (void)**releaseDMALock**

Releases the lock associated with DMA. This method panics if this IODirectDevice doesn't hold the DMA lock.

Most drivers don't need to use DMA locking. However, the floppy drive (and possibly other devices) tends to have DMA underruns when the bus is saturated. As a result, the floppy driver and drivers for devices that tend to saturate the bus use DMA locking to avoid performing I/O at the same time. DMA locking is ignored by all other device drivers.

You don't have to use DMA locking unless your device is having DMA underruns or

is causing another device to have underruns. Sometimes these underruns occur on ISA computers, but not EISA ones. If your device is causing the floppy drive to have underruns, you'll see the following error on the console while your device is performing I/O:

```
fd0: DMA Over/underrun
```

See also: – **reserveDMALock**

releaseInterrupt:

– (void)**releaseInterrupt:**(unsigned int)*localInterrupt*

Releases the interrupt corresponding to *localInterrupt* so that another device can use the interrupt.

See also: – **reserveInterrupt:**

releasePortRange:

– (void)**releasePortRange:**(unsigned int)*localPortRange*

Releases the range of I/O ports corresponding to *localPortRange*.

See also: – **reservePortRange:**

reserveChannel:

– (IOReturn)**reserveChannel:**(unsigned int)*localChannel*

Reserves the DMA channel corresponding to *localChannel* so that no other device can use the channel. Returns IO_R_NOT_ATTACHED if *localChannel* doesn't correspond to a DMA channel or if the DMA channel is reserved by another device. Otherwise, returns IO_R_SUCCESS.

You don't normally have to invoke this method, since **initFromDeviceDescription:** reserves all the device's DMA channels.

See also: – **releaseChannel:**

reserveDMALock

– (void)**reserveDMALock**

Reserves the lock associated with DMA. See **releaseDMALock** for information on DMA locking.

reserveInterrupt:

– (IOReturn)**reserveInterrupt:**(unsigned int)*localInterrupt*

Reserves the interrupt corresponding to *localInterrupt* so that no other device can use it. Returns IO_R_NOT_ATTACHED if *localInterrupt* doesn't correspond to an interrupt or if another device has already reserved the interrupt. Otherwise, returns IO_R_SUCCESS.

You don't normally have to invoke this method, since **initFromDeviceDescription:** reserves all the device's interrupts.

See also: – **releaseInterrupt:**

reservePortRange:

– (IOReturn)**reservePortRange:**(unsigned int)*localPortRange*

Releases the range of I/O ports corresponding to *localPortRange* and returns IO_R_SUCCESS.

You don't normally have to invoke this method, since **initFromDeviceDescription:** reserves all the device's I/O ports.

See also: – **releasePortRange:**

setAutoinitialize:forChannel:

– (IOReturn)**setAutoinitialize:**(BOOL)*flag* **forChannel:**(unsigned int)*localChannel*

Sets the specified channel's autoinitialize DMA mode to on if *flag* is YES; otherwise, sets it off. The new autoinitialize mode stays in effect until this method is invoked again or the computer is rebooted. By default, autoinitialize mode is disabled.

If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **setIncrementMode:forChannel:**, – **setTransferMode:forChannel:**

setDMATiming:forChannel:

– (IOReturn)**setDMATiming:**(IOEISADMATiming)*timing*
forChannel:(unsigned int)*localChannel*

Makes the specified channel use the specified DMA bus cycle—ISA-compatible (IO_Compatible), Type A (IO_TypeA), Type B (IO_TypeB), or burst (IO_Burst), which is also known as Type C. This method is valid only on EISA systems.

If the system is ISA-based, this method does nothing and returns IO_R_UNSUPPORTED. If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

setDMATransferWidth:forChannel:

– (IOReturn)**setDMATransferWidth:**(IOEISADMATransferWidth)*width*
forChannel:(unsigned int)*localChannel*

Makes the specified channel use the specified width for DMA transfers. The width can be 8-bit (IO_8Bit), 16-bit (IO_16BitByteCount), or 32-bit (IO_32Bit). The 16-bit mode requires byte counting, not word counting (which is unsupported). This method is valid only on EISA systems.

If the system is ISA-based, this method does nothing and returns IO_R_UNSUPPORTED. If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

setEOPAsOutput:forChannel:

– (IOReturn)**setEOPAsOutput:**(BOOL)*flag* **forChannel:**(unsigned int)*localChannel*

Selects whether the specified channel's EOP pin is an output signal (the default) or an input signal. This method is valid only on EISA systems.

If the system is ISA-based, this method does nothing and returns IO_R_UNSUPPORTED. If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns

IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

setIncrementMode:forChannel:

– (IOReturn)**setIncrementMode:**(IOIncrementMode)*mode*
forChannel:(unsigned int)*localChannel*

This method lets the driver specify how the start address and length of its DMA buffers should be interpreted. By default, the increment mode is IO_Increment, so each DMA buffer is interpreted so that if the start address is *n* and the length is *m*, the data in addresses *n* through $n + m - 1$ are transferred. By setting the increment mode to IO_Decrement, however, the driver specifies that the affected addresses should be *n* through $n - m + 1$. The new increment mode is in effect until this method is invoked again or until the computer is rebooted.

If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Note: IO_Decrement mode is not currently supported.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **setAutoinitialize:forChannel:**, – **setTransferMode:forChannel:**

setStopRegisterMode:forChannel:

– (IOReturn)**setStopRegisterMode:**(IOEISAStopRegisterMode)*mode*
forChannel:(unsigned int)*localChannel*

Enables or disables the specified channel's Stop register. By default, the Stop register is disabled. You can enable it by specifying *mode* to be IO_StopRegisterEnable. This method is valid only on EISA systems.

Note: Enabling the Stop register isn't currently supported.

If the system is ISA-based or *mode* is IO_StopRegisterEnable, this method does nothing and returns IO_R_UNSUPPORTED. If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until

after this category's implementation of **initFromDeviceDescription:** is invoked.

setTransferMode:forChannel:

– (IOReturn)**setTransferMode:**(IODMATransferMode)*mode*
forChannel:(unsigned int)*localChannel*

Sets the specified channel's transfer mode to *mode*. The new transfer mode stays in effect until this method is invoked again or the computer is rebooted.

If *localChannel* doesn't correspond to a DMA channel, this method does nothing and returns IO_R_INVALID_ARG. If the DMA channel isn't reserved by this device, this method does nothing and returns IO_R_NOT_ATTACHED. Otherwise, this method returns IO_R_SUCCESS.

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

See also: – **setAutoinitialize:forChannel:**, – **setIncrementMode:forChannel:**

startDMAForBuffer:channel:

– (IOReturn)**startDMAForBuffer:**(IOEISADMABuffer)*buffer*
channel:(unsigned int)*localChannel*

Begins DMA with *buffer* on the DMA channel specified by *localChannel*, and returns IO_R_SUCCESS. DMA isn't started if *localChannel* doesn't correspond to a DMA channel (in which case IO_R_INVALID_ARG is returned), if the DMA channel isn't assigned, or if no DMA frames could be allocated (IO_R_NO_FRAMES is returned).

Because this method uses a local equivalent of a resource, it can't be invoked until after this category's implementation of **initFromDeviceDescription:** is invoked.

unmapMemoryRange:from:

– (void)**unmapMemoryRange:**(unsigned int)*localMemoryRange*
from:(vm_address_t)*address*

Unmaps the device memory corresponding to *localMemoryRange* from the calling task's address space. The value of *address* must be the same as the value returned by the *destinationAddress* argument of **mapMemoryRange:to:findSpace:cache**: for the same *localMemoryRange*.

See also: – **mapMemoryRange:to:findSpace:cache:**

Class Methods (PCI Architecture)

getPCIConfigData:atRegister:withDeviceDescription:

+ (IOReturn)**getPCIConfigData:**(unsigned long *)*data*
  atRegister:(unsigned char)*address*
  withDeviceDescription:*description*

Reads from the device's configuration space at the byte address *address* using the IOPCIDeviceDescription *description*. All accesses are 32 bits wide and *address* must be aligned as such.

getPCIConfigSpace:withDeviceDescription:

+ (IOReturn)**getPCIConfigSpace:**(IOPCIConfigSpace *)*configurationSpace*
  withDeviceDescription:*description*

Reads the device's entire configuration space using the IOPCIDeviceDescription *description*. Returns IO_R_SUCCESS if successful. If this method fails, the driver should make no assumptions about the state of the data returned in the IOPCIConfigSpace **struct**.

isPCIPresent

+ (BOOL)**isPCIPresent**

Returns YES if PCI Bus support is enabled. Returns NO otherwise.

setPCIConfigData:atRegister:withDeviceDescription:

+ (IOReturn)**setPCIConfigData:**(unsigned long)*data*
  atRegister:(unsigned char)*address*
  withDeviceDescription:*description*

Writes to the device's configuration space at the byte address *address* using the IOPCIDeviceDescription *description*. All accesses are 32 bits wide and *address* must be aligned as such.

setPCIConfigSpace:withDeviceDescription:

+ (IOReturn)**setPCIConfigSpace:**(IOPCIConfigSpace *)*configurationSpace*
  withDeviceDescription:*description*

Writes the device's entire configuration space using the IOPCIDeviceDescription *description*. Returns IO_R_SUCCESS if successful. If this method fails, the driver should make no assumptions about the state of the device's configuration space.

Instance Methods (PCI Architecture)

getPCIConfigData:atRegister:

– (IOReturn)**getPCIConfigData:**(unsigned long *)*data*
atRegister:(unsigned char)*address*

Reads from the device's configuration space at the byte address *address*. All accesses are 32 bits wide and *address* must be aligned as such.

getPCIConfigSpace:

– (IOReturn)**getPCIConfigSpace:**(IOPCIConfigSpace *)*configurationSpace*

Reads the device's entire configuration space. Returns IO_R_SUCCESS if successful. If this method fails, the driver should make no assumptions about the state of the data returned in the IOPCIConfigSpace **struct**.

isPCIPresent

– (BOOL)**isPCIPresent**

Returns YES if PCI Bus support is enabled. Returns NO otherwise.

setPCIConfigData:atRegister:

– (IOReturn)**setPCIConfigData:**(unsigned long)*data*
atRegister:(unsigned char)*address*

Writes to the device's configuration space at the byte address *address*. All accesses are 32 bits wide and *address* must be aligned as such.

setPCIConfigSpace:

– (IOReturn)**setPCIConfigSpace:**(IOPCIConfigSpace *)*configurationSpace*

Writes the device's entire configuration space. Returns IO_R_SUCCESS if successful. If this method fails, the driver should make no assumptions about the state of the device's configuration space.

Instance Methods (PCMCIA Architecture)

mapAttributeMemoryTo:findSpace:

– (IOReturn)**mapAttributeMemoryTo:**(vm_address_t *)*destinationAddress*
findSpace:(BOOL)*findSpace*

Maps attribute memory to *destinationAddress* in *findSpace*.

See also: – **unmapAttributeMemory:**

unmapAttributeMemory:

– (void)**unmapAttributeMemory**

Unmaps attribute memory.

See also: – **mapAttributeMemoryTo:findSpace:**

IODisplay

Inherits From: IODirectDevice : IODevice : Object

Conforms To: IOScreenEvents

Declared In: driverkit/IODisplay.h

Class Description

IODisplay is an abstract class for controlling video displays. To write a display driver, you need to create a subclass of one of NeXT's IODisplay subclasses—IOFrameBufferDisplay and IOSVGADisplay. IOFrameBufferDisplay is the preferred basis for display drivers, but it can only be used for video cards that linearly map the entire frame buffer into memory. Other video cards require IOSVGADisplay subclasses.

Note: All VGA cards work even without special drivers. However, they have a small display area (640×480) and are 2-bit grayscale.

Most of what you need to create a display driver is described in the IOFrameBufferDisplay and IOSVGADisplay class specifications. In addition, both kinds of display drivers need to specify certain display-specific configuration keys and provide IODisplayInfo structures.

Display Configuration Keys

Your driver's configuration table must have values for the "VGA Memory Maps" and "Display Mode" keys. "VGA Memory Maps" must be equal to "0xa0000-0xbffff 0xc0000-0xcffff"; those addresses must also be specified for the "Memory Maps" key.

An IOFrameBufferDisplay might specify the following in its default configuration table.

```
"Memory Maps" = "0x7e00000-0x7ffffff 0xa0000-0xbffff
0xc0000-0xcffff";
"VGA Memory Maps" = "0xa0000-0xbffff 0xc0000-0xcffff";
```

An IOSVGADisplay would have the following:

```
"Memory Maps" = "0xa0000-0xbffff 0xc0000-0xcffff";
"VGA Memory Maps" = "0xa0000-0xbffff 0xc0000-0xcffff";
```

Note: The first range for the "Memory Maps" key must be the range that the

window server will use for access to the screen. For example, for IOFrameBufferDisplays, the first range must be that of the linear frame buffer.

A default display mode is usually specified with the “Display Mode” key in the default configuration table. The mode can also be set by the user, with Configure’s display inspector. You specify the modes your driver supports with the “Display Modes” key in the **Localizable.strings** file in the *Language*.**lproj** directories of your driver’s bundle. (Driver bundles are discussed in Chapter 4.) An example of specifying display modes is below.

```
"Display Modes" = "
    Height: 600 Width: 800 Refresh: 60Hz ColorSpace: RGB:555/16;
    Height: 768 Width:1024 Refresh: 60Hz ColorSpace: BW:8;
    Height: 768 Width:1024 Refresh: 70Hz ColorSpace: BW:8;
    Height: 768 Width:1024 Refresh: 72Hz ColorSpace: BW:8;
    Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:555/16;
    Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:555/16;
    Height:1024 Width:1280 Refresh: 60Hz ColorSpace: BW:8;
    Height:1024 Width:1280 Refresh: 60Hz ColorSpace: RGB:555/16;
    Height: 600 Width: 800 Refresh: 60Hz ColorSpace: RGB:888/32;
    Height: 768 Width:1024 Refresh: 60Hz ColorSpace: RGB:888/32;
    Height: 768 Width:1024 Refresh: 72Hz ColorSpace: RGB:888/32";
```

See the “Configuration Keys” section of this chapter for more information on configuration keys.

IODisplayInfo

Display drivers need to have an IODisplayInfo structure for each mode the driver supports. Drivers that support multiple modes use one of the **selectMode:...** methods (provided by IOFrameBufferDisplay and IOSVGADisplay) to find the IODisplayInfo that corresponds to the value of the “Display Mode” key. Once the driver has determined which mode it will be in, it needs to set the value returned by **displayInfo** so that it points to the appropriate IODisplayInfo structure. The display subsystem, as well as the driver, uses the IODisplayInfo to get information about the driver’s mode.

The IODisplayInfo type is defined in the **driverkit/displayDefs.h** header file as the following:

```
typedef struct {
    int width;
    int height;
    int totalWidth;
    int rowBytes;
    int refreshRate;
    void *frameBuffer;
    IOBitsPerPixel bitsPerPixel;
    IOColorSpace colorSpace;
    IOPixelEncoding pixelEncoding;
    unsigned int flags;
```

```
    void *parameters;
} IODisplayInfo;
```

The **width** and **height** fields hold the width and height in pixels of the display area. Generally, the width should be at least 640, and the height at least 480. The **totalWidth** is the width including any undisplayed pixels that might be included for efficiency reasons; often, it's equal to **width**. The value of **rowBytes** is equal to **totalWidth** multiplied by the number of bytes used to address each pixel, as shown in the following table.

