Network Device Driver Programming Guide

Hardware & Drivers > Networking

2008-03-11

Contents

Figures

Introduction to Network Device Driver Programming Guide

> **Important:** This document is in a preliminary stage of completion. Although it has received some techincal review, there may be changes and/or additions to some of the information provided here.

Network Device Driver Programming Guide is an introduction to developing network device drivers, and a companion to the source code available in the Darwin Projects Directory, http://www.opensource.apple.com/darwinsource/Current. You will find this document most useful if you examine a sample network driver as you read it. This document will refer to the AppleUSBCDCDriver. Code for this driver can be found at http://www.opensource.apple.com/darwinsource/tarballs/apsl/AppleUSBCDCDriver-314.4.1.tar.gz.

This book assumes some familiarity with programming the Mac OS X kernel and the I/O Kit. For a broad overview of the Mac OS X kernel see *Kernel Programming Guide*. If you need more information about the I/O kit, please read *I/O Kit Device Driver Design Guidelines* and *I/O Kit Fundamentals*.

Who Should Read This Document?

Network Device Driver Programming Guide is intended for anyone who wants to develop network drivers for Mac OS X.

Information on the Web

Apple maintains several websites where developers can go for general and technical information on Mac OS X.

- Darwin Projects Directory—Open source drivers for use with Darwin (http://www.opensource.apple.com/darwinsource/Current/)

- Apple Developer Connection—Developer Documentation (http://developer.apple.com/documentation). Features the same documentation that is installed on Mac OS X, except that often the documentation is more up-to-date. Also includes legacy documentation.

- AppleCare Knowledge Base (http://www.apple.com/support). Contains technical articles, tutorials, FAQs, technical notes, and other information.

- Apple Developer Connection—Mac OS X (http://developer.apple.com/macosx). Offers SDKs, release notes, product notes and reviews, and other resources and information related to Mac OS X.

I/O Networking Family API

The network family defines a framework for building drivers of network controllers. It includes the superclasses for network drivers, which transfer network packets to and from the controller hardware, and for network interfaces, which serve as nubs connecting the BSD networking stack to the driver. Writing a driver involves creating a subclass of the appropriate superclass for a controller driver. The network family currently supports ethernet controllers, with an abstract superclass for controller drivers and a concrete interface class. It's also possible to add support for new controller types, such as token ring or FDDI, by creating new classes for drivers and interfaces.

A network driver is a provider within the network family, but it is also a client of whatever bus the physical device connects to. The CDC Ethernet driver, for example, uses an IOUSBDevice nub to connect to the USB bus. For information on client setup, see the document appropriate for that device.

Network Family Architecture

The network family comprises a small group of core driver and nub classes, along with a host of utility and helper classes. Figure 2-1 (page 9) shows the inheritance hierarchy for the network family. The principal driver classes are IONetworkController, which is the superclass for any kind of network driver, and IOEthernetController, which is the superclass for an ethernet controller driver. A network controller object publishes a nub that forms an interface between the driver and the BSD Data Link Interface Layer, or DLIL. The superclass for a network interface is, IONetworkInterface; the IOEthernetInterface class defines the interface for ethernet controllers. You create a new interface class if you are extending the network family to add support for a new networking hardware protocol, such as FDDI.

Figure 1-1   Network family inheritance hierarchy
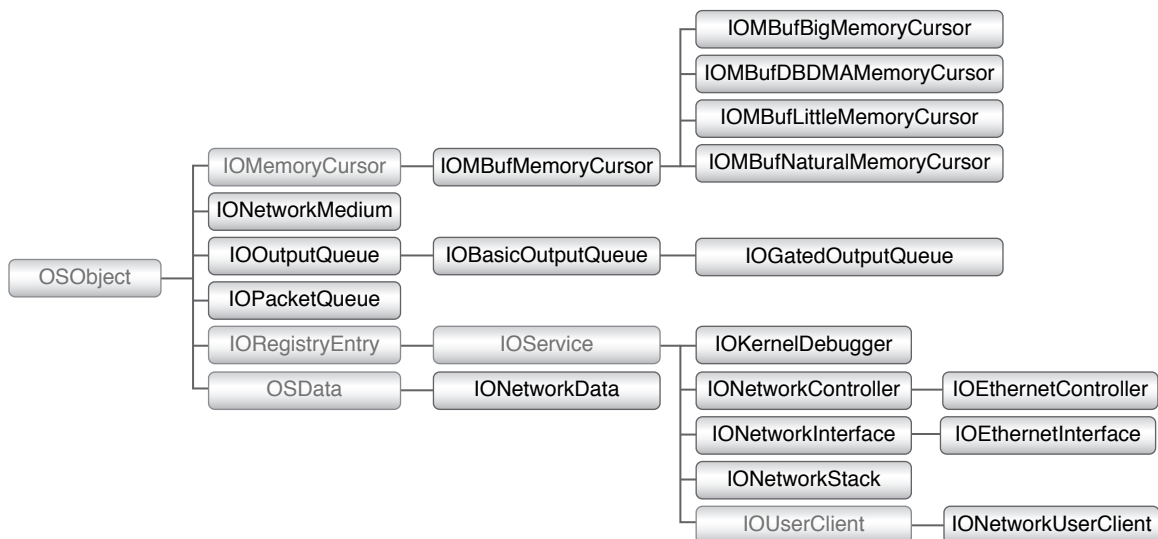
The remaining classes are mostly helpers or managers of data for the network controller and interface objects. An IONetworkStack object connects a network interface object to the BSD networking stack. The various IOMBufMemoryCursor classes work like the IOMemoryCursor classes used in the rest of the I/O Kit, but on BSD `mbuf_t` structures rather than on IOMemoryDescriptor objects. An IONetworkMedium describes a physical link medium of the hardware. IOOutputQueue and its subclasses, along with IOPacketQueue, perform packet queuing for network controller objects. IONetworkData is used to hold standard C structures used by the BSD networking layer so that I/O Kit driver can access them.

