VxWorks® 5.5

NETWORK PROGRAMMER'S GUIDE

WindRiver®

Corporate Headquarters
Wind River Systems, Inc.
500 Wind River Way
Alameda, CA 94501-1153
U.S.A.

toll free (U.S.): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/749-2010

For additional contact information, please visit the Wind River URL:

http://www.windriver.com

For information on how to contact Customer Support, please visit the following URL:

http://www.windriver.com/support

VxWorks Network Programmer's Guide, 5.5

14 Aug 02
Part #: DOC-14618-ND-00

Contents

1
Overview

1.1 Introduction

This guide describes the standard VxWorks network stack, which is based on the 4.4 BSD TCP/IP release. Chapters two through nine explain how to configure VxWorks to include or to exclude a particular network protocol or utility. They also describe how to set any configuration parameters associated with the supplied protocols. If your application can use the supplied protocols and drivers as shipped, you need read only these chapters.

The remaining chapters and appendices are intended for developers who need to add a new data link layer driver or network service. This information comes from the *VxWorks Network Protocol Toolkit Programmer's Guide*, which is merged with this manual now that the Network Protocol Toolkit is bundled with the standard VxWorks stack.

NOTE: Although this manual describes how to write code that interfaces with the MUX, this manual is not a tutorial on how to write a protocol or data link layer driver.

Protection Domains

The protection domain feature was introduced to VxWorks with Tornado 3.0. Using protection domains, it is possible to create protected groupings of system resources such as tasks, code modules, semaphores, message queues, as well as physical and virtual memory pages. Sharing resources between protection domains is supported and expected, but it is possible only under very controlled circumstances. This control, enforced using the memory management unit, lets you insulate and protect sensitive elements of a VxWorks system from errant code and other misbehaviors.

VxWorks under Tornado 2.0.2 and recent Tornado 2.*n* releases do not support protection domains. However, the network stack implementation that follows the Tornado 2.0.2 and Tornado 3.1 releases is identical. Thus, this manual is appropriate for network programmers working with either Tornado/VxWorks variant.

Developers working with versions of VxWorks that support protection domains should note that direct access to some network stack features is limited to the kernel protection domain, the protection domain that hosts the network stack. Whenever such a limit applies, this manual notes the fact. If you are working with a non-AE version of VxWorks, you can ignore these notes.

1.2 Chapter Overviews

After the stack overview in *2. The VxWorks Network Stack*, chapters three through nine describe individual network components. The remaining chapters focus on adding new drivers and services to the existing network stack.

- *3. Data Link Layer Components*
- *4. Configuring the Network Stack*
- *5. Network Configuration Protocols*
- *6. Dynamic Routing Protocols*
- *7. Sockets under VxWorks*
- *8. Remote Access Applications*
- *9. DNS and SNTP*
- *10. Integrating a New Network Interface Driver*
- *11. Integrating a New Network Service*

The manual also includes an index, the first place to look when using this manual as a reference.

Data Link Layer Components

3. Data Link Layer Components discusses the data link layer, its general configuration needs, and network drivers. These drivers handle the specifics of communicating over networking hardware, such as an Ethernet board, or even the shared-memory backplane. These drivers are the foundation of the network stack. This chapter also discusses the Berkeley Packet Filter.

NOTE: The PPP, SLIP and CSLIP implementations previously described in this chapter are now deprecated for future use and will be removed from the next major release of Tornado. In anticipation of this change, the configuration information for the PPP, SLIP and CSLIP implementations has been moved to *C. PPP, SLIP, and CSLIP*. For more information on the discontinuance of these features, please contact your local Wind River account manager.

If you require a PPP solution, please ask your Wind River account manager about WindNet PPP. WindNet PPP is a reliable and manageable PPP solution built upon an extensible Remote Access Framework.

TCP/IP under VxWorks

4. Configuring the Network Stack introduces the TCP/IP protocol suite. In particular, this chapter focuses on components such as TCP/IP itself, as well as UDP, ARP, IGMP, and the use of IP over unnumbered interfaces.

VxWorks uses the MUX interface to communicate with the data link layer. The purpose of the MUX is to decouple the data link and network layers. This makes it easier to add new network drivers under an existing protocol. It also makes it easier for an alternative protocol to run over the VxWorks data link layer. For more information on the MUX, see *10. Integrating a New Network Interface Driver*.

The discussion of IP, TCP, and UDP presented in *4. Configuring the Network Stack* is a simple overview that prepares you for a discussion of their configuration needs under VxWorks. It is not a primer on these protocols, which are thoroughly described in a variety of published works (see *1.4 Recommended Reading*, p.11).

The discussion of ARP and proxy ARP are somewhat more detailed. ARP provides dynamic mapping from an IP address to the corresponding media address. Using ARP, VxWorks implements a proxy ARP scheme that can make distinct networks appear to be one logical network. This proxy ARP scheme is an alternative to the use of explicit subnets for accessing the shared-memory network. This helps conserve IP addresses. So too does the use of unnumbered interfaces on point-to-point links (see *4.9 Using Unnumbered Interfaces*, p.90).

IGMP supports the correct delivery of multicast packets for TCP/IP. The standard VxWorks stack supports both the host-side and the router-side of the IGMP v2 protocol, as specified in RFC 2236. For more information, see *4.6 IGMP under VxWorks*, p.71.

Network Configuration Protocols

5. Network Configuration Protocols discusses the network configuration protocols:

- DHCP, Dynamic Host Configuration Protocol
- BOOTP, Bootstrap Protocol
- SNMP, Simple Network Management Protocol

VxWorks can use either DHCP or BOOTP to set up and maintain its network configuration information. At boot time, both DHCP and BOOTP can provide the client with IP addresses and related information. However, BOOTP assigns IP addresses permanently. DHCP extends BOOTP to support the leasing of IP addresses. If necessary, the lease need not time out, but if the client will need the IP address for a limited time, you can limit the length of the lease appropriately.

When the lease expires, the IP address is available for use by another client. If the original client still needs an IP address when the lease expires, the client must renegotiate the lease. Because of the need for lease renegotiation, a DHCP client remains active during run-time. This is not the case for the BOOTP client.

Although SNMP can provide network configuration information, it differs significantly from BOOTP and DHCP in that it was not designed for use at boot time. Instead, you use it to set up a network management station (NMS) from which you can remotely configure, monitor, and control network-connected devices called *agents*. Thus, SNMP is a network configuration protocol, but in a very different sense of the term.

Dynamic Routing Protocols

6. Dynamic Routing Protocols discusses RIP, the Routing Information Protocol. RIP maintains routing information within small inter-networks. The RIP server provided with VxWorks is based on the 4.4 BSD **routed** program. The VxWorks RIP server supports three modes of operation: Version 1 RIP, Version 2 RIP with multicasting, and Version 2 RIP with broadcasting. The RIP implementation also supports an interface exclusion list that you can use to exclude RIP from specific interfaces as they are brought on line.

Sockets under VxWorks

7. Sockets under VxWorks discusses the VxWorks implementation of sockets. Using sockets, applications can communicate across a backplane, within a single CPU, across an Ethernet, or across any connected combination of networks. Socket communications can occur between any combination of VxWorks tasks and host system processes. VxWorks supports a standard BSD socket interface to TCP and UDP. Using these standard BSD sockets, you can:

- communicate with other processes
- access the IP multicasting functionality
- review and modify the routing tables

In addition to the standard BSD socket interface, VxWorks also supports zbuf sockets—an alternative set of socket calls based on a data abstraction called the zbuf (the zero-copy buffer). Using zbuf sockets, you share data buffers (or portions of data buffers) between separate software modules on a VxWorks target. Because zbuf sockets share data buffers, you can avoid time-consuming data copies. The zbuf socket interface is WRS-specific, but visible only to applications running on the VxWorks target. The other end of the socket connection can use a standard BSD socket interface.

NOTE: VxWorks also supports a registration mechanism that you can use to add a socket back end for protocols that you have ported to VxWorks. For more information, see *11. Integrating a New Network Service*.

Remote Access Applications

8. Remote Access Applications discusses the applications that provide remote access to network-connected resources. VxWorks supports the following:

- RPC (Remote Procedure Call, for distributed processing)
- RSH (Remote Shell, for remote command execution)
- FTP (File Transfer Protocol, for remote file access)
- NFS (Network File System, for remote file access)
- TFTP (Trivial File Transfer Protocol, for remote file access)
- **rlogin** (for remote login)
- **telnet** (for remote login)

Other Network Applications

9. DNS and SNTP provides information on how to configure and use DNS and SNTP under VxWorks.

DNS is a distributed database that most TCP/IP applications can use to translate host names to IP addresses and back. DNS uses a client/server architecture. The client side is known as the *resolver*. The server side is called the *name server*. VxWorks provides the resolver functionality in **resolvLib**. For detailed information on DNS, see *RFC-1034* and *RFC-1035*.

SNTP is a Simple Network Time Protocol. Using an SNTP client, a target can maintain the accuracy of its internal clock based on time values reported by one or more remote sources. Using an SNTP server, the target can provide time information to other systems.

Integrating New Data Link Layer Drivers with the MUX

10. Integrating a New Network Interface Driver describes how to integrate new network interface drivers with the MUX. The purpose of the MUX is to provide an interface that insulates network services from the particulars of network interface drivers and vice versa.

Currently, the MUX supports two network driver interface styles, the END interface and the Network Protocol Toolkit (NPT) driver interface. ENDs are frame-oriented drivers that exchange frames with the MUX. All drivers now shipped with the standard network stack are ENDs. The NPT style drivers are packet-oriented drivers that exchange packets with the MUX.

- **Loading the Network Devices** The **muxDevLoad( )** routine loads network devices into the system. This function returns a *cookie* that identifies the device and that is used thereafter in all other calls that refer to that device. The system automatically calls **muxDevLoad( )** for each of the interfaces defined in the system device table. For information on how to set up this table, see *Adding an NPT Driver to VxWorks*, p.204 or *Adding an END to VxWorks*, p.189.

- **Starting the Network Devices** After a network device loads successfully, VxWorks uses **muxDevStart( )** to activate it (see *Launching the Driver*, p.205).

Integrating New Network Services with the MUX

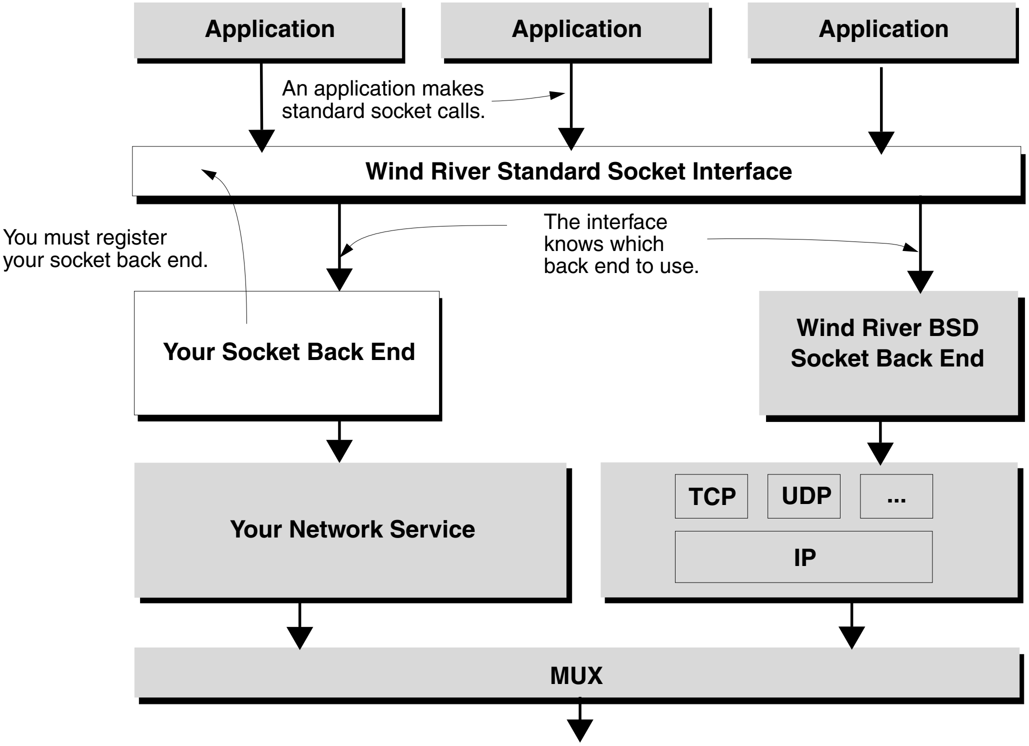
11. Integrating a New Network Service tells you how to integrate a new network service with MUX and thus with VxWorks.

This manual defines a network service as an implementation of the network and transport layers of the OSI network model. Under VxWorks, network services communicate with the data link layer through the MUX interface. Part of porting a new network service to VxWorks is porting its data link layer access code to use the MUX interface.

- **Registering a Socket Back End for New Network Services** To give applications access to your network service, the VxWorks socket interface provides **sockLibAdd( )**. This registration function simplifies adding socket back ends for your network service. The standard socket interface is designed to support the coexistence of distinct socket back ends for each protocol layer implementation.

Thus, adding your socket library does not interfere with the socket back end for any other network service. A layered architecture makes this possible (see Figure 1-1).

Figure 1-1 **The Standard Socket Interface**


Application
Application
Application
An application makes standard socket calls.
Wind River Standard Socket Interface
You must register your socket back end.
The interface knows which back end to use.
Your Socket Back End
Wind River BSD Socket Back End
Your Network Service
TCP
UDP
...
IP
MUX

Because adding a new network service might change your address mapping needs, the VxWorks network stack also includes a registration mechanism for address mapping services.

- **Registering Service Address Mapping Routines** To configure service address mapping (such as address resolution and multicast mapping functions) with the MUX, use functions such as **muxAddrResFuncAdd( )** and **muxMCastAddrAdd( )**. Register service address mapping functions for every pairing of a network service type and a network driver type for which service address mapping functions will be needed. The network service will then retrieve these registered functions from the MUX when they are needed. See *B.2.1 muxAddrResFuncAdd( )*, p.270 and *B.2.13 muxMCastAddrAdd( )*, p.277.

- **Initializing Network Services** Typically, a network service provides a routine that assigns the service to a network interface. Internally, this routine must bind the service to a network driver interface using the **muxTkBind( )** routine. After this step is taken, the network service is ready to send and receive packets over the corresponding device. See *B.2.16 muxTkBind( )*, p.278.

NOTE: You can set the **MUX_MAX_BINDS** configuration parameter to change the maximum number of bind instances that the MUX will allow.

Memory Management and the Network Stack

A. Using netBufLib provides advice on how to use **netBufLib** to create and manage memory pools. Currently, the default VxWorks stack includes two memory pools. Managing these pools is described in *4.3.3 Network Memory Pool Configuration*, p.50. However, if you need to create a totally new memory pool for a new network driver or protocol, you need to read *A. Using netBufLib*.

1.3 UML Notation

Some schematic diagrams in this guide use a form of Unified Modeling Language (UML) notation. This section gives an overview of this UML dialect.

Figure 1-2 shows how a class is represented in UML notation, along with its data members and functional methods. In the VxWorks stack, classes are implemented as data structures.

Figure 1-3 shows class inheritance in UML notation. In the VxWorks stack, class inheritance is implemented by making the first member of the structure that represents the subclass be a structure of the superclass. In UML notation, a subclass points to its superclass with a closed arrowhead at the end of a solid line.

Figure 1-4 shows interface implementation and use in UML notation. Interfaces are represented in much the same way as classes, except that their names are in italics and they always have empty "Attributes" sections. Classes that implement an interface point to that interface with a closed arrowhead at the end of a dashed

Figure 1-2 **A Class in UML Notation**

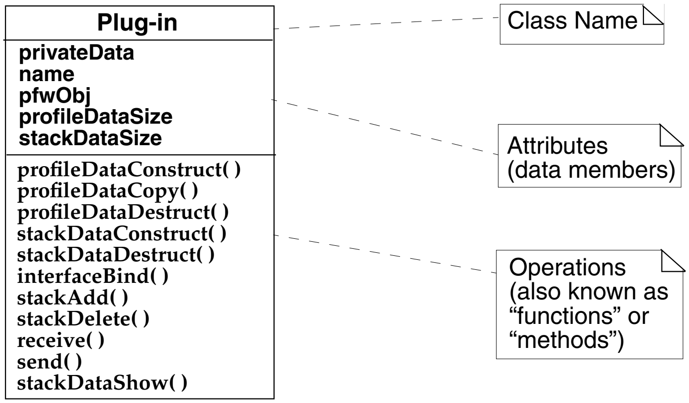
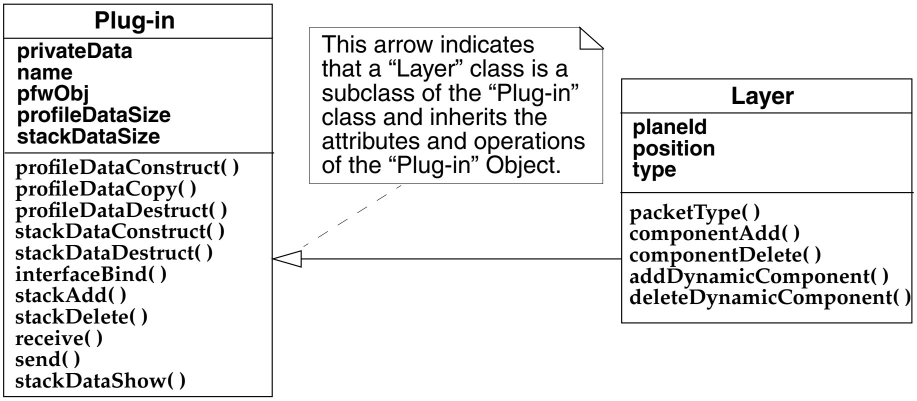


Figure 1-3 **Class Inheritance in UML Notation**

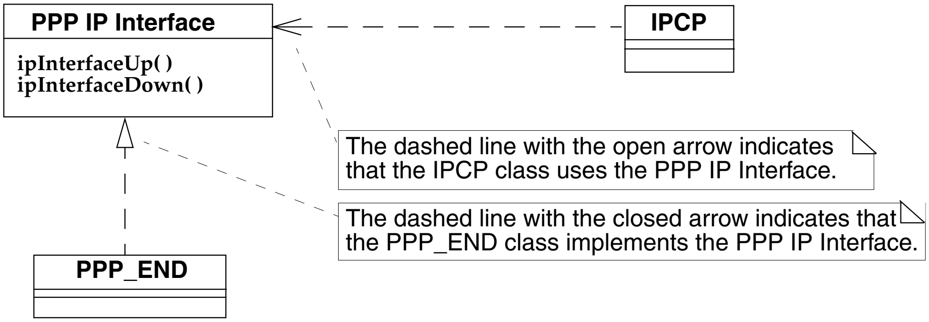


line. Classes that use an interface that another class implements point to that interface with an open arrowhead at the end of a dashed line.

Figure 1-5 shows how class relationships are represented in UML. A solid line between two classes indicates that one of the classes has a reference to one or more objects of the other class (or that the classes mutually reference objects of each other′s class). This relationship can be made more explicit with additional notation.

