8

Network Modules

Loadable kernel servers that serve as network modules use special functions and interfaces in addition to the ones available to all loadable kernel servers. This chapter discusses how to write network modules. The special functions that network modules can use are described in detail in Chapter 10, ªKernel Support Functions,º under the section ªNetwork Functions.º

The NeXT Mach kernel supports the following types of network modules:

- *Network device drivers*. A network device driver sends and receives packets to and from some network media.
- *Protocol handlers*. On input, a protocol handler receives packets from network device drivers and forwards the data to the interested programs. On output, the protocol handler takes data from programs, puts the data into packets, and sends these packets to the appropriate network device driver.
- *Packet sniffers*. A packet sniffer examines input packets for diagnostic purposes.

If you're familiar with UNIX 4.3BSD networking primitives, you'll find many similarities to what's described in this chapter. The biggest difference is that a common programming interface like the socket mechanism isn't defined. While sockets work well for TCP/IP, they don't generalize well to other protocols.

If you're writing a protocol handler and want to open it up to programmers, you must define your own interface for communication between user programs and your protocol handler.

This chapter first gives an overview of NeXT networking support, and then discusses the objects you'll use in your network module. The next section has details on the routines that you should implement. The chapter ends with notes on implementing specific interfaces.

Overview

Here's a simplified view of what happens when a network packet is received by a NeXT computer:

1. The packet is received by the appropriate network device driver, which puts the packet into a data structure called a *netbuf* (netbufs are discussed in the next section).
2. The driver calls the dispatcher (by calling **if_handle_input()**).
3. The dispatcher polls all registered packet sniffers and protocol handlers until it finds a protocol handler that accepts the packet.
4. If the protocol handler is an IP (Internet Protocol) handler, it sends the packet up to the kernel by calling **inet_queue()**.

When a packet is sent out onto the network, the following events happen:

1. The output function of the protocol handler is called. One of the arguments is a netbuf containing the packet to

be sent. (This netbuf must have been previously allocated by the network device driver; how netbufs are allocated is described later in this chapter.)

2. The protocol handler calls the appropriate network device driver's output function, passing it the netbuf containing the packet.

3. The device driver puts the packet out onto the network.

Note that there's one extra step in the case of the input packet: A dispatcher is called. This happens because a network device driver doesn't know what its associated protocol handler is, but the protocol handler knows which driver to call. The dispatcher doesn't query the modules in any particular order, except that it queries all packet sniffers before querying any protocol handlers.

Network Objects

The NeXT kernel includes two abstractions especially for network modules:

- Network buffers, known as *netbufs*
- Network interfaces, known as *netifs*

Each of these abstractions is discussed in the following subsections. The associated C functions are described in detail in Chapter 10.

Network Buffers (Netbufs)

The NeXT kernel uses netbufs for dealing with network packet buffers. Netbufs are an interface to an abstract sequence of bytes that can be read and written. The sequence has an original starting point and ending point, but these can be changed. An input network packet typically has its starting point advanced as the various headers are pulled off. Similarly, an output packet has its starting point retreated as headers are inserted.

Operating beyond the range of the original starting and ending points isn't currently detected as an error. This means that an outgoing netbuf should be copied into a larger netbuf if the information being added to its top requires more bytes than are available between the current ending point and the original starting point.

Network Interfaces (Netifs)

Netifs are used to handle the installation and usage of network modules. Remember that a network module is one of three things: a network device driver, a protocol handler, or a packet sniffer.

Each network module initializes and installs its netif (thus registering itself) by calling **if_attach()**. A network device driver should immediately register itself by calling **if_attach()** at load time. Protocol handlers and packet sniffers, on the other hand, don't have to register themselves until their services are required. They determine whether to register themselves in a callback function that they supply as an argument to the function **if_registervirtual()**. This callback function is called once for each network device driver; it should call **if_attach()** if the module isn't already registered and it wants to receive input packets from the specified driver.