A few other classes are less relevant to most network drivers. An IOKernelDebugger object is used in place of a network interface object when a computer is being debugged through the Kernel Debugger Protocol (KDP). The drivers for the built-in network controllers on Macintosh computers include support for KDP; third-party drivers rarely need to support it. An IONetworkUserClient object gives user-space programs direct access to networking services. Since networking services are typically made available to the whole system, this class is also of less relevant for third-party network drivers.

A driver in the network family is a subclass of the abstract IONetworkController class, typically through an intermediate protocol-specific class such as IOEthernetController. Figure 2-2 (page 10) shows a simplified view of the IORegistry stack for an ethernet driver. In this figure the driver is a client of the family representing the bus that the controller card uses, and a provider to a network interface object. In the same manner as IONetworkController, IONetworkInterface is the abstract superclass for interfaces, with subclasses for specific network protocols.

Figure 1-2   Network family objects in the IORegistry
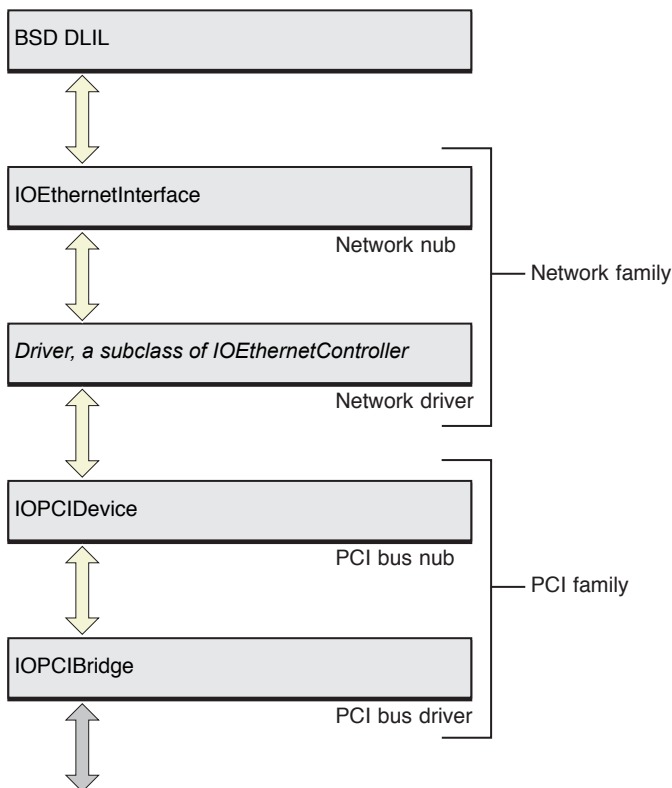
The network interface object represents the generic interface to a given networking protocol. In its provider role, it offers a device-independent networking interface to the operating system, and in its client role it relies on the driver to implement its services. The network family currently includes an interface object class for the ethernet protocol. It is possible to add support for other protocols by creating new subclasses of the abstract controller and interface classes.

Above the network interface is the BSD data link interface layer (DLIL). The interface object presents itself to the DLIL as a standard BSD netif-style interface, providing the appropriate callbacks and invoking DLIL routines for handling network traffic. Drivers don't interact with the DLIL, but interfaces do. If you're extending the network family to add support for a new hardware protocol, you'll need to interface with the DLIL. See *Network Kernel Extensions Programming Guide* for more information.