An open arrowhead indicates that the object of the class doing the pointing knows about the link between the two and should know which object or objects of the

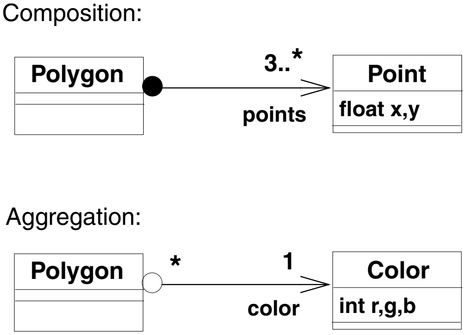
Figure 1-4 **Interface Implementation in UML Notation**



The dashed line with the open arrow indicates that the IPCP class uses the PPP IP Interface.

The dashed line with the closed arrow indicates that the PPP_END class implements the PPP IP Interface.

Figure 1-5 **Aggregates and Composites in UML Notation**



This indicates that a polygon is composed of three or more points, and that a point object used to compose a polygon cannot be used to compose a second polygon. If the polygon object is deleted, the point objects that compose that polygon are also deleted.

This indicates that a polygon has a color object, but a color object may be shared by more than one polygon, and if the polygon object is deleted, the associated color object will not be deleted

other class make up the relationship. Numbers, ranges of numbers (such as **1..5**), or an asterisk (meaning any value) show how many objects are in the relationship.

A circle at the end of the line indicating the relationship can indicate how closely coupled the objects of the two classes are. If object A creates and holds a reference to object B, and then deletes object B as part of its own delete processing, this can be indicated by putting a black circle at the class A side of the line connecting class A and class B. If the two objects are created and deleted independently of one another, a white circle can be used to indicate this.

1.4 Recommended Reading

The focus of this manual is the configuration of the VxWorks network stack. Although this manual includes some networking background information, it is beyond the scope of this manual to provide a thorough description of socket usage, routing, protocol implementation, and how to write a network interface driver. For information of that sort, consider the following sources:

- *The Design and Implementation of the 4.4 BSD Operating System* by Marshall Kirk McKusick, Keith Bostic, Michael J. Kraals, John S. Quarterman
- *TCP/IP Illustrated, Vol. 1*, by Richard Stevens
- *TCP/IP Illustrated, Vol. 2*, by Gary Wright and Richard Stevens
- *Unix Network Programming*, by Richard Stephens
(for information on socket programming)

2
The VxWorks Network Stack

2.1 Introduction

This chapter provides an overview of the components that comprise the VxWorks network stack. Also included in this chapter is a description of scheduling issues associated with the priority of **tNetTask**.

VxWorks includes drivers that support network connections over serial lines (using SLIP or CSLIP) or Ethernet networks (IEEE 802.3). It also supports connections over a backplane bus using shared memory. The VxWorks network stack uses the Internet protocols, based on the 4.4 BSD TCP/IP release, for all network communications.

In addition to the remote access provided by Tornado, VxWorks supports remote command execution, remote login, and remote source-level debugging. VxWorks also supports standard BSD socket calls, remote procedure calls, SNMP, remote file access, boot parameter access from a host, and proxy ARP networks.

2.2 Supported Protocols and Utilities

The VxWorks network stack includes support for the following protocols and utilities:

- SLIP and CSLIP
- IP, Internet Protocol

- TCP and UDP
- DHCP
- BOOTP
- DNS
- IGMP
- ARP and Proxy ARP
- RIP
- Sockets (TCP, UDP, multicasting, routing, and Zbuf)
- RPC
- RSH, rlogin and telnet
- FTP and TFTP
- NFS

MUX Interface

VxWorks supports a network driver interface called the MUX. This interface provides support for features such as multicasting, polled-mode Ethernet, and zero-copy transmission. This interface also decouples the network driver and network protocol layers. This decoupling lets you add new network drivers without the need to alter the network protocol. Likewise, the decoupling lets you add a new network protocol without the need to modify the existing MUX-based network interface drivers.

Earlier versions of the network stack used drivers based on the BSD 4.3 or 4.4 model. Drivers of the BSD model are no longer supported and should be upgraded to the MUX interface model. More information about the process of adding new drivers and protocols to the network stack can be found in the *10. Integrating a New Network Interface Driver* and *11. Integrating a New Network Service*.

Sockets

This network stack implementation includes two sets of socket calls: one set is source-compatible with BSD 4.4 UNIX, the other set is the *zbuf socket interface*. This second set of socket calls is useful when you need to streamline throughput.[1] Both interface styles provide a highly abstracted communication mechanism that hides environmental details. Data written to one socket of a connected pair is read transparently from the other socket.

Because of this transparency, the two tasks do not necessarily know whether they are communicating with an agent on the same host or on another host, or with an agent running under some other host operating system. Similarly, communicating agents using the zbuf socket interface are not aware of whether their

[1] The TCP subset of the zbuf interface is sometimes called "zero-copy TCP."

communications partners are using standard sockets or are also using the zbuf interface.

For information on using sockets, see *7 Sockets under VxWorks*, p.123, and the reference entries for **sockLib** and **zbufSockLib**. For information on adding a socket back end to a new protocol implementation, see *11. Integrating a New Network Service*.

Remote Procedure Calls (RPC)

Remote Procedure Call (RPC) is a protocol that allows a process on one machine to call a procedure that is executed by another process on another machine. Thus with RPC, a VxWorks task or host machine process can invoke routines that are executed on other VxWorks or host machines, in any combination. For more information, see the RPC documentation (publicly and commercially available) and the reference entry for **rpcLib**.

Remote File Access: NFS, RSH, FTP, and TFTP

VxWorks implements the remote file access protocols NFS, RSH, FTP, and TFTP. Using these protocols, a VxWorks target can access the files on a remote network-connected machine as easily as if the files were local to the VxWorks system. Conversely, the VxWorks implementation of all of these protocols (except RSH) lets remote machines access files on a VxWorks target just as transparently – programs running on the host need not know that the files they use are maintained on the VxWorks system.

See the reference entries for **nfsLib**, **nfsdLib**, **remLib**, **ftpLib**, **ftpdLib**, **tftpLib**, and **tftpdLib**, and the following sections: *8.2 RSH, FTP, and netDrv*, p.158, *8.3 NFS and nfsDrv*, p.163, and *8.4 TFTP*, p.169.

Boot Parameter Access from Host

BOOTP is a basic bootstrap protocol. Using BOOTP, a booting target can get its boot parameters from a network-connected host instead of getting the information from local non-volatile RAM or ROM. Included in the information supplied using BOOTP can be items such as the IP address for the target and the name/location of the target’s run-time image. BOOTP cannot supply the image itself. Typically, this file transfer is left to a protocol such as TFTP.

If BOOTP is inadequate to your needs, you can use DHCP instead. DHCP, an extension of BOOTP, is designed to supply clients with all of the Internet configuration parameters defined in the Host Requirements documents (RFCs 1122 and 1123) without manual intervention. Like BOOTP, DHCP allows the permanent allocation of configuration parameters to specific clients. However, DHCP also supports the assignment of a network address for a finite lease period. This feature allows serial reassignment of network addresses. The DHCP implementation provided with VxWorks conforms to the Internet standard *RFC 2131*.

Proxy ARP Networks

Proxy ARP provides transparent network access by using Address Resolution Protocol (ARP) to make distinct networks appear as one logical network. The proxy ARP scheme implemented in VxWorks provides an alternative to the use of explicit subnets for access to the shared memory network.

With proxy ARP, nodes on different physical subnets are assigned addresses with the same subnet number. Because they appear to reside on the same logical network, and because they can communicate directly, they use ARP to resolve each other's hardware address. The gateway node that responds to ARP requests is called the *proxy server.*

2.3 Setting Task Priorities Relative to the Networking Task

The **tNetTask** task provides packet-processing services outside the ISR. Processing network packets outside the ISR minimizes the amount of time spent with interrupts disabled. By default, **tNetTask** runs at a priority of 50. If you launch a task that depends on network services, make sure your new task runs at a lower priority than that of **tNetTask**. When assigning a priority to a task dependent upon network services, keep in mind the following:

- an ISR interrupts even a priority 0 task
- when **tNetTask** is the highest priority task ready to run, it runs
- if a user task (typically priority 100) is ready, it runs instead of **tNetTask**
- while **tNetTask** does not run, packets are not processed, although ISRs continue to receive the packets (by default, up to 85 before dropping packets)

Priority Inversion

After a task takes a semaphore with priority inversion protection, its task priority is elevated if another higher priority task tries to take the semaphore. The new task priority is equal to that of the highest priority task waiting for the semaphore. This priority elevation is temporary. The priority of the elevated task drops back down to its normal level after it releases the semaphore with priority inversion protection.

If a task dependent on **tNetTask** takes a semaphore with priority inversion protection, and if a higher priority task subsequently tries to take the same semaphore, the **tNetTask**-dependent task inherits the higher task priority. Thus, it is possible for a network-dependent task to elevate in priority beyond that of **tNetTask**. This locks **tNetTask** out until after the **tNetTask**-dependent task gives back the problematic semaphore or semaphores.

NOTE: For more information on priority inversion protection and semaphores, see the reference entry for **semMLib**.

3
Data Link Layer Components

3.1 Introduction

The data link layer consists of the drivers that directly transmit and receive frames on the physical network medium. The VxWorks stack bundles Ethernet drivers and the shared-memory backplane network driver, which provides communication over a backplane.

➔ **NOTE:** The PPP, SLIP and CSLIP implementations previously described in this chapter are now deprecated for future use and will be removed from the next major release of Tornado. In anticipation of this change, the configuration information for the PPP, SLIP and CSLIP implementations has been moved to *C. PPP, SLIP, and CSLIP*. For more information on the discontinuance of these features, please contact your local Wind River account manager.

If you require a PPP solution, please ask your Wind River account manager about WindNet PPP. WindNet PPP is a reliable and manageable PPP solution built upon an extensible Remote Access Framework.

3.2 Ethernet Driver Support

Ethernet is one medium among many over which the VxWorks network stack can operate. Ethernet is a local area network specification that is supported by numerous vendors. If you are writing or porting an Ethernet driver to the VxWorks network stack, it should conform to the MUX interface for network drivers. This interface includes support for features such as multicasting and polled-mode Ethernet. For information on how to write a driver that works with the MUX, see *10. Integrating a New Network Interface Driver*.

3.2.1 BPF, the BSD Packet Filter

This network stack implementation supports the BSD Packet Filter (BPF) as a method of inspecting incoming data. One advantage of using BPF rather than an input hook (a technique now deprecated) is that you can use BPF from outside of the kernel protection domain. This is possible because packets caught by the filter can be copied to an area of data that is available outside of the kernel protection domain.

NOTE: The input hook depends on the **etherInputHookAdd( )** functionality supplied by **etherLib**, a library now deprecated for future use. Although this library is still supported, it will be removed from the next major release or Tornado.

To create a BPF device, take the following steps:

1. Call **bpfDrv( )** to initialize the BPF driver – you should check to make sure this routine does not return **ERROR**.

2. Call **bpfDevCreate( )** to create a BPF device.

```
STATUS bpfDevCreate
    (
    char *  pDevName,       /* I/O system device name */
    int     numUnits,       /* number of device units */
    int     bufSize         /* block size for the BPF device */
    )
```

   The device name should be something like **/dev/bpf** or **/bpf/foo**. The number of units should be set to the maximum number of BPF units for the device – for example, if you want to use **/dev/bpf0** and **/dev/bpf1**, **numUnits** should be set to at least **2**. The buffer size should be less than or equal to the MTU, and the buffer you use to read data from the BPF device should be at least as large as this **bufSize**.

3. Call **open( )** to get a file descriptor for one of these BPF units, for example:

```
bpffd = open( "/dev/bpf0", 0, 0 )
```

4. Set any desired options for this unit by using *ioctl* commands – for instance:

   ```
   int arg = 1;
   int status = ioctl (bpffd, BIOCIMMEDIATE, (int)&arg);
   ```

5. Use the **BIOCSETF** *ioctl* command to set a filter for the BPF device. This filter is written according to the **bpf_program** template. Incoming packets that match this filter will be passed to the associated BPF device.

6. Use the **BIOCSETIF** *ioctl* command to attach the file descriptor associated with your BPF device unit to a specific network interface.

3.2.2 Additional Filter Syntax

The **BPF_TYPE** alias finds the type of link level frame. Use it in statements such as:

```
BPF_STMT(BPF_LD+BPF_TYPE,0)  /* Save lltype in accumulator */
```

The **BPF_HLEN** alias determines the header length, independently of the variety of link layer header in use. Use it in statements such as:

```
BPF_STMT(BPF_LD+BPF_H+BPF_ABS+BPF_HLEN, 6) /* IP fragment field */
```

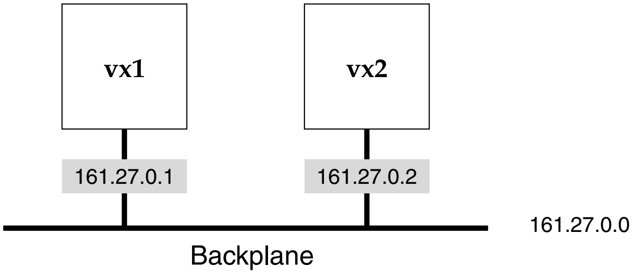
3.3 The Shared-Memory Backplane Network Driver

The **smEnd** (Tornado 3) or **if_sm** (Tornado 2) shared-memory backplane network driver allows multiple processors to communicate over their common backplane as if they were communicating over a network by using a standard 4.4 BSD compatible (**if_sm**) or MUX-capable (**smEnd**) network driver.[1]

A multiprocessor backplane bus is an Internet network with its own network / subnet number. The hosts on this network (the processors) have their own unique IP addresses. In the example shown in Figure 3-1, two CPUs are on a backplane. The Internet address for the shared-memory backplane network is 161.27.0.0. Each CPU on the shared-memory backplane network has a unique Internet address, 161.27.0.1 for **vx1** and 161.27.0.2 for **vx2**.

[1] For more information on the MUX and on END and NPT drivers, see *10. Integrating a New Network Interface Driver*.

Figure 3-1 **Shared-Memory Backplane Network**



Processors can communicate with other hosts on the Internet, or with each other, by using the **smEnd** or **if_sm** driver. The driver behaves as any other network driver, and thus a variety of network services can communicate through it.

3.3.1 The Backplane Shared-Memory Region

The contiguous memory region accessible to all processors on the backplane is the physical medium that lets the **smEnd** or **if_sm** driver simulate driver-style communication.[1]

Backplane Processor Numbers

Each processor on the backplane is assigned a unique *backplane processor number* starting with 0. The assignment of numbers is arbitrary, except for processor 0, which by convention and by default is the shared-memory backplane network master, described in *The Shared-Memory Network Master*, p.23.

You set the processor numbers in the boot-line parameters passed to the boot image. You can burn these parameters into ROM, set them in the processor's NVRAM (if available), or enter them manually.

NOTE: Using the **smEnd** available with Tornado 3, you can set up two shared memory networks on a single backplane with a single processor being a node on each network. However, if you are using the optional VxMP product, you can use only one shared memory network over the backplane. In this case, the processor number of the master node is assigned to and fixed at zero.

[1] The backplane is a type of bus. In this document, the terms are used interchangeably.

The Shared-Memory Network Master

One processor on the backplane is the *shared-memory network master.* The shared-memory network master has the following responsibilities:

- Initialize the shared-memory region and the *shared-memory anchor*
- Maintain the *shared-memory backplane network heartbeat*
- Function (usually) as the gateway to the external network
- Allocate the shared-memory region (if requested)

No processor can use the shared-memory backplane network until the shared-memory network master has initialized it. However, the master processor is *not* involved in the actual transmission of packets on the backplane between other processors. After the shared-memory region is initialized, all of the processors, including the master, are peers.

Under Tornado 3, a configuration parameter determines the processor number of the master. Knowing this, a Tornado 3 node can determine at run time whether it is the master node by comparing the configured processor number with that assigned to it in the boot parameters. Under Tornado 2, the processor number of the master is always zero. A Tornado 2 node can determine at run time whether it is the master by examining the processor number assigned to it in the boot parameters.

Typically, the master has two Internet addresses in the system: its Internet address on the external network, and its address on the shared-memory backplane network. (See the reference entry for **usrConfig**.)

The other processors on the backplane can boot indirectly over the shared-memory backplane network, using the master as the gateway. They need only have an Internet address on the shared-memory backplane network. These processors specify the shared-memory backplane network interface, **esm**, (Tornado 3) or **sm** (Tornado 3), as the boot device in the boot parameters.

The Shared-Memory Anchor

The location of the shared-memory region depends on the system configuration. All processors on the shared-memory backplane network must be able to access the shared-memory region within the same bus address space as the anchor.

The shared-memory anchor serves as a common point of reference for all processors. The anchor structure and the shared memory region may be located in the dual-ported memory of one of the participating boards (the master by default) or in the memory of a separate memory board.

NOTE: Some BSPs support locating the anchor and shared memory regions on a participating non-master board. For examples of how to do this, see the Compact PCI bus BSPs **mcp750** and **mcpn750**.

The anchor contains an offset to the actual shared-memory region. The master sets this value during initialization. The offset is relative to the anchor itself. Thus, the anchor and region must be in the same bus address space so that the offset is linear and valid for all processors.

The anchor address is established by configuration parameters or by boot parameters. For the shared-memory network master, the local anchor address is assigned in the master's configuration at the time the system image is built.

Under Tornado 3, you set the shared memory anchor backplane bus address, *as seen by all boards*, at build time. Two values determine the anchor location. These values are a backplane bus address space specification (configuration parameter: **SM_ADRS_SPACE**) and a bus address within that space (configuration parameter: **SM_ANCHOR_ADRS**). Local, PCI and VME buses are supported.

Under Tornado 2, the shared memory anchor bus address is always a local bus address. You can set this address using the **SM_ANCHOR_OFFSET** and **SM_ANCHOR_ADRS** configuration parameters accessible from the **INCLUDE_SM_COMMON** configuration component. You can also set this address at run time as an aspect of the mapping between a slave's local bus address space and the backplane bus address space. (For more information, see the reference entries for **sysBusToLocalAdrs( )** and **sysLocalToBusAdrs( )**.)

For the slave processors on the shared-memory backplane network, a default anchor bus address can also be assigned during configuration in the same way. However, this requires burning boot ROMs with that configuration, because the other processors must, at first, boot from the shared-memory backplane network. For this reason, the anchor address can also be specified in the boot parameters if the shared-memory backplane network interface is the boot device.

Under Tornado 3, the boot line format is *bootDevName*=*busAdrsSpace*:*busAdrs*. For example:

```
esm=0x0d:0x10010000
```

In this case, this is the backplane bus address of the anchor as seen by all processors. Note that the *busAdrsSpace* section should not specify the local bus if the board is a slave.

Under Tornado 2, the boot line format is *bootDevName*=*localAdrs*. For example:

```
sm=0x10010000
```

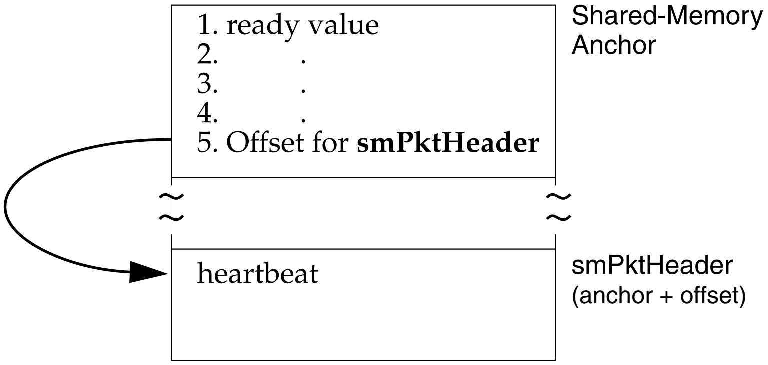
This is the local address of the anchor *as seen by the processor being booted*.