Functions Implemented in the Network Module

Besides a callback function, your network module needs to supply certain functions so that other modules can call it. When your network module calls **if_attach()**, you must specify the locations of five functions:

- Initialization functionÐDoes any initialization that's required to change the module's state to ªon.º
- Input functionÐReceives packets from lower layers and either consumes them or passes them on to other modules.
- Output functionÐSends packets from higher layers.
- Getbuf functionÐProvides netbufs for higher layers to use in impending sends.
- Control functionÐProvides any necessary operations the above functions don't.

Note: You should specify null to **if_attach()** for any unimplemented function.

These five functions, along with the callback function, are described in more detail in the following subsections.

Callback Function

A *callback function* is required in protocol handlers and packet sniffers, but isn't appropriate in network device drivers. It must have the following syntax:

void *callback_func*(void **private*, netif_t *realif*)

The purpose of the callback function is to determine whether its network module is interested in a particular device driver and, if necessary, to register its module (using **if_attach()**). The callback function is called once for each current and future network device driver, so it can keep information about more than one network device driver.

The callback function is specified in the network module's call to **if_registervirtual()**. The *private* argument is the data that was specified in the call to **if_registervirtual()**. The *realif* argument is a pointer to the network device driver for which this function is being called. The following code is an example of a typical callback function for a protocol handler.

```
static void myhandler_attach(void *private, netif_t rifp)
{
    netif_t ifp;
    const char *name;
    int unit;
    void *ifprivate;

    if (strcmp(if_type(rifp), IFTYPE_ETHERNET) != 0) {
        return;
    }

    ifprivate = (void *)kalloc(sizeof(myhandler_private_t));
    name = MYNAME;
    unit = MYUNIT;
    ifp = if_attach(NULL, myhandler_input, myhandler_output,
        myhandler_getbuf, myhandler_control, name, unit, IFTYPE_IP,
        MYMTU, IFF_BROADCAST, NETIFCLASS_VIRTUAL, ifprivate);

    (myhandler_private_t *)if_private(ifp)->rifp = rifp;

    if_control(rifp, IFCONTROL_GETADDR, MYHANDLER_ADDRP(ifp));

    if (verbose) {
        printf("IP protocol enabled for interface %s%d, type
            \"%s\"\n", name, unit, MYDRIVER_TYPE);
    }
    return;
}

void myhandler_config(void)
{
```

```
    if_registervirtual(myhandler_attach, NULL);
}
```

Initialization Function

An *initialization function* is not required but is often found in network device drivers. It must have the following syntax:

int *init_func*(netif_t *netif*)

The initialization function takes a pointer to its module's netif structure and performs any necessary initialization. For example, a network device driver should perform any steps necessary to have its hardware ready to run. You can determine what the integer return value (if any) should be. The following is an example of an initialization function.

```
int mydriver_init(netif_t netif)
{
    unsigned unit = if_unit(netif);
    register struct mydriver_data_t *is = &mydriver_data[unit];

    if (is->is_flags & HW_RUNNING)
        return;

    is->is_flags |= HW_RUNNING;
    /* Initialize software structures and the hardware. */
    /* ... */
    return;
}
```

Input Function

An *input function* is required in protocol handlers and packet sniffers, but not in network device drivers. It must have the following syntax:

int *input_func*(netif_t *netif*, netif_t *realnetif*, netbuf_t *packet*, void€**extra*)

The input function takes a pointer to its module's netif (*netif*), a pointer to the calling network device driver (*realnetif*), the input packet, and optional extra data. This function should examine the input packet and decide if it wants the packet. If so, this function should return zero and take responsibility for freeing the packet. Otherwise, this function should return EAFNOSUPPORT to allow other modules to receive the packet. Packet sniffers should always return EAFNOSUPPORT.

For example, an IP handler getting packets from an Ethernet device would check if an Ethernet packet's protocol number is the value for IP. If so, the IP handler should handle the packet and return zero.

Since this function might be called at interrupt priority, it should only queue packets. Another thread should pull the packets off of the queue and process them.