Color Mode	Value of rowBytes
SVGA 2-bit grayscale	**totalWidth**/4
8-bit grayscale	**totalWidth**
8-bit color	**totalWidth**
16-bit RGB (either 12 or 15 bits per pixel)	**totalWidth** x 2
24-bit RGB (24 bits per pixel)	**totalWidth** x 4

The **refreshRate** field holds the monitor refresh rate, in Hz.

The **frameBuffer** field should contain the first virtual address that screen memory is mapped to. You get this address during initialization by invoking **mapFrameBufferAtPhysicalAddress:length:**, as documented in the IOFrameBufferDisplay and IOSVGADisplay specifications.

The next three fields specify how the display subsystem should interpret the screen memory. The value of **bitsPerPixel** should be IO_2BitsPerPixel, IO_8BitsPerPixel, IO_12BitsPerPixel, IO_15BitsPerPixel, or IO_24BitsPerPixel. You shouldn't specify IO_VGA, since it's used only by NeXT-supplied VGA support. For IOFrameBufferDisplays, the value of **colorSpace** is IO_OneIsWhiteColorSpace for black-and-white modes and IO_RGBColorSpace for RGB modes. For IOSVGADisplays, **colorSpace** is always IO_OneIsBlackColorSpace. The value of **pixelEncoding** is a string that specifies how each bit of a pixel should be interpreted. Some common values of **pixelEncoding** are shown below.

Value of pixelEncoding	Description
"--------RRRRRRRRGGGGGGGGBBBBBBBB"	24-bit RGB, 8 bits per component; ignore the most significant byte
"-RRRRRGGGGGBBBBB"	16-bit RGB, 5 bits per component; ignore the most significant bit
"RRRRGGGGBBBB----"	16-bit RGB, 4 bits per component; ignore the 4 least significant bits
"WWWWWWWW"	8-bit grayscale

“WW”	2-bit grayscale; used by IOSVGADisplays

The **flags** field contains caching instructions and optionally some flags, combined using the bitwise OR operator. The value you specify for caching depends on whether your hardware supports burst reads. If your hardware supports burst reads (as most ’486 hardware does), you should specify IO_DISPLAY_CACHE_WRITETHROUGH to specify that screen memory be cached *write-through*. Write-through caching means that each write—even to memory that is in the cache—is immediately written to the device. If your hardware doesn’t support burst reads, you should use IO_DISPLAY_CACHE_OFF to turn the cache off for screen memory. The third caching option, IO_DISPLAY_CACHE_COPYBACK, isn’t currently supported; it specifies that writes have to be explicitly flushed, instead of being passed through the cache immediately. If **flags** contains no caching instructions, write-through caching is used.

IO_DISPLAY_NEEDS_SOFTWARE_GAMMA_CORRECTION and IO_DISPLAY_HAS_TRANSFER_TABLE are the two optional flags. IO_DISPLAY_NEEDS_SOFTWARE_GAMMA_CORRECTION should be specified if the display is in a 16-bit RGB mode with 5 bits per component *and* the hardware does not support gamma correction in this mode. It tells the display subsystem to use a default gamma correction table when converting the 4 bits used internally into the 5 bits required by the hardware.

IO_DISPLAY_HAS_TRANSFER_TABLE should be specified if the hardware supports gamma correction in this mode. Generally, drivers that support gamma correction should implement the **setTransferTable:...** method, which is described in IOFrameBufferDisplay.

The following example shows how a driver specifies **flags** when its hardware supports gamma correction only for 8-bit modes.

```
/* displayMode points to the IODisplayInfo for this mode */
displayMode->flags = IO_DISPLAY_CACHE_WRITETHROUGH;
if (displayMode->bitsPerPixel == IO_15BitsPerPixel)
    displayMode->flags |=
IO_DISPLAY_NEEDS_SOFTWARE_GAMMA_CORRECTION;
else if (displayMode->bitsPerPixel == IO_8BitsPerPixel)
    displayMode->flags |= IO_DISPLAY_HAS_TRANSFER_TABLE;
```

The **parameters** field is ignored by the display system. You can use it for whatever you want.

The following code example shows some typical IODisplayInfo structures, taken from a driver based on IOFrameBufferDisplay. See the description of **displayInfo** for an example for an IOSVGADisplay.

```
/* The frameBuffer field is initialized to 0, since it’s determined
 * at runtime.  The flags field is also determined at runtime.  The
 * parameters field in this driver points to a mode-specific
 * structure that specifies values for the hardware registers.*/

const IODisplayInfo S3_928_ModeTable[] = {
```

```
    {   /* 800 x 600, 15bpp, 60Hz. */
        800, 600, 1024, 2048, 60, 0, IO_15BitsPerPixel,
        IO_RGBColorSpace, "-RRRRRGGGGGBBBBB", 0,
        (void *)&S3_928_800x600x15,
    },
    {   /* 800 x 600, 24bpp, 60Hz. */
        800, 600, 1024, 4096, 60, 0, IO_24BitsPerPixel,
        IO_RGBColorSpace, "--------RRRRRRRRGGGGGGGGBBBBBBBB",
        0, (void *)&S3_928_800x600x24,
    },
    {   /* 1024 x 768, 8bpp, 60Hz. */
        1024, 768, 1024, 1024, 60, 0, IO_8BitsPerPixel,
        IO_OneIsWhiteColorSpace, "WWWWWWWW", 0,
        (void *)&S3_928_1024x768x8,
    },
    {   /* 1024 x 768, 8bpp, 70Hz. */
        1024, 768, 1024, 1024, 70, 0, IO_8BitsPerPixel,
        IO_OneIsWhiteColorSpace, "WWWWWWWW", 0,
        (void *)&S3_928_1024x768x8,
    },
    {   /* 1024 x 768, 8bpp, 72Hz. */
        1024, 768, 1024, 1024, 72, 0, IO_8BitsPerPixel,
        IO_OneIsWhiteColorSpace, "WWWWWWWW", 0,
        (void *)&S3_928_1024x768x8,
    },
    {   /* 1024 x 768 x 15bpp, 60Hz. */
        1024, 768, 1024, 2048, 60, 0, IO_15BitsPerPixel,
        IO_RGBColorSpace, "-RRRRRGGGGGBBBBB", 0,
        (void *)&S3_928_1024x768x15,
    },
    {   /* 1024 x 768 x 15bpp, 72Hz. */
        1024, 768, 1024, 2048, 72, 0, IO_15BitsPerPixel,
        IO_RGBColorSpace, "-RRRRRGGGGGBBBBB", 0,
        (void *)&S3_928_1024x768x15,
    },
    {   /* 1024 x 768 x 24bpp, 60Hz. */
        1024, 768, 1024, 4096, 60, 0, IO_24BitsPerPixel,
        IO_RGBColorSpace, "--------RRRRRRRRGGGGGGGGBBBBBBBB",
        0, (void *)&S3_928_1024x768x24,
    },
    {   /* 1024 x 768 x 24bpp, 72Hz. */
        1024, 768, 1024, 4096, 72, 0, IO_24BitsPerPixel,
        IO_RGBColorSpace, "--------RRRRRRRRGGGGGGGGBBBBBBBB",
        0, (void *)&S3_928_1024x768x24,
    },
    {   /* 1280 x 1024, 8bpp, 60Hz. */
        1280, 1024, 1280, 1280, 60, 0, IO_8BitsPerPixel,
        IO_OneIsWhiteColorSpace, "WWWWWWWW", 0,
        (void *)&S3_928_1280x1024x8,
    },
    {   /* 1280 x 1024, 15bpp, 60Hz. */
        1280, 1024, 2048, 4096, 60, 0, IO_15BitsPerPixel,
        IO_RGBColorSpace, "-RRRRRGGGGGBBBBB", 0,
        (void *)&S3_928_1280x1024x15,
    },
};
```

Instance Variables

None declared in this class.

Adopted Protocols

IOScreenEvents	– devicePort
	– hideCursor:
	– moveCursor:frame:token:
	– setBrightness:token:
	– showCursor:frame:token:

Method Types

Getting information about this display	– displayInfo
Getting and setting the registration token for this display	– setToken:
	– token
Getting parameters	– getIntValues:forParameter:count:
Getting the device port	– devicePort

Note: The IOScreenEvents protocol method **devicePort** is reimplemented in this class.

Instance Methods

devicePort

– (port_t)**devicePort**

Returns the device port, which should be obtained from this instance's IODeviceDescription. This method in the IOScreenEvents protocol is reimplemented in this class.

displayInfo

– (IODisplayInfo *)**displayInfo**

Returns the IODisplayInfo that describes this display. Each display driver instance

must use this method to obtain its IODisplayInfo structure. The driver must then set the fields in the structure so that they describe the display's configuration. For example, the following code initializes the IODisplayInfo associated with this instance. See the class description for information on IODisplayInfo structures.

```
static const IODisplayInfo modeTable[] = {
    {1024, 768, 1024, 256, 60, 0, IO_2BitsPerPixel,
     IO_OneIsBlackColorSpace, "WW", 0, 0,
    },
    /* Add more modes here. */
};
#define modeTableCount  (sizeof(modeTable) / sizeof(IODisplayInfo))
#define defaultMode 0

- initFromDeviceDescription:deviceDescription
{
    IODisplayInfo *displayInfo;
    const IORange *range;

    if ([super initFromDeviceDescription:deviceDescription] == nil)
        return [super free];

    /* selectedMode is a driver-defined instance variable */
    selectedMode = [self selectMode:modeTable count:modeTableCount
                     valid:NULL];
    if (selectedMode < 0) {
        IOLog("%s: Sorry, cannot use requested display mode.\n",
              [self name]);
        selectedMode = defaultMode;
    }

    displayInfo = [self displayInfo];
    *displayInfo = modeTable[selectedMode];

    displayInfo->frameBuffer = (void *)
            [self mapFrameBufferAtPhysicalAddress:0 length:0];
    if (displayInfo->frameBuffer == 0)
        return [super free];

    IOLog("%s: Initialized @ %d Hz.\n", [self name],
          displayInfo->refreshRate);
    return self;
}
```

getIntValues:forParameter:count:

– (IOReturn)**getIntValues:**(unsigned int *)*array*
forParameter:(IOParameterName)*parameter*
count:(unsigned int *)*count*

Handles NeXT-internal parameters specific to IODisplays; forwards the handling of all other parameters to **super**.

setToken:

– (void)**setToken:**(int)*token*

Sets the registration token for this display.

See also: – **token**

token

– (int)**token**

Gets the registration token for this display.

See also: – **setToken**

IODisplayInspector

Inherits From: IODeviceInspector : Object

Conforms To: IOConfigurationInspector

Declared In: driverkit/IODisplayInspector.h

Class Description

This class provides inspectors used by the Configure application for all displays. It provides an accessory View to IODeviceInspector that displays the current display mode and has a button. When the button is clicked, the IODisplayInspector puts up a panel that lets the user select the display mode for the device.

The panel shows all display modes specified for the "DisplayModes" key in the driver bundle's *Language***.lproj/Localizable.strings** file. The mode that's selected is placed in the device's **Instance*n*.table** file as the value of the "Display Mode" key.

Instance Variables

id **displayAccessoryHolder**;

id **displayMode**;

id **panel**;

id **modes**;

id **okButton**;

id **selectButton**;

id **modeText**;

IODisplayMode ***modeRecs**;

unsigned int **modeCount**;

displayAccessoryHolder	The View where the display inspector's own accessory View (as opposed to the IODeviceInspector's accessory View) is placed
displayMode	The accessory View provided to the IODeviceInspector

panel	The Select Display Mode panel
modes	The DBTableView where valid display modes are listed and can be selected
okButton	The OK button in **panel**
selectButton	The Select button in **displayMode**
modeText	The text in **displayMode** that shows the current display mode
modeRecs	An array of IODisplayModes, initialized during **setTable:** with the modes specified in the device's **Default.table**
modeCount	The number of IODisplayModes in **modeRecs**

Method Types

Initializing the IODisplayInspector
- init

Setting attributes
- setAccessoryView:
- setTable:

Displaying the Select Display Mode panel
- runPanel:
- panelDone:

Target and Action methods
- cancel:
- doubleClicked:

Instance Methods

cancel:

– **cancel:***sender*

Exits the Select Display Modes panel without changing the current display mode. Returns **self**.

See also: – **runPanel:**

doubleClicked:

– **doubleClicked:***sender*

Clicks the OK button in the Select Display Modes panel. This method is invoked when the user double-clicks an item in the display modes DBTableView. Returns **self**.

See also: – **panelDone:**

init

– **init**

Initializes and returns a newly allocated IODisplayInspector. Returns **nil** and frees itself if an error occurs.

panelDone:

– **panelDone:***sender*

Dismisses the Select Display Modes panel. This method is invoked when the user clicks the panel's OK button. Returns **self**.

See also: – **runPanel:**

runPanel:

– **runPanel:***sender*

Runs the Select Display Modes panel in a modal loop. Before displaying the panel, this method reads the supported display modes from the driver bundle's **Localizable.strings** file, puts the modes in the panel's DBTableView, and selects the current mode. When the user clicks the Cancel or OK button the loop is broken, the panel is hidden, and, if the button was OK, the new display mode is written to the driver's configuration table. Returns **self**.

See also: – **cancel:**, – **panelDone:**

setAccessoryView:

– **setAccessoryView:***aView*

Sets the IODisplayInspector's accessory View to *aView*. Because IODisplayInspector's inspector View is implemented as IODeviceInspector's accessory View, *aView* is an accessory View within an accessory View. Use this method to add a device-specific View to the inspector. Returns **self**.

setTable:

– **setTable:**(NXStringTable *)*instanceTable*

Specifies *instanceTable* as the configuration table associated with this device and uses the value of *instanceTable*'s "Display Mode" key to initialize the display modes for this device. The data in *instanceTable* is written out to its corresponding file (**Instance*n*.table**) when the user saves the configuration.

The Configure application invokes this method whenever the user selects this device for inspection. Returns **self**.

IOEISADeviceDescription

Inherits From: IODeviceDescription : Object

Declared In: driverkit/i386/IOEISADeviceDescription.h

Class Description

IOEISADeviceDescriptions encapsulate information about IODirectDevices that run on ISA- and EISA-compliant computers. Usually, you need only to pass around IOEISADeviceDescription objects, without creating them, subclassing them, or sending messages to them. IOEISADeviceDescriptions are created by the system and initialized from IOConfigTables. They are then passed to the **probe:** method to instantiate the driver, using the device description.

Instance Variables

None declared in this class.

Method Types

Getting and setting the list of DMA channels
- channel
- channelList
- numChannels
- setChannelList:num:

Getting and setting the list of I/O port ranges
- portRangeList
- numPortRanges
- setPortRangeList:num:

Getting the EISA slot number and ID
- getEISASlotNumber
- getEISASlotID

Instance Methods

channel

– (unsigned int)**channel**

Returns the first DMA channel associated with this device. The return value is undefined if this device has no DMA channels associated with it.