The Shared-Memory Backplane Network Heartbeat

The processors on the shared-memory backplane network cannot communicate over that network until the shared-memory region initialization is finished. To let the other processors know when the backplane is "alive," the master maintains a *backplane network heartbeat*. This heartbeat is a counter that is incremented by the master once per second. Processors on the shared-memory backplane network determine that the shared-memory backplane network is alive by watching the heartbeat for a few seconds.

The shared-memory backplane heartbeat is located in the first 4-byte word of the shared-memory packet header. The offset of the shared-memory packet header is the fifth 4-byte word in the anchor, as shown in Figure 3-2.

Figure 3-2 **Shared-Memory Backplane Network Heartbeat**



Shared-Memory Location

Assigning the shared-memory location differs according to whether you are working under Tornado 2 or Tornado 3.

Tornado 3

Shared memory is assigned a fixed location at compile time. The location is determined by the value of the shared memory address set through a logical region entry in the **00region.sdf** file for that board's BSP. Because all processors on the backplane access the shared-memory region, you must configure that memory as non-cacheable. This can be configured in the **00region.sdf** file.

Tornado 2

Shared memory is, by default, assigned a fixed location and size at compile time. The location is determined by the value of the shared memory address set through the **SM_MEM_ADRS** parameter in the **INCLDUE_SM_COMMON** component for that board. Because all processors on the backplane access the shared-memory region, you must configure that memory as non-cacheable.

The shared memory backplane network region (not including the anchor) can also be allocated at run time if you set **SM_MEM_ADRS** to **NONE**. In this case, a region of size **SM_MEM_SIZE** is allocated and made non-cacheable.

Shared Memory Size

The size of the shared-memory backplane network area is set in the build-time configuration information. The relevant configuration parameter is **SM_NET_MEM_SIZE** (Tornado 3) or **SM_MEM_SIZE** (Tornado 2).

A related area, the shared-memory object area, used by the optional VxMP product, is governed by the configuration parameter **SM_OBJ_MEM_SIZE**. If either the network or VxMP is not installed or used, set the corresponding size parameter to zero. You must do this because the total size of shared memory allocated or assigned is the sum of the network and VxMP sizes.

The size required for the shared-memory backplane network area depends on the number of processors and the expected traffic. There is less than 2KB of overhead for data structures. After that, the shared-memory backplane network area is divided into 2KB packets. Thus, the maximum number of packets available on the backplane network is (*areasize* – 2KB) / 2KB. A reasonable minimum is 64KB. A configuration with a large number of processors on one backplane and many simultaneous connections can require as much as 512KB. If you reserve a backplane network memory area that is too small, you will slow network communication.

Test-and-Set to Shared Memory

To prevent more than one processor from simultaneously accessing certain critical data structures of the shared-memory region, the shared memory backplane network driver uses an indivisible test-and-set (TAS) instruction to obtain exclusive use of a shared-memory data structure. This translates into a *read-modify-write* (RMW) cycle on the backplane bus.[1]

It is important that the selected shared memory supports the RMW cycle on the bus and guarantees the indivisibility of such cycles. This is especially problematic if the memory is dual-ported, as the memory must then lock out one port during a RMW cycle on the other.

Some processors do not support RMW indivisibly in hardware, but do have software hooks to provide the capability. For example, some processor boards have a flag that can be set to prevent the board from releasing the backplane bus, after it is acquired, until that flag is cleared. You can implement these techniques for a processor in the **sysBusTas( )** routine of the system-dependent library **sysLib**. The shared memory backplane network driver calls this routine to set up mutual exclusion on shared-memory data structures.

⚠ **CAUTION:** Configure the shared memory test-and-set type (configuration parameter: **SM_TAS_TYPE**) to either **SM_TAS_SOFT** or **SM_TAS_HARD**. If even one processor on the backplane lacks hardware test and set, all processors in the backplane must use the software test and set (**SM_TAS_SOFT**).

➡ **NOTE:** The shared memory backplane network driver does not support the specification of TAS operation size. This size is architecture dependent.

3.3.2 Interprocessor Interrupts

Each processor on the backplane has a single *input queue* for packets received from other processors. To attend to its input queue, a processor can either poll or rely on interrupts (either bus interrupts or mailbox interrupts). When using polling, the processor examines its input queue at fixed intervals. When using interrupts, the sending processor notifies the receiving processor that its input queue contains packets.

[1] Or a close approximation to it. Some hardware cannot generate RMW cycles on the VME bus and the PCI bus does not support them at all.

Interrupt-driven communication using either bus interrupts or mailbox interrupts is more efficient than polling in that it invests as few cycles in communication as is possible (although at a cost of greater latency). Unfortunately, the bus interrupt mechanism can handle only as many processors as there are interrupt lines available on the backplane (for example, VMEbus has seven). In addition, not all processor boards are capable of generating bus interrupts.

As an alternative to bus interrupts, you can use *mailbox interrupts*, also called *location monitors* because they monitor the access to specific memory locations. A mailbox interrupt specifies a bus address that, when written to or read from, causes a specific interrupt on the processor board. Using hardware jumpers or software registers, you can set each board to use a different address for its mailbox interrupt.

To generate a mailbox interrupt, a processor accesses the specified mail box address and performs a configurable read or write of a configurable size. Because each interrupt requires only a single bus address, there is no meaningful limit on the number of processors that can use mailbox interrupts. Most modern processor boards include some kind of mailbox interrupt.

Each processor must tell the other processors which notification method it uses. Each processor enters its *interrupt type* and up to three related parameters in the shared-memory data structures. The shared-memory backplane network drivers of the other processors use this information when sending packets.

The interrupt type and parameters for each processor are specified during configuration. The relevant configuration parameter is **SM_INT_TYPE** (also **SM_INT_ARG***n*). The possible values are defined in the header file **smLib.h**. Table 3-1 summarizes the available interrupt types and parameters.

3.3.3 Sequential Addressing

Sequential addressing is a method of assigning IP addresses to processors on the backplane network based on their processor number. Addresses are assigned in ascending order, with the master having the lowest address, as shown in Figure 3-3.

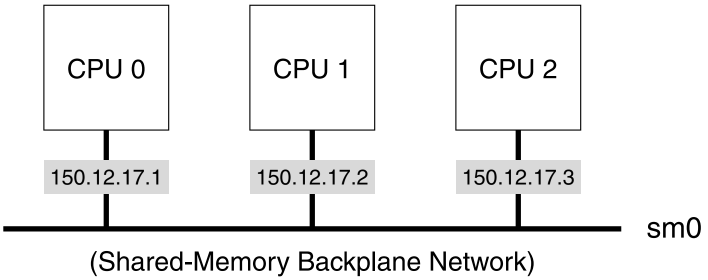
Using sequential addressing, a target on the shared-memory backplane network can determine its own IP address. Only the master's IP address need be entered manually. All other processors on the backplane determine their IP address by adding their processor number to the master's IP address.

Sequential addressing simplifies network configuration. By explicitly assigning an IP address to the master processor, you implicitly assign IP addresses to other processors. This simplifies setting up the boot parameters, in that only the master's

Table 3-1 **Backplane Interrupt Types**

Type	Arg 1	Arg 2	Arg 3	Description
SM_INT_NONE	-	-	-	Polling
SM_INT_BUS	level	vector	-	Bus interrupt
SM_INT_MAILBOX_1	address space	address	value	1-byte write mailbox
SM_INT_MAILBOX_2	address space	address	value	2-byte write mailbox
SM_INT_MAILBOX_4	address space	address	value	4-byte write mailbox
SM_INT_MAILBOX_R1	address space	address	-	1-byte read mailbox
SM_INT_MAILBOX_R2	address space	address	-	2-byte read mailbox
SM_INT_MAILBOX_R4	address space	address	-	4-byte read mailbox
SM_INT_USER_1	user defined	user defined	user defined	first user-defined method
SM_INT_USER_2	user defined	user defined	user defined	second user-defined method

Figure 3-3 **Sequential Addressing**



parameters need to specify the backplane network IP address. The boot parameters of the slave processors need no backplane IP addresses. Thus, when setting up a shared-memory backplane network with sequential addressing, choose a block of IP addresses and assign the lowest address in this block to the master.

When the master initializes the shared-memory backplane network driver, the master passes in its IP address as a parameter. The shared-memory backplane network stores this information in the shared-memory region. If any other address is specified in the **inet on backplane (b)** boot parameter, the specified address overrides the sequential address. To determine the starting IP address for an active shared-memory network, use **smNetShow( )**, if you are working under Tornado 3, or **ifShow("sm")** from the master if you are working under Tornado 2.

Tornado 3 Example

In the following example, the master's IP address is 150.12.17.1.

```
[vxKernel] -> smNetShow
```

The following output displays on the standard output device:

```
Anchor Local Addr: 0x4100, Hard TAS
Sequential addressing enabled.
Master IP address: 150.12.17.1   Local IP address: 150.12.17.2

heartbeat = 56, header at 0xe0025c, free pkts = 57.

cpu int type     arg1       arg2       arg3    queued pkts
--- -------- ---------- ---------- ---------- -----------
 0  mbox-1          0xd 0xfb000000       0x80       0
 1  mbox-1          0xd 0xfb001000       0x80       2

             PACKETS                          ERRORS
    Unicast           Brdcast
 Input    Output   Input    Output        Input    Output
=======  =======  =======  =======    +  =======  =======
     26       27        2        2    |        0        1
value = 0 = 0x0
[vxKernel] ->
```

With sequential addressing, when booting a slave, the backplane IP address and gateway IP boot parameters are not necessary. The default gateway address is the address of the master. Another address can be specified if this is not the desired configuration.

```
[vxWorks Boot]   : p
boot device      : esm=0xD:0x800000
processor number : 1
file name        : /folk/fred/wind/target/config/bspname/vxWorks
host inet (h)    : 150.12.1.159
user (u)         : moose
flags (f)        : 0x0
[vxWorks Boot]   : @

boot device          : esm
unit number          : 0
processor number     : 1
host name            : host
file name            :/folk/fred/wind/target/config/bspname/vxWorks
```

```
inet on backplane (b): 150.12.17.2:ffffff00
host inet (h)        : 150.12.1.159
user (u)             : moose
flags (f)            : 0x0
target name (tn)     : t207-2

Attaching to SM net with memory anchor at 0x10004100...
SM address: 150.12.17.2
Attached TCP/IP interface to esm0.
Gateway inet address: 150.12.17.1
Attaching interface lo0...done
Loading /folk/fred/wind/target/config/bspname/vxWorks/boot.txt

sdm0=/folk/fred/wind/target/config/bspname/vxWorks/vxKernel.sdm
0x000d8ae0 + 0x00018cf0 + 0x00011f70 + (0x0000ccec) + 0x00000078 + 0x0000
015c
```

Sequential addressing can be enabled during configuration. The relevant component is **INCLUDE_SM_SEQ_ADDR**.

Tornado 2 Example

In the following example, the master's IP address is 150.12.17.1.

```
[vxKernel] -> ifShow("sm")
```

The following output displays on the standard output device:

```
sm (unit number 0):
    Flags: (0x8063) UP BROADCAST MULTICAST ARP RUNNING
    Type: ETHERNET_CSMACD
    Internet address: 147.11.207.1
    Broadcast address: 147.11.207.255
    Netmask 0xffff0000 Subnetmask 0xffffff00
    Ethernet address is 00:02:e2:00:00:00
    Metric is 0
    Maximum Transfer Unit size is 2178
    0 packets received; 1 packets sent
    0 multicast packets received
    0 multicast packets sent
    0 input errors; 0 output errors
    0 collisions; 0 dropped
value = 29= 0x1d
[vxKernel] ->
```

With sequential addressing, when booting a slave, the backplane IP address and gateway IP boot parameters are not necessary. The default gateway address is the address of the master. Another address can be specified if this is not the desired configuration.

```
[vxWorks Boot]: p
boot device      : sm=0x800000
processor number : 1
```

```
file name        : /folk/fred/wind/target/config/bspname/vxWorks
host inet (h)    : 150.12.1.159
user (u)         : moose
flags (f)        : 0x0
[vxWorks Boot]: @

boot device          : sm
unit number          : 0
processor number     : 1
host name            : host
file name            :/folk/fred/wind/target/config/bspname/vxWorks
inet on backplane (b): 150.12.17.2:ffffff00
host inet (h)        : 150.12.1.159
user (u)             : moose
flags (f)            : 0x0
target name (tn)     : t207-2

Backplane anchor at 0x10004100... Attaching network interface sm0... done
Attaching interface lo0...done
Loading... 950064
Starting at 0x100000,,,
```

Sequential addressing can be enabled during configuration. The relevant component is **INCLUDE_SM_SEQ_ADDR**.

3.3.4 Shared-Memory Backplane Network Configuration

For UNIX, configuring the host to support a shared-memory backplane network uses the same procedures outlined elsewhere for other types of networks. In particular, a shared-memory backplane network requires that:

- All shared-memory backplane network host names and addresses are present in **/etc/hosts**.
- All shared-memory backplane network host names are present in **.rhosts** in your home directory or in **/etc/hosts.equiv** if you are using RSH.
- A gateway entry specifies the master's Internet address on the external network as the gateway to the shared-memory backplane network. The gateway entry is not needed if you are using proxy ARP. For more information, see *4.8 Proxy ARP for Transparent Subnets*, p.81.

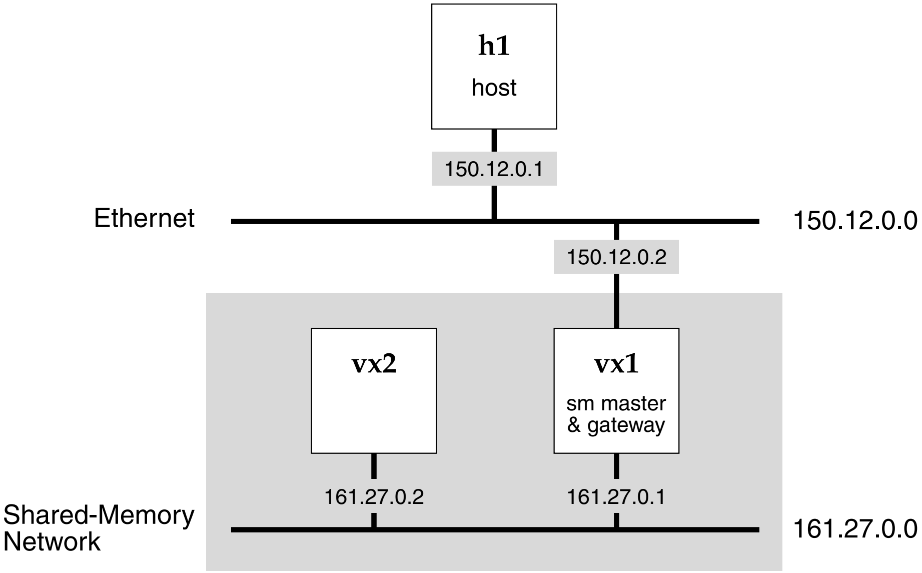
For Windows hosts, the steps required to configure the host are determined by your version of Windows and the networking software you are using. See that documentation for details.

Example Configuration

This section presents an example of a simple shared-memory backplane network. The network contains a single host and two target processors on a single backplane. In addition to the target processors, the backplane includes a separate memory board for the shared-memory region, and an Ethernet controller board. The additional memory board is not essential, but provides a configuration that is easier to describe.

Figure 3-4 illustrates the overall configuration. The Ethernet network is assigned network number 150, subnet 12.0, and the shared-memory backplane network is assigned network number 161, subnet 27.0. The host **h1** is assigned the Internet address 150.12.0.1.

Figure 3-4 **Example Shared-Memory Backplane Network**



The shared memory master is **vx1**, and functions as the gateway between the Ethernet and shared-memory backplane networks. It therefore has two Internet addresses: 150.12.0.2 on the Ethernet network and 161.27.0.1 on the shared-memory backplane network.

The other backplane processor is **vx2**; it is assigned the shared-memory backplane network address 161.27.0.2. It has no address on the Ethernet because it is not directly connected to that network. However, it can communicate with **h1** over the shared-memory backplane network, using **vx1** as a gateway. Of course, all gateway use is handled by the IP layer and is completely transparent to the user. Table 3-2 shows the example address assignments.

Table 3-2 **Network Address Assignments**

Name	Inet on Ethernet	Inet on Backplane
h1	150.12.0.1	-
vx1	150.12.0.2	161.27.0.1
vx2	-	161.27.0.2

To configure the UNIX system for our example, the **/etc/hosts** file must contain the Internet address and name of each system. Note that the backplane master has two entries. The second entry, **vx1.sm**, is not actually necessary. This is because the host system never accesses that system with that address. Still, it is useful to include it in the file because that ensures that the address is not used for some other purpose.

The entries in **/etc/hosts** are as follows:

```
150.12.0.1    h1
150.12.0.2    vx1
161.27.0.1    vx1.sm
161.27.0.2    vx2
```

To allow remote access from the target systems to the UNIX host, the **.rhosts** file in your home directory, or the file **/etc/hosts.equiv**, must contain the names of the target systems:

```
vx1
vx2
```

To inform the UNIX system of the existence of the Ethernet-to-shared-memory backplane network gateway, make sure the following line is in the file **/etc/gateways** *at the time the route daemon* **routed** *is started.*

```
net 161.27.0.0 gateway 150.12.0.2 metric 1 passive
```

Alternatively, you can add the route manually (effective until the next reboot) with the following UNIX command:

```
% route add net 161.27.0.0 150.12.0.2 1
```

To prepare a run-time image for **vx1**, the backplane master shown in Figure 3-4, include the following configuration components:

- **Tornado 3**
  - **INCLUDE_SMEND** — includes the shared memory END (**smEnd**)
  - **INCLUDE_SM_COMMON** — includes configuration parameters common to memory sharing utilities
  - **INCLUDE_SECOND_SMEND_BOOT** — attaches the **smEnd** as a secondary boot device
  - **SM_OFF_BOARD** — included because the memory is physically off-board

  Within the **INCLUDE_SMEND** and **INCLUDE_SM_COMMON** components, you must set the parameters as shown in Table 3-3.

Table 3-3 **Configuration Parameters from INCLUDE_SM_COMMON and INCLUDE_SMEND**

Parameter[1]	**Value**	**Comment**
SM_NET_DEV_NAME	"esm"	The name of the device.
SM_ADRS_SPACE	0xD e.g., **`esm=0xD:0x800000`**	Address space of shared-memory region as seen by **vx1**.
SM_ANCHOR_ADRS	0x800000 e.g., **`esm=0xD:0x800000`**	Address of anchor as seen by **vx1** in **SM_ADRS_SPACE**.
SM_NET_MEM_SIZE	0x80000	Size of the shared-memory network area, in bytes.
SM_OBJ_MEM_SIZE	(set to zero if not used)	Size of the shared-memory object area, in bytes.
SM_INT_TYPE	**SM_INT_MAILBOX_1**	Interrupt targets with 1-byte write mailbox.
SM_INT_ARG1	**VME_AM_SUP_SHORT_IO**	Mailbox in short I/O space.

SM_INT_ARG2	`(0xc000|`
	`(sysProcNum * 2))`
		Mailbox at:
		0xc000 for **vx1**
		0xc002 for **vx2**

SM_INT_ARG3	0	Write 0 value to mailbox.
SM_PKTS_SIZE	**DEFAULT_PKTS_SIZE**	Shared memory packet size.