Figure 1-3   Network family objects used by a driver
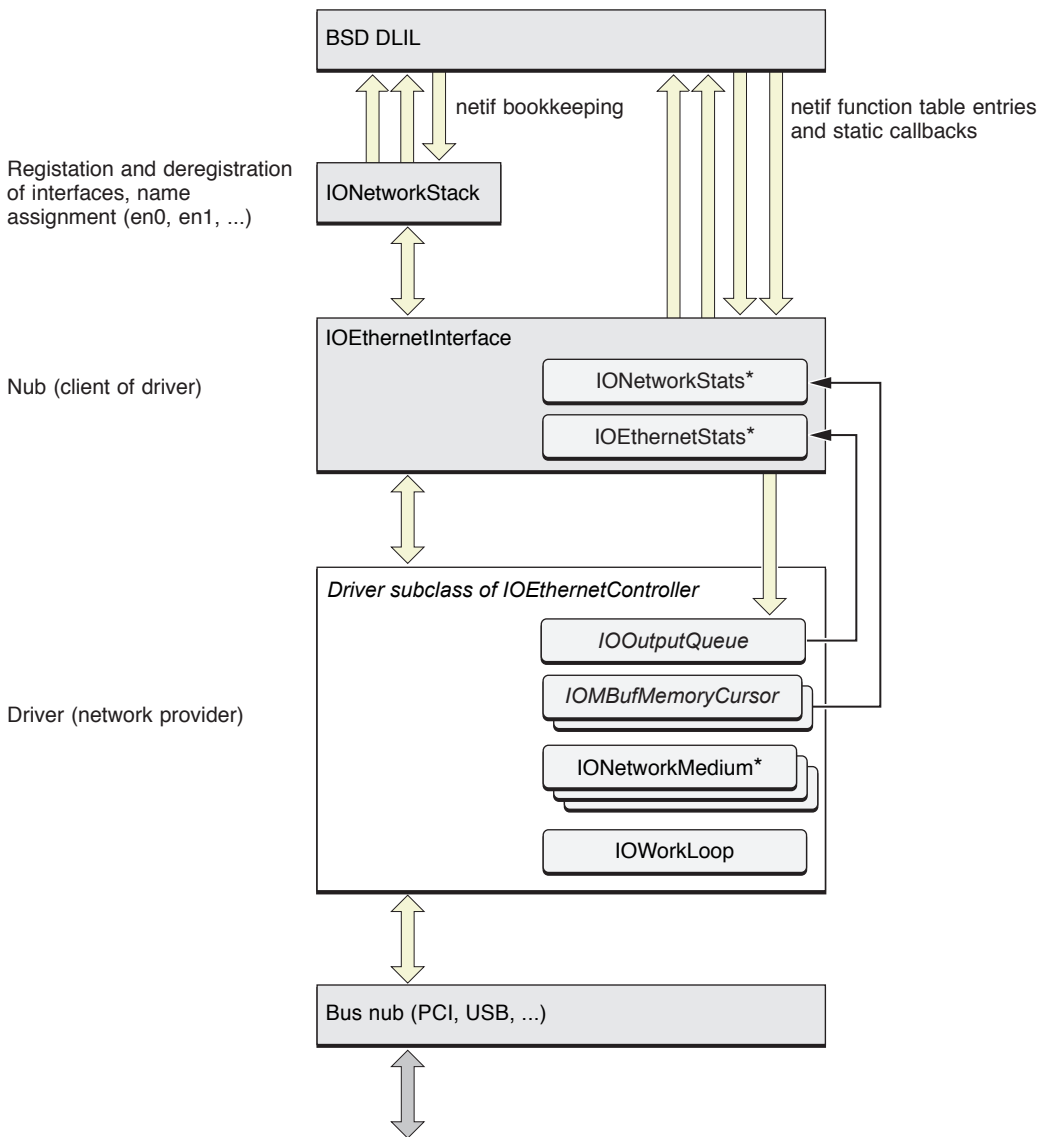
* Visible in the I/O Kit registry (as in IORegistryExplorer)

The driver itself makes use of several other classes and structures defined by the network family and by the I/O Kit in general. These are shown in Figure 2-3 (page 11), which presents a more complete picture of how a network driver interacts with these objects. The IOOutputQueue class defines an abstract interface for queuing packets to be transmitted; subclasses provide different locking and synchronization mechanisms. IOMBufMemoryCursor objects help the driver manage BSD `mbuf_t` structures, building scatter/gather lists and coalescing buffers as necessary. IONetworkMedium objects describe the physical link media of the hardware. The driver can collect statistics using two structures maintained by the network interface, IONetworkStats and a protocol-specific structure, in this case IOEthernetStats. As a driver that handles interrupts, a network driver typically creates its own IOWorkLoop object. An IONetworkStack object moderates the registration and deregistration of interfaces, assigning unique names as expected by BSD (such as `en0`) and performing other netif bookkeeping.