See also: – **channelList**, – **numChannels**, – **setChannelList:num:**

channelList

– (unsigned int *)**channelList**

Returns all the DMA channels associated with this device. You can get the number of items in the returned array by invoking **numChannels**. You should never free the data returned by this method.

See also: – **channel**, – **numChannels**, – **setChannelList:num:**

getEISASlotID

– (IOReturn)**getEISASlotID**:(unsigned long *)*slotID*

Returns the EISA slot identifier for the device. In this identifier, the device ID is in the lower 16 bits, and the vendor ID is in the upper 16 bits.

getEISASlotNumber

– (IOReturn)**getEISASlotNumber**:(unsigned int *)*slotNumber*

Returns the EISA slot number for the device.

numChannels

– (unsigned int)**numChannels**

Returns the total number of DMA channels associated with this device. This number is determined from the IOConfigTable from which this IOEISADeviceDescription is initialized.

See also: – **channel**, – **channelList**, – **setChannelList:num:**

numPortRanges

– (unsigned int)**numPortRanges**

Returns the total number of I/O port ranges associated with this device.

portRangeList

– (IORange *)**portRangeList**

Returns all the I/O port ranges associated with this device. You can get the number of items in the returned array by invoking **numPortRanges**. You should never free the data returned by this method.

See also: – **numPortRanges**, – **setPortRangeList:num:**

setChannelList:num:

– (IOReturn)**setChannelList:**(unsigned int *)*aList* **num:**(unsigned int)*numChannels*

Sets the array and number of DMA channels associated with this device. You shouldn't normally invoke this method, since it overrides the normal configuration scheme (which is documented in Chapter 4).

See also: – **channel**, – **channelList**, – **numChannels**

setPortRangeList:num:

– (IOReturn)**setPortRangeList:**(IORange *)*aList* **num:**(unsigned int)*numPortRanges*

Sets the array and number of I/O port ranges associated with this device. You shouldn't normally invoke this method, since it overrides the normal configuration scheme (which is documented in Chapter 4).

See also: – **numPortRanges**, – **portRangeList**

IOEthernet

Inherits From: IODirectDevice : IODevice : Object

Conforms To: IONetworkDeviceMethods

Declared In: driverkit/IOEthernet.h

Class Description

IOEthernet is an abstract class for controlling Ethernet devices. It provides a framework for sending and receiving packets, handling interrupts, and setting and detecting transmission timeouts. It also provides an IONetwork instance that connects the driver with the kernel networking subsystem, as well as an I/O thread from which most of the IOEthernet instance methods are invoked.

IOEthernet includes support for multicast mode and promiscuous mode. It doesn't currently provide **gdb** support for non-NeXT drivers. (**gdb** support enables the kernel running the IOEthernet driver to be debugged over the network.)

IOEthernet's multicast support consists mainly of keeping a list of the multicast addresses at which multicast packets should be received and providing methods for configuring multicast addresses. Depending on the hardware's capability, you can either implement **enableMulticastMode** and **disableMulticastMode** to allow and disallow receptions of all multicast packets or implement **addMulticastAddress:** and **removeMulticastAddress:** to configure the hardware for individual addresses.

Most hardware implementations don't guarantee filtering based on individual addresses. For this reason, the **isUnwantedMulticastPacket:** method exists to indicate packets that aren't bound for an address in the list of valid multicast addresses. A subclass of IOEthernet written for hardware that implements partial or no filtering based upon individual addresses should invoke this method each time it receives a multicast packet to determine whether it should be discarded or not.

To write an Ethernet driver, you create a subclass of IOEthernet.

Implementing a Subclass

Your subclass of IOEthernet must do the following:

- Implement **probe:** and **initFromDeviceDescription:**. The implementation of **probe:** should allocate an instance and invoke **initFromDeviceDescription:**. See

the IODevice specification for more information on implementing **probe:**.

- Implement **transmit:**, **resetAndEnable:**, **interruptOccurred**, and **timeoutOccurred**. These methods perform the real work of the driver. **interruptOccurred** is invoked from the I/O thread whenever the Ethernet hardware interrupts. See the EISA/ISA method descriptions in the IODirectDevice specification for more information on **interruptOccurred** and **timeoutOccurred**.

If your subclass supports multicast mode, you must implement either **enableMulticastMode** and **disableMulticastMode** or **addMulticastAddress:** and **removeMulticastAddress:**.

If your subclass supports promiscuous mode, you must implement **enablePromiscuousMode** and **disablePromiscuousMode**.

IONetworkDeviceMethods Protocol Implementation

In IOEthernet's implementation, **finishInitialization** invokes **resetAndEnable:YES** if [self isRunning] == YES.

Instance Variables

None declared in this class.

Adopted Protocols

IONetworkDeviceMethods	– allocateNetbuf – finishInitialization – outputPacket:address: – performCommand:data:

Method Types

Creating and destroying IOEthernet instances	– free – initFromDeviceDescription: – attachToNetworkWithAddress:
Handling interrupts	– interruptOccurred (IODirectDevice)
Transmitting packets	– transmit: – performLoopback:

Setting and handling hardware timeouts	– setRelativeTimeout: – relativeTimeout – clearTimeout – timeoutOccurred (IODirectDevice)
Setting and getting the state of the hardware	– isRunning – resetAndEnable
Supporting multicast	– enableMulticastMode – disableMulticastMode – addMulticastAddress: – removeMulticastAddress: – isUnwantedMulticastPacket:
Supporting promiscuity	– disablePromiscuousMode – enablePromiscuousMode

Instance Methods

addMulticastAddress:

– (void)**addMulticastAddress:**(enet_addr_t *)*address*

Does nothing. Subclasses that support multicast mode can implement this method so that it notifies the hardware that it should receive packets sent to *address*. Some subclasses that support multicast don't implement this method because their hardware doesn't provide filtering based upon individual multicast addresses. Instead, they inspect all multicast packets, using **isUnwantedMulticastPacket:** to weed out packets to unwanted multicast addresses. This method, followed by **enableMulticastMode**, is invoked in the I/O thread every time a new multicast address is added to the list that IOEthernet maintains.

See also: – **enableMulticastMode**, – **isUnwantedMulticastPacket:**, – **removeMulticastAddress:**

attachToNetworkWithAddress:

– (IONetwork *)**attachToNetworkWithAddress:**(enet_addr_t)*address*

Creates an IONetwork instance and attaches to the network subsystem by sending the IONetwork an **initForNetworkDevice:...** message. Before returning, this method logs a message including the ethernet address. Returns the IONetwork instance just created.

You invoke this method at the end of your implementation of **initFromDeviceDescription:**. You must invoke **resetAndEnable:NO** before invoking this method, as described under **initFromDeviceDescription:**.

clearTimeout

– (void)**clearTimeout**

If a transmission timeout is scheduled, unschedules the timeout. This method is normally invoked from a subclass's implementation of **interruptOccurred**.

See also: **setRelativeTimeout:**, **relativeTimeout**, **timeoutOccurred**

disableMulticastMode

– (void)**disableMulticastMode**

Does nothing. Subclasses that support multicast mode and implement **enableMulticastMode** should implement this method so that it disables the hardware's support for multicast mode. This method is invoked in the I/O thread when the last multicast address has been removed from the list that IOEthernet maintains.

See also: – **enableMulticastMode**

disablePromiscuousMode

– (void)**disablePromiscuousMode**

Does nothing. Subclasses that support promiscuous mode must implement this method so that it disables the hardware's support for promiscuous mode. This method is invoked in the I/O thread by the networking subsystem.

See also: – **enablePromiscuousMode**

enableMulticastMode

– (BOOL)**enableMulticastMode**

Does nothing and returns YES. Subclasses that support multicast mode can implement this method so that it enables the hardware's support for multicast mode. Every time a new multicast address is added to the list that IOEthernet maintains, **addMulticastAddress:** and this method are invoked in the I/O thread.

See also: – **disableMulticastMode**

enablePromiscuousMode

– (BOOL)**enablePromiscuousMode**

Does nothing and returns YES. Subclasses that support promiscuous mode must implement this method so that it enables the hardware's support for promiscuous mode. This method is invoked in the I/O thread by the networking subsystem.

See also: – **enablePromiscuousMode**

free

– **free**

Frees the IOEthernet instance and returns **nil**.

initFromDeviceDescription:

– **initFromDeviceDescription:**(IODeviceDescription *)*deviceDescription*

Initializes a newly allocated IOEthernet instance. This includes invoking **initFromDeviceDescription:** on **super**; invoking **startIOThread**; setting the name, kind, and unit of this instance; and invoking **registerDevice**.

Subclasses of IOEthernet should implement this method so that it invokes **[super initFromDeviceDescription:]** and then performs any device-specific initialization. The subclass implementation should invoke **resetAndEnable:NO** and should finish by invoking **attachToNetworkWithAddress:**. An example of a subclass implementation of this method is below. Italicized text delineated in angle brackets, that is << >>, is to be filled in with device-specific code.

```
- initFromDeviceDescription:(IODeviceDescription *)devDesc
{
    IOEISADeviceDescription *deviceDescription =
                            (IOEISADeviceDescription *)devDesc;
    IORange                *io;

    if ([super initFromDeviceDescription:devDesc] == nil)
            return nil;

    << Perform any 1-time hardware initialization. >>

    /* NOTE: This implementation of resetAndEnable: sets myAddress.
*/
    [self resetAndEnable:NO];  // Finish initializing the hardware

    << Perform any additional software initialization. >>

    network = [self attachToNetworkWithAddress:myAddress];
    return self;
```

```
}
```

Returns **self** if the instance was successfully initialized; otherwise, frees itself and returns **nil**.

isRunning

– (BOOL)**isRunning**

Returns YES if the hardware is currently capable of communication with other stations in the network; otherwise, returns NO.

See also: – s**etRunning:**

isUnwantedMulticastPacket:

– (BOOL)**isUnwantedMulticastPacket:**(ether_header_t *)*header*

Determines whether the specified packet is to a multicast address that this device shouldn't listen to. Returns YES if the packet should be dropped; otherwise, returns NO.

See also: – **addMulticastAddress:**

performLoopback:

– (void)**performLoopback:**(netbuf_t)*packet*

Determines whether the outgoing packet should be received by this device (because it's a broadcast packet, for example, or a multicast packet for an enabled address). If so, simulates reception by sending a copy of *packet* to the protocol stack. You should invoke this method in your **transmit:** method if your hardware device can't receive its own packets.

relativeTimeout

– (unsigned int)**relativeTimeout**

Returns the number of milliseconds until a transmission timeout will occur. If no transmission timeout is currently scheduled, this method returns zero.

See also: **clearTimeout**, **setRelativeTimeout:**, **timeoutOccurred**

removeMulticastAddress:

– (void)**removeMulticastAddress:**(enet_addr_t *)*address*

Does nothing. Subclasses that support multicast mode can implement this method so that it notifies the hardware that it should stop listening for packets sent to *address*.

See also: – **addMulticastAddress:**, – **disableMulticastMode**

resetAndEnable:

– (BOOL)**resetAndEnable:**(BOOL)*enable*

Does nothing and returns YES. Subclasses of IOEthernet must implement this method so that it resets and initializes the hardware. Interrupts should be enabled if *enable* is YES; otherwise, they should be left disabled. In either case, this method should invoke **setRunning:** to record the basic state of the device.

This method should return YES if it encounters no errors (no matter what the value of *enable* is); if it encounters errors, it should return NO. For example, the result from **resetAndEnable:NO** should be YES if the reset is successful.

The only time this method is invoked, with the exception of any invocations from your IOEthernet subclass implementation, is during initialization. Specifically, **resetAndEnable:YES** is invoked once in the I/O thread after **attachToNetworkWithAddress:** is invoked.

See also: – **setRunning:**

setRelativeTimeout:

– (void)**setRelativeTimeout:**(unsigned int)*timeout*

Schedules a timeout to occur in *timeout* milliseconds. This method is generally invoked by the IOEthernet's **transmit:** method. When *timeout* milliseconds pass without the timeout being cleared (with **clearTimeout**), the method **timeoutOccurred** is invoked.

See also: **clearTimeout**, **relativeTimeout**, **timeoutOccurred**

setRunning:

– (void)**setRunning:**(BOOL)*running*

Sets whether the hardware is on line. The value of *running* should be YES to indicate that the hardware is on line; otherwise, it should be NO. This method is invoked only by methods in IOEthernet subclasses—not by IOEthernet's own method implementations. You should invoke this method in your implementation of **resetAndEnable:**.

See also: – **isRunning**

transmit:

– (void)**transmit:**(netbuf_t)*packet*

Does nothing except free *packet*, using the **nb_free()** function. This method is invoked by the kernel networking subsystem when the hardware should transmit a packet.

Subclasses of IOEthernet must implement this method. To determine the number of bytes of data to be transmitted, use the **nb_size()** function. To get a pointer to the data, use **nb_map()**. After getting the information you need from *packet*, you must free it with **nb_free()**. Just before transmitting the packet, you can set a timeout with **setRelativeTimeout:**. If your hardware can't receive packets it transmits, you must invoke **performLoopback:** in your implementation of this method.

This method can be invoked in many contexts, not just from the I/O thread (or from the I/O task). For example, **transmit:** and **interruptOccurred** can run at the same time, so any common structures they both use must be protected with locks.

IOFrameBufferDisplay

Inherits From: IODisplay : IODirectDevice : IODevice : Object

Conforms To: IOScreenEvents

Declared In: driverkit/IOFrameBufferDisplay.h

Class Description

IOFrameBufferDisplay is an abstract class for managing display cards that support linear-mode frame buffers. IOFrameBufferDisplay's close interaction with the window server and event system means that your driver needs to do very little.

Note: If your display adapter doesn't allow you to linearly address the entire frame buffer at once, use the IOSVGADisplay class instead.

IOFrameBufferDisplay currently supports the following bit depths:

- 2-bit grayscale
- 8-bit grayscale
- 8-bit color
- 16-bit RGB (5-5-5 or 4-4-4—both with 4096 colors)
- 24-bit RGB (8-8-8)

Most of the work in writing a IOFrameBufferDisplay driver is determining how to put the hardware into an advanced mode in which the frame buffer is linearly addressable. Some drivers support several advanced modes, which the user chooses between using the Configure application. The IODisplayInfo specification describes how to specify the modes your driver supports.

When specifying your driver's memory ranges in its default configuration table, you must first specify the addresses of the linear frame buffer, and then the addresses 0xa0000-0xbffff and 0xc0000-0xcffff.

```
"Memory Maps" = "0x7e00000-0x7ffffff 0xa0000-0xbffff
0xc0000-0xcffff";
```

See the IODisplayInfo specification for information on display-specific configuration keys.

Implementing a Subclass

In your subclass of IOFrameBufferDisplay, you must implement the following

methods:

- initFromDeviceDescription:
- enterLinearMode
- revertToVGAMode

You might also need to implement two more methods:

- If the hardware supports setting brightness, you must implement **setBrightness:**.
- To support multiple gamma correction tables, implement **setTransferTable:count:**.