Table 3-3 **Configuration Parameters from INCLUDE_SM_COMMON and INCLUDE_SMEND** *(Continued)*

Parameter[1]	Value	Comment
SM_CPUS_MAX	**DEFAULT_CPUS_MAX**	Maximum number of CPUs for the shared network.

[1] The **SM_PKTS_SIZE** and **SM_CPUS_MAX** parameters are part the **INCLUDE_SMEND** configuration component. The other parameters are in **INCLUDE_SM_COMMON**.

- **Tornado 2**
  - **INCLUDE_SM_NET** —includes the shared memory driver (**if_sm**)
  - **INCLUDE_SM_COMMON**—includes configuration parameters common to memory sharing utilities
  - **INCLUDE_SECOND_SMNET—**attaches the **if_sm** as a secondary boot device
  - **INCLUDE_SM_NET_SHOW**—includes the **smNetShow( )** routine

Within the **INCLUDE_SM_NET** and **INCLUDE_SM_COMMON** components, you must set the parameters as shown in Table 3-4.

Table 3-4 **Configuration Parameters from INCLUDE_SM_COMMON and INCLUDE_SM_NET**

Parameter[1]	Value	Comment
SM_OFF_BOARD	**TRUE**	shared memory is on separate board
SM_ANCHOR_ADRS	0x800000 e.g., **sm=0x800000**	Address of anchor as seen by local CPU.
SM_MEM_SIZE	0x80000	Size of the shared-memory backplane network area, in bytes.
SM_OBJ_MEM_SIZE	(set to zero if not used)	Size of the shared-memory object area, in bytes.
SM_INT_TYPE	**SM_INT_MAILBOX_1**	Interrupt targets with 1-byte write mailbox.
SM_INT_ARG1	**VME_AM_SUP_SHORT_IO**	Mailbox in short I/O space.

SM_INT_ARG2	`(0xc000|`
	`(sysProcNum * 2))`
		Mailbox at:
		0xc000 for **vx1**
		0xc002 for **vx2**

Table 3-4 **Configuration Parameters from INCLUDE_SM_COMMON and INCLUDE_SM_NET** *(Continued)*

Parameter[1]	Value	Comment
SM_INT_ARG3	0	Write 0 value to mailbox.
SM_PKTS_SIZE	**DEFAULT_PKTS_SIZE**	Shared memory packet size.
SM_CPUS_MAX	**DEFAULT_CPUS_MAX**	Maximum number of CPUs for the shared network.

[1] The **SM_PKTS_SIZE** and **SM_CPUS_MAX** parameters are part the **INCLUDE_SM_NET** configuration component. The other parameters are in **INCLUDE_SM_COMMON**.

When booting the backplane master, **vx1**, specify boot line parameters such as the following:

```
boot device              : gn
processor number         : 0
host name                : h1
file name                : /usr/wind/target/config/bspname/vxWorks
inet on ethernet (e)     : 150.12.0.2
inet on backplane (b)    : 161.27.0.1:ffffff00
host inet (h)            : 150.12.0.1
gateway inet (g)         :
user (u)                 : darger
ftp password (pw) (blank=use rsh) :
flags (f)                : 0
```

NOTE: For more information on boot devices, see the *Tornado User's Guide: Getting Started*. To determine which boot device to use, see the BSP's documentation.

The other target, **vx2**, would use the following boot parameters:[1]

Under Tornado 3:

```
boot device              : esm
processor number         : 1
host name                : h1
file name                : /usr/wind/target/config/bspname/vxWorks
inet on ethernet (e)     :
inet on backplane (b)    : 161.27.0.2
host inet (h)            : 150.12.0.1
gateway inet (g)         : 161.27.0.1
user (u)                 : darger
ftp password (pw) (blank=use rsh)†:
flags (f)                : 0
```

[1] The parameters **inet on backplane (b)** and **gateway inet (g)** are optional with sequential addressing.

Under Tornado 2:

```
boot device              : sm
processor number         : 1
host name                : h1
file name                : /usr/wind/target/config/bspname/vxWorks
inet on ethernet (e)     :
inet on backplane (b)    : 161.27.0.2
host inet (h)            : 150.12.0.1
gateway inet (g)         : 161.27.0.1
user (u)                 : darger
ftp password (pw) (blank=use rsh)†:
flags (f)                : 0
```

Troubleshooting

Getting a shared-memory backplane network configured for the first time can be tricky. If you have trouble, use the following troubleshooting procedures—taking one step at a time.

1. Boot a single processor in the backplane without any additional memory or processor cards.

2. Power off and add the memory board, if you are using one. Power on and boot the system again. Using the VxWorks boot ROM commands for display memory (**d**) and modify memory (**m**), verify that you can access the shared memory at the address you expect, with the size you expect.

3. Reboot the system, filling in the **inet on backplane** parameter. This initializes the shared-memory backplane network. The following message appears during the reboot:

   Under Tornado 3:

   ```
   Backplane anchor at anchor-addrs...Attaching network interface esm...don
   e.
   ```

   Under Tornado 2:

   ```
   Backplane anchor at anchor-addrs...Attaching network interface sm...done
   .
   ```

4. After VxWorks is booted, you can display the state of the shared-memory network with the **smNetShow( )** routine, as follows:

   ```
   -> smNetShow ["interface"] [, 1]
   value = 0 = 0x0
   ```

Under Tornado 3, the interface parameter is **esm** by default. Under Tornado 2, it is **sm0**. Normally, **smNetShow( )** displays cumulative activity statistics to the standard output device; specifying 1 (one) as the second argument resets the totals to zero.

5. Now test the host connection to the shared-memory master by pinging both of its IP addresses from the host. On the host console , type:

   ```
   ping 150.12.0.2
   ```

   This should succeed and produce a message something like:

   ```
   150.12.0.2 is alive
   ```

   Then type:

   ```
   ping 161.27.0.1
   ```

   This should also succeed. If either **ping** fails, the host is not configured properly, or the shared-memory master has incorrect boot parameters.

6. Power off and add the second processor board. Remember that the second processor must *not* be configured as the system controller board. Power on and stop the second processor from booting by typing any key to the boot ROM program. Boot the first processor as you did before.

7. If you have trouble booting the first processor with the second processor plugged in, you have some hardware conflict. Check that only the first processor board is the system controller. Check that there are no conflicts between the memory addresses of the various boards.

8. On the second processor's console, use the **d** and **m** boot ROM commands to verify that you can see the shared memory from the second processor. This is either the memory of the separate memory board (if you are using the off-board configuration) or the dual-ported memory of the first processor (if you are using the on-board configuration).

9. Use the **d** command on the second processor.

   Under Tornado 3, use the **d** command to look for the local address of the shared-memory anchor as mapped to the two-part bus address space and anchor location within that space. You can also look for the shared-memory heartbeat; see *The Shared-Memory Backplane Network Heartbeat*, p.25.

   Under Tornado 2, use the **d** command on the second processor to look for the shared-memory anchor. You can also look for the shared-memory heartbeat; see *The Shared-Memory Backplane Network Heartbeat*, p.25.

10. After you have found the anchor from the second processor, enter the boot parameter for the boot device with the appropriate anchor bus address:

    Under Tornado 3, enter the two-part backplane bus anchor address:

    boot device: esm=0x0d:0x10010000

    Under Tornado 2, enter the local bus anchor address:

    boot device: sm=0x10010000

    Enter the other boot parameters and try booting the second processor.

11. If the second processor does not boot, you can use **smNetShow( )** on the first processor to see if the second processor is correctly attaching to the shared-memory backplane network. If not, then you have probably specified the anchor bus address incorrectly on the second processor or have a mapping error between the local and backplane buses. If the second processor is attached, then the problem is more likely to be with the gateway or with the host system configuration.

12. You can use host system utilities, such as **arp**, **netstat**, **etherfind**, and **ping**, to study the state of the network from the host side; see the *Tornado User's Guide: Getting Started*.

13. If all else fails, call your technical support organization.

3.4 Custom Interfaces

You can write a driver to provide a custom interface to existing or new communication media. If you write the driver to use the MUX/END interface, the VxWorks network stack can use your custom interface as readily as it uses the Ethernet interface.

4
Configuring the Network Stack

4.1 Introduction

This chapter tells you how to include and configure VxWorks components on a VxWorks target. It also tells you how to set any parameter values associated with those components. Configuring VxWorks involves activities such as:

- including VxWorks components in the VxWorks image at build time
- setting the parameter values associated with those components
- configuring the network interfaces
- manually adjusting the contents of the forwarding table (optional)

NOTE: By default, IP forwarding is turned on. This is typical for a router. If you want a host only product, you could turn off IP forwarding. To do this, remove the **IP_DO_FORWARDING** flag from the **IP_FLAGS_DFLT** configuration parameter.

4.2 Summary of Configuration Settings

The following summary lists all compile-time components and parameters, boot-line parameters, and run-time function calls associated with configuring the VxWorks stack. The more familiar you are with these components, parameters, functions, and the functionality associated with each, the easier it will be to configure your VxWorks stack.

Compile-time Configuration Components and Parameters

ARP_MAX_ENTRIES — limit size of ARP table

INCLUDE_TCP — include the TCP protocol

INCLUDE_UDP — include the UDP protocol

INCLUDE_ICMP — include the ICMP protocol

INCLUDE_IGMP — include the IGMP protocol (host side)

INCLUDE_IGMP_ROUTER — include the IGMP protocol (router side)

INCLUDE_PING — include PING client

INCLUDE_SM_NET — include shared memory network support

INCLUDE_PROXY_SERVER — include the proxy ARP server

INCLUDE_PROXY_CLIENT — include the proxy ARP client

INCLUDE_SM_SEQ_ADDR — assign sequential addresses to hosts on the shared memory back plane

INCLUDE_PROXY_DEFAULT_ADDR — use default IP address (**ead** plus 1) for interface to shared memory backplane

SM_OFF_BOARD — identifies whether shared memory is off or on local board

TCP_FLAGS_DFLT — TCP Default Flags

TCP_SND_SIZE_DFLT — TCP Send Buffer Size

TCP_RCV_SIZE_DFLT — TCP Receive Buffer Size

TCP_CON_TIMEO_DFLT — TCP Connection Time out

TCP_REXMT_THLD_DFLT — TCP Retransmission Threshold

TCP_MSS_DFLT — Default TCP Maximum Segment Size

TCP_RND_TRIP_DFLT — Default Round Trip Interval

TCP_IDLE_TIMEO_DFLT — TCP Idle Time-out Value

TCP_MAX_PROBE_DFLT — TCP Probe Limit

UDP_FLAGS_DFLT — UDP Configuration Flags

UDP_SND_SIZE_DFLT — UDP Send Buffer Size

UDP_RCV_SIZE_DFLT — UDP Receive Buffer Size

ICMP_FLAGS_DFLT — ICMP Configuration Flags

IP_FLAGS_DFLT — IP Configuration Flags

IP_MAX_UNITS — maximum number of interfaces attached to IP layer

IP_TTL_DFLT — IP Time-to-live Value

IP_QLEN_DFLT — IP Packet Queue Size

IP_FRAG_TTL_DFLT — IP Time-to-live Value for packet fragments

NUM_64 — Number of 64 byte clusters for network data memory pool

NUM_128 — Number of 128 byte clusters for network data memory pool

NUM_256 — Number of 256 byte clusters for network data memory pool

NUM_512 — Number of 512 byte clusters for network data memory pool

NUM_1024 — Number of 1024 byte clusters for network data memory pool

NUM_2048 — Number of 2048 byte clusters for network data memory pool

NUM_CL_BLKS — Number of **clBlk**s for network data memory pool

NUM_NET_MBLKS — Number of **mBlk**s for network data memory pool

NUM_SYS_64 — Number of 64 byte clusters for network system memory pool

NUM_SYS_128 — Number of 128 byte clusters for network system memory pool

NUM_SYS_256 — Number of 256 byte clusters for network system memory pool

NUM_SYS_512 — Number of 512 byte clusters for network system memory pool

NUM_SYS_CL_BLKS — Number of **clBlk**s for network system memory pool

NUM_SYS_MBLKS — Number of **mBlk**s for network system memory pool

Boot Line Configuration Values

ead — boot line parameter specifying IP address and mask for boot interface

bootDev — boot line parameter specifying name of boot network interface

unitNum — boot line parameter specifying number of boot network interface

gad — address and mask of the intermediate gateway to host supplying boot image

bad — address and mask assigned to interface representing the shared memory back plane. If you are using sequential addressing and proxy default addressing, the target generates a **bad** value from the **ead** parameter and the CPU number.

Configuration Functions Callable at Run-Time

igmpRouterLibInit( ) — initialize IGMPv2 router

igmpRouterLibQuit( ) — shut down IGMPv2 router

igmpInterfaceEnable( ) — enable IGMPv2 on specified interface

igmpInterfaceDisable( ) — disable IGMPv2 on specified interface

igmpLibInit( ) — initialize the IGMPv2 host

ipAttach( ) — attach IP to the MUX

ipDetach( ) — detach IP from the MUX

ifMaskSet( ) — set network mask for an interface

ifAddrSet( ) — set IP address for an interface and associated broadcast address

ifBroadcastSet( ) — set custom broadcast IP address

mRouteAdd( ) — add a static route to the routing table

mRouteDelete( ) — delete a static route from the routing table

mRouteEntryAdd( ) — add a protocol-owned route to the routing table

mRouteEntryDelete( ) — delete a protocol-owned route from the routing table

proxyPortFwdOn( ) — allow forwarding of broadcasts for specified port

proxyPortFwdOff( ) — block forwarding of broadcasts for specified port

smNetShow( ) — find the shared memory anchor

sysBusToLocalAddr( ) — get the correct bus address for the shared memory anchor

4.3 Configuring the Network Stack at Build Time

At compile time, you select the protocols and facilities that are included in the networking stack. You may also specify resources allocated to the networking stack as well as set behavior configuration values for IP, TCP, UDP, and ICMP.

4.3.1 Network Protocol Scalability

The default VxWorks stack includes the code implementing the TCP, UDP, ICMP, and IGMP protocols. If you want to exclude one of these protocols, reconfigure VxWorks using the configuration parameters listed below:

INCLUDE_TCP	Include the TCP protocol.
INCLUDE_UDP	Include the UDP protocol.
INCLUDE_ICMP	Include the ICMP protocol.
INCLUDE_IGMP	Include the IGMP protocol (host side).
INCLUDE_IGMP_ROUTER	Include the IGMP protocol (router side).

About UDP — User Datagram Protocol

UDP provides a simple *datagram*-based end-to-end communication mechanism. UDP extends the message address to include a *port address* in addition to the host Internet address. The port address identifies one of several distinct destinations within a single host. Thus, UDP accepts messages addressed to a particular port on a particular host, and tries to deliver them, using IP to transport the messages between the hosts. Like IP, UDP makes no guarantees that messages are delivered correctly or even delivered at all.

However, this relatively low-overhead delivery mechanism makes UDP useful to many other protocols and utilities, such as BOOTP, DHCP, DNS, RIP, SNMP, and NFS.

About TCP — Transmission Control Protocol

TCP provides reliable, flow-controlled, two-way, process-to-process transmission of data. TCP is a *connection*-based communication mechanism. This means that before data can be exchanged over TCP, the two communicating processes must first establish a connection through a distinct connection phase. Data is then sent and received as a byte stream at both ends.

Like UDP, TCP extends the connection address to include a port address in addition to the host Internet address. That is, a connection is established between a particular port in one host and a particular port in another host. TCP *guarantees* that the delivery of data is correct, in the proper order, and without duplication.

About ICMP — Internet Control Message Protocol

ICMP provides information on the success of data transfer. This protocol defines a set of messages that the TCP/IP stack uses to detect a variety of transmission failure types as well as time information. How the TCP/IP stack processes these messages and how these messages change TCP/IP stack configuration often depends upon higher level protocols, such as TCP and interested users of UDP. Still, ICMP is fundamental to monitoring transmission success.

About IGMP — Internet Group Management Protocol

The TCP/IP stack uses IGMP to support multicasting. The standard VxWorks stack supports both the host-side and the router-side of the IGMP v2 protocol.

IGMP v2 is specified in RFC 2236. According to the RFC, IGMP routers listen on enabled interfaces for membership reports from attached hosts. Using this information, the IGMP routers maintain lists of the multicast addresses to which the IGMP hosts are listening. The IGMP router keeps separate lists for each interface. To discover when it needs to prune entries from a list, the IGMP router periodically transmits queries to the multicast groups on a given interface. If no reply arrives within a specifiable time or if a leave message is processed, the IGMP router removes that group from the list for that interface.

To configure the host-side VxWorks IGMP implementation, you need only include it in your image. It starts up automatically at boot time and requires no system management attention. Configuring the router-side implementation can be as simple, although you have more options. For more information, see *4.6 IGMP under VxWorks*, p.71.

4.3.2 Configuring the ARP, IP, TCP, UDP, IGMP, and ICMP Protocols

This section describes the configuration for the network layer protocols. Table 4-1 describes all configuration options. For some options, the default value is specified using symbolic constants. These configuration components are defined in **netLib.h**. To override any default values assigned to these constants, reconfigure VxWorks with the appropriate values set.

Table 4-1 **Network Configuration Options**

Configuration Component	Default Value and Description
ARP Cache Size	Default Value: 20
ARP_MAX_ENTRIES	Limits the number of entries in the ARP cache. Each entry requires two 64-byte clusters and one 256-byte cluster.

Table 4-1 **Network Configuration Options** *(Continued)*

Configuration Component	Default Value and Description
TCP Default Flags **TCP_FLAGS_DFLT**	Default Value: **TCP_DO_RFC1323** Includes RFC 1323 support. RFC 1323 is a specification to support networks that have high bandwidth and longer round trip times. This option is enabled by default. If the peer cannot negotiate this option, it should drop the option. If the host does not understand this option, it closes the connection. For such hosts, you must turn off this option. You can also use this option in conjunction with **IP_FLAGS_DFLT** to turn off software checksum computation. See *IP_FLAGS_DFLT*, p. 48.
TCP Send Buffer Size **TCP_SND_SIZE_DFLT**	Default Value: 8192 Sets the default send buffer size of a TCP connection.
TCP Receive Buffer Size **TCP_RCV_SIZE_DFLT**	Default Value: 8192 Sets the default receive buffer size of a TCP connection.
TCP Connection Timeout **TCP_CON_TIMEO_DFLT**	Default Value: 150 (75 seconds) Sets the timeout on establishing a TCP connection.
TCP Retransmission Threshold **TCP_REXMT_THLD_DFLT**	Default Value: 3 Sets the number of duplicate ACKs needed to trigger the fast retransmit algorithm. Typically, TCP receives a duplicate ACK only if a segment is lost.
Default TCP Maximum Segment Size (**TCP_MSS_DFLT**)	Default Value: 512 Sets the default maximum segment size to use if TCP cannot establish the maximum segment size of a connection.
Default Round Trip Interval **TCP_RND_TRIP_DFLT**	Default Value: 3 (seconds) Sets the round-trip time to use if TCP cannot get an estimate within 3 seconds. The round trip time of a connection is calculated dynamically.
TCP Idle Timeout Value **TCP_IDLE_TIMEO_DFLT**	Default Value: 14400 (4 hours, in seconds) Sets the idle time for a connection. Idle times greater than this value trigger a keep-alive probe. After the first keep-alive probe, a probe is sent every 75 seconds for a number of times restricted by the TCP Probe Limit.