The following code is a typical input function of a protocol handler.

```
static int venip_input(netif_t ifp, netif_t rifp, netbuf_t nb,
    void *extra)
{
    short etype;
    short offset;
    short size;
    trailer_data_t trailer_data;

    /* Do we want packets from this driver? */
    if ((myhandler_private_t *)if_private(ifp)->rifp != rifp) {
```

```
        return (EAFNOSUPPORT);
    }

    /*
     * Check fields in the packet to see whether they match
     * the protocol we understand.
     */
    nb_read(nb, MYTYPEOFFSET, sizeof(etype), &etype);
    etype = htons(etype);
    /*
     * Handle ethernet trailer protocol.
     */
    if (etype >= ETHERTYPE_TRAIL &&
        etype < ETHERTYPE_TRAIL + ETHERTYPE_NTRAILER) {
        offset = (etype - ETHERTYPE_TRAIL) * 512;
        if (offset == 0 || (ETHERHDRSIZE + offset +
                    sizeof(trailer_data) >=
                    nb_size(nb))) {
            return (EAFNOSUPPORT);
        }
        nb_read(nb, ETHERHDRSIZE + offset, sizeof(trailer_data),
            &trailer_data);
        etype = htons(trailer_data.etype);
        if (etype != ETHERTYPE_IP &&
            etype != ETHERTYPE_ARP) {
            return (EAFNOSUPPORT);
        }
        size = htons(trailer_data.length);
        if (ETHERHDRSIZE + offset + size > nb_size(nb)) {
            return (EAFNOSUPPORT);
        }
        /*
         * trailer_fix() is a private function that converts trailer
         * packet to regular ethernet packet.
         */
        trailer_fix(nb, offset, size - sizeof(trailer_data));
    }
    switch (etype) {
    case ETHERTYPE_IP:
        nb_shrink_top(nb, ETHERHDRSIZE);
        if_ipackets_set(ifp, if_ipackets(ifp) + 1);
        inet_queue(ifp, nb);
        break;
        /* Put other cases here as necessary. */
        default:
        /*
         * Do not free buf:  let others handle it
         */
        return (EAFNOSUPPORT);
    }
    return (0);
}
```

Output Function

All network modules except packet sniffers must have an *output function*. The syntax of this function must be the following:

int *output_func*(netif_t *netif*, netbuf_t *packet*, void **address*)

The output function takes a pointer to the module's netif, a pointer to a packet, and an address. How this function works depends on whether it's part of a protocol handler or of a network device driver.

If this function is part of a protocol handler, it should assume the packet and address are strictly protocol-level

entities, containing no device-dependent information. The function should add network device information to the packet and call the network device driver's output routine. The netbuf that holds the packet should have been returned by this module's getbuf function, as described later in this chapter.

If this function is part of a network device driver, it should assume the packet and address are device-level entities. The function should simply deliver the packet to the given device-level address. Its return value should be zero if no error occurred; otherwise, return an error number from the header file **sys/errno.h**.

The following example illustrates a typical output function.

```
static int venip_output(netif_t ifp, netbuf_t nb, void *addr)
{
    struct sockaddr *dst = (struct sockaddr *)addr;
    struct ether_header eh;
    struct in_addr idst;
    int off;
    int usetrailers;
    netif_t rifp = VENIP_RIF(ifp);
    int error;

    switch (dst->sa_family) {
    case AF_UNSPEC:
        bcopy(dst->sa_data, &eh, sizeof(eh));
        break;
    case AF_INET:
        idst = ((struct sockaddr_in *)dst)->sin_addr;
        /* ... */
        /*
         * Resolve the en address using arp.  Return 0 if the address
         * wasn't resolved.
         */
         /*
         * XXX:  trailers not supported for output
         */
        eh.ether_type = htons(ETHERTYPE_IP);
        break;
    default:
        nb_free(nb);
        return (EAFNOSUPPORT);
    }
    nb_grow_top(nb, ETHERHDRSIZE);
    nb_write(nb, ETYPEOFFSET, sizeof(eh.ether_type),
         (void *)&eh.ether_type);
    error = if_output(rifp, nb, (void *)&eh.ether_dhost);
    if (error == 0) {
        if_opackets_set(ifp, if_opackets(ifp) + 1);
    } else {
        if_oerrors_set(ifp, if_oerrors(ifp) + 1);
    }
    return (error);
}
```

Getbuf Function

A *getbuf function* is required in all modules except packet sniffers. It must have the following syntax:

netbuf_t *getbuf_func*(netif_t *netif*)

This function returns a netbuf to be used for an impending output call. Only network device drivers should allocate these netbufs. Protocol handlers should instead call the appropriate network device driver's getbuf function to do the allocation. After allocation from the network device driver and before returning the result, the protocol handler should leave enough room at the top of the netbuf for its own output function to later insert a header.