Instance Variables

None declared in this class.

Method Types

Creating and initializing IOFrameBufferDisplays	+ probe: – initFromDeviceDescription:
Getting and setting parameters	– getIntValues:forParameter:count: – setCharValues:forParameter:count: – setIntValues:forParameter:count:
Handling the cursor	– hideCursor: – moveCursor:frame:token: – showCursor:frame:token:
Setting screen brightness	– setBrightness:token:
Setting the gamma correction table	– setTransferTable:count:
Mapping the frame buffer	– mapFrameBufferAtPhysicalAddress:length:
Choosing display modes	– enterLinearMode – revertToVGAMode – selectMode:count: – selectMode:count:valid:

Class Methods

probe:

+ (BOOL)**probe:***deviceDescription*

Without checking for the presence of hardware, allocates and initializes an IOFrameBufferDisplay. You shouldn't reimplement this method.

If the initialization (done with **initFromDeviceDescription:**) is unsuccessful, this method returns NO. Otherwise, this method sets the device kind to "Linear Framebuffer", invokes **registerDevice**, and returns YES.

See also: – **initFromDeviceDescription:**

Instance Methods

enterLinearMode

– (void)**enterLinearMode**

Implemented by subclasses to put the display into linear frame buffer mode. This method is invoked by the system when appropriate, such as when the window server starts running.

See also: – revertToVGAMode

getIntValues:forParameter:count:

– (IOReturn)**getIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int *)*count*

Handles NeXT-internal parameters specific to IOFrameBufferDisplays; forwards the handling of all other parameters to **super**.

See also: – **getIntValues:forParameter:count:** (IODevice)

hideCursor:

– **hideCursor:**(int)*token*

Implements this method, as described in the IOScreenEvents protocol specification. You should never need to invoke or implement this method.

initFromDeviceDescription:

– **initFromDeviceDescription:***deviceDescription*

Invokes **initFromDeviceDescription:** on **super**. If successful, sets the unit number and the name (to “Display” followed by the unit number). Frees itself if initialization was unsuccessful.

Subclasses must implement this method so that it performs all initialization necessary to set up the device and the driver. This includes setting the IODisplayInfo structure (as described in the IODisplay class description) and invoking **mapFrameBufferAtPhysicalAddress:length:**. If possible, this method should also check the hardware to see if it matches the IOConfigTable. If the hardware doesn’t match, the driver should do what it can to ensure that the display is still usable.

See also: + **probe:**

mapFrameBufferAtPhysicalAddress:length:

– (vm_address_t)**mapFrameBufferAtPhysicalAddress:**(unsigned int)*address*
length:(int)*numBytes*

Maps the physical memory for this instance into virtual memory for use by the device driver. If *address* is 0, this method maps the physical memory corresponding to local memory range 0, and *numBytes* is ignored. If *address* is not 0, the reserved resources are overridden—*address* is used as the physical memory address and *numBytes* is used as the length. The mapped memory range is cached as specified in the IODisplayInfo for this instance.

Note: When overriding reserved resources, you can’t map memory outside of the memory range reserved for the device. However, you can map a subset of the memory range.

You should invoke this method during initialization.

Returns the virtual address that corresponds to *address*. If the memory mapping failed, this method logs an error message and returns NULL.

See also: – **initFromDeviceDescription:**

moveCursor:frame:token:

– **moveCursor:**(Point *)*cursorLoc*
frame:(int)*frame*
token:(int)*token*

Implements this method, as described in the IOScreenEvents protocol specification. You should never need to invoke or implement this method.

revertToVGAMode

– (void)**revertToVGAMode**

Implemented by subclasses to remove the display from whatever advanced mode it's in and enter a mode in which it can be used as a standard VGA device.

See also: – enterLinearMode

selectMode:count:

– (int)**selectMode:**(const IODisplayInfo *)*modeList* **count:**(int)*count*

Invokes **selectMode:count:valid:**, specifying 0 for the last argument.

selectMode:count:valid:

– (int)**selectMode:**(const IODisplayInfo *)*modeList*
count:(int)*count*
valid:(const BOOL *)*isValid*

Determines which IODisplayInfo in the driver-supplied *modeList* matches the value of the "Display Mode" key in the device's IOConfigTable. Drivers that support multiple advanced modes should invoke this method during initialization. When the driver receives a **enterLinearMode** message, it should enter the mode selected by this method. If this method doesn't find a valid mode, the driver should determine a mode that will work.

The "Display Mode" key is a configuration key that can be used by drivers to support multiple modes—for example, both 8-bit gray and 16-bit RGB. IODisplayInfo is defined in the header file **driverkit/displayDefs.h**.

The *modeList* argument should contain a IODisplayInfo for each advanced mode the driver supports. The *count* argument should specify the number of IODisplayInfos in *modeList*. *isValid* should either be 0 (in which case it's ignored) or an array that corresponds to the *modeList*. If *isValid*[1] is NO, for example, then this method ignores the IODisplayInfo pointed to by *modeList*[1].

If this method finds a match, it returns the index of the matching IODisplayInfo in *modeList*. If the "Display Mode" key is missing or its value is improperly formatted, or if a corresponding IODisplayInfo isn't found, this method returns -1.

See the IODisplay class description for information on display modes and the IODisplayInfo type.

setBrightness:token:

– **setBrightness:**(int)*level* **token:**(int)*token*

Checks whether *level* is between EV_SCREEN_MIN_BRIGHTNESS and EV_SCREEN_MAX_BRIGHTNESS (inclusive). If not, this method logs an error message. Subclasses that support brightness changes should override this method and implement it as described in the IOScreenEvents protocol specification.

Returns **self**.

setCharValues:forParameter:count:

– (IOReturn)**setCharValues:**(unsigned char *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int)*count*

Handles NeXT-internal parameters specific to IOFrameBufferDisplays; forwards the handling of all other parameters to **super**.

See also: – **setCharValues:forParameter:count:** (IODevice)

setIntValues:forParameter:count:

– (IOReturn)**setIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int)*count*

Handles NeXT-internal parameters specific to IOFrameBufferDisplays; forwards the handling of all other parameters to **super**.

See also: – **setIntValues:forParameter:count:** (IODevice)

setTransferTable:count:

– **setTransferTable:**(const unsigned int *)*table* **count:**(int)*numEntries*

Specifies new gamma correction values to be used by the hardware. The default implement does nothing but return **self**. Subclasses that support multiple gamma correction transforms must override this method so that it sets the hardware to reflect the values in *table*.

Gamma correction is necessary because displays respond nonlinearly to linear ranges of voltage. For example, consider a pixel that can have red, green, and blue values between 0 and 15. This pixel's brightness when the values are (7, 0, 0) might be more than 7/15 its brightness when the values are (15, 0, 0). Gamma correction lets the hardware adjust the voltage of the beam—for example, using 6.5/15 of maximum voltage instead of 7/15, so that the pixel isn't too bright.

Each entry in *table* specifies the gamma correction (a value scaled to be between 0

and 255, inclusive) for the corresponding pixel component values. For example, for RGB color modes, *table***[7]** specifies the gamma corrections for a red value of 7, a green value of 7, and a blue value of 7 (using one byte of the entry per component). If a pixel's value is (0, 5, 15), for example, the hardware should use the red gamma correction from *table***[0]**, the green gamma correction from *table***[5]**, and the blue gamma correction from *table***[15]**. Which bytes you use from each table entry depends on whether the transfer table is for a color or black-and-white mode; you can determine the mode from the value of *numEntries*.

When *numEntries* is IO_2BPP_TRANSFER_TABLE_SIZE or IO_8BPP_TRANSFER_TABLE_SIZE (as defined in the header file **driverkit/displayDefs.h**), the table is for a black-and-white display. In this case, each table entry has only one meaningful byte: the least significant byte.

When *numEntries* is IO_12BPP_TRANSFER_TABLE_SIZE, IO_15BPP_TRANSFER_TABLE_SIZE, or IO_24BPP_TRANSFER_TABLE_SIZE, the table is for an RGB display, and each entry has three meaningful bytes. The most significant byte holds the red gamma correction, the next most significant byte holds the green gamma correction, and the next holds the blue gamma correction. The least significant byte holds no information.

The following example shows how to copy the correction information from the transfer table to a particular type of hardware.

```
/* This driver implements setTransferTable: so that it copies the
 * table values into a table that contains first all the red
values,
 * then all the green values, and then all the blue values.  It
 * defines 3 instance variables -- redTransferTable,
 * greenTransferTable, and blueTransferTable -- to point to where
 * each component's values begin in the copied table. Finally,
 * it puts the values in the hardware gamma correction table. */
- setTransferTable:(unsigned int *)table count:(int)numEntries
{
    int k;

    /* redTransferTable, greenTransferTable, and blueTransferTable
     * are driver-defined instance variables
    if (redTransferTable != 0)
        IOFree(redTransferTable, 3 * transferTableCount);

    transferTableCount = numEntries;

    redTransferTable = IOMalloc(3 * numEntries);
    greenTransferTable = redTransferTable + numEntries;
    blueTransferTable = greenTransferTable + numEntries;

    switch ([self displayInfo]->bitsPerPixel) {
    case IO_2BitsPerPixel:
    case IO_8BitsPerPixel:
        for (k = 0; k < numEntries; k++) {
            redTransferTable[k] = greenTransferTable[k] =
            blueTransferTable[k] = table[k] & 0xFF;
        }
        break;
```

```
    case IO_12BitsPerPixel:
    case IO_15BitsPerPixel:
    case IO_24BitsPerPixel:
        for (k = 0; k < numEntries; k++) {
            redTransferTable[k] = (table[k] >> 24) & 0xFF;
            greenTransferTable[k] = (table[k] >> 16) & 0xFF;
            blueTransferTable[k] = (table[k] >> 8) & 0xFF;
        }
        break;

    default:
        IOFree(redTransferTable, 3 * numEntries);
        redTransferTable = 0;
        break;
    }
    [self setGammaTable]; /* subclass method */
    return self;
}

/* subclass function */
static void
SetGammaValue(unsigned int r, unsigned int g, unsigned int b,
              int level)
{
    /* EV_SCALE_BRIGHTNESS is a macro defined in bsd/dev/ev_types.h
     * that scales a pixel value to the specified brightness level.
*/
    outb(PALETTE_DATA, EV_SCALE_BRIGHTNESS(level, r));
    outb(PALETTE_DATA, EV_SCALE_BRIGHTNESS(level, g));
    outb(PALETTE_DATA, EV_SCALE_BRIGHTNESS(level, b));
}

/* subclass method */
- setGammaTable
{
    unsigned int i, j, g;
    const IODisplayInfo *displayInfo;

    displayInfo = [self displayInfo];

    outb(PALETTE_WRITE, 0x00);

    /* brightnessLevel is a subclass ivar initialized to
     * EV_SCREEN_MAX_BRIGHTNESS; setBrightness: changes it */
    if (redTransferTable != 0) {
        for (i = 0; i < transferTableCount; i++) {
            for (j = 0; j < 256/transferTableCount; j++) {
                SetGammaValue(redTransferTable[i],
                              greenTransferTable[i],
                              blueTransferTable[i],
brightnessLevel);
            }
        }
    }
    return self;
}
```

Gamma correction transforms are set using the **setframebuffertransfer** PostScript operator. The Window Server uses the functions specified in **setframebuffertransfer**

to fill the values used in *table*. It then passes the values down the display system so that eventually the **setTransferTable:count:** message is invoked.

Note: The default transfer table cannot be specified using NetInfo, despite the claims of the **setframebuffertransfer** documentation.

See also: **setframebuffertransfer** PostScript Operator (*NEXTSTEP General Reference*)

showCursor:frame:token:

– **showCursor:**(Point *)*cursorLoc*
frame:(int)*frame*
token:(int)*token*

Implements this method, as described in the IOScreenEvents protocol specification. You should never need to invoke or implement this method.

IONetbufQueue

Inherits From: Object

Declared In: driverkit/IONetbufQueue.h

Class Description

IONetbufQueue is used by network device drivers to store packets until they're transmitted. IONetbufQueue is a first-in first-out (FIFO) queue.

Instance Variables

None declared in this class.

Method Types

Creating and initializing instances
- – initWithMaxCount:

Adding and removing netbufs from the queue
- – enqueue:
- – dequeue

Getting the size of the queue
- – count
- – maxCount

Instance Methods

count

– (unsigned int)**count**

Returns the number of netbufs in the IONetbufQueue.

See also: – **maxCount**

dequeue

– (netbuf_t)**dequeue**

Removes and returns the netbuf that has been in the queue the longest. Returns NULL if no netbufs are in the queue.

enqueue:

– (void)**enqueue:**(netbuf_t)*netbuf*

Adds the specified netbuf to the queue, unless the queue already has reached its maximum length. If the queue is at its maximum length, the netbuf is freed.

See also: – **count**, – **maxCount**

initWithMaxCount:

– **initWithMaxCount:**(unsigned int)*maxCount*

Initializes and returns a newly allocated IONetbufQueue. The maximum number of netbufs in the queue is set to *maxCount*.

maxCount

– (unsigned int)**maxCount**

Returns the maximum number of netbufs that can be in the IONetbufQueue. This number is set at initialization time.

See also: – **maxCount**, – **initWithMaxCount:**

IONetwork

Inherits From: Object

Declared In: driverkit/IONetwork.h

Class Description

The IONetwork class connects direct drivers, such as Ethernet drivers, into the kernel network interface. One IONetwork object is associated with each instance of a network direct driver. In the future, support may be added for indirect network drivers, as well.

Network direct drivers must implement the IONetworkDeviceMethods protocol, so that the IONetwork can send them messages.

Note: Network drivers must run in the kernel.

See the IOEthernet specification for information on how to write Ethernet drivers, and the IOTokenRing specification for information on writing Token Ring drivers. See Chapter 8, "Network Modules," in *NEXTSTEP Operating System Software* for more information about network drivers.

Instance Variables

None declared in this class.

Method Types

Initializing an IONetwork instance
- initForNetworkDevice:name:unit:type: maxTransferUnit:flags:
- finishInitialization

Passing packets from the driver up to the protocol stack
- handleInputPacket:extra:

Outputting a packet
- outputPacket:address:

Performing a command
- performCommand:data:

Allocating a network buffer	– allocateNetbuf
Keeping statistics	– collisions – incrementCollisions – incrementCollisionsBy: – incrementInputErrors – incrementInputErrorsBy: – incrementInputPackets – incrementInputPacketsBy: – incrementOutputErrors – incrementOutputErrorsBy: – incrementOutputPackets – incrementOutputPacketsBy: – inputErrors – inputPackets – outputErrors – outputPackets

Instance Methods

allocateNetbuf

– (netbuf_t)**allocateNetbuf**

This method creates and returns a netbuf to be used for an impending output.

This method doesn't always have to return a buffer. For example, you might want to limit the number of buffers your driver instance can allocate (say, 200 kilobytes worth) so that it won't use too much wired-down kernel memory. When this method fails to return a buffer, it should return NULL.