Table 4-1 **Network Configuration Options** *(Continued)*

Configuration Component	Default Value and Description
TCP Probe Limit **TCP_MAX_PROBE_DFLT**	Default Value: 8 Sets the maximum number of keep-alive probes sent out on an idle TCP connection. If TPC sends out the maximum number of keep-alive probes but receives no response, TCP drops the connection.
UDP Configuration Flags **UDP_FLAGS_DFLT**	Default Value: **UDP_DO_CKSUM_SND | UDP_DO_CKSUM_RCV** Specifies a calculation of data checksum for both send and receive UDP datagrams.
UDP Send Buffer Size **UDP_SND_SIZE_DFLT**	Default Value: 9216 Sets the default send buffer size of a UDP socket.
UDP Receive Buffer Size **UDP_RCV_SIZE_DFLT**	Default Value: 41600 Sets the default receive buffer size of a UDP socket.
ICMP Configuration Flags **ICMP_FLAGS_DFLT**	Default Value: **ICMP_NO_MASK_REPLY** The default value specifies no ICMP mask replies. If this option is enabled on a VxWorks host, and the host receives an ICMP mask query, the VxWorks host replies with its network interface mask.
IP Configuration Flags **IP_FLAGS_DFLT**	Default Value: **IP_DO_FORWARDING | IP_DO_REDIRECT | IP_DO_CHECKSUM_SND | IP_DO_CHECKSUM_RCV** The **IP_DO_FORWARDING** flag enables packet forwarding on systems with multiple external interfaces. This is a typical router configuration. If you want a product that does not forward packets, you must remove this flag from the default value. The **IP_DO_REDIRECT** flag enables ICMP redirects, which are messages sent by a receiving host that was not the optimal recipient of a packet. The redirect message provides the address of a better host. **IP_DO_CHECKSUM_SND** and **IP_DO_CHECKSUM_RCV** enable software checksums for packets sent and received. Both checksums are required for RFC compliance, but, if the network chip handles IP checksum computation and insertion, you do not need it in software. To prevent all software checksums, clear both **IP_DO_CHECKSUM_SND** and **IP_DO_CHECKSUM_RCV**. You must also set **UDP_FLAGS_DFLT** to zero.

Table 4-1 **Network Configuration Options** *(Continued)*

Configuration Component	Default Value and Description
IP Time-to-live Value **IP_TTL_DFLT**	Default Value: 64 Sets the IP default time to live, an upper limit on the number of routers through which a datagram can pass. This value limits the lifetime of a datagram. It is decremented by one by every router that handles the datagram. If a host or router gets a packet whose time to live is zero (this value is stored in a field in the IP header), the datagram is thrown out and the sender is notified with an ICMP message. This behavior prevents packets from wandering in the networks forever.
IP Packet Queue Size **IP_QLEN_DFLT**	Default Value: 50 Sets the default length of the IP queue and the network interface queue. IP packets are added to the IP queue when packets are received. Packets are added to the network interface queue when transmitting.
IP Time-to-live Value for packet fragments **IP_FRAG_TTL_DFLT**	Default Value: 60 (30 seconds for received fragments) Sets the default time to live value for an IP fragment. To transmit a packet bigger than the MTU size, the IP layer breaks the packet into fragments. On the receiving side, IP re-assembles these fragments to form the original packet. Upon receiving a fragment, IP adds it to the IP fragment queue. Each fragment waiting to be re-assembled is removed after the time-to-live expires. If the network is extremely busy, the IP fragment queue can accumulate many fragments. This accumulation of fragments can use up a large amount of system memory. To alleviate this problem, reduce the value of this configuration parameter.

TCP Window Sizes

For TCP sockets, the socket buffer sizes also limit the maximum send and receive window sizes for a connection. While TCP window sizes under BSD 4.4 can theoretically exceed a billion bytes, the maximum socket receive or send buffer size of 258,111 bytes determines the actual upper limit.

The window size for a connection must be set with **setsockopt( )** before the **listen( )** call (on the server) or the **connect( )** call (on the client), because the maximum window sizes are among the parameters negotiated when the connection is established. If you change the buffer size after the connection has been established, this change will not have any effect.

Periodically TCP sends window updates informing the peer how much space is available in the receive buffer. The advertised window size cannot exceed the maximum set when the connection was established, but will decrease to zero if the receive buffer becomes full.

4.3.3 Network Memory Pool Configuration

VxWorks allocates and initializes memory for the network stack during network initialization. This memory is allocated to two pools, a system pool and a data pool. To get status information on these pools, use **netStackSysPoolShow( )** and **netStackDataPoolShow( )**.

WARNING: Failure to configure these two pools correctly is one of the single biggest causes of "frozen" network applications. The default settings for these pools are just enough to get the stack up, running, and able to respond to simple tests, such as **ping**. Deployed applications require more resources. Therefore, study carefully the information in *Determining Memory Pool Usage*, p.55.

The memory in these pools is organized and managed using **mBlk** structures, **clBlk** structures, cluster buffers (simple character arrays), and routines supplied by **netBufLib**. The **mBlk** and **clBlk** structures provide information necessary to support buffer sharing and buffer chaining for the data that is stored in clusters. The clusters come in sizes that are determined by the **CL_DESC** table that describes the memory pool. The **netBufLib** functions provide an interface that you can use to get or return buffers associated with a memory pool.

The **CL_DESC** tables for the network system memory pool and the network data memory pool are defined in **target/config/comps/src/net/usrNetLib.c** as **sysClDescTbl[ ]** and **clDescTbl[ ]** respectively.

Clusters

Valid cluster sizes are powers of two to up to 64KB (65536). Whether a particular cluster size is valid within a particular memory pool depends on the contents of the **CL_DESC** table that describes the pool. The two pools integral to the network stack use the **CL_DESC** tables, **clDescTbl[ ]** and **sysClDescTbl[ ]**.

- **Data pool cluster sizes**

  Data pool cluster sizes are defined in **usrNetLib.c** by **clDescTbl[ ]** as follows:

```
CL_DESC clDescTbl [] =  /* network cluster pool configuration table */
            {
            /*
            clusterSize         num             memArea         memSize
            -----------         ----            -------         -------
            */
            {64,                NUM_64,         NULL,           0},
            {128,               NUM_128,        NULL,           0},
            {256,               NUM_256,        NULL,           0},
            {512,               NUM_512,        NULL,           0},
            {1024,              NUM_1024,       NULL,           0},
            {2048,              NUM_2048,       NULL,           0}
            };
```

- **System pool cluster sizes**

  System pool cluster sizes are defined in **usrNetLib.c** by **sysClDescTbl[ ]** as follows:

```
CL_DESC sysClDescTbl [] =
    {
    /*
    clusterSize         num             memArea         memSize
    -----------         ----            -------         -------
    */
    {64,                NUM_SYS_64,     NULL,           0},
    {128,               NUM_SYS_128,    NULL,           0},
    {256,               NUM_SYS_256,    NULL,           0},
    {512,               NUM_SYS_512,    NULL,           0},
    };
```

A value in the first column specifies a valid cluster size in bytes. A value in the second column, the "num" column, specifies how many clusters of the specified size are allocated for the table. In the tables shown above, all values in the second column are specified using symbolic constants. You can assign the values of these constants using the standard Tornado configuration utility. In particular, look for these values in the configuration component for network buffer initialization.

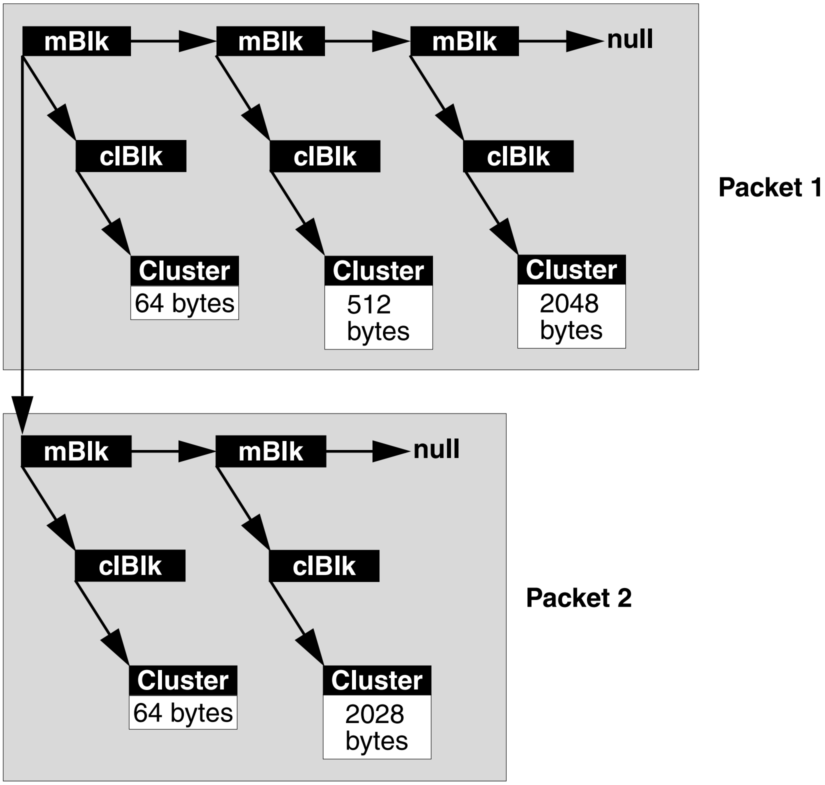
If you do not modify the network stack internals, you should not need to edit (add or delete rows) either of these tables directly. However, you will need to adjust the values assigned to the **NUM_*** constants. See *Setting the Number of Clusters* on p. 53.

mBlks and clBlks

The **clBlk** provides the first level of abstraction above the data in a cluster. For each cluster in a memory pool, there needs to be a corresponding **clBlk** structure. Contained in the **clBlk** is information such as a pointer to the cluster data, the cluster size, and an external reference count.

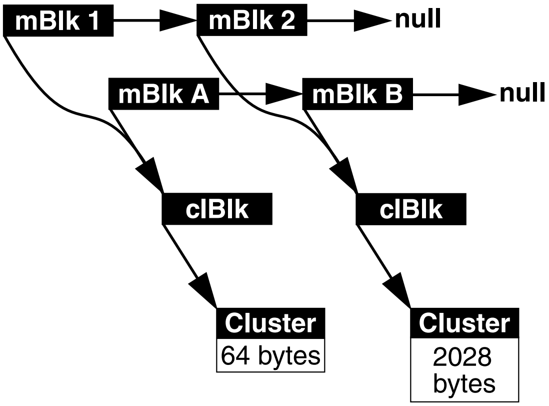
Above the **clBlk**, is the **mBlk** structure. This structure stores a link to a **clBlk** and can store a link to another **mBlk**. By chaining **mBlk**s, you can reference an arbitrarily large amount of data, such as a packet chain (see Figure 4-1).

Figure 4-1 **Presentation of Two Packets in One mBlk Chain**



Because the **mBlk** references the cluster data through a **clBlk**, duplicating an **mBlk** does not copy the cluster data. For example, by duplicating **mBlk 1** in Figure 4-2, you can produce **mBlk A**. However, this duplication did not create a new copy of the underlying cluster. When you use the **netBufLib** routines to duplicate an **mBlk** (or **mBlk** chain), the library automatically increments the external reference counter in the underlying **clBlk**s. This is important when it comes time to free data back to the memory pools.

Figure 4-2 **Different mBlks Can Share the Same Cluster**



If you use **netBufLib** routines to free an **mBlk**, the **mBlk** returns to the pool and the reference count in the underlying **clBlk** is decremented. If this reference count drops to zero (indicating that no **mBlk**s reference the **clBlk**), **netBufLib** frees the **clBlk** and its associated cluster back to the memory pool.

The number of **mBlk**s allocated for the network stack system and data memory pools are determined by the configuration constants **NUM_NET_MBLK** and **NUM_SYS_MBLK**. See Table 4-2 and Table 4-3.

Setting the Number of Clusters

Table 4-2 describes the **NUM_*** constants used to configure the network data memory pool. Table 4-3 describes the **NUM_*** constants used to configure the network system memory pool.

⚠ **WARNING:** Failure to configure these two pools correctly is one of the single biggest causes of "frozen" network applications. The default settings for these pools are just enough to get the stack up, running, and able to respond to simple tests, such as **ping**. Deployed applications require more resources. Therefore, study carefully the information provided in *Determining Memory Pool Usage*, p.55.

Table 4-2 **Configuration Parameters for Network Data Memory Pool**

Parameter	Description
NUM_NET_MBLK Default: 400	Specifies the number of **mBlk** structures to allocate for the network data memory pool. At a minimum, there should be at least as many **mBlk**s as there are clusters.
NUM_64 Default: 100	Specifies the number of 64-byte clusters to allocate for the network data memory pool.
NUM_128 Default: 100	Specifies the number of 128-byte clusters to allocate for the network data memory pool.
NUM_256 Default: 40	Specifies the number of 256-byte clusters to allocate for the network data memory pool.
NUM_512 Default: 40	Specifies the number of 512-byte clusters to allocate for the network data memory pool.
NUM_1024 Default: 25	Specifies the number of 1024-byte clusters to allocate for the network data memory pool.
NUM_2048 Default: 25	Specifies the number of 2048-byte clusters to allocate for the network data memory pool.
NUM_CL_BLKS Default: the sum of **NUM_64** through **NUM_2048**	This value specifies the number of **clBlk** structures to allocate. You need exactly one **clBlk** structure per cluster. If you add another cluster pool to **clDescTbl[ ]**, be sure you increment this value appropriately.

Table 4-3 **Configuration Parameters for Network System Memory Pool**

Parameter	Description
NUM_SYS_MBLK Default value: 2 times **NUM_SYS_CL_BLKS**	Specifies the number of **mBlk** structures to allocate for the network system memory pool. At a minimum, there should be at least as many **mBlk**s as there are clusters.
NUM_SYS_64 Default value: 40	Specifies the number of 64-byte clusters to allocate for the network system memory pool.

Table 4-3 **Configuration Parameters for Network System Memory Pool** *(Continued)*

Parameter	Description
NUM_SYS_128 Default value: 40	Specifies the number of 128-byte clusters to allocate for the network system memory pool.
NUM_SYS_256 Default value: 40	Specifies the number of 256-byte clusters to allocate for the network system memory pool.
NUM_SYS_512 Default value: 20	Specifies the number of 512-byte clusters to allocate for the network system memory pool.
NUM_SYS_CL_BLKS Default: the sum of **NUM_SYS_64** through **NUM_SYS_512**	This value specifies the number of **clBlk** structures to allocate. You need exactly one **clBlk** structure per cluster.

The defaults assigned to these parameters are tuned to let you run Tornado out of the box. However, if you add an application that makes significant use of the network, you will need to adjust these values.

Determining Memory Pool Usage

Estimating the demands on the network stack memory pools requires a detailed understanding of the applications making the demands. Based on that understanding, you should be able to estimate the number of simultaneous open socket connections, the number of routing table and ARP entries needed, and the number of network interfaces needed. Each of these elements creates a predictable demand on the memory pool. For example:

- **TCP socket:** 128 byte cluster, a 256 byte cluster, and a 512 byte cluster; these are used for the generic protocol control block, the TCP protocol control block, and the socket structure respectively. [1]

- **UDP socket:** one 128-byte cluster and one 512-byte cluster; these are used for the generic protocol control block and socket structures.

- **Routing table entries:** four 64-byte clusters and two 256-byte clusters

[1] If your applications open more than 50 sockets, increase the default **NUM_FILES** value (in the I/O system component), which currently defaults to fifty.

- **ARP cache:** two 64-byte clusters and one 256-byte cluster per cache entry. To limit the size of the ARP cache, use **ARP_MAX_ENTRIES**.
- **Network interface instance:** two 64-byte clusters, two 128-byte clusters, and one 256-byte cluster

However, if you are unsure of how your application uses the above resources, you can run the application under the debugger, pause it at critical points, and then compare network stack pool usage at each point. To determine memory pool usage, use the following functions:

netStackSysPoolShow( )
Show network stack system pool statistics.

netStackDataPoolShow( )
Show network stack data pool statistics.

memShow( )
Show blocks and statistics for the current heap partition.

By scaling such single-event values according to your knowledge of how your application uses sockets, the routing table, and so on, you should be able to make some initial estimates on how to set the **NUM_*** parameters (Table 4-2 and Table 4-3). However, it is critical that you test these values before deploying the system. Failure to adequately reserve memory resources for the network stack is one of the biggest reasons for "frozen" network applications. In addition, you must avoid creating a network stack that reserves nearly all available memory to itself and thus locks out all other applications.

4.3.4 Testing Network Connections

You can use the **ping( )** utility from a target to test whether a particular system is accessible over the network. Like the UNIX command of the same name, **ping( )** sends one or more packets to another system and waits for a response. You can identify the other system either by name or by its numeric Internet address. This feature is useful for testing routing tables and host tables, or determining whether another machine is receiving and sending data.

The following example shows **ping( )** output for an unreachable address:

```
-> ping "150.12.0.1",1
no answer from 150.12.0.1
value = -1 = 0xffffffff = _end + 0xfff91c4f
```

If the first argument uses a host name, **ping( )** uses the host table to look it up, as in the following example:

```
    -> ping "caspian",1
    caspian is alive
    value = 0 = 0x0
```

The second argument specifies how many packets to expect back (typically, when an address is reachable, that is also how many packets are sent). If you specify more than one packet, **ping( )** displays more elaborate output, including summary statistics. For example, the following test sends packets to a remote network address until it receives ten acknowledgments. Then the test reports on the time it takes to get replies:

```
-> ping "198.41.0.5",10
PING 198.41.0.5: 56 data bytes
64 bytes from 198.41.0.5: icmp_seq=0. time=176. ms
64 bytes from 198.41.0.5: icmp_seq=1. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=2. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=3. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=4. time=80. ms
64 bytes from 198.41.0.5: icmp_seq=5. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=6. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=7. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=8. time=64. ms
64 bytes from 198.41.0.5: icmp_seq=9. time=64. ms

----198.41.0.5 PING Statistics----
10 packets transmitted, 10 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 64/76/176
value = 0 = 0x0
```

The report format matches the format used by the UNIX **ping** utility. Timings are based on the system clock; its resolution could be too coarse to show any elapsed time when communicating with targets on a local network.

Applications can use **ping( )** periodically to test whether another network node is available. To support this use, the **ping( )** routine returns a **STATUS** value and accepts a **PING_OPT_SILENT** flag as a bit in its third argument to suppress printed output, as in the following code fragment:

```
    /* Check whether other system still there */

    if (ping (partnerName, 1, PING_OPT_SILENT) == ERROR)
        {
        myShutdown();              /* clean up and exit */
        }
    ...
```

You can set one other flag in the third **ping( )** argument: **PING_OPT_DONTROUTE** restricts **ping( )** to hosts that are directly connected, without going through a gateway.

4.3.5 Supporting Multiple Network Interface Drivers

The VxWorks stack lets you use multiple network interface cards simultaneously. You can use multiple cards of the same variety, or different types of cards, with any combination of END and NPT drivers.[1]

Configuring VxWorks for Multiple Drivers

If you want to set up VxWorks to function as a router, you must configure it to support multiple network interfaces. As a first step, make sure that you have compiled all the necessary network drivers into your image. You may also need to increase the value of the configuration parameter **IP_MAX_UNITS**.