"Writing a Driver for an Ethernet Controller" (page 17) describes how to set up these structures for your network driver.

Tips on Bringing Up a UNIX Network Driver

If you've never brought up a network driver on a UNIX operating system before, you'll want to learn about the tools you can use to set up interfaces, test packet transmission and reception, and gather information. Detailed documentation of these tools is outside the scope of this manual, but you can always use the UNIX `man` command to display the standard system manual pages in a Terminal window.

This chapter assumes you're using two machines to bring up your driver, one that has your driver, and the other to act as a source or destination for network traffic. Figure 3-1 (page 13) shows a typical setup, with the ethernet and IP addresses that will be used. The machine that contains your driver will be referred to as the target. The second machine is typically the machine you use to debug your driver, and will be referred to here as the debug host.

Figure 2-1   Two-machine target-debug setup

en1: 00:90:27:94:8f:21
IP: 192.168.1.1

en2: 00:90:27:23:12:f3
IP: 192.168.1.2
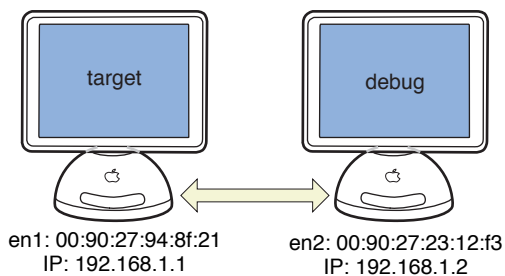
The IP addresses shown are merely examples. If you are debugging on the wider network (which is not recommended), you should use legal IP addresses for that network. If you are debugging over a private link between the two machines, you can choose any IP addresses, as long as you set up static address binding as shown below.

Activating the Network Link

After you load your network driver using `kextload`, you have to find out which network interface name it was assigned, and then bring up a network link for that interface. You use the `ifconfig` program to do this. If you are using a dedicated link between two machines to avoid flooding the wider network, you may also have to bring up the dedicated link on the debug host.

> **Note:** You must be logged in as `root`, or preface your command with `sudo`, to set up a network interface.

To get a list of network interfaces, run `ifconfig` with the `-a` flag. The interface for your driver should be listed without an "inet" field, and with the ethernet address of the controller. Here's an example for the target machine shown in Figure 3-1 (page 13):

```
# ifconfig -a
lo0: flags=8009<UP,LOOPBACK,MULTICAST> mtu 16384
    inet 127.0.0.1 netmask 0xff000000
en0: flags=8863<UP,BROADCAST,b6,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    inet 17.202.40.235 netmask 0xfffffc00 broadcast 17.202.43.255
    ether 00:05:02:3b:45:cb
en1: flags=822<BROADCAST,b6,SIMPLEX> mtu 1500
    ether 00:90:27:94:8f:21
```

In this case, the existing interface `en0` represents the built-in ethernet port on the computer. The interface `en1` represents the new driver being brought up. To bring up the link on the added interface, run `ifconfig` on it with an IP address and a netmask, and specify `up`, as shown here:

```
# ifconfig en1 inet 192.168.1.1 netmask 255.255.255.0 up
```

When you run `ifconfig` with the `up` argument, you are directly calling the device driver's `enable` function. This function will be described in depth in "Writing a Driver for an Ethernet Controller" (page 17).

If you are using a private link, and have not configured the debug machine to set up its link at startup, you can use `ifconfig` on it as well. With the example setup, the command would look like this:

```
# ifconfig en2 inet 192.168.1.2 netmask 255.255.255.0 up
```

To make some aspects of testing easier, you may want to set up static IP address binding for the two machines. Normally this task is performed automatically by your computer, but if your network driver is not transmitting and receiving packets, the binding will never complete. You do this by running the `arp` command on each machine, giving the IP address and the hardware address for the other machine. Based on the examples above, on the target host you would enter this command:

```
# arp -s 192.168.1.2 00:90:27:23:12:f3
```

And on the debug host you would enter this command:

```
# arp -s 192.168.1.1 00:90:27:94:8f:21
```

With that, you have a network connection between the two systems, which you can use to test your driver.

Testing I/O

While developing your driver, you will want to test that either receiving or transmitting is working before both are. Since your driver is not yet capable of handling round-trip network traffic, you have to use different tools to check these two cases. You will likely be using plenty of logging in your driver as well to make sure things are working as you expect.