Here's an example of implementing **allocateNetbuf**.

```
#define my_HDR_SIZE    14
#define my_MTU        1500
#define my_MAX_PACKET  (my_HDR_SIZE + my_MTU)

- netbuf_t allocateNetbuf
{
    if (_numbufs == _maxNumbufs)
        return(NULL);
    else {
        _numbufs++;
        return(nb_alloc(my_MAX_PACKET));
    }
}
```

See also: **nb_alloc()** (*NEXTSTEP Operating System Software)*

collisions

– (unsigned int)**collisions**

Returns the total number of network packet collisions that have been detected since boot time.

finishInitialization

- (int)**finishInitialization**

This method should perform any initialization that hasn't already been done. For example, it should make sure its hardware is ready to run. You can specify what the integer return value (if any) should be.

If you implement this method, you need to check that [self isRunning] == YES.

handleInputPacket:extra:

– (int)**handleInputPacket:**(netbuf_t)*packet* **extra:**(void *)*extra*

Increments the number of input packets and passes *packet* to the kernel for processing. The kernel dispatches the packet to the appropriate protocol handler, as described <<*only in the OS book, for now*>>.

A network device driver should invoke this method after it's processed a newly received packet. The value of *extra* should be zero, unless the protocol handler requires another value. For instance, token ring drivers need to return a valid pointer to a token ring header. This method returns EAFNOSUPPORT if no protocol handler accepts the packet; otherwise, it returns zero.

incrementCollisions

– (void)**incrementCollisions**

Increments by one the total number of network packet collisions that have been detected since boot time.

incrementCollisionsBy:

– (void)**incrementCollisionsBy:**(unsigned int)*increment*

Increments by *increment* the total number of network packet collisions that have been detected since boot time.

incrementInputErrors

– (void)**incrementInputErrors**

Increments by one the total number of packet input errors that have been detected since boot time.

incrementInputErrorsBy:

– (void)**incrementInputErrorsBy:**(unsigned int)*increment*

Increments by *increment* the total number of packet input errors that have been detected since boot time.

incrementInputPackets

– (void)**incrementInputPackets**

Increments by one the total number of packets that have been received by the computer since boot time. You usually don't need to invoke this method because **handleInputPacket:extra:** does so for you.

incrementInputPacketsBy:

– (void)**incrementInputPacketsBy:**(unsigned int)*increment*

Increments by *increment* the total number of packets that have been received by the computer since boot time.

incrementOutputErrors

– (void)**incrementOutputErrors**

Increments by one the total number of packet output errors that have been detected since boot time.

incrementOutputErrorsBy:

– (void)**incrementOutputErrorsBy:**(unsigned int)*increment*

Increments by *increment* the total number of packet output errors that have been

detected since boot time.

incrementOutputPackets

– (void)**incrementOutputPackets**

Increments by one the total number of packets that have been transmitted by the computer since boot time.

incrementOutputPacketsBy:

– (void)**incrementOutputPacketsBy:**(unsigned int)*increment*

Increments by *increment* the total number of packets that have been transmitted by the computer since boot time.

initForNetworkDevice:name:unit:type:maxTransferUnit:flags:

– **initForNetworkDevice:***device*
name:(const char *)*name*
unit:(unsigned int)*unit*
type:(const char *)*type*
maxTransferUnit:(unsigned int)*mtu*
flags:(unsigned int)*flags*

Initializes and returns the IONetwork instance associated with the specified direct device driver *device*. This method connects *device* into the kernel's networking subsystem. It's typically called from a network driver's implementation of **initFromDeviceDescription**. You shouldn't invoke **initForNetworkDevice:...** directly. IOEthernet and IOTokenRing invoke this method on behalf of their subclasses and return an IONetwork object in their respective **attachToNetworkWithAddress:** methods.

The *name* argument should be set to a constant string that names this type of network device. For example, Ethernet drivers are named "en", and Token Ring drivers are named "tr". The *unit* is an integer greater than or equal to zero that's unique for *name*. For example, the first instance of an Ethernet driver is unit 0, the second is unit 1, and so on.

The *type* is a constant string that describes this module. For example, Ethernet drivers supply the constant IFTYPE_ETHERNET (which is defined in **net/etherdefs.h** to be "10MB Ethernet").

The *mtu* is the maximum amount of data your module can send or receive. For example, Ethernet drivers use the value `ETHERMTU`, which is defined in the header file

net/etherdefs.h as **1500**.

Finally, *flags* defines the initial flags for the interface. Possible values are:

IFF_UP:	If true, this interface is working.
IFF_BROADCAST:	If true, this interface supports broadcast.
IFF_LOOPBACK:	If true, this interface is local only.
IFF_POINTTOPOINT:	If true, this is a point-to-point interface.

inputErrors

– (unsigned int)**inputErrors**

Returns the total number of packet input errors that have been detected since boot time.

inputPackets

– (unsigned int)**inputPackets**

Returns the total number of packets that have been received by the computer since boot time.

outputErrors

– (unsigned int)**outputErrors**

Returns the total number of packet output errors that have been detected since boot time.

outputPacket:address:

– (int)**outputPacket:**(netbuf_t)*packet* **address:**(void *)*address*

This method should deliver the specified packet to the given address. Its return value should be zero if no error occurred; otherwise, return an error number from the header file **sys/errno.h**.

If you implement this method, you need to check that [self isRunning] == YES. If so, insert the necessary hardware addresses into the packet and check it for minimum length requirements.

outputPackets

– (unsigned int)**outputPackets**

Returns the total number of packets that have been transmitted by the computer since boot time.

performCommand:data:

– (int)**performCommand:**(const char *)command **data:**(void *)data

This method performs arbitrary control operations; the character string *command* is used to select between these operations. Although you don't have to implement any operations, there are five standard operations. You can also define your own operations.

The standard commands are listed in the following table. The constant strings listed below are declared in the header file **net/netif.h** (under the **bsd** directory of **/NextDeveloper/Headers**).

Command	Operation
IFCONTROL_SETFLAGS	Request to have interface flags turned on or off. The *data* argument for this command is of type **union ifr_ifru** (which is declared in the header file **net/if.h**).
IFCONTROL_SETADDR	Set the address of the interface.
IFCONTROL_GETADDR	Get the address of the interface.
IFCONTROL_AUTOADDR	Automatically set the address of the interface.
IFCONTROL_UNIXIOCTL	Perform a UNIX **ioctl()** command. This is only for compatibility; **ioctl()** isn't a recommended interface for network drivers. The argument is of type **if_ioctl_t ***, where the **if_ioctl_t** structure contains the UNIX ioctl request (for example, SIOCSIFADDR) in the **ioctl_command** field and the ioctl data in the **ioctl_data** field.

An example of implementing **performCommand:data:** follows.

```
- (int)performCommand:(const char *)command data:(void *)data
{
    int error = 0;

    if (strcmp(command, IFCONTROL_SETFLAGS) == 0)
        /* do nothing */;
    else
    if (strcmp(command, IFCONTROL_GETADDR) == 0)
        bcopy(&my_address, data, sizeof (my_address));
    else
```

```
        error = EINVAL;

    return (error);
}
```

IOPCIDeviceDescription

Inherits From: IOEISADeviceDescription : IODeviceDescription : Object

Declared In: driverkit/i386/IOPCIDeviceDescription.h

Class Description

IOPCIDeviceDescription objects encapsulate information about IODirectDevices that run on PCI-compliant computers. Usually, you need only to pass around IOPCIDeviceDescriptions, without creating them, subclassing them, or sending messages to them. IOPCIDeviceDescriptions are created by the system and initialized from IOConfigTables.

This object encapsulates the PCI Configuration Space address of the device. This address contains three fields:

- Device number, ranging from 0 to 31
- Function number, ranging from 0 to 7
- Bus number, ranging from 0 to 255

Instance Variables

None declared in this class.

Method Types

Getting config address of PCI device

– getPCIdevice:function:bus:

Instance Methods

getPCIdevice

– (IOReturn)**getPCIdevice:**(unsigned char *)*deviceNumber*
function:(unsigned char *)*functionNumber*
bus:(unsigned char *)*busNumber*

This method allows callers to get the PCI config address of the PCI device associated with this device description. If all goes well, the three parameters are filled in and IO_R_SUCCESS is returned. There are a variety of reasons that the address couldn’t be known, in which case an appropriate code is returned and the parameters are left untouched. It is acceptable for any of the parameter pointers to be **nil**.

IOPCMCIADeviceDescription

Inherits From: IOEISADeviceDescription : IODeviceDescription : Object

Declared In: driverkit/i386/IOPCMCIADeviceDescription.h

Class Description

IOPCMCIADeviceDescription objects encapsulate information about IODirectDevices that run on PCMCIA-compliant computers. Usually, you need only to pass around IOPCMCIADeviceDescriptions, without creating them, subclassing them, or sending messages to them. IOPCMCIADeviceDescriptions are created by the system and initialized from IOConfigTables.

Instance Variables

None declared in this class.

Method Types

Getting information about tuples – numTuples
– tupleList

Instance Methods

numTuples

– (unsigned)**numTuples**

Returns the number of items in the tuple list.

See also: **tupleList**

tupleList

– (id *)**tupleList**

Returns the tuple list.

See also: **numTuples**

IOPCMCIATuple

Inherits From: IOPCMCIATuple : Object

Declared In: driverkit/i386/IOPCMCIATuple.h

Class Description

IOPCMCIATuple objects encapsulate configuration information about IODirectDevices that run on PCMCIA-compliant computers. Data from a “tuple” is from information stored on the PCMCIA card; each tuple stores a separate piece of information. IOPCMCIADeviceDescription objects typically contain a list of IOPCMCIATuple objects, containing such configuration data as electrical requirements, I/O port ranges, and timing information.

Usually, you need only to pass around IOPCMCIATuple objects, without creating them, subclassing them, or sending messages to them. IOPCMCIATuples are created by the system.

Instance Variables

None declared in this class.

Method Types

Getting information from a tuple

– code
– data
– length

Instance Methods

code

– (unsigned char)**code**

Returns a code describing the contents of the tuple, as described in the PCMCIA

standard.

See also: **data, length**

data

– (unsigned char *)**data**

Returns the tuple data, which is in machine readable form.

See also: **code, length**

length

– (unsigned)**length**

Returns the length of the tuple data in bytes.

See also: **code, data**

IOSCSIController

Inherits From: IODirectDevice : IODevice : Object

Conforms To: IOSCSIControllerExported

Declared In: driverkit/IOSCSIController.h

Class Description

IOSCSIController is an abstract class for managing SCSI controllers. It provides a framework for making SCSI requests and providing standard statistics. It also provides an I/O thread.

Implementing a Subclass

To write a driver for a SCSI controller, you create a subclass of IOSCSIController. Your subclass must do the following:

- Implement **probe:** (as documented in IODevice) and **initFromDeviceDescription:**. These let your driver create instances of itself.
- Implement **executeRequest:buffer:client:** and **resetSCSIBus**.
- Implement timeouts, as described in "Implementing Timeouts," below.
- Implement **interruptOccurred**, as documented in IODirectDevice.

To support standard statistics, you should implement **sumQueueLengths**, **maxQueueLength**, **numQueueSamples**, and **resetStats**, as described in "Supporting Standard Statistics," below.

Implementing Timeouts

To implement timeouts, you need to implement the **timeoutOccurred:** method (as documented in IODirectDevice) and make sure that your driver sends a timeout message whenever a request has taken too much time. To do the latter, your **executeRequest:buffer:client:** method should use **IOScheduleFunc()** to schedule a function; the method should then start I/O. If the I/O finishes before the function has executed, **executeRequest:buffer:client:** should unschedule the function. Otherwise, the function should send a timeout message (one with a **msg_id** field set to IO_TIMEOUT_MSG) to the instance's interrupt port. An example is below.

Italicized text delineated in angle brackets, that is << >>, is to be filled in with device-specific code.

```
In executeRequest:buffer:client:
    << ...Construct a device-dependent command buffer “ccb”...
       Since the function we schedule won’t be called from the I/O
       task, it must use msg_send_from_kernel.  This means that we
       must convert the interrupt port from the I/O task space to
one
       that’s valid in the regular kernel space.  We do this in
       initFromDeviceDescription: as follows:
               interruptPortKern = IOConvertPort([self
interruptPort],
               IO_KernelIOTask,  IO_Kernel); >>
    ccb->timeoutPort = interruptPortKern;
    IOScheduleFunc(myTimeout, ccb, scsiRequest->timeoutLength);
    << ...Start the I/O and wait for it to finish... >>
    (void) IOUnscheduleFunc(myTimeout, ccb);

/* This method just logs a warning and sends a timeout message. */
static void myTimeout(void *arg)
{
    struct ccb    *ccb = arg;
    msg_header_t   msg;

    if(!ccb->in_use) {
        /* Race condition - this CCB got completed another way. */
        return;
    }

    msg.msg_remote_port = ccb->timeoutPort;
    msg.msg_id = IO_TIMEOUT_MSG;
    IOLog("mySCSIController timeout\n");
    (void) msg_send_from_kernel(&msg, MSG_OPTION_NONE, 0);
}
```

Supporting Standard Statistics

Subclasses of IOSCSIController can provide information used by the **iostat** command and any other statistics-gathering modules. To provide this information, the IOSCSIController must look at the number of requests in its queue of I/O requests, keeping track of the following:

- The total number of requests detected in the queue. The IOSCSIController should implement **sumQueueLengths** so that it returns this value.
- The highest number of requests in the queue at one time. This value should be returned by **maxQueueLength**.
- The number of times the driver has looked at the queue. The **numQueueSamples** method should return this value.

For example, assume the IOSCSIController has looked at its list of outstanding I/O requests three times, and found 1 request the first time, 5 the second, and 2 the third. At this point, **sumQueueLengths** should return 8, **maxQueueLength** should return 5,

and **numQueueSamples** should return 3. The average number of requests in the list is **sumQueueLengths** divided by **numQueueSamples**.

The IOSCSIController should reset all these values to 0 whenever it receives a **resetStats** message.

Instance Variables

None declared in this class.

Adopted Protocols

IOSCSIControllerExported
- allocateBufferOfLength:actualStart:actualLength:
- executeRequest:buffer:client:
- getDMAAlignment:
- maxTransfer
- releaseTarget:lun:forOwner:
- reserveTarget:lun:forOwner:
- resetSCSIBus
- returnFromScStatus:

Method Types

Initializing a newly allocated IOSCSIController
- initFromDeviceDescription:

Reserving target/lun pairs
- numReserved

Getting and setting parameters
- getIntValues:forParameter:count:
- setIntValues:forParameter:count:

Collecting statistics
- maxQueueLength
- numQueueSamples
- sumQueueLengths
- resetStats

Instance Methods

getIntValues:forParameter:count:

– (IOReturn)**getIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int *)*count*

Handles the two parameters specific to SCSI controllers—IO_SCSI_CONTROLLER_STATS and IO_IS_A_SCSI_CONTROLLER—and forwards the handling of all other parameters to **super**. The array of values returned for IO_SCSI_CONTROLLER_STATS is set to the numbers returned by **maxQueueLength**, **numQueueSamples**, and **sumQueueLengths**. No array is returned for IO_IS_A_SCSI_CONTROLLER; only IO_R_SUCCESS is returned, to indicate that this IODevice is indeed a SCSI controller.