If the image includes all the necessary drivers, you are ready to configure the target to load, start, and configure the driver. For deployed systems, you will probably want to set up the image so that it does this automatically at boot time. For information on how to do this, see:

- *Adding an END to VxWorks*, p. 189, to add an END driver
- *Adding an NPT Driver to VxWorks*, p. 204, to add an NPT driver

However, as shown in *Manually Starting Additional Network Interfaces at Run-Time*, p. 67, is also possible to load, start, and configure a driver manually.

4.4 Overview of TCP/IP

IP is the heart of TCP/IP. The goal of the IP protocol is to accept packets addressed to a particular host and then transmit the packets to that host. To accomplish this

[1] Some BSPs and drivers may impose their own limitations on the number of interfaces and units they support.

task, IP works within a *stack* of cooperating protocols. Using a four-layer model, the layers of the VxWorks stack are:

- *Application*—Telnet, FTP, and others
- *Transport*—TCP and UDP
- *Network*—IP
- *Link*—the MUX (a feature specific to VxWorks), its subordinate drivers, and various supporting protocols, such as ARP

Data Plane Functionality

The four-layer model mentioned above describes the *data plane* functionality of a TCP/IP stack. The data plane describes how the TCP/IP stack organizes the flow of data from an application, down through the stack, out onto the network, and then back. As packets flow down the stack, each layer adds a header to the packet and pushes the packet down to the next level. As packets flow up the stack, each layer strips its header, uses the header to determine the recipient of the packet, and pushes the packet up to the appropriate recipient in the next higher layer.

To deliver a packet, IP hands the packet to the link layer after a *routing table* lookup determines that the packet is deliverable. If the routing table cannot match the destination address, IP considers the packet undeliverable and discards it. A matching routing table entry gives IP all the information needed to transmit the packet to a destination on the local network.

If the packet is destined for a remote host, the routing table match provides access information for a local gateway. Gateways (also known as routers) have multiple network interfaces (at least two) that let the gateway transfer data between different networks. When a gateway accepts a packet, its IP layer checks the destination of the packet. If the packet is not destined for the gateway itself, IP does a routing table search on the destination address of the packet.

This lookup duplicates the process that took place on the machine originating the packet. Depending on the results of the routing table search, the packet is dropped, delivered, or forwarded. If the packet is forwarded, the process continues until the packet is delivered, dropped, or expired. A packet expires if it cannot reach its destination within a configurable number of hops (gateway forwardings).

If the packet expires, or IP drops the packet, IP generates an ICMP error message. IP uses its data plane functionality to transmit this message, but the message itself is processed by the TCP/IP control plane functionality.

Control Plane Functionality

The *control plane* functionality of the TCP/IP stack allows the protocols to detect transmission failures and the like.

Maintaining Routing Information

How you set up and manage a routing table for your VxWorks stack depends on your application. For example, you could use RIP to initialize, populate, and manage the routing table. RIP uses IP functionality (through UDP) to announce its presence on the network and discover peers with which to exchange information on network topology.

In addition to a routing protocol, VxWorks includes functionality that you can use to set up and manage a routing table manually. Although you might not use this manual functionality in your deployed application, it is often very useful when you are first developing and debugging your network communications for your application.

4.5 Configuring the IP-to-Link Layer Interface

Configuring the IP-to-Link Layer interface involves the following:

1. Bind IP to the MUX.
2. Set the network mask for an interface.
3. Assign an IP address to each interface.
4. Assign broadcast addresses.

Binding IP to the MUX happens by default for the boot interface. If your target includes only one network interface, you can set the interface network mask and address from the boot line (additional interfaces will require explicit configuration after booting). Assigning the netmask also sets a default broadcast address—the IP address with all the host bits set. However, VxWorks includes functionality that you can use to override this default.

4.5.1 Binding IP to the MUX (Link Layer)

The MUX interface provides data link layer access to registered network protocols. The MUX decouples IP and the data link layer and makes it possible to run different network protocols over the same network hardware. To bind the TCP/IP stack to a particular interface in the MUX, you must call **ipAttach( )**. If you need to remove this binding, call **ipDetach( )**.

For the boot device, the built-in TCP/IP stack initialization code automatically calls **ipAttach( )**. To bind the VxWorks stack to additional interfaces, you must make an explicit **ipAttach( )** call for each additional interface.

For more information on the MUX interface, see *10. Integrating a New Network Interface Driver*.

4.5.2 Assigning an IP Address and Network Mask to an Interface

A *network interface* represents a physical connection to a network through a specific device. To use a specific network device with IP, you must assign a 32-bit Internet (*inet*) address and a 32-bit network mask to the interface. The *network mask* distinguishes the network address portion of the IP address from the host address portion. IP uses this information to determine whether a particular IP address is local (directly reachable). This information is also used to create local entries in the routing table.

When setting the IP address and network mask value for a network interface, you must specify the mask first and then the IP address. To assign a network mask to an interface, call **ifMaskSet( )**. To assign an IP address to an interface, call **ifAddrSet( )**. Because you must assign the mask first, you should call **ifMaskSet( )** before calling **ifAddrSet( )**. This is because **ifAddrSet( )** uses the net mask when assigning the broadcast address. Calling **ifMaskSet( )** does not update these structures.

If your target boot line already includes an **ead** value (an *IPaddress*:*mask* value) for a network interface, you do not need to call **ifMaskSet( )** and **ifAddrSet( )** for that network interface. That is done for you automatically. For any additional interfaces attached to the target, you will need to call **ifMaskSet( )** and **ifAddrSet( )** explicitly.

For detailed information on these functions, see the **ifLib** reference entries.

Interfaces Configured from the Boot Line

If the boot line for the target specifies an **ead** value, your target uses this value as the IP address of the network interface identified in the boot line by the **bootDev** and **unitNum** values.

To specify the network mask for the interface, append a colon and network mask value to the IP address in the **ead** value. For example, to assign an IP address of 147.38.1.2 and a network mask of 0xFFFFFFC0 in the boot line, you would specify an **ead** value of:

```
147.38.1.2:FFFFFFC0
```

The base-16 network mask after the colon does not take a 0x prefix.

Consider the boot line:

```
ln(0, 0) bear:/usr/wpwr/target/config/mz7122/vxworks e=90.10.50.2:FFFFFFC0
b=90.10.50.2 h=100.0.0.4 g=90.10.50.3 u=papa pw=ornery f=0x80 tn=goldilox
s=bear:/usr/papa/startupScript o=
```

This line assigns an IP address of 90.10.50.2 with a network mask of 0xFFFFFFC0 to the **ln0** interface.

Assigning the Net Mask to a Network Interface

IP routing table entries use the network number of an interface to determine which IP addresses are reachable through that interface. If the network number of an IP address matches the network number of a local network interface, IP can reach the destination through that network interface.

Class-Based IP Addresses and Network Mask Values

Before Classless Inter-domain Routing (CIDR), the Internet address space was divided into address classes (see Figure 4-3), each with its own default network mask (see Table 4-4). IP could determine the class of an address by reading the high-order bits of an Internet address as shown in Figure 4-3.

The default class-oriented address masks fell on byte boundaries. Masks with this degree of granularity carved up the Internet address space into rather large chunks. A network with a class A network number could manage 16,777,216 IP addresses, a class B address network number allowed a network of 61,696 IP addresses, and a class C network number could manage 256 IP addresses.

Figure 4-3 **Pre-CIDR Internet Address Classes**

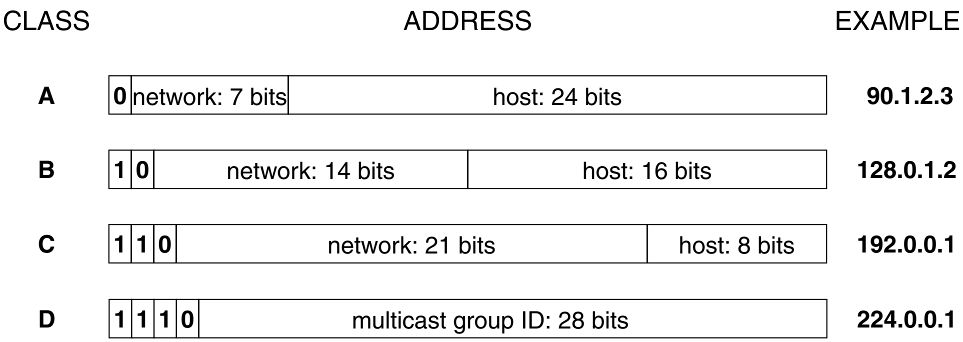


Table 4-4 **Pre-CIDR Internet Address Ranges and Masks**

Class	High Order Bits	Default Address Mask	Address Range
A	0	`0xff000000`	0.0.0.0 – 126.255.255.255
Reserved			127.0.0.0 – 127.255.255.255
B	10	`0xffff0000`	128.0.0.0 – 191.255.255.255
C	110	`0xffffff00`	192.0.0.0 – 223.255.255.255
D	1110	None	224.0.0.0 – 239.255.255.255

NOTE: The number of hosts on an IP network is not quite as large as its IP address space. This is because IP requires each network to reserve at least one address as the broadcast address. If the network is divided into subnets, each subnet must similarly reserve a local IP address for broadcasting to the local subnet. By default, VxWorks reserves at least two addresses for broadcasting. One reserved address follows the current convention for broadcast addresses—all host address bit are set. The other reserved address follows the obsolete "all host address bits are cleared" convention. In addition, the **ifBroadcastSet( )** routine lets you designate still another address as the local broadcast address.

CIDR IP Addresses and Network Masks

Under CIDR, IP routing no longer assumes a network mask based on the Internet address class. Although, if you fail to set a network mask value for a network interface before calling **ifAddrSet( )**, the VxWorks routing table software assumes a default value based on the class system.

In the current CIDR environment, your network is still described by a network number and a mask. However, the mask no longer falls on byte boundaries. Masks can now be assigned by the bit. When choosing a network number, you can select a network mask that has enough bits to include as many IP addresses as you expect to need for your network.

For example, if you need a network of 1000 IP addresses, you would need a netmask of at least 10 bits. Such a mask would give you an address space of 2^{10} addresses (1024 possible IP addresses).

Determining the Network Mask for an Interface

When adding a host or router to an existing network, use the same network mask as all the other interfaces attached to that network. If you did not set up that network, you can get the network mask by asking the network administrator. The network administrator can also assign you an available IP address for the host you are adding.

For a simple example of how you would use masks to divide an address space into subnets, consider a network with the network number 147.38.1.0 and a mask of 24 bits, which you can represent as 255.255.255.0 or 0xFFFFFF00. This gives you an address space of 256 IP addresses (147.38.1.00 through 147.38.1.255).

If you want to divide this space into four equal subnets, you typically need five interfaces, one connecting to the outside world and four connecting to the internal subnets.

To subdivide your address space, you extend the assigned network mask. Each added bit of mask subdivides your address space by powers of two. Thus, if you extend the mask by one bit, you divide your address space into two subnets, two bits subdivides your address space into four subnets, three bits into eight subnets, four bits into 16 subnets, and so on.

If you extend the assigned mask by two bits, the last field of the mask contains a value of:

$$11\ 00\ 00\ 00 = 2^7 + 2^6 + 0 + 0 + 0 + 0 + 0 + 0 = 192 = \mathtt{0xC0}$$

Joining this to the network mask assigned to your overall network gives you the mask to use on each of the four subnets:

```
255.255.255.192 = 0xFFFFFFC0
```

The network number of each subnet is the assigned network number plus all the values expressible within the mask extension:

```
00 00 00 00 = 0    + 0    + 0  + 0  + 0  + 0  + 0  + 0  =   0 =  0x0
01 00 00 00 = 0    + $2^6$ + 0  + 0  + 0  + 0  + 0  + 0  =  64 = 0x10
10 00 00 00 = $2^7$ + 0    + 0  + 0  + 0  + 0  + 0  + 0  = 128 = 0x80
11 00 00 00 = $2^7$ + $2^6$ + 0  + 0  + 0  + 0  + 0  + 0  = 192 = 0xC0
```

This gives the four network numbers shown in Table 4-5.

Table 4-5 **Example Two-Bit Subnets under 147.38.1.00/0xFFFFFF00**

Network Address/Network Mask	IP Address Range (inclusive)		
147.38.1.0/**0xFFFFFFC0**	147.38.1.0	to	147.38.63
147.38.1.64/**0xFFFFFFC0**	147.38.1.64	to	147.38.127
147.38.1.128/**0xFFFFFFC0**	147.38.1.128	to	147.38.191
147.38.1.192/**0xFFFFFFC0**	147.38.1.192	to	147.38.255

NOTE: Although the example divides the network into subnets of equal size, it is possible to create subnets of different sizes. For example, the address/mask values 147.38.1.128/0xFFFFFF00, 147.38.1.0/0xFFFFFFC0, and 147.38.1.64/0xFFFFFFC0 create three subnets. The subnets with the longer 0xFFFFFFC0 masks create subnets for the address ranges 147.38.1.0 to 147.38.63 and 147.38.1.64 to 147.38.127. With these subnets already carved out of the 147.38.1.0 to 147.38.1.255 address space, the 147.38.1.128/0xFFFFFF00 subnet contains the 147.38.128 to 147.38.255 space.

Assigning the Internet Address for a Network Interface

On a UNIX system, you assign an IP address to network interface using the **ifconfig** command. For example, to assign the Internet address 147.38.1.64. to the ln0 network interface, you would enter:

```
% ifconfig ln0 147.38.1.64
```

This is usually handled in the UNIX start-up file **/etc/rc.boot**. For more information, see the UNIX reference entry for **ifconfig**.

Under VxWorks, use **ifAddrSet( )** to assign an IP address to a local interface.

NOTE: Before calling **ifAddrSet( )**, call **ifMaskSet( )** to set the network mask. Otherwise, the network interface assumes the default netmask from the now obsolete class-based system. Such a mask will likely be too large. This is not necessarily a disaster, but the routing table may incorrectly indicate that it has local access to addresses that actually reside on a remote network. For that reason, you will see a lot of ARP errors and transmission failures.

For example, to assign the Internet address 147.38.1.64 to the ln0 interface, enter:

```
    ifAddrSet ("ln0", "147.38.1.64");
```

The **ifAddrSet( )** routine does not let you specify a mask value. Prior to calling **ifAddrSet( )**, you must set the subnet mask for the interface by calling **ifMaskSet( )**.

Fixing Misconfigured Interfaces

A call to **ifAddrSet( )** automatically creates a local entry in the routing table. Local entries in the routing table identify network interfaces on the local host. If you did not assign the correct address to the interface or if you forgot to call **ifMaskSet( )** before calling **ifAddrSet( )**, you need to delete the local host table entry and then reconfigure the interface. The sequence should be:

```
mRouteDelete ( "old IP address for network interface", "old mask value for network" )
ifMaskSet ( "new IP address for network interface", "new mask value for network" )
ifAddrSet ( "new IP address for network interface", "new mask value for network" )
```

The IP address for a network interface can be any available IP address within the IP address space of the network to which the interface connects.

Conventions for Assigning Interface Addresses

A free address is any address not already used by another interface or a service, such as broadcasting. By convention, the broadcast address is the local address with all its host bits set. Thus, within 147.38.1.64/0xFFFFFFC0, the conventional broadcast address is 147.38.1.127:

```
      Mask: 01 00 00 00 = 0  + 2^6 + 0   + 0   + 0   + 0   + 0   + 0   =  64 = 0x10
 Broadcast: 01 11 11 11 = 0  + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 127 = 0x7F
```

An old-style broadcast address would be 147.38.1.64. When VxWorks on a subnet sees a packet addressed to either address style, it treats it as a broadcast packet for the local network. Beyond avoiding the broadcast address, there are no general conventions for assigning IP addresses. However, many organizations have their own conventions. For example, some organizations reserve the network address plus one as the address of the gateway machine.

Manually Starting Additional Network Interfaces at Run-Time

Although you can configure VxWorks at build time to automatically load, start, and configure multiple network interfaces, it is also possible to do it manually at run-time.[1]

NOTE: You may need to increase the value of the configuration parameter **IP_MAX_UNITS**.

To start additional network interfaces manually at run-time:

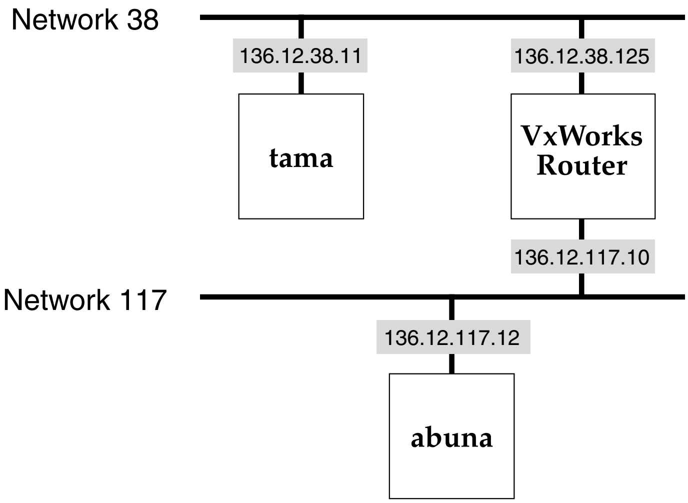
1. Use **ifShow( )** to display information on each currently loaded interface. When adding a new interface, you do not want to conflict with any interface already loaded.
2. Use **muxDevLoad( )** to load the driver for the network interface.
3. Use **muxAddrResFuncAdd( )** to install an address resolution function if necessary. If you loaded an Ethernet driver, you can probably skip this step. Drivers that register as Ethernet drivers automatically use **arpresolve( )** to handle address resolution, so you need not call **muxAddrResFuncAdd( )**.
4. Use **muxDevStart( )** to initialize the network interface.
5. Call the network layer's *foo***Attach( )** routine to attach the driver to the service. For example, to attach to the standard VxWorks stack, call **ipAttach( )**.
6. Configure the interface. If working with the standard VxWorks stack, this means assigning an IP address and a netmask. Use:
   - **ifMaskSet( )** — assign a netmask to the interface
   - **ifAddrSet( )** — assign an IP address to the interface
7. Use **hostAdd( )** — to add the interface to the host table.
8. Check that the interface was loaded and configured correctly:
   - **ifShow( )** — list configuration information for network devices
   - **routeShow( )** — check that the routing table has an entry for the device
   - **hostShow( )** — check that the device was added to the host table

[1] For information on configuring VxWorks to automatically start multiple network interfaces, see *Adding an END to VxWorks*, p.189, and *Adding an NPT Driver to VxWorks*, p.204.

NOTE: For information on the exact inputs expected by the routines named above, see the relevant reference entries.

Consider Figure 4-4. The device acts as a router for tama. When tama pings abuna, the packet goes out 136.12.38.11 to 136.12.38.125, from where it is routed out 136.12.117.10 to 136.12.117.12, the network interface to abuna.