To verify that your driver is receiving packets, you can use the `ping` utility from the debug host to send packets to the target machine, and use logging in the driver's packet-reception code to verify that packets are being received. Running `ping` normally performs an ARP request to get the hardware address for the target. If your driver can't transmit yet, `ping` will fail if you did not set up a static IP address binding with the `arp` command. If you do not want to set up a static IP address binding, you can run `ping` on the broadcast address for the private network, bypassing address resolution.

To verify that your driver is sending packets, you can use `ping` on the target machine to send packets to the debug host. If your driver does not yet support receiving, you can use the `tcpdump` utility on the debug host to verify that packets are coming from the target machine. To do this, specify the `-v` flag for verbose output, and the `-i` flag to name the interface to examine.

> **Note:** You must be logged in as `root`, or preface your command with `sudo`, to set up a network interface

If the debug machine is connected to the target on the interface `en2`, for example, you would enter this command:

```
# tcpdump -v -i en2
```

With `tcpdump` running, you can start `ping` on the target machine and examine the output of `tcpdump` on the debug host. If there is no output, your driver is not transmitting. If there is output, verify that the IP address for your driver's network interface on the target system appears in the output. If it does, your driver is successfully transmitting packets.

Once your driver can both receive and transmit, you can just use `ping` without setting up static IP address binding, and perform tests to verify data integrity.

Testing Statistics-Gathering

If you implement general statistics-gathering in your network driver, you can verify that it is working with the `netstat` program. Run this program with the `-I` flag, followed by the name of your driver's network interface. Here is an example:

```
% netstat -I en1
Name  Mtu    Network       Address            Ipkts Ierrs   Opkts Oerrs  Coll
en1   1500   <Link>        00.90.27.94.8f.21    17     0      19     0     0
en1   1500   192.168.1     192.168.1.1          17     0      19     0     0
```

The output shows that the driver has received 17 packets (Ipkts, or incoming packets) and transmitted 19 (Opkts, or outgoing packets). See the UNIX man page for `netstat` for more information.

Unloading the Driver

Once you have enabled IP communication on a device using `ifconfig`, the IP protocol is attached to your driver inside the kernel's networking stack. (See *Network Kernel Extensions Programming Guide* for more information about the networking stack.) Once a protocol is attached, your driver cannot be unloaded until you detach the protocol.

To detach the IP protocol from your network device, you can issue the following command from the command line:

```
ipconfig set en1 NONE
```

Substitute the appropriate interface name instead of `en1`, of course. This will disable IP networking for the interface. You can then unload the driver KEXT using `kextunload`.

For more information, see the manual pages for `ipconfig(8)` and `kextunload(8)`.

Writing a Driver for an Ethernet Controller

This chapter describes what a network driver does to set up its interface object, to handle I/O, and to perform its other tasks. Follow the CDC Ethernet driver provided at http://www.opensource.apple.com/darwinsource/tar-balls/apsl/AppleUSBCDCDriver-314.4.1.tar.gz for specific examples of how to implement the functionality required of a network driver. In particular this document will refer to the functions in `AppleUSBCDCEMData.cpp`.

Driver Overview

Before diving into the specifics of what you need to be do to write a network driver, it may be helpful to see the basic functions which you need to override. These descriptions are meant to give you an idea of what is entailed in creating a very simple network driver.

`start`

The `start` function should initialize the device to a working state. It also needs to create a network object and make it visible to the networking stack as an interface.

`stop`

The `stop` function must free anything allocated in `start` and also release the network object created in `start`.

`enable`

As mentioned in "Tips on Bringing Up a UNIX Network Driver" (page 13), the `enable` function is run when the system sets the driver's status to `up`. This function is also responsible for starting the hardware's transmit and receive capabilities. It should inform the system about the link status of the hardware.

`disable`

The `disable` function releases anything allocated, and stops any functions started in `enable`.

`getHardwareAddress`

The `getHardwareAddress` function returns the MAC address of the network device.

`outputPacket`

The `outputPacket` function sends the packet to the hardware for transmission. It will be called from multiple threads, so it needs to be thread-safe.

Each of these functions are described in more depth further in this chapter.

Startup and Shutdown

A network driver's `start` function is responsible for setting up the resources the driver needs all the time, whether the driver is enabled or disabled. These include the network interface object, a work loop, and an output queue, along with whatever specific resources the driver needs. In addition, the driver should retain

its provider nub (that is, the nub for the bus it is attached to, such as an IOUSBDevice nub). The `stop` function reverses all this, releasing the nub and disposing of any resources created in `start`. See the example driver for samples of these two functions.

The `start` function is typically not the place for a driver to allocate its transmit and receive buffers and other such runtime resources that are needed only while the driver is enabled and handling network traffic. Creation and disposal of these resources is managed by the `enable` and `disable` functions, described in "Enabling and Disabling the Driver" (page 20).

Opening the Provider Nub

You invoke the `start` function with a single argument, the nub of the hardware device that your ethernet controller is connected to. The driver should do three things with the nub: verify that it is of the appropriate class, retain a reference to it, and open it in order to access its services.

Your driver has two options before returning from the `start` function. If you close the nub at the end of the `start` function, then you must reopen it in the `enable` function. However, you also have the option to leave the nub open, so that it will be ready to use in the `enable` function. It is recommended that the former technique be used, but because a network driver is rarely "downed," either technique is acceptable. The sample code uses the latter method.

Setting up Output Queuing

IONetworkController, the superclass of all network drivers, defines the `outputPacket` function for packet transmission. This function is typically called indirectly by the network interface through the network family's standard queuing mechanism, defined by IOOutputQueue and its subclasses. A driver can choose to have it invoked directly by the network interface, however, and implement its own internal queuing mechanism.

A driver's output queue is created by the `createOutputQueue` function. The network driver superclass, IONetworkController, invokes this function automatically on startup. The driver should override this function to create an instance of an IOOutputQueue subclass suitable for the driver, or not override it at all if the driver performs its own queuing—in which case the network interface will pass outgoing packets directly to the driver.

The simplest IOOuputQueue subclass to use is IOGatedOutputQueue. This class queues packets using a lock and dequeues them one at a time using a command gate on the driver's work loop. It invokes `outputPacket` once for each packet, breaking up `mbuf_t` packet chains so that the driver doesn't have to handle them. Using the work loop means that transmit and receive operations are mutually exclusive. If your network hardware has completely distinct transmit and receive engines, using an IOGatedOutputQueue object may not be the most efficient option. Even so, it makes bringing up a driver quite simple, as you do not have to worry about locking while you establish control of the hardware itself.

If your network hardware supports multithreaded access, as it does when the transmit and receive engines are distinct, you may want to use an IOBasicOutputQueue instead. This class is the actually the superclass of IOGatedOutputQueue, and it defines the locking mechanism that IOGatedOutputQueue uses. For dequeuing, however, it negotiates multiple queuing threads so that only a single thread at a time dequeues packets and invokes the driver's `outputPacket` function. This queue subclass also breaks up `mbuf_t` packet chains for the driver. Because dequeuing isn't synchronized with the work loop, however, both transmission and reception can occur at the same time.

If a driver performs queuing internally, it doesn't override `createOutputQueue`, and it implements its own queuing mechanism. If you are porting a driver that already performs its own output locking, for example, you will probably choose this option. In this case, when the network interface needs to transmit packets, it invokes the driver's `outputPacket` function directly. The driver is then responsible for all locking of its internal queue and related data, for handling stall conditions, and further for handling `mbuf_t` packet chains. The output queue classes guarantee that a single packet at a time is passed to `outputPacket`; this is not the case when the driver forgoes using an output queue object.

Another option a driver has is to create its own subclass of IOOutputQueue. See the class reference documentation and the source code for more information.

> **Note:** While this document refers to the new `mbuf_t` structure, introduced in Mac OS v10.4, the sample code still uses an `mbuf` structure. The `mbuf` structure is supported in order to preserve binary compatibility of drivers on pre-Mac OS v10.4 systems. However, if you are writing a driver for a Mac OS v10.4 system, it is highly recommended that your driver use the `mbuf_t` structure.

Setting up a Network Interface

As with any driver, a network driver is responsible for creating its nub. A network nub is an instance of a subclass of IONetworkInterface. Each type of network controller has a corresponding subclass of IONetworkInterface; an IOEthernetController, for example, uses an IOEthernetInterface. Creating the network interface is implemented by the driver's superclass; all the driver need do is call the `attachInterface` function and implement a few auxiliary functions. If the driver includes a custom subclass of the network interface class, it can override `createInterface` to create an instance of the custom subclass rather than the default class.

The `attachInterface` function takes two parameters: a pointer to the network interface object, which is filled by the function, and an optional "register with the network stacks" flag. This flag is `true` by default, meaning that the interface object will be registered with the DLIL, that the driver's `enable` function may be invoked during startup and that the driver may receive requests to transmit packets before `attachInterface` returns (and before `start` completes). If your driver needs to perform additional initialization, you can pass `false` to delay registering with the network stacks. If you do this, your driver must invoke the interface's `registerService` function when it becomes ready to handle network traffic.

Because a network interface can't be reconfigured after it registers itself with the BSD network stack, it invokes a callback on the driver from the `attachInterface` function. The callback is `configureInterface`, and the driver can implement it to set up maximum transfer sizes, filter tap modes, and other such settings as described in the IONetworkInterface reference documentation. Network drivers typically use this function to get their references to the statistics structures maintained by the network interface before any network traffic comes by. (See below for information on how to do this.)

For ethernet drivers, another function related to setting up the interface is `getHardwareAddress`, which the interface object invokes in order to register the ethernet address with the BSD network stack. Before invoking `attachInterface`, the driver should determine in the `start` function what the hardware address is, and implement `getHardwareAddress` to provide this address on request. Other networking protocols may require a similar function.

Defining Capabilities, Restrictions, and Modes

This section presents some useful functions that will be explained more fully in later versions of this book.

A network driver overrides a number of IONetworkController functions in order to advertise its capabilities and restrictions. Some of these functions, and their uses, are:

- `setMaxTransferUnit`, which allows clients to set the largest single packet size to transfer.

- `getPacketBufferConstraints`, which informs the driver's superclass of alignment constraints for packet buffers.

- `getVendorString`, which provides the vendor name for the hardware controller that the driver is operating.

- `getModelString`, which provides the model name for the hardware controller that the driver is operating.

Enabling and Disabling the Driver

When a network interface is brought up—for example, using the `ifconfig` command—the driver's `enable` function is invoked. This function is responsible for preparing the driver to transmit and receive packets. Here are a few of the operations typically performed in `enable`:

- Opening the driver's provider nub (if the nub was closed in the `start` function).

- Creating any resources needed by the driver for operation, such as hardware-specific transmit and receive buffers, memory cursors for managing scatter/gather lists, and event sources for the work loop.

- Resetting the hardware so that it's ready to transmit and receive packets.

- Starting I/O by enabling hardware interrupts as well as interrupt event sources and setting timer event sources, restarting the output queue (if the driver uses one), and then starting the transmit and receive engines on the hardware.

The `disable` function must reverse this process, disabling what was enabled, shutting down what was started, disposing of resources that were created, and, if necessary, closing the provider nub. The driver should also reset the hardware when disabling, to leave it in a known state.

You invoke the `enable` and `disable` functions within a synchronized context through the driver's superclass, using an IOCommandGate on the driver's work loop. These functions are intended to be overridden, and not directly invoked.

Opening and Closing the Provider Nub

Both `enable` and `disable` are invoked with the provider nub the driver was originally started with. The `enable` function is responsible for invoking `open` on the nub, returning a failure result if it can't open the nub. Similarly, `disable` must invoke `close` on the nub. Because network driver code can be running in multiple threads, however, the driver should implement a mutex. The lock should ensure that the `enable` function is allowed to run only once at a time, and that the `disable` function can run only after the `enable` function. The sample driver uses a Boolean variable, `fNetifEnabled`, as the lock.

Creating and Destroying Resources

In order to preserve kernel resources, a driver should never consume more system memory than necessary. This means that the driver should delay creating resources until they're actually needed, as when the driver becomes active, and should dispose of those resources when it becomes inactive. Such resources typically include any hardware-specific transmit and receive buffers, the BSD `mbuf_t` structures used to pass packets up and down, memory cursor objects used to manage scatter/gather lists based on the `mbuf_t` structures, and any event sources needed for operation.

Hardware-specific resources are necessarily outside the scope of this document. Whatever the hardware-specific resources are, however, they will typically include a receive buffer that contains `mbuf_t` structures. Your driver can use its network interface object to allocate and free these structures through the `allocatePacket` and `freePacket` functions defined by IONetworkController.

Memory cursors, represented by the IOMBufMemoryCursor group of classes, manage the translation between `mbuf_t` structures and scatter/gather lists used by the hardware. Specific classes are available to handle data in big-endian, little-endian, or CPU-native byte order, and to handle data used with DBDMA engines. Your driver will typically create one or more memory cursors, depending on whether it's single- or multithreaded and whether the transmit and receive buffers used by hardware are similar or different.

Starting and Stopping I/O

To start I/O, the `enable` function should enable the interrupt and timer event sources and then enable any hardware-specific interrupts. Following this, it should start up the output queue (if it has one) using IOOutputQueue's `setCapacity` and `start` functions. Finally, it should start any I/O engines on the hardware.

The `disable` function stops I/O by roughly reversing this process. It should disable hardware-specific interrupts, disable the interrupt event source and cancel any pending timeout. It should then stop the I/O engines and reset the hardware if necessary. Then, it must stop and flush the output queue by invoking its `stop` function, setting its capacity to zero with `setCapacity`, and invoking `flush`.

Performing I/O

Everything that has been covered up to now is essentially support for the real purpose of a network driver: to send and receive packets through the network controller. The network family specifies the means for a driver to get an output request from the network interface object, and to hand received packets up to it, as well as defining data structures for gathering statistics. Additional functions defined by the network interface and controller superclasses provide support for managing packet buffers, and the `mbuf_t` memory cursor classes aid in the use of scatter/gather lists based on those buffers.

Transmitting Packets

A network driver's entry point for transmission is the `outputPacket` function. This function is invoked either by an IOOutputQueue or by the network interface directly. Depending on the type of queue used (see above), this function may or may not be invoked within the protected context of the driver's work loop.

The sole argument to `outputPacket` is an `mbuf_t` for the packet or packets to be transmitted. A driver that uses an output queue object is guaranteed to be passed a single-packet `mbuf_t`. A driver that does its own queuing must be able to process a chain of packets in a single `mbuf_t` pointer. The `mbuf_t` passed in becomes the property of the driver.

To actually output the packet or packets, the driver must check its hardware-specific resources and prepare a buffer. Depending on the state of the resources, the driver may need to return a packet status of stalled (`kIOReturnStall`), as when the hardware transmit buffers are all full, or dropped (`kIOReturnDropped`), as when an error occurs in processing the packet. If the driver returns `kIOReturnDropped`, it should also put the `mbuf_t` back into the network stack's common pool by invoking the superclass's `freePacket` function.

In order to generate a scatter/gather list for the packet, the driver uses an IOMBufMemoryCursor, invoking its `getPhysicalSegmentsWithCoalesce` function to create a list of physical location/length pairs for the memory segments of the `mbuf_t`. Once it has this information the driver can insert it into the hardware buffers and issue the go-ahead to the hardware controller.

After the packet has been transmitted by the hardware, the driver should reclaim its transmit buffer and put the `mbuf_t` back into the network stack's common pool by invoking the superclass's `freePacket` function.

Receiving Packets

Receiving packets is typically done through an interrupt handler, or possibly a timer for a device that requires polling. In either case, reception is always handled within the protected context of the driver's work loop

To process incoming packets, the driver must extract the `mbuf_t` structure for each from its hardware receive buffers and pass it up to the network interface object by invoking that object's `inputPacket` function. For efficient replacement of an `mbuf_t` structure containing a received packet, IONetworkController defines the functions `copyPacket`, `replacePacket`, and `replaceOrCopyPacket`, which performs the most efficient operation based on the size of the received packet. While it is recommended that you use these functions, the example driver does not take advantage of them.

The `inputPacket` function has an optional argument to allow for queueing of multiple packets. If the driver uses input queueing, it must invoke the network interface's `flushInputQueue` function to ensure that the packets find their way up the network stack.

Gathering Network Statistics

The network family defines several structures for recording network statistics. If you want your driver to do this, it must get the addresses of these structures from the network interface object and then update the relevant fields during operation.

The structures are available from the network interface object through its `getNetworkData` function. This function takes the name of the relevant network data structure, for which constants are defined by the network interface classes. The constant `kIONetworkStatsKey`, for example, indicates the generic network statistics structure, which contains fields for number of input and output packets, among others.

The `getNetworkData` function returns an IONetworkData object, from which you can retrieve the data buffer address using its `getBuffer` function. Casting the returned pointer to the appropriate type gives your driver direct access to the network data structure.

Advertising and Changing Filter Modes

To indicate what kinds of packet filtering (addressing) a driver supports, it overrides `getPacketFilters`. This function is invoked with a pointer to a bitfield indicating which addressing modes are supported, such as unicast, broadcast, and multicast. The driver's implementation of this function should set the bits for the modes it supports and return a success code.

In order to support promiscuous and multicast modes, a network driver overrides the `setPromiscuousMode`, `setMulticastMode`, and `setMulticastList` functions defined by IONetworkController. The set-mode functions can be implemented to just set a flag in the driver; actually supporting the modes requires hardware-specific code in the I/O handling functions, of course. `setMulticastList` is invoked with a list of hardware addresses, which the driver should pass down to the hardware.

Advertising and Changing Media

When a network controller driver starts up, it typically examines its hardware for the media supported and currently active, for example 10Base-T and 100Base-T for ethernet. A driver can advertise these media by creating an instance of IONetworkMedium for each one, collecting them in an OSDictionary that it makes available to the network family by invoking the `setMediumDictionary` function defined by IONetworkController.

The driver should also invoke `setCurrentMedium` to establish the current selected medium. Similarly, when the driver notes that the network link has come up or gone down, it should invoke the `setLinkStatus` function to report the status to the network interface object.

Document Revision History

This table describes the changes to *Network Device Driver Programming Guide*.

Date	Notes
2008-03-11	Added note about how to remove the IP protocol from an interface to permit KEXT unloading.
2005-06-22	Major content update for Mac OS X v10.4.
2000-10-01	First version of this document, which describes how to create a network device driver.