See also: – **setIntValues:forParameter:count:**

initFromDeviceDescription:

– **initFromDeviceDescription:***deviceDescription*

Initializes a new IOSCSIController instance. After invoking IODirectDevice's version of **initFromDeviceDescription:**, this method starts an I/O thread (with **startIOThread**) and sets its unit, name, and device kind. Each IOSCSIController has its own unit number. The first instance's unit is 0, the second is 1, and so on. The name is the concatenation of "sc" and the unit (for example, "sc0"), and the device kind is "sc".

This method also determines the alignment restrictions for the hardware, using the **getDMAAlignment:** method. The alignment restrictions are used by the method **allocateBufferOfLength:actualStart:actualLength:**.

This method returns **nil** and frees the instance if initialization failed; otherwise, it returns **self**.

You should implement this method to invoke IOSCSIController's version and to then perform any driver-dependent initialization, including initializing the hardware and (at the very end) invoking **registerDevice**.

maxQueueLength

– (unsigned int)**maxQueueLength**

Returns zero. Subclasses that support standard statistics should implement this method so that it returns the highest number of requests queued since this instance was initialized or **resetStats** was last called. See the class description for more information on supporting standard statistics.

numQueueSamples

– (unsigned int)**numQueueSamples**

Returns zero. Subclasses that support standard statistics should implement this method so that it returns the number of times the instance has collected information about its queue of I/O requests. This number should be reset to 0 when this instance is initialized and when **resetStats** is called. See the class description for more information on supporting standard statistics.

numReserved

– (unsigned int)**numReserved**

Returns the number of target/lun pairs that are reserved. Each pair corresponds to an active device on the SCSI bus that this instance controls.

See also: – **reserveTarget:lun:forOwner:** and – **releaseTarget:lun:forOwner:** (IOSCSIControllerExported protocol)

resetStats

– (void)**resetStats**

Does nothing. Subclasses that support standard statistics should implement this method so that it resets to zero the numbers that are returned by **maxQueueLength**, **numQueueSamples**, and **sumQueueLengths**. See the class description for more information on supporting standard statistics.

setIntValues:forParameter:count:

– (IOReturn)**setIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int)*count*

Handles the IO_SCSI_CONTROLLER_STATS parameter, forwarding the handling of all other parameters to **super**. The IO_SCSI_CONTROLLER_STATS parameter resets (using **resetStats**) the standard statistical data kept by this instance.

See also: – **getIntValues:forParameter:count:**

sumQueueLengths

– (unsigned int)**sumQueueLengths**

Returns zero. Subclasses that support standard statistics should implement this method so that it returns the total number of requests detected in its queue of I/O requests. This number should be reset to 0 when this instance is initialized and when **resetStats** is called. See the class description for more information on supporting standard statistics.

IOSVGADisplay

Inherits From: IODisplay : IODirectDevice : IODevice : Object

Conforms To: IOScreenEvents

Declared In: driverkit/IOSVGADisplay.h

Class Description

IOSVGADisplay is an abstract class for managing Super VGA (SVGA) video displays. It provides most of the functionality needed by SVGA drivers. Functionality that varies from card to card must be provided by subclasses of IOSVGADisplay. In particular, different SVGA cards have different ways of setting the current plane and segment and of entering and exiting SVGA modes.

IOSVGADisplay supports 2-bit grayscale modes; it doesn't currently support 8-bit gray or color modes. To provide 2-bit grayscale support, IOSVGADisplay uses 2 of the 8 planes associated with screen pixels. The IOSVGADisplay subclass maps the values in the two planes into four entries in the hardware color palette, as described in **enterSVGAMode**.

Because the VGA specification allows only 64KB of screen memory to be mapped (from 0xa0000 to 0xbffff), the screen is split up into segments of 64KB or less. The display system tells the driver which segment and plane to map into the 64KB of available space. For a screen that's 768 pixels high by 1024 wide, the first 64KB segment (segment 0) consists of the top 512 rows of pixels. The next segment consists of the bottom 256 rows.

To write an SVGA display driver, you need to write two software modules:

- A subclass of IOSVGADisplay
- Five functions to be loaded into a user-level PostScript driver

How to write these modules is described below. See the IODisplay class description for additional notes on implementing a display driver.

Implementing a Subclass

In your subclass of IOSVGADisplay, you must implement the following methods:

- initFromDeviceDescription:
- enterSVGAMode

- revertToVGAMode
- savePlaneAndSegmentSettings
- restorePlaneAndSegmentSettings
- setReadPlane:
- setReadSegment:
- setWritePlane:
- setWriteSegment:

If the hardware supports setting brightness, you should also implement **setBrightness:token:**.

Writing Functions for the PostScript Driver

Besides implementing a subclass of IOSVGADisplay, you also need to write five C functions. One, named **IOSetSVGAFunctions()**, should fill in the structure it's passed with pointers to the other four functions. It should return zero on success. The other four functions correspond exactly to four methods that you must also implement; each function should have exactly the same code as its corresponding method.

Function Prototype	Corresponding Method
void *setReadPlane*(unsigned char *num*)	setReadPlane:
void *setReadSegment*(unsigned char *num*)	setReadSegment:
void *setWritePlane*(unsigned char *num*)	setWritePlane:
void *setWriteSegment*(unsigned char *num*)	setWriteSegment:

Here's an example of how to implement **IOSetSVGAFunctions()**.

```
void SetReadPlane(unsigned char num) . . .
void SetReadSegment(unsigned char num) . . .
void SetWriteSegment(unsigned char num) . . .
void SetWritePlane(unsigned char num) . . .

int IOSetSVGAFunctions(IOSVGAFunctions *funcs)
{
    funcs->setReadSegment = SetReadSegment;
    funcs->setWriteSegment = SetWriteSegment;
    funcs->setReadPlane = SetReadPlane;
    funcs->setWritePlane = SetWritePlane;

    return 0;
}
```

Note: The functions must contain only C code; Objective C code won't work.

The five functions should be defined in a user-level executable, to be loaded into the SVGA PostScript driver at run time. You need to inform the PostScript driver of the executable's location using the configuration key "SVGA PostScript Driver Extension". You also need to specify the SVGA PostScript driver (**/usr/lib/NextStep/Displays/SVGA_psdrvr**) with the "PostScript Driver" key. For example, the lines below specify that the SVGA PostScript driver should load the executable named **TsengLabsET4000_psdrvr** from the driver's configuration bundle.

```
"SVGA PostScript Driver Extension" = "TsengLabsET4000_psdrvr";
"PostScript Driver" = "/usr/lib/NextStep/Displays/SVGA_psdrvr";
```

Note: See the IODisplay class description for other configuration keys that must be specified.

Instance Variables

None declared in this class.

Method Types

Creating and initializing IOSVGADisplays	+ probe: – initFromDeviceDescription:
Getting and setting parameters	– getIntValues:forParameter:count: – setIntValues:forParameter:count:
Handling the cursor	– hideCursor: – moveCursor:frame:token: – showCursor:frame:token:
Setting screen brightness	– setBrightness:token:
Mapping memory	– mapFrameBufferAtPhysicalAddress:length:
Choosing video modes	– enterSVGAMode – revertToVGAMode – selectMode:count: – selectMode:count:valid:
Setting planes and segments	– savePlaneAndSegmentSettings – restorePlaneAndSegmentSettings – setReadPlane: – setReadSegment: – setWritePlane: – setWriteSegment:

Class Methods

probe:

+ (BOOL)**probe:***deviceDescription*

Without checking for the presence of hardware, allocates and initializes an IOSVGADisplay. You shouldn't reimplement this method.

If initialization (done with **initFromDeviceDescription:**) is unsuccessful, this method returns NO. Otherwise, this method sets the device kind to "frame buffer", invokes **registerDevice**, and returns YES.

Instance Methods

enterSVGAMode

– (void)**enterSVGAMode**

Implemented by subclasses to put the display into SVGA mode. This method is invoked by the system when appropriate, such as when the window server starts running. This method should set up all the registers necessary for the selected mode, set the color palette, and clear the screen.

You should set the color palette to contain values for the four supported shades of gray in the first four entries; the rest of the entries should be zeroed out. NeXT drivers currently use the palette values shown in the following table.

Color	Palette Index	Value
Black	0	0
Dark gray	1	0x26
Light gray	2	0x34
White	3	0x3F

See also: – **revertToVGAMode**

getIntValues:forParameter:count:

– (IOReturn)**getIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int *)*count*

Handles NeXT-internal parameters specific to IOSVGADisplays; forwards the handling of all other parameters to **super**.

hideCursor:

– **hideCursor:**(int)*token*

Implements this method, as described in the IOScreenEvents protocol specification. You should never need to invoke or implement this method.

initFromDeviceDescription:

– **initFromDeviceDescription:***deviceDescription*

Invokes **initFromDeviceDescription:** on **super**. If successful, sets the unit number and the name (to "SVGADisplay" followed by the unit number). Frees itself if initialization was unsuccessful.

Subclasses must implement this method so that it performs all initialization necessary to set up the device and the driver. After invoking **initFromDeviceDescription:** on **super**, this method should determine its mode (invoking **selectMode:count:** or **selectMode:count:valid:**, if necessary) and set **[self displayMode]** to the IODisplayInfo appropriate for the mode. The driver should finish by invoking **mapFrameBufferAtPhysicalAddress:length:** and setting the IODisplayInfo's **frameBuffer** field to the value returned.

If possible, this method should check the hardware to see if it matches the IOConfigTable. If the hardware doesn't match, the driver should do what it can to ensure that the display is still usable.

See also: IODisplay class description ("IODisplayInfo")

mapFrameBufferAtPhysicalAddress:length:

– (vm_address_t)**mapFrameBufferAtPhysicalAddress:**(unsigned int)*address*
length:(int)*numBytes*

Maps the physical memory for this instance into virtual memory for use by the device driver. If *address* is 0, this method maps the physical memory corresponding to local memory range 0, and *numBytes* is ignored. If *address* is not 0, the reserved resources are overridden—*address* is used as the physical memory address and *numBytes* is used as the length. The mapped memory range is cached as IO_WriteThrough.

Note: When overriding reserved resources, you can't map memory outside of the memory range reserved for the device. However, you can map a subset of the memory range.

You should invoke this method during initialization.

Returns the virtual address that corresponds to *address*. If the memory mapping failed, this method logs an error message and returns NULL.

moveCursor:frame:token:

– **moveCursor:**(Point *)*cursorLoc*
frame:(int)*frame*
token:(int)*token*

Implements this method, as described by the IOScreenEvents protocol. You should never need to invoke or implement this method.

restorePlaneAndSegmentSettings

– (void)**restorePlaneAndSegmentSettings**

Implemented by subclasses to restore the plane and segment settings to the saved values. This method is invoked by IOSVGADisplay's cursor handling methods. The cursor handling methods invoke **savePlaneAndSegmentSettings**, do whatever is necessary to update the cursor, and then invoke **restorePlaneAndSegmentSettings** to restore the display's state.

Here's an example of implementing this method by saving the current settings into subclass-defined instance variables.

```
- (void)restorePlaneAndSegmentSettings
{
    IOWriteRegister(EIDR_SEQ_ADDR, SEQ_AT_MPK, writePlane);
    IOWriteRegister(EIDR_GCR_ADDR, GCR_AT_READ_MAPS, readPlane);
    outb(EIDR_GCR_SEGS, readSegment);
    outb(EIDR_GCR_SEGS, writeSegment);
}
```

revertToVGAMode

– (void)**revertToVGAMode**

Implemented by subclasses to remove the display from whatever advanced mode it's in and enter a mode in which it can be used as a standard VGA device. Implementing this method usually consists of setting registers that aren't used by VGA.

savePlaneAndSegmentSettings

– (void)**savePlaneAndSegmentSettings**

Implemented by subclasses to save the current plane and segment settings. This method is invoked by IOSVGADisplay's cursor handling methods. The cursor handling methods invoke **savePlaneAndSegmentSettings**, do whatever is necessary to update the cursor, and then invoke **restorePlaneAndSegmentSettings** to restore the display's state.

Each invocation of **savePlaneAndSegmentSettings** is followed by exactly one invocation of **restorePlaneAndSegmentSettings**, with no intervening invocations of

savePlaneAndSegmentSettings. In other words, the driver only has to remember one group of settings at a time.

Here's an example of implementing this method by saving the current settings into subclass-defined instance variables.

```
- (void)savePlaneAndSegmentSettings
{
    writePlane = IOReadRegister(EIDR_SEQ_ADDR, SEQ_AT_MPK);
    readPlane = IOReadRegister(EIDR_GCR_ADDR, GCR_AT_READ_MAPS);
    readSegment = inb(EIDR_GCR_SEGS);
    writeSegment = inb(EIDR_GCR_SEGS);
}
```

selectMode:count:

– (int)**selectMode:**(const IODisplayInfo *)*modeList* **count:**(int)*count*

Invokes **selectMode:count:valid:**, specifying 0 for the last argument.

selectMode:count:valid:

– (int)**selectMode:**(const IODisplayInfo *)*modeList*
count:(int)*count*
valid:(const BOOL *)*isValid*

Determines which IODisplayInfo in the driver-supplied *modeList* matches the value of the "Display Mode" key in the device's IOConfigTable. Drivers that support multiple advanced modes should invoke this method during initialization. When the driver receives a **enterSVGAMode** message, it should enter the mode selected by this method. If this method doesn't find a valid mode, the driver should determine a mode that will work.

The "Display Mode" key is a configuration key that can be used by drivers to support multiple modes—for example, 66 Hz and 72 Hz. IODisplayInfo is defined in the header file **driverkit/displayDefs.h**.

The *modeList* argument should contain an IODisplayInfo for each advanced mode the driver supports. The *count* argument should specify the number of IODisplayInfos in *modeList*. *isValid* should either be 0 (in which case it's ignored) or an array that corresponds to the *modeList*. If *isValid*[1] is NO, for example, then this method ignores the IODisplayInfo pointed to by *modeList*[1].

If this method finds a match, it returns the index of the matching IODisplayInfo in modeList. If the "Display Mode" key is missing or its value is improperly formatted, or if a corresponding IODisplayInfo isn't found, this method returns -1.

See the IODisplay class description for information on display modes and the IODisplayInfo type.

setBrightness:token:

– **setBrightness:**(int)*level* **token:**(int)*token*

Checks whether *level* is between EV_SCREEN_MIN_BRIGHTNESS and EV_SCREEN_MAX_BRIGHTNESS (inclusive). If not, logs an error message. Subclasses that support brightness changes should override this method. A typical implementation has code like this:

```
/*  Color palette constants (gamma 2.2, for typical CRT displays)
*/
#define WHITE_PALETTE_VALUE 0x3F
#define LIGHT_GRAY_PALETTE_VALUE 0x34
#define DARK_GRAY_PALETTE_VALUE 0x26
#define BLACK_PALETTE_VALUE 0
.
.
.
unsigned char val;

val = EV_SCALE_BRIGHTNESS(level, WHITE_PALETTE_VALUE);
/* Write val to the hardware's color palette entry for white */

val = EV_SCALE_BRIGHTNESS(level, LIGHT_GRAY_PALETTE_VALUE);
/* Write val to the entry for light gray */

val = EV_SCALE_BRIGHTNESS(level, DARK_GRAY_PALETTE_VALUE);
/* Write val to the entry for dark gray */

val = EV_SCALE_BRIGHTNESS(level, BLACK_PALETTE_VALUE);
/* Write val to the entry for black */
```

Returns **self**.

setIntValues:forParameter:count:

– (IOReturn)**setIntValues:**(unsigned int *)*parameterArray*
forParameter:(IOParameterName)*parameterName*
count:(unsigned int)*count*

Handles NeXT-internal parameters specific to IOSVGADisplays; forwards the handling of all other parameters to **super**.