Figure 4-4 **Using a VxWorks Target as a Router**



The routing table on tama, a Solaris box, contains the following entries:

```
Routing Table:
  Destination           Gateway           Flags  Ref   Use   Interface
-------------------- -------------------- ----- ----- ------ ---------
136.12.38.0          136.12.38.11          U        3     78  le0
default              136.12.38.125         UG       0     32
```

To manually add the fei0 interface shown in Figure 4-4, you would access a command shell on the target and enter the following **muxDevLoad( )**, **muxDevStart( )**, **ipAttach( )**, **ifAddrSet( )**, and **hostAdd( )** commands:

```
[vxKernel] -> fei2=muxDevLoad(0,fei82557EndLoad,"-1:0x00:0x20:0x20:0x00",1,0)
fei2 = 0x2e2650: value = 0 = 0x0 (PD NAME: vxKernel)

[vxKernel] -> muxDevStart(fei2)
value = 62 = 0x3e = '>'

[vxKernel] -> ipAttach(0,"fei")
value = 0 = 0x0
```

```
[vxKernel] -> ifMaskSet("fei0",0xffffff00)
value = 0 = 0x0

[vxKernel] -> ifAddrSet("fei0","136.12.117.10")
value = 0 = 0x0

[vxKernel] -> hostAdd("woof-route-10","136.12.117.10")
value = 0 = 0x0

[vxKernel] -> muxShow
Device: elPci Unit: 0
Description: 3COM 3c90X Fast Etherlink Endhanced Network Driver.
Protocol: Wind Debug Agent      Type: 257       Recv 0x1b87d0   Shutdown 0x0
Protocol: IP 4.4 ARP    Type: 2054      Recv 0x14e480   Shutdown 0x14e780
Protocol: IP 4.4 TCP/IP Type: 2048      Recv 0x14e480   Shutdown 0x14e6a0
Device: fei Unit: 0
Description: Intel 82557 Ethernet Enhanced Network Driver
Protocol: IP 4.4 ARP    Type: 2054      Recv 0x14e480   Shutdown 0x14e780
Protocol: IP 4.4 TCP/IP Type: 2048      Recv 0x14e480   Shutdown 0x14e6a0
value = 0 = 0x0

[vxKernel] -> hostShow
hostname         inet address       aliases
--------         ------------       -------
localhost        127.0.0.1
t38-125          136.12.38.125
tama             136.12.38.11
woof-route-10    136.12.117.10
value = 0 = 0x0

[vxKernel] -> ifShow
elPci (unit number 0):
     Flags: (0x8863) UP BROADCAST MULTICAST ARP RUNNING
     Type: ETHERNET_CSMACD
     Internet address: 136.12.38.125
     Broadcast address: 136.12.38.255
     Netmask 0xffff0000 Subnetmask 0xffffff00
     Ethernet address is 00:60:97:d1:5e:ce
     Metric is 0
     Maximum Transfer Unit size is 1500
     0 octets received
     0 octets sent
     1899 packets received
     0 packets sent
     1899 unicast packets received
     0 unicast packets sent
     0 non-unicast packets received
     0 non-unicast packets sent
     0 input discards
     617 input unknown protocols
     0 input errors
     0 output errors
     0 collisions; 0 dropped
lo (unit number 0):
     Flags: (0x8069) UP LOOPBACK MULTICAST ARP RUNNING
     Type: SOFTWARE_LOOPBACK
     Internet address: 127.0.0.1
     Netmask 0xff000000 Subnetmask 0xff000000
     Metric is 0
     Maximum Transfer Unit size is 32768
     0 packets received; 0 packets sent
     0 multicast packets received
     0 multicast packets sent
     0 input errors; 0 output errors
     0 collisions; 0 dropped
fei (unit number 0):
     Flags: (0x8063) UP BROADCAST MULTICAST ARP RUNNING
     Type: ETHERNET_CSMACD
     Internet address: 136.12.117.10
     Broadcast address: 136.12.117.255
     Netmask 0xffff0000 Subnetmask 0xffffff00
     Ethernet address is 00:02:b3:1d:29:a8
     Metric is 0
     Maximum Transfer Unit size is 1500
     0 octets received
     0 octets sent
     0 packets received
     1 packets sent
     0 unicast packets received
     1 unicast packets sent
     0 non-unicast packets received
     0 non-unicast packets sent
     0 input discards
     0 input unknown protocols
     0 input errors
     0 output errors
     0 collisions; 0 dropped
value = 1 = 0x1

[vxKernel] -> mRouteShow
Destination   Mask      TOS  Gateway        Flags RefCnt Use Interface  Proto
127.0.0.1     0         0    127.0.0.1      5     0      0    lo0       0
136.12.38.0   ffffff00  0    136.12.38.125  101   0      0    elPci0    0
136.12.117.0  ffffff00  0    136.12.117.10  101   0      0    fei0      0
value = 0 = 0x0
```

4.5.3 Configuring IP Broadcast Addresses

Many physical networks support the notion of *broadcasting* a packet to all hosts on the network. A special Internet *broadcast address* is interpreted by the network subsystem to mean "all systems" when specified as the destination address of a datagram message (UDP).

Unfortunately, there is ambiguity concerning which address is the broadcast address. The Internet specification now states that the broadcast address is an Internet address with a host part of all ones (1). However, some older systems use an Internet address with a host part of all zeros as the broadcast address.

Most new network stacks, including VxWorks, *accept* either address on incoming packets as being a broadcast packet. However, when an application *sends* a broadcast packet, it must use the correct broadcast address for its system.

VxWorks normally uses a host part of all ones as the broadcast address. Thus a datagram sent to Internet address 150.255.255.255 (**0x96FFFFFF**) is broadcast to all systems on network 150. However, to allow compatibility with other systems, VxWorks allows the broadcast address to be reassigned for each network interface by calling the routine **ifBroadcastSet( )**. For more information, see the reference entry for **ifBroadcastSet( )**.

In addition, VxWorks supports multicasting — transmission to a subset of hosts on the network. For more information on multicasting, see *Using a Datagram (UDP) Socket to Access IP Multicasting*, p. 129.

4.6 IGMP under VxWorks

According to RFC 2236, an IGMPv2 router listens on IGMP-enabled interfaces for membership reports from networked hosts. Using the membership reports, an IGMP router constructs and maintains per-interface lists of the multicast addresses to which IGMP hosts are listening. To discover when it needs to prune entries from a list, an IGMP router periodically transmits queries to the multicast groups accessible through a given interface. If no replies arrive after a specific time, or if a leave message is processed, the IGMP router removes the group from the list for that interface.

To support its IGMPv2 router implementation, VxWorks relies on **tIGMPtask**, an IGMP-dedicated task that waits on **igmpMsgQ**. This message queue collects IGMP timer expiration messages as well as IGMP packet-arrival messages from **tNetTask**.

In response to a timer expiration message, **tIGMPtask** either removes a multicast destination from the interface on which it has expired, or the task sends a query. In response to a packet arrival message, **tIGMPtask** processes the packet. This processing can involve adding an address to an interface (when a new membership report arrives) or deleting an address (when a leave report arrives).

To support the IGMPv2 host implementation, VxWorks does not spawn an independent task. Processing for the IGMPv2 host implementation takes place within the context of **tNetTask**.

4.6.1 Including IGMPv2

The IGMPv2 host is included by default, but IGMPv2 routing is not. To include the IGMPv2 routing:

1. Access the project facility and select the VxWorks tab.
2. Select the **IGMPv2 Routing** component.
3. Select the Project -> Add/Include -> Component(s) menu option.

   Because IGMPv2 routing depends on kernel multicast routing, the project facility will prompt to add **INCLUDE_MCAST_ROUTING** (if it is not already included).
4. Select OK.

The configuration parameters associated with the IGMPv2 host and router are **INCLUDE_IGMP** and **INCLUDE_IGMP_ROUTER**.

4.6.2 IGMPv2 APIs

The IGMPv2 router side API consists of the following functions:

- **igmpRouterLibInit( )**—initialize the IGMPv2 router
- **igmpRouterLibQuit( )**—shut down the IGMPv2 router
- **igmpInterfaceEnable( )**—enable IGMPv2 on the specified interface
- **igmpInterfaceDisable( )**—disable IGMPv2 on the specified interface
- **igmpNameToPort( )**—return a port number (VIF) for the specified interface

For detailed information on the above functions, see the **igmpRouterLib** reference entries.

The IGMPv2 host side API consists of the function:

- **igmpLibInit( )**—initialize the IGMPv2 host

IGMPv2 Host Initialization

If you include the IGMPv2 host in your application, the network initialization code calls **igmpLibInit( )**. You should have no need to call **igmpLibInit( )** explicitly. Because the IGMPv2 host does not launch an independent task or reserve significant system resources, there is no IGMPv2 host termination function.

IGMPv2 Router Initialization and Termination

If you used the project tool to enable IGMPv2 routing, the VxWorks start-up code automatically calls **igmpRouterLibInit( )**. You should have no need to call **igmpRouterLibInit( )** explicitly. If you need to shut down IGMPv2 routing before shutting down the target, you can call **igmpRouterLibQuit( )**.

NOTE: An IGMP router necessarily requires that the host device support at least two network interfaces on which IGMP routing is enabled. Simply including IGMP is not enough. Neither is enabling it on only one interface.

igmpRouterLibInit( )—initialize the IGMP router

Calling **igmpRouterLibInit( )** spawns the IGMP router task and does almost everything necessary to initialize (but not start) the router side of the IGMPv2 implementation. A call to **igmpRouterLibInit( )** returns OK if successful, or **ERROR** otherwise, such as when IGMP has already been started. To actually start IGMP routing, you must enable it on at least two host-local interfaces.

igmpRouterLibQuit( )—shut down the IGMP router

Calling **igmpRouterLibQuit( )** ends IGMP routing by closing the IGMP socket, deleting the router task, and generally cleaning up. An **igmpRouterLibQuit( )** call returns OK if successful, **ERROR** otherwise, such as when IGMP has not been started.

IGMPv2 Router Control

Nothing in VxWorks automatically calls **igmpInterfaceEnable( )**. You must call **igmpInterfaceEnable( )** explicitly for each network interface on which you want to enable IGMPv2 routing. Similarly, nothing in VxWorks automatically calls **igmpInterfaceDisable( )**. If an IGMPv2 routing enabled interface goes down, you must explicitly call **igmpInterfaceDisable( )** for that interface.

igmpInterfaceEnable( )—enable IGMP on the specified interface

Calling **igmpInterfaceEnable( )** enables IGMP on an interface. A call to **igmpInterfaceEnable( )** applies on the router side only. The host side of IGMP does not have a notion of enabled or disabled interfaces. You can call this function for any interface that is capable of receiving multicast packets.

If you use **igmpInterfaceEnable( )** to enable more than one interface on a target, IGMP routing occurs. If there are fewer than two enabled interfaces on a target, there is no possibility of multicast routing. This routine is also responsible for populating the appropriate elements of the IGMP control structure.

An **igmpInterfaceEnable( )** call returns **OK** if successful, **ERROR** otherwise, such as when an interface is not multicast capable.

igmpInterfaceDisable( )—disable IGMP on the specified interface

igmpInterfaceDisable( ) disables IGMP on an interface. If the target supports fewer than two IGMP-enabled interfaces, IGMP on that target stops acting as a router. **igmpInterfaceDisable( )** returns **OK** if successful, **ERROR** otherwise, such as when the interface was not enabled.

Working with VIFs (Ports) and ifnet Structure Pointers

Virtual Interfaces (VIFs), also known as ports, are implemented as indexes into a system-internal array of **ifnet** structures. When working with the VxWorks IGMPv2 implementation, it is often more convenient to work with ports than with pointers to **ifnet** structures.

- **igmpNameToPort( )**—return a port number (VIF) for the specified interface

4.7 Manually Editing the Routing Table

When IP needs to transmit a packet, it searches the routing table for an entry that provides the address information it needs to supply when it hands the packet to the link layer for transmission. The information in that table is often entered automatically by a protocol such as RIP, but you can also manually add and delete routing table entries using **routeLib** functions.

A review of the **routeLib** API in Table 4-6 shows a redundancy of add and delete functions. This redundancy supports backward compatibility with earlier **routeLib** and routing table implementations.

Table 4-6 **routeLib Routines**

Routine	Definition
routeAdd( )	Add a static route (class based).*
routeNetAdd( )	Add a route to a destination that is a network.*
routeDelete( )	Delete a static route (class based).*
mRouteAdd( )	Add mask-distinguished static routes to a destination. (CIDR)
mRouteEntryAdd( )	Add a protocol-specific route. (CIDR)
mRouteEntryDelete( )	Delete a protocol-specific route. (CIDR)
mRouteDelete( )	Delete a static route from the routing table. (CIDR)

* These routines have been deprecated. Use the **mRoute*( )** equivalents.

Deprecated Functions

The **routeAdd( )**, **routeDelete( )**, and **routeNetAdd( )** functions date from the class-based VxWorks routing table implementation. Thus, when they add routes, they use the netmasks assumed by the class-based system. As a result, the routes they add are often disastrously inappropriate. These functions still work as documented, but you should consider them obsolete. Do not use them in any new code. You should also upgrade existing code to use **mRouteAdd( )** and **mRouteDelete( )** to manage the static routes in your table.

4.7.1 Adding Gateways (Routers) to a Network

A gateway or router is a machine that is able to forward packets from one network to another. Thus, a gateway has a physical connection to two or more networks. If the destination of a packet is local to a network attached to the gateway, it can deliver the packet directly. Otherwise, the gateway passes the packet to still another gateway (if one is available). This process, called routing, continues until the packet is delivered or is dropped.

To support routing, VxWorks depends on the routing table to distinguish addresses on the local network from addresses that are accessible through a gateway only. Using a routing protocol, such as RIP, VxWorks is able to discover gateways and add or delete them from its routing table dynamically. In addition to these dynamic routes, **routeLib** supplies functions you can use to create static gateway routes. You can use these static routes to provide initial routing table information.

Finally, the VxWorks routing table supports the idea of a default gateway. The gateway you assign to 0.0.0.0 serves as the gateway of last resort. If the routing table does not contain any other entry for a destination IP address, it returns the gateway you assign to 0.0.0.0.

Adding a Gateway on Windows

The procedures vary according to your version of Windows and your networking software package. For the details, see the documentation for your system.

Adding a Gateway on UNIX

A UNIX system can be told explicitly about a gateway in one of two ways: by editing **/etc/gateways** or by using the **route** command. When the UNIX route daemon **routed** is started (usually at boot time), it reads a static routing configuration from **/etc/gateways**. Each line in **/etc/gateways** specifies a network gateway in the following format:

```
net destinationAddr gateway gatewayAddr metric n passive
```

where *n* is the *hop count* from the host system to the destination network (the number of gateways between the host and the destination network) and "passive" indicates the entry is to remain in the routing tables.

For example, consider a system on network 150. The following line in **/etc/gateways** describes a gateway between networks 150 and 161, with an Internet address 150.12.0.1 on network 150. A hop count (metric) of 1 specifies that the gateway is a direct connection between the two networks:

```
net 161.27.0.0 gateway 150.12.0.1 metric 1 passive
```

After editing **/etc/gateways**, you must kill the route daemon and restart it, because it only reads **/etc/gateways** when it starts. After the route daemon is running, it is not aware of subsequent changes to the file.

Alternatively, you can use the **route** command to add routing information explicitly:

```
# route add destination-network gatewayAddr [metric]
```

For example, the following command configures the gateway in the same way as the previous example, which used the **/etc/gateways** file:

```
# route add net 161.27.0.0 150.12.0.1 1
```

Note, however, that routes added with this manual method are lost the next time the system boots.

You can confirm that a route is in the routing table by using the UNIX command **netstat -r**.

Adding a Gateway on VxWorks

The **routeLib** API provides a number of functions that you can use to add routes to the routing table. Some of these functions are now obsolete and included for backward compatibility only. All new development should use the following routines:

mRouteAdd( )	Add mask-distinguished static routes.
mRouteEntryAdd( )	Add a protocol-specific dynamic route.

As input to these functions, you can specify IP addresses using either dotted decimal notation or host names.

▪ About Destination IP Addresses and Destination Networks

Using an IP address, you can specify a single interface on the Internet. Using an IP address and a mask, you can specify a group of IP addresses — a network.

When creating a host entry in your routing table, you specify the destination as a combination of the following values:

– an IP address
– the **RTF_HOST** flag

When creating a network entry in your routing table, you specify the destination as a combination of the following values:

– an IP address
– a network mask

In the current implementation, a VxWorks routing table entry also stores:

– type of service
– protocol ID
– weight (router stack implementation only)

In the router stack, it is possible for the routing table to store multiple distinct entries to the same destination. Only one of these route entries is externally visible to protocols such as IP. This representative route is chosen based on its assigned weight. The entry with the lowest weight value is used as the representative route.

If you add a route using a function that does not let you specify a weight, type of service, or protocol ID, VxWorks assigns appropriate default values for these route characteristics.

Inspecting the Routing Table

Before you edit the table, it is generally a good idea to look at what is already there. To inspect the contents of the routing table, use **routeShow( )**. If a VxWorks host boots through an Ethernet network interface, a typical **routeShow( )** call would display the following:[1]

```
-> routeShow()

ROUTE NET TABLE
destination      gateway           flags  Refcnt  Use       Interface
----------------------------------------------------------------------
136.12.44.0      136.12.44.165     101    0       0              ei0
----------------------------------------------------------------------
ROUTE HOST TABLE
destination      gateway           flags  Refcnt  Use       Interface
----------------------------------------------------------------------
127.0.0.1        127.0.0.1         5      1       0              lo0
----------------------------------------------------------------------
value = 77 = 0x4d = 'M'
```

In the output shown above, the route entry for 136.12.44.0 shows that the flags **RTF_CLONING** (0x100) and **RTF_UP** (0x001, signifying that the route is available for use) are set. This route entry is set when the Ethernet network device "ei0" is initialized. This is a network route and the network mask associated with this route is **0xFFFFFF00**.

Using mRouteAdd( ) to Add Static Gateways and the Default Gateway

To use **mRouteAdd( )** to add a gateway to your routing table, you need to specify a destination, a gateway, a net mask, and a type of service value for the route. The general format of an **mRouteAdd( )** call is as follows:

```
mRouteAdd ( "destination", "gateway", netmask, type-of-service, flags)
```

To use **mRouteAdd( )** to add a default entry to the routing table:

```
mRouteAdd ("0.0.0.0", "gatewayAddrs", 0, 0, 0);
```

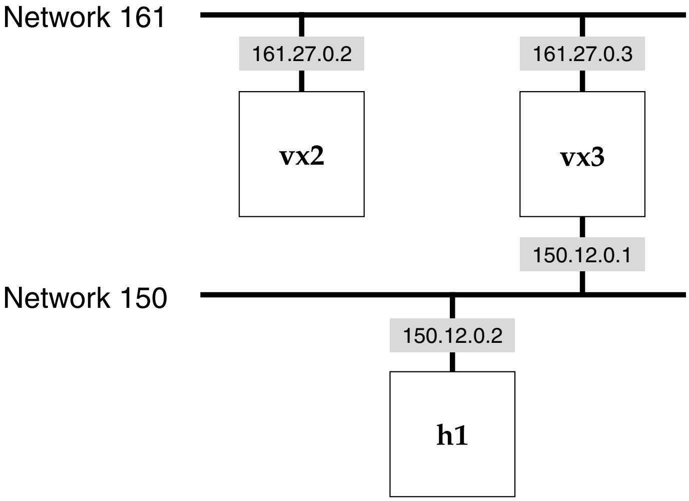
[1] This assumes that VxWorks is configured to include network show routines. The relevant configuration parameter is **INCLUDE_NET_SHOW**.

If the routing table contains a route to 0.0.0.0, the gateway assigned to 0.0.0.0 serves as the default for any destination for which there is no better match. Note also that the netmask is all zeros.

NOTE: Although the boot line **gad** value (if any) automatically adds an entry to your routing table, the **gad** is not used as a default gateway (the gateway associated with 0.0.0.0). Instead, the **gad** gateway has a destination value equal to that of the network (IP address and mask) of the remote network connected to the boot host.