A getbuf function doesn't always have to return a buffer. For example, you might want to limit the number of

buffers your module can allocate (say, 200 kilobytes worth) so that it won't use up too much wired-down kernel memory. When a getbuf function fails to return a buffer, it should return null.

In a protocol handler:

```
static netbuf_t venip_getbuf(netif_t ifp)
{
    netif_t  rifp = VENIP_RIF(ifp);
    netbuf_t nb;

    nb = if_getbuf(rifp);
    if (nb == NULL) {
        return(NULL);
    }
    nb_shrink_top(nb, ETHERHDRSIZE);
    return(nb);
}
```

In a driver:

```
static netbuf_t engetbuf(struct ifnet *ifp)
{
    if (numbufs == MAXALLOC)
        return(NULL);
    else {
        numbufs++;
        return(nb_alloc(HDR_SIZE + ETHERMTU));
    }
}
```

Control Function

The *control function* isn't required, but it's useful in all three kinds of network modules. It must have the following syntax:

int *control_func*(netif_t *netif*, const char **command*, void **data*)

The control function performs arbitrary operations; the character string *command* is used to select between these operations. There are five standard operations that you can choose to implement, although you can also define your own. The command strings corresponding to the standard operations are listed in the following table; constants for the strings (such as IFCONTROL_SET_FLAGS for ªsetflagsº) are declared in the header file **net/netif.h** (under the **bsd** directory of **/NextDeveloper/Headers**).

Command	Operation
ªsetflagsº	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**).
ªsetaddrº	Set the address on the interface.
ªgetaddrº	Get the address of the interface.
ªautoaddrº	Automatically set the address of the interface.
ªunix-ioctlº	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 a control function follows.

```
static int
venip_control(netif_t ifp, const char *command, void *data)
{
```

```
    netif_t rifp = VENIP_RIF(ifp);
    unsigned ioctl_command;
    void *ioctl_data;
    int s;
    struct sockaddr_in *sin = (struct sockaddr_in *)data;

    if (strcmp(command, IFCONTROL_AUTOADDR) == 0) {
        /*
         * Automatically set the address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        /* ... */
    } else if (strcmp(command, IFCONTROL_SETADDR) == 0) {
        /*
         * Manually set address
         */
        if (sin->sin_family != AF_INET) {
            return (EAFNOSUPPORT);
        }
        if_flags_set(ifp, if_flags(ifp) | IFF_UP);
        if_init(rifp);
        VENIP_PRIVATE(ifp)->vp_ipaddr = sin->sin_addr;
        /* ... */
    } else {
        /*
         * Let lower layer handle
         */
        return (if_control(rifp, command, data));
    }
    return (0);
}
```

Notes for Specific Interfaces

This section contains notes about implementing Ethernet and TCP/IP interfaces.

Ethernet Interfaces

Network device drivers that implement the 10-megabit-per-second Ethernet protocol should register their type as IFTYPE_ETHERNET (defined in the header file **net/etherdefs.h**). One 10-megabit Ethernet network device driver comes standard with the NeXT operating system. The type of the address passed to the Ethernet driver's output function for output should be a 6-byte character array (which is cast to **void ***).

TCP/IP Interfaces

IP protocol handlers can hand over their input packets to the kernel for processing by calling **inet_queue()**.

IP protocol handlers should specify their type as ªInternet Protocolº when they call **if_attach()**. The NeXT operating system comes with two TCP/IP modulesÐone for delivery over Ethernet, and one for delivery over loopback. The type of address used by€IP€protocol handlers should be **struct sockaddr_in**, which is defined in the header file€**netinet/in.h**.