See also: – **setIntValues:forParameter:count:** (IODevice)

setReadPlane:

– (void)**setReadPlane:**(unsigned char)*planeNum*

Implemented by subclasses to set which of two planes the display subsystem will

read from. Only one plane can be active at a time. Here's an example of implementing this method.

```
#define GRAPHICS_CONTROLLER_REGISTER_ADDRESS 0x03CE
#define SEGMENT_REGISTER_INDEX               0x09

- (void)setReadSegment: (unsigned char)segmentNum
{
    IOWriteRegister(GRAPHICS_CONTROLLER_REGISTER_ADDRESS,
                    SEGMENT_REGISTER_INDEX,
                    (segmentNum << 4));
}
```

See also: – **setWritePlane:**

setReadSegment:

– (void)**setReadSegment:**(unsigned char)*segmentNum*

Implemented by the subclass to set the 64KB segment the display subsystem will read from.

```
#define GRAPHICS_CONTROLLER_REGISTER_ADDRESS   0x03CE
#define PLANE_REGISTER_INDEX                   0x04
#define PROTECT_HIGH_REGISTER_BITS             0xFC

- (void)setReadPlane: (unsigned char)planeNum
{
    IOReadModifyWriteRegister(GRAPHICS_CONTROLLER_REGISTER_ADDRESS,
                              PLANE_REGISTER_INDEX,
                              PROTECT_HIGH_REGISTER_BITS,
                              planeNum);
}
```

See also: – **setWriteSegment:**

setWritePlane:

– (void)**setWritePlane:**(unsigned char)*planeNum*

Implemented by subclasses to set which of two planes the display subsystem will write to. Only one plane can be active at a time.

See also: – **setReadPlane:**

setWriteSegment:

– (void)**setWriteSegment:**(unsigned char)*segmentNum*

Implemented by the subclass to set the 64KB segment the display subsystem will read from.

See also: – **setReadSegment:**

showCursor:frame:token:

– **showCursor:**(Point *)*cursorLoc*
frame:(int)*frame*
token:(int)*token*

Implements this method, as described in the IOScreenEvents protocol specification. You should never need to invoke or implement this method.

IOTokenRing

Inherits From: IODirectDevice : IODevice : Object

Conforms To: IONetworkDeviceMethods

Declared In: driverkit/IOTokenRing.h

Class Description

IOTokenRing is an abstract class for controlling Token Ring devices. It provides a framework for sending and receiving packets (also known as *frames*), handling interrupts, and setting and detecting timeouts. It also provides an IONetwork instance that connects the driver with the kernel networking subsystem, as well as an I/O thread from which most of the IOTokenRing instance methods are invoked. To write a Token Ring driver, you create a subclass of IOTokenRing.

Implementing a Subclass

Your subclass of IOTokenRing must do the following:

- Implement **probe:** and **initFromDeviceDescription:**. These let your driver create instances of itself. The implementation of **probe:** should allocate an instance, if necessary, and invoke **initFromDeviceDescription:**. See the IODevice specification for more information on implementing **probe:**.
- Implement **resetAndEnable:**, and **interruptOccurred**. (**interruptOccurred** is documented in the IODirectDevice specification.)
- Implement either **transmit:** or **outputPacket:address:**.

IONetwork Method Usage

When your driver invokes IONetwork's method **handleInputPacket:extra:** to hand off a packet to the kernel, it needs to pass a valid pointer to a tokenHeader_t **struct** as the **extra:** argument. Passing 0 for this argument (as ethernet drivers do) won't suffice.

IONetworkDeviceMethods Protocol Implementation

In IOEthernet's implementation, finishInitialization invokes resetAndEnable:YES if

[self isRunning] == YES.

Recommended Reading

Besides the documentation for your hardware, see the references in the “Network Drivers” section of “Suggested Reading” in the Appendix to help you write a Token Ring driver.

Instance Variables

None declared in this class.

Adopted Protocols

IONetworkDeviceMethods
- allocateNetbuf
- finishInitialization
- outputPacket:address:
- performCommand:data:

Method Types

Creating and destroying IOTokenRing instances
- free
- initFromDeviceDescription:
- attachToNetworkWithAddress:

Transmitting packets
- transmit:

Setting and handling hardware timeouts
- setRelativeTimeout:
- relativeTimeout
- clearTimeout

Setting and getting the state of the hardware
- setRunning:
- isRunning
- resetAndEnable:

Setting and getting maximum sizes
- setMaxInfoFieldSize:
- maxInfoFieldSize

Getting other configuration information

- earlyTokenEnabled
- nodeAddress
- ringSpeed
- shouldAutoRecover

Instance Methods

attachToNetworkWithAddress:

– (IONetwork *)**attachToNetworkWithAddress:**(token_addr_t)*address*

Invokes **registerDevice**, sets the node address to *address*, creates an IONetwork instance, and attaches to the network subsystem by sending the IONetwork an **initForNetworkDevice:...** message. Besides starting up the IP protocol stack for the device, this method also starts up an 802.2-compliant Null SAP interface. Finally, this method logs a message stating the node address. Returns the IONetwork instance just created.

To determine the value to specify for *address*, first invoke **nodeAddress**. If **nodeAddress** returns a nonzero value, use that value. Otherwise, use the hardware's burnt-in address.

You invoke this method at the end of your implementation of **initFromDeviceDescription:**. You must invoke **resetAndEnable:NO** before invoking this method, as described under **initFromDeviceDescription:**, later in this specification.

clearTimeout

– (void)**clearTimeout**

If a transmission timeout is scheduled, unschedules the timeout. This method is normally invoked from a subclass's implementation of **interruptOccurred**.

See also: – **setRelativeTimeout:**, – **relativeTimeout**

earlyTokenEnabled

– (BOOL)**earlyTokenEnabled**

Returns YES if Early Token Release (ETR) is supported by the station; otherwise, returns NO. Stations that support ETR can co-exist with non-ETR stations in the ring. The value returned by this method is set by **initFromDeviceDescription:**.

free

– **free**

Frees the IOTokenRing instance and its resources and returns **nil**.

initFromDeviceDescription:

– **initFromDeviceDescription:**(IODeviceDescription *)*devDesc*

Invokes the superclass implementation, starts an I/O thread (using **startIOThread**), and sets the device name, kind, and unit.

Next, it examines the device configuration table for such parameters as ring speed and early token enablement. It then sets the maximum packet size, based on the ring speed. If the ring speed is 4 megabits per second, the maximum info field size is MAC_INFO_4MB. If the ring speed is 16, the maximum info field size is MAC_INFO_16MB. (The maximum packet size is the maximum info field size plus MAC_HDR_MAX.) These constants are defined in the header file **bsd/net/tokendefs.h**.

Subclasses of IOTokenRing should implement this method so that it invokes the superclass version of **initFromDeviceDescription:**, makes sure the configuration is correct, invokes **setMaxInfoFieldSize:**, does any other device-specific software and hardware initialization, and invokes **attachToNetworkWithAddress:**.

This method should free the instance and return **nil** on failure; otherwise, it should return **self**. A rough example of implementing this method is below.

```
- initFromDeviceDescription:(IODeviceDescription *)devDesc
{
    if([super initFromDeviceDescription:devDesc] == nil)
        return nil;

    /* Perform any 1-time hardware initialization. */

    /* Finish initializing the hardware. */
    [self resetAndEnable:NO];

    /* Do any additional software initialization; set the max info
     * field size; get the node address (as described in the
     * documentation of attachToNetworkWithAddress: */

    IOLog("%s: Token-Ring at port=%x irq=%d dma=%d speed=%d\n",
        [self name], base, myIrq, myDmaChan, [self ringSpeed]);

    network = [super attachToNetworkWithAddress:myNodeAddress];
    return self;
}
```

isRunning

– (BOOL)**isRunning**

Returns YES if the hardware is currently inserted in the ring; otherwise, returns NO.

See also: – s**etRunning:**

maxInfoFieldSize

– (unsigned int)**maxInfoFieldSize**

Returns the maximum size of the info field. This value is used by **allocateNetbuf**. It's also used as the maximum transfer unit specified to the network subsystem.

See also: – **setMaxInfoFieldSize:**

nodeAddress

– (token_addr_t)**nodeAddress**

Returns the node address for this station. Currently, only burnt-in addresses are supported. In the future, however, IOTokenRing will be able to initialize the node address from the device configuration table. The value returned by this method is set by **attachToNetworkWithAddress:**.

relativeTimeout

– (unsigned int)**relativeTimeout**

Returns the number of milliseconds until a transmission timeout will occur. If no transmission timeout is currently scheduled, this method returns zero.

See also: – **clearTimeout**, – **setRelativeTimeout:**

resetAndEnable:

– (BOOL)**resetAndEnable:**(BOOL)*enable*

Does nothing and returns YES. Subclasses of IOTokenRing must implement this method so that it resets and initializes the hardware. This method should invoke **setRunning:** to record the basic state of the device.

If *enable* is YES and the station is already in the ring, this method should do nothing but invoke **setRunning:** with a YES argument and return YES. If *enable* is YES and the station isn't in the ring, interrupts should be enabled and the station inserted in the ring; **setRunning:** should be used to update the device running status to YES or NO,

depending on the success of the insertion. If *enable* is NO, interrupts should be left disabled, the station should be removed from the ring, and **setRunning:** should be invoked with a NO argument.

This method should return YES if it encounters no errors (no matter what value *enable* has); if it encounters errors, it should return NO. For example, the result from **resetAndEnable:NO** should be YES if the reset is successful.

The only time this method is invoked, with the exception of any invocations from your IOTokenRing subclass implementation, is during initialization. Specifically, **resetAndEnable:YES** is invoked once in the I/O thread after **attachToNetworkWithAddress:** is invoked.

See also: – **setRunning:**

ringSpeed

– (unsigned int)**ringSpeed**

Returns the speed of the Token Ring, in megabits per second. This value, which is either 4 or 16, is set to the amount specified by the "Ring Speed" key in the device configuration table. If the value is missing or invalid, the ring speed is set to 16.

setMaxInfoFieldSize:

– (void)**setMaxInfoFieldSize:**(unsigned int)*size*

Sets the maximum size of the info field. This value is used by **allocateNetbuf**. It's also used as the maximum transfer unit specified to the network subsystem. Your subclass should invoke this method in its implementation of **initFromDeviceDescription:**.

See also: – **maxInfoFieldSize**

setRelativeTimeout:

– (void)**setRelativeTimeout:**(unsigned int)*timeout*

Schedules a timeout to occur in *timeout* milliseconds. When *timeout* milliseconds pass without the timeout being cleared (with **clearTimeout**), **timeoutOccurred** is invoked.

See also: – **clearTimeout**, – **relativeTimeout**, – **timeoutOccurred** (IODirectDevice)

setRunning:

– (void)**setRunning:**(BOOL)*running*

Sets whether the hardware is inserted into the ring. The value of *running* should be YES to indicate that the hardware is inserted; otherwise, it should be NO. This method is invoked only by methods in IOTokenRing subclasses—not by IOTokenRing's own method implementations. You should invoke this method in your implementation of **resetAndEnable:**.

See also: – **isRunning**

shouldAutoRecover

– (BOOL)**shouldAutoRecover**

Returns YES if the device should try to recover from a failed attempt at inserting itself into the ring or from an unexpected removal from the ring; otherwise, returns NO. IOTokenRing sets this value depending on the value of the "Auto Recovery" key in the device configuration table. This method is provided as a convenience for IOTokenRing subclasses that support automatic recovery.

transmit:

– (void)**transmit:**(netbuf_t)*packet*

Does nothing except free *packet*, using the **nb_free()** function. This method is invoked by the kernel networking subsystem when the hardware should transmit a packet.

Subclasses of IOTokenRing can implement this method or they can reimplement the method that invokes it: **outputPacket:address:**. To determine the number of bytes of data to be transmitted, use the **nb_size()** function. To get a pointer to the data, use **nb_map()**. After getting the information you need from *packet*, you should free it with **nb_free()**.

See also: – **outputPacket:address:** (IONetworkDeviceMethods protocol)

Configuration Keys

This section describes keys that can be used in **.table** files in driver and system configuration bundles (in **/NextLibrary/Devices**). The configuration system and Configure application are described in Chapter 4.

Some keys can have several values, expressed as a space-delimited list. Space-delimited lists have one space between elements, with nothing before the first or after the last element.

Key values that specify addresses are expressed as ranges. Ranges include both the start and end address. If a range consists of a single byte, it's indicated by specifying the same start and end address—for example, "0x0-0x0".

Driver Configuration Keys

The keys described in this section can be used in **.table** files in a driver's bundle. You can also specify your own keys. User and kernel modules alike can get the value of any key using the IOConfigTable class. Configure inspectors, which set some key values, use NXStringTable to do so; the NXStringTable corresponding to the instance configuration is available through the **table** instance variable of IODeviceInspector.

Here's an example of a default configuration file:

```
"Class Names" = "myTestDriver";
"Family" = "Example";
"Instance" = "0";
"Version" = "1.1";
"Driver Version" = "myTestDriver, 3.2 version, built by kw
8/20/93";
"DMA Channels" = "1";
"I/O Ports" = "0x0-0x0";
"IRQ Levels" = "2";
"Valid IRQ Levels" = "1 2 3 4";
"Memory Maps" = "0x20000-0x200ff";
"Server Name" = "myTestDriver";
```

See the driver bundles under **/NextLibrary/Devices** for more examples.

The following table shows the keys and explains when they must be used. Each key is explained in detail later in this section.