For an example of using **mRouteAdd( )** to add a gateway to the routing table, consider the **vx2** and **vx3** VxWorks targets shown in Figure 4-5. Both have network interfaces that link them to network 161.27.0.0:FFFFFF00. Because **vx3** also has an interface on network 150.12.0.0:FFFFFF00, it can serve as a gateway linking networks 150.12.0.0:FFFFFF00 and 161.27.0.0:FFFFFF00.

Figure 4-5 **Configuring vx3 as a Gateway**



On **vx2**, you can use the following calls to establish **vx3** as a gateway to 150:

```
-> mRouteAdd ("150.12.0.0", "vx3", 0xFFFFFF00, 0, 0 );
```

or:

```
-> mRouteAdd ("150.12.0.0", "161.27.0.3", 0xFFFFFF00, 0, 0 );
```

To confirm that a route is in the routing table, call **routeShow( )**.[1]

Using Masks to Distribute Traffic to the Same Destination IP Address

Unique routing table entries are defined by a destination address, a network mask (or **RTF_HOST** flag for host routes), and a type of service value — not simply a destination IP address. For example, consider the network routing table entries created by the calls:

```
mRouteAdd( "90.0.0.0", "91.0.0.3",   0xFFFFFF00, 0, 0 );
mRouteAdd( "90.0.0.0", "91.0.0.254", 0xFFFF0000, 0, 0 );
```

Both calls specify a destination IP address of 90.0.0.0 but differ in their network mask values. When the routing software searches the routing table, it tries to match a destination IP address with a routing entry based on the stored network address and network mask.

The first **mRouteAdd( )** call creates a routing table entry that matches with destination IP addresses 90.0.0.0 through 90.0.0.255. The second **mRouteAdd( )** call uses a shorter mask and so matches more entries, 90.0.0.0 through 90.0.255.255. This overlaps the range of the first **mRouteAdd( )** call. However, because routing software prefers matches with longer masks, searches for IP addresses within the 90.0.0.0 to 90.0.0.255 range match the 91.0.0.3 interface first.

Thus, these two entries divide the 90.0.0.0:0xFFFFFF00 traffic between 91.0.0.3 and 91.0.0.254. Packets destined to addresses in the 90.0.0.0 through 90.0.0.255 range go to 91.0.0.3. Packets destined to addresses in the 90.0.1.0 through 90.0.255.255 range go to 91.0.0.254.

If you were to delete the 91.0.0.3 route, all its traffic would go out through 91.0.0.254. However, deleting the 91.0.0.254 route would not result in all its traffic going to 91.0.0.3. Only addresses matching 90.0.0.0 within the 0xFFFFFF00 mask can map to the 91.0.0.3 entry. Thus, addresses in the 90.0.1.0 through 90.0.255.255 range are directed to the default gateway (if any) or are discarded as unreachable.

Using mRouteDelete( ) to Delete a Static Entry in the Routing Table

To delete a static routing table entry, use **mRouteDelete( )**. For example, to delete an entry with a destination of 161.27.0.51 with a mask of 0xFFFFFF00, a type of service value of 0, and a flags value of **RTF_HOST**, you would call **mRouteDelete( )** as follows:

```
mRouteDelete( "161.27.0.51", 0xFFFFFF00, 0, RTF_HOST );
```

[1] This function is not built into the Tornado shell. The relevant configuration parameter is **INCLUDE_NET_SHOW**.

The destination IP address as well as the values for network mask, type of service, and flags that you specify in the parameters must match the values for the table entry. Otherwise, the **mRouteDelete( )** call fails.

Adding and Managing Dynamic Routes Manually

Although dynamic routes are normally added and deleted by a protocol, it is possible to use **mRouteEntryAdd( )** to impersonate a protocol when adding a route.

4.8 Proxy ARP for Transparent Subnets

The IP routing discussed previously relies on the assignment of a separate network number to each physical network. This restriction prevents communication between hosts on different physical networks unless each host's routing table contains an entry for the appropriate network number.

Proxy ARP provides a method for assigning the same logical network number to different physical networks without altering the routing table entries for existing hosts. This feature is particularly valuable when creating a shared-memory network on the back plane (see *3.3 The Shared-Memory Backplane Network Driver*, p.21).

RFCs Relevant to Proxy ARP

Proxy ARP is described in Request For Comments (RFC) 925 "Multi LAN Address Resolution," and an implementation is discussed in RFC 1027 "Using ARP to Implement Transparent Subnet Gateways." The ARP protocol is described in RFC 826 "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48-bit Ethernet address for transmission on Ethernet hardware." The implementation of Proxy ARP for VxWorks is based on RFC 925. However, it is a limited subset of that proposal.

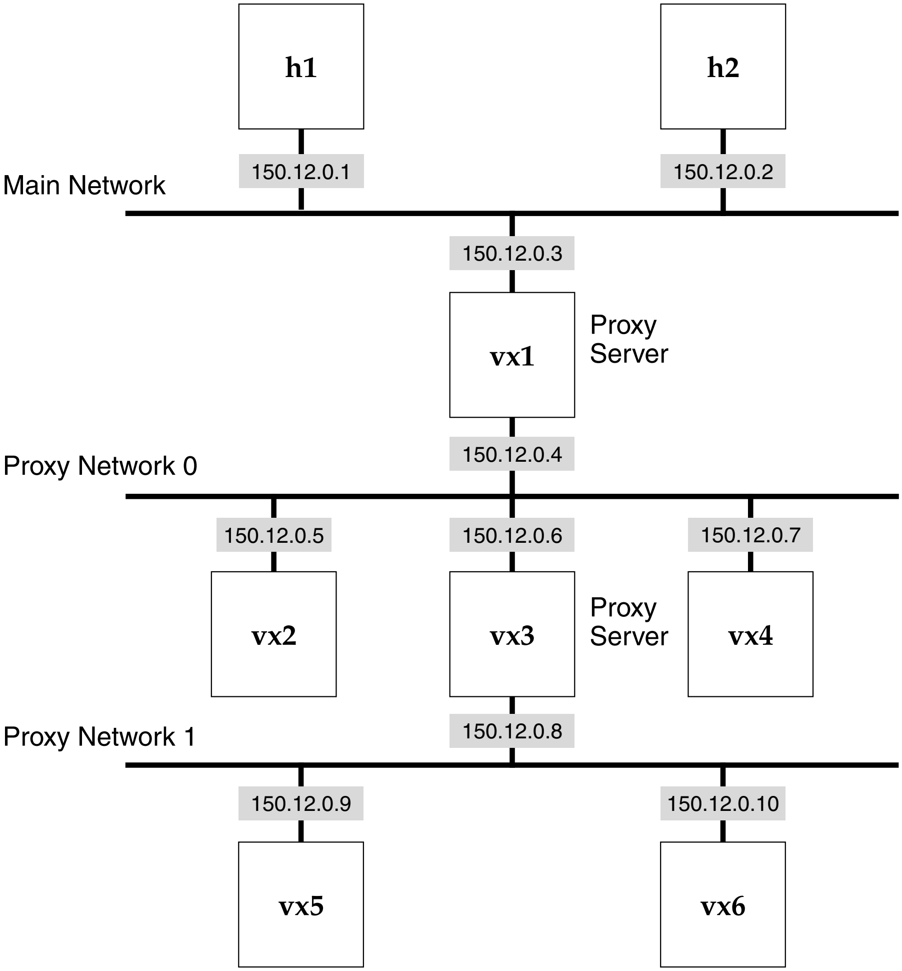
4.8.1 Proxy ARP Protocol Overview

Proxy ARP uses the address resolution protocol (ARP) to provide transparent data transfer across physical network boundaries. The IP transmission process uses ARP to find the required link-level address information for a specific destination address. Ordinarily, ARP messages are restricted to a single physical network. Running a proxy ARP server on a multi-homed host lets ARP requests from hosts on physically separate networks to succeed. From the perspective of individual hosts, the result completely disguises the physical separation of the networks.

A Single Proxy ARP Instance Cannot Serve Both as a Server and a Client

A single proxy ARP instance can act as a server or as a client but not both. Thus, configurations such as that in Figure 4-6 are not supported.

Figure 4-6 **Multi-Tier Configurations *CANNOT* Be Used with Proxy ARP**



This restriction provides built-in protection against the accidental creation of network circles, broadcast storms, and eternally forwarded ARP requests. In addition, the restriction helps avoid routing table scalability issues that could arise because the proxy ARP server edits the local routing table to add one host-specific route per host on the proxy network.

This is not to say that a VxWorks target running VxWorks cannot be the node linking multiple proxy networks. For example, the configuration shown in Figure 4-7 is possible.

Two Instances of Proxy ARP on a Single Target Allow Chaining

In Figure 4-6, the **vx3** target is labeled as a Proxy Server. As a simple proxy server, it cannot fulfill all its functions. The **vx3** target must function both as a proxy server for **vx5** and **vx6** and as a proxy client to **vx1**. To do this, **vx3** must run two instances of proxy ARP; you can configure one instance to function as a server for **vx5**. The other instance should be configured as a proxy client to **vx1**. Still, use such configurations with extreme caution. Used casually, you risk accidentally creating network circles, broadcast storms, and eternally forwarded ARP requests.

4.8.2 Routing and the Proxy ARP Server

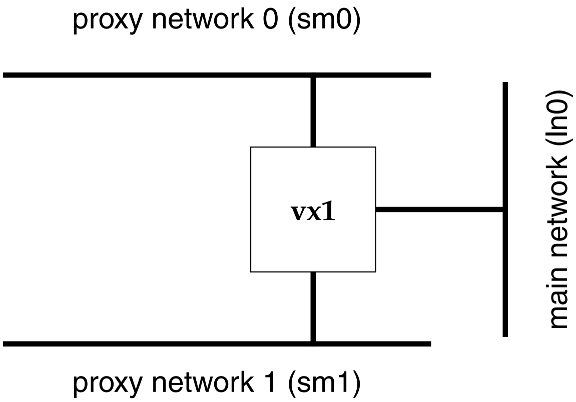
Although a response to an ARP request is necessary for a host to begin an IP data transfer, it is not enough to actually accomplish the data transfer. The data can reach the final destination only if the proxy ARP server added appropriate entries to the local routing table. Setting up a network interface ordinarily creates a single route entry that serves for all hosts on the network attached to the network interface. By default, the proxy ARP server automatically creates host-specific routing table entries for proxy clients.

4.8.3 Proxy ARP and Broadcast Datagrams

When you broadcast an IP packet on a network, all nodes on that network are expected to receive the packet. To make the proxy network truly appear part of the main network, the proxy ARP server can forward broadcasts to and from the proxy network. Thus, if a broadcast datagram originates on a proxy network (and the port is enabled), the server forwards the broadcast to the main network, and to all other proxy networks that have the same main network.

For example, in Figure 4-7, if a broadcast originates on **sm1**, proxy ARP may forward the broadcast to **ln0** and **sm0**.

Figure 4-7 **Broadcast Datagram Forwarding**



If the broadcast originates on the main network (and the port is enabled), the server forwards the broadcast to all appropriate proxy networks. For example, in Figure 4-7, a broadcast from **ln0** is forwarded to both **sm0** and **sm1**. To prevent forwarding loops, broadcasts forwarded onto proxy networks are given a time-to-live value of 1.

However, most broadcasts are not of interest to any host outside the true physical network. To give you selective control over which broadcasts are forwarded and which are not, you can configure the proxy ARP server to forward the broadcasts of specific destination UDP ports only. To enable proxy ARP broadcast forwarding on a UDP port, call **proxyPortFwdOn( )**. To disable forwarding for a port, call **proxyPortFwdOff( )**. By default, forwarding is disabled on all ports.

4.8.4 Proxy ARP Configuration

To include proxy ARP in VxWorks, reconfigure the image and rebuild it to include the proxy server. The relevant configuration parameter is **INCLUDE_PROXY_SERVER**.

On the target with processor zero (the shared-memory network master), the proxy ARP server assumes the main network is on the other side of the default local network interface. This interface uses the IP address:mask value specified in the **ead** boot parameter, **inet on ethernet (e)**.

For a shared memory configuration, the address of the interface between the proxy ARP server and the backplane depends on whether you have set the **INCLUDE_PROXY_DEFAULT_ADDR** configuration parameter. If this configuration parameter is not set, the interface to the back plane gets its address from the **bad** boot parameter, **inet on backplane (b)**. If **INCLUDE_PROXY_DEFAULT_ADDR** is set, the target hosting the proxy ARP server generates the address for the back plane interface by adding one to the **ead** boot parameter.

For a shared memory configuration, how you assign an address to each host on the proxy network depends on whether you have configured VxWorks with the configuration parameter **INCLUDE_SM_SEQ_ADDR** set.

If **INCLUDE_SM_SEQ_ADDR** is not set, each slave target uses its **ead** value as the IP address for its interface to shared memory back plane.

If **INCLUDE_SM_SEQ_ADDR** is set, the slave targets generate their IP address by adding their CPU number (1, 2, 3, and so on) to the IP address of the interface between the proxy ARP master and the shared memory back plane. For example, if the proxy ARP server has a backplane address of 150.12.0.4, the first slave is 150.12.0.5, the second slave 150.12.0.6, and so on.

NOTE: When using proxy ARP, it is no longer necessary to specify the gateway. Each target on the shared-memory network (except the proxy server) can register itself as a proxy client by specifying the proxy ARP flag, **0x100**, in the boot flags instead of specifying the gateway.

Proxy ARP not Limited To a Shared Memory Network

Although this document describes the use of Proxy ARP over a shared memory network, the current Proxy ARP implementation is no longer limited to the shared memory network.

Proxy ARP with Shared Memory and IP Routing

Even if you are using the same board for the master and the slaves, the master and slaves need separate BSP directories because they have different configurations. For more information on configuration, see the *Tornado User's Guide: Customizing VxWorks AE*.

Proxy ARP and Shared Memory Configuration Parameters:

(1) PING client (configuration parameter: **INCLUDE_PING**)

(2) Shared memory network initialization (**INCLUDE_SM_NET**)

(3) Proxy ARP server (**INCLUDE_PROXY_SERVER**)

(4) Auto address setup (**INCLUDE_SM_SEQ_ADDR**)—required for default addressing of proxy clients, but required in both client and server

(5) Default address for back plane—required only for default addressing

Parameters for proxy ARP sever**:**
INCLUDE_PROXY_SERVER
SM_OFF_BOARD=FALSE

Parameters for proxy client**:**
INCLUDE_PROXY_CLIENT
SM_OFF_BOARD=TRUE

Setting Up Boot Parameters and Booting

See *3.3 The Shared-Memory Backplane Network Driver*, p.21 for information on booting shared memory networks. After booting **vx1** (the master, Figure 4-8), use **smNetShow( )** to find the shared memory anchor, which will be used with the slave boot device (for **vx2**, **vx3**, and **vx4**). You will need to run **sysLocalToBusAddr( )** on the master and **sysBusToLocalAddr( )** on each type of target to get the correct bus address for the anchor.

Creating Network Connections

From vx1 (the master): Use **mRouteAdd( )** to tell the master (the proxy server) about the IP routing network:

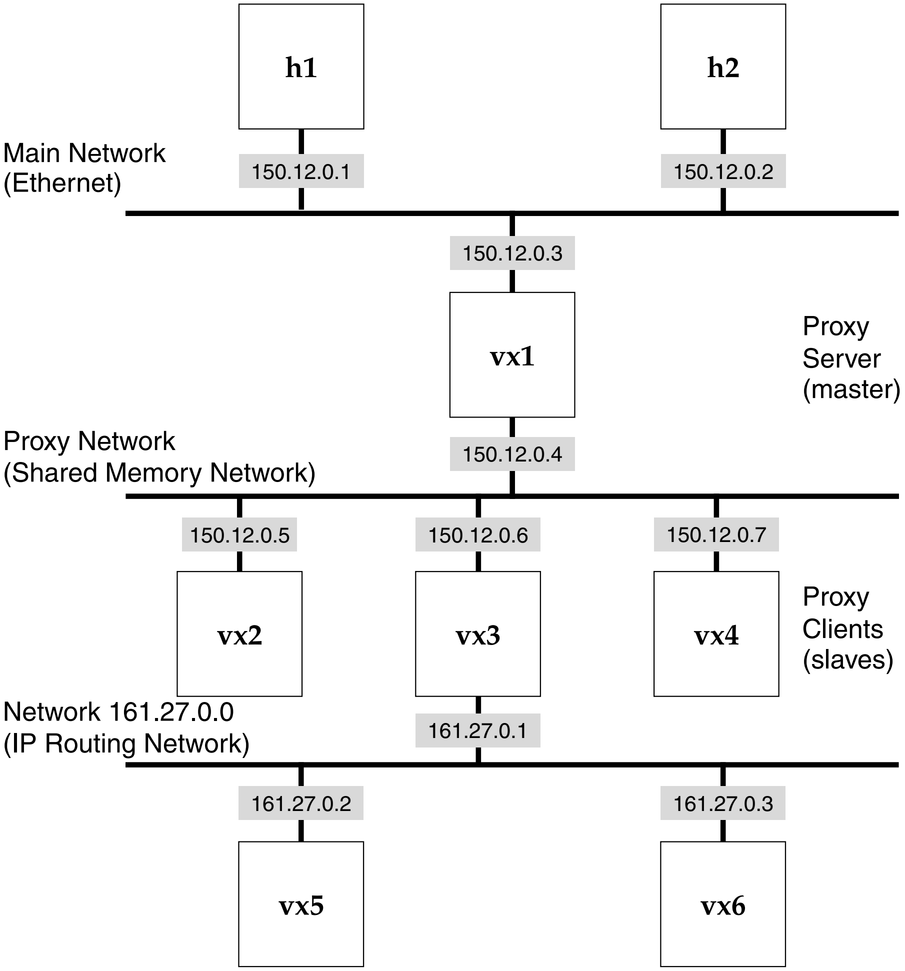
```
-> mRouteAdd ("161.27.0.0", "150.12.0.6", 0xffff0000, 0, 0)

value = 0 = 0x0
```

From vx3: Since **vx3** boots from the shared memory network, it needs to have its connection to the IP routing network brought up explicitly. The following example shows how to do this for **vx3** in Figure 4-8:

```
-> usrNetIfAttach ("ln", "161.27.0.1")
Attaching network interface ln0...done.
value = 0 = 0x0
-> usrNetIfConfig ("ln", "161.27.0.1", "t0-1", 0xffff0000)
value = 0 = 0x
```

Figure 4-8 **Multi-Tier Example Using Proxy ARP and IP Routing**



NOTE: Substitute the appropriate network boot device for *"ln"*. The correct boot device is given in the output from a boot prompt **?** command.

Diagnosing Shared Memory Booting Problems

See *Troubleshooting*, p.38 for information on debugging the shared memory network.

Diagnosing Routing Problems

The following routines can be useful in locating the source of routing problems:

ping( )
Starting from **vx1**, ping other processors in turn to see if you get the expected result. The function returns **OK** if it reaches the other machine, or **ERROR** if the connection fails.

smNetShow( )
This routine displays cumulative activity statistics for all attached processors.

arpResolve( )
This function tells you the hardware address for a specified Internet address. This function supercedes the functionality provided by the now obsolete **etherAddrResolve( )**.

arpShow( )
This routine displays the current Internet-to-Ethernet address mappings in the system ARP table.

arptabShow( )
This routine displays the known Internet-to-Ethernet address mappings in the ARP table

routeShow( )
This routine displays the current routing information contained in the routing table.

ifShow( )
This routine displays the attached network interfaces for debugging and diagnostic purposes.

proxyNetShow( )
This routine displays the proxy networks and their associated clients.