Key	Used For

“Auto Detect IDs”	Drivers that support device auto detection
“Auto Recovery”	IOTokenRing drivers
“Block Major”	Drivers with UNIX block entry points; optional
“Boot Driver”	Drivers that must be loaded at boot time
“Bus Type”	Drivers that aren’t EISA- or ISA-based
“Character Major”	Drivers with UNIX character entry points; optional
“Class Names”	All drivers that don’t specify “Driver Name”
“Default Table”	Instance tables only (inserted by Configure)
“Display Mode”	Display drivers
“DMA Channels”	Drivers that support DMA
“Driver Name”	Alternative to the preferred “Class Names”
“Driver Version”	All drivers
“Family”	All drivers
“Instance”	All drivers
“I/O Ports”	Drivers that need access to I/O ports
“IRQ Levels”	Drivers that support interrupts
“Location”	All drivers; optional
“Memory Maps”	Drivers that need access to mapped device memory
“Post-Load”	Drivers that require user-level help after loading
“PostScript Driver”	IOSVGADisplay drivers
“Pre-Load”	Drivers that require user-level help before loading
“Ring Speed”	IOTokenRing drivers
“Server Name”	All drivers (inserted by Driver Kit makefiles)
“Share IRQ Levels”	Drivers that use shared interrupts
“SVGA PostScript Driver Extension”	Display drivers that require a special PostScript driver
“Valid DMA Channels”	Drivers that support DMA; optional but recommended
“Valid IRQ Levels”	Drivers that support interrupts; optional but recommended
“Version”	All drivers
“VGA Memory Maps”	Display drivers
“16Mb Early Token”	IOTokenRing drivers

Keys

Auto Detect IDs

Example: “Auto Detect IDs” = “CPQ1234”;
or
“Auto Detect IDs” = “0x71789004 0x0e111234”;

This is a string used by Configure and installation software to identify hardware that

can be controlled by the device driver. The string is a space separated list of *auto detect IDs*, each of which is an identifier that can be used to match a device connected to an I/O bus.

The auto detect ID contains both a vendor ID and a 16-bit device ID. An ANSI committee assigns vendor IDs; the vendor assigns device IDs. The auto detect ID takes the form of a 7 character string described in the EISA specification. It consists of two fields: *VVVdddd*, where *V* is an upper-case letter, and *d* is a hexadecimal digit. The three letters *VVV* represent the vender code; the four digit hexadecimal number *dddd* represents the device ID. The combination of these two fields is guaranteed to be unique. For example, “CPQ” is the vendor ID for Compaq, so an ID of “CPQ1234” represents the Compaq device with device ID “1234”.

The 7 character format is the preferred form of the auto detect ID. However, this ID can also be expressed as a 32-bit hexadecimal number. The vendor ID is translated into a 16-bit hexadecimal number; the device ID is the same as in the other format. The layout in this format differs for each bus type. For the EISA bus, the *device* ID is in the lower 16 bits, and the *vendor* ID is in the upper 16 bits. For the PCI bus, the *vendor* ID is in the lower 16 bits, and the *device* ID is in the upper 16 bits.

Auto Recovery

Example: “Auto Recovery” = “YES”;

Used in IOTokenRing drivers to specify whether the driver should support automatic recovery from errors. See the IOTokenRing class specification for more information.

Block Major

Example: “Block Major” = “1”;

Used by some drivers with UNIX entry points to specify the device’s block major number. See the IODevice class specification for more information.

Boot Driver

Example: “Boot Driver”;

Specifies that the driver must be loaded at boot time. For example, SCSI controller drivers must typically be loaded at boot time so that the system can use the disks attached to the controller.

Bus Type

Example: “Bus Type” = “PCI”;

Indicates the type of bus the device uses. The current valid values are “EISA” (which includes ISA), “PCI” and “PCMCIA”. If the key isn’t present or valid, it defaults to “EISA”.

Character Major

Example: "Character Major" = "15";

Used by some drivers with UNIX entry points to specify the device's character major number. See the IODevice class specification for more information.

Class Names

Example: "Class Names" = "FloppyController FloppyDisk";
"Class Names" = "AHAController";

A space-delimited list of the classes in the relocatable object file that should receive **probe:** messages. This key is preferred to the "Driver Name" key, which may become obsolete.

Default Table

Example: "Default Table" = "ATIUltraPro";

Automatically inserted into **Instance*n*.table** files by Configure when necessary. You should never have to specify this key.

Display Mode

Example: "Display Mode" = "Width: 1024 Height: 768 Refresh: 76Hz ColorSpace: RGB:555/16";

Used by display drivers to specify the mode the display should be in. This key's value should be equivalent to one of the values assigned to the "Display Modes" key in the bundle's *Language*.**lproj/Localizable.strings** file.

The key's value should be of the form:

"Width:*width* **Height:***height* **ColorSpace:**(**BW:***bits* | **RGB:***xyz*/*w*) **Refresh:***rate* **Hz"**

White space and ordering are ignored, but correct capitalization is required. The color space parameter should be either **BW:** followed by the bits per pixel, or **RGB:** followed by the bits per color component and then the bits per pixel.

For example, the string shown below describes a display mode that's 800 pixels wide and 600 high, supports color at 16 bits per pixel (5 bits each of red, green, and blue per pixel), and has a refresh rate of 60 Hz.

```
Width: 800 Height: 600 ColorSpace: RGB:555/16 Refresh: 60 Hz
```

DMA Channels

Example: “DMA Channels” = “2”;
“DMA Channels” = “3 7”;

A space-delimited list of DMA channels that should be reserved for the device. You must specify default values with this key if your device performs DMA. The user can change the default values with the Configure application, subject to restrictions that you impose with the “Valid DMA Channels” key.

Driver Name

Example: “Driver Name” = “AHAController”;

This is obsolete; use the “Class Names” key instead. The “Driver Name” key is identical to the “Class Names” key, except that it doesn’t allow you to specify more than one class.

Driver Version

Example: “Driver Version” = “PROGRAM:Wingine PROJECT:displayDrivers-14 DEVELOPER:mflynn BUILT:NO DATE SET (-B used)”;
“Driver Version” = “myTestDriver, 3.2 version, built by kw 8/18/93”;

A string uniquely identifying the driver version. In the future, the system may display this string when appropriate.

Family

Example: “Family” = “Display”;

The family the device belongs to. Configure uses this key to group devices and to make sure that all essential device families are represented. Valid values are listed in the table below.

Value	Configure View
“Display”	Display (at least one is required in the system configuration)
“Pointing Device”	Mouse (at least one pointing device is required)
“Network”	Network
“SCSI”	SCSI
“Audio”	Audio
“Keyboard”	Other (at least one keyboard is required)
“Disk”	Other

The “SCSI” value should be used only for SCSI controllers—not for SCSI devices such as tape drives. The “Disk” value should be used for both disks and disk controllers (except for SCSI controllers). For example, the IDE disk and IDE controller drivers (which are in the same relocatable object file) have the value

“Disk” in their default configuration file.

Values besides those listed in the table above are permitted, but aren’t treated specially. They’re included in the Configure view labeled Other. Examples of other values in use include “Parallel” and “Serial”.

Instance

Example: “Instance” = “0”;

The instance number of this configuration file. Configure automatically specifies this key in **Instance*n*.table** files, but you should specify “Instance” = “0” in default files.

I/O Ports

Example: “I/O Ports” = “0x170-0x177”;
“I/O Ports” = “0x3f8-0x3ff 0x2f8-0x2ff”;

A space-delimited list of I/O port ranges that should be reserved for the device. You must specify default values with this key if your driver uses I/O ports to get access to the device. If your driver uses Configure’s default inspector, the user can change the starting address of the first range (but not the length of the range) using the inspector.

IRQ Levels

Example: “IRQ Levels” = “1”;
“IRQ Levels” = “4 3”;

A space-delimited list of interrupts (IRQs) that should be reserved for the device. You must specify default values with this key if your device interrupts. The user can change the default values with the Configure application, subject to restrictions that you impose with the “Valid IRQ Levels” key.

Location

Example: “Location” = “Slot 3”;

The location of the device. This string is set automatically by the device auto detection software and has a different format for each bus.

EISA	
“Slot *n*”	where *n* is replaced by a slot number, as in “Slot 1”.
PCI	
“Dev:*d* Func:*f* Bus:*b*”	where *d* is the device number, *f* is the function number, and *b* is the bus number; “Dev:6 Func:0 Bus:0”, for example.

Memory Maps

Example: "Memory Maps" = "0x0D0000-0xD3FFF";
"Memory Maps" = "0xa0000-0xbffff 0xc0000-0xcffff";

A space-delimited list of memory ranges that should be reserved for the device. You must specify default values with this key if your driver needs access to mapped device memory. If your driver uses Configure's default inspector, the user can change the starting address of the first range (but not the length of the range) using the inspector.

Post-Load

Example: "Post-Load" = "InstallPPDev";

A user-level program to be run just after the driver is loaded. In the example above, the executable file **InstallPPDev** is a file in the driver's bundle that installs the driver's device files.

PostScript Driver

Example: "PostScript Driver" = "/usr/lib/NextStep/Displays/SVGA_psdrvr";

Used by display drivers to specify the PostScript driver that matches them. IOFrameBufferDisplays don't specify this key, since they use the default PostScript driver. IOSVGADisplay drivers, however, must specify the SVGA PostScript driver, as shown above. See the IOSVGADisplay class description for more information.

Pre-Load

Example: "Pre-Load" = "RemovePPDev"
;

A user-level program to be run just before the driver is loaded. In the example above, the executable file **RemovePPDev** is a file in the driver's bundle that removes the driver's old device files before the driver is loaded.

Ring Speed

Example: "Ring Speed" = "4"
;

Used by IOTokenRings to specify the speed of the Token Ring. This must be either 4 or 16. See the IOTokenRing class specification for more information.

Server Name

Example: "Server Name" = "ATI";

Indicates the name of this driver's bundle, minus the **.config** suffix. You shouldn't need to specify this key, since it's inserted automatically by the Driver Kit makefiles. For information on using the Driver Kit makefiles, refer to Chapter 4.

Share IRQ Levels

Example: "Share IRQ Levels" = "Yes";

Indicates whether the device uses shared interrupts or not. On EISA and PCI systems, using shared interrupts implies using level-triggered interrupts. The value is either "Yes" or "No" with the default being "No". Shared interrupts are not supported on ISA bus computers.

SVGA PostScript Driver Extension

Example: "SVGA PostScript Driver Extension" = "CirrusLogicGD542X_psdrvr";

Used by IOSVGADisplay drivers to specify the driver-specific module to be loaded into the SVGA PostScript driver. See the IOSVGADisplay class description for more information.

Valid DMA Channels

Example: "Valid DMA Channels" = "0 1 3 5 6 7";
"Valid DMA Channels" = "2"
;

A space-delimited list of DMA channels that can be used by the device. When the user inspects the device, Configure automatically dims every DMA channel that isn't valid, so that the user can select only valid channels. See also the "DMA Channels" key.

Valid IRQ Levels

Example: "Valid IRQ Levels" = "1";
"Valid IRQ Levels" = "11 12 14 15"
;

A space-delimited list of interrupts (IRQs) that can be used by the device. When the user inspects the device, Configure automatically dims every IRQ that isn't valid, so that the user can select only valid IRQs. See also the "IRQ Levels" key.

Note: IRQ 2 can't be used on ISA- and EISA-based machines, so it should never be in the "Valid IRQ Levels" list.

Version

Example: "Version" = "1.0";
"Version" = "2.1";

A floating point number that describes the version of this driver. In the future, the system may warn the user whenever the user attempts to install a driver that has a lower version than the already installed version of the same driver. By convention, the number before "." should change only when the driver is incompatible (for user-level clients) from earlier versions. Configure display this version string.

VGA Memory Maps

Example: "VGA Memory Maps" = "0xa0000-0xbffff 0xc0000-0xcffff";

A space-delimited list of memory ranges used for VGA access. Every display driver's default configuration table must include this key with the value "0xa0000-0xbffff 0xc0000-0xcffff".

16Mb Early Token

Example: "16Mb Early Token" = "YES";

Used in IOTokenRing drivers to specify whether the driver should support early token release. See the IOTokenRing class specification for more information.

System Configuration Keys

The keys described in this section are used in **.table** files in the system bundle. You don't usually have to specify any of the keys except perhaps the "Kernel Flags" key.

Active Drivers

Example: "Active Drivers" = "EtherExpress16 ParallelPort ATI Beep";

Drivers to be loaded automatically and probed after boot time. Configure automatically adds drivers to either this list or the "Boot Drivers" list whenever the user adds a driver to the system configuration. By default, drivers are added to this list; if the default table contains the "Boot Driver" key, however, the driver is added to the "Boot Drivers" list.

Boot Drivers

Example: "Boot Drivers" = "PS2Keyboard BusMouse DPT2012 IDE Floppy";

Drivers to be loaded and probed at boot time. See also "Active Drivers", above.

Boot Graphics

Example: “Boot Graphics” = “Yes”;

Specifies whether graphics (instead of system messages) should be displayed during boot time.

Bus Type

Example: “Bus Type” = “ISA”;
“Bus Type” = “EISA”;

The system bus architecture. This key isn’t currently used for the System Configuration.

Kernel

Example: “Kernel” = “mach_kernel”;

The name of the kernel to use.

Kernel Flags

Example: “Kernel Flags” = “rootdev=sd1a”;

Options to pass to the kernel.

Machine Name

Example: “Machine Name” = “Dell 450DE/2 DGX”;

The system manufacturer name/model. This key isn’t currently used.

Version

Example: “Version” = “1.0”;

Used by the Configure application.

Suggested Readings on Writing Device Drivers

These references provide useful information in a variety of areas for driver writers.

NeXT Documentation

NEXTSTEP General Reference

This reference manual describes the Mach Kit, which contains the NXLock and NXConditionLock classes.

NEXTSTEP Development Tools and Techniques

This manual tells how to use development tools such as ProjectBuilder and **gdb**.

NEXTSTEP Operating System Software

This manual has information on the Mach Operating System and using Mach messages. It contains extensive material on writing Loadable Kernel Servers.

NEXTSTEP Object-Oriented Programming and the Objective C Language

This book explains the basic concepts of Objective C including Objective C messages, protocols, and categories.

NeXTanswers on archive sites

These files contain much useful information on NeXT device drivers and the Driver Kit.

Mach Operating System

Programming under Mach. Joseph Boykin, David Kirschen, Alan Langerman, and Susan LoVerso. Addison-Wesley, 1993.

An introduction to Mach tasks, threads, interprocess communication, and memory management.

General Driver Writing

Writing a UNIX™Device Driver, Second Edition. Janet I. Egan and Thomas J. Teixeira. John Wiley and Sons, 1992.

An excellent general introduction to UNIX drivers. Make sure you specify the second edition since the first one is very specific to System V and MassComp in particular.

Writing Device Drivers for SCO™UNIX™, A Practical Approach. Peter Kettle and Steve Statler. Addison-Wesley, 1993.

This book includes some details of Intel hardware. It contains a good reference section.

Buses

EISA System Architecture, Second Edition. Tom Shanley. Mindshare Press, 1993.

ISA System Architecture, Second Edition. Tom Shanley and Don Anderson. Mindshare Press, 1993.

PCI System Architecture, Second Edition. Tom Shanley. Mindshare Press, 1993.

This book tells how to work with a version 2.0 compliant bus.

PCMCIA System Architecture, Tom Shanley. Mindshare Press, 1994.

All of these books are distributed by Computer Literacy Bookshops.

Display Drivers

Programmer's Guide to the EGA and VGA Cards, Second Edition. Richard F. Ferraro. Addison-Wesley.

Network Drivers

Besides the documentation for your hardware, the following references can help you write a Token Ring driver.

Computer Networks. Andrew S. Tanenbaum, Prentice Hall, 1981.

Has information on networking, in general.

International Standard ISO/IEC 8802-3; ANSI/IEEE Std. 802.3.

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 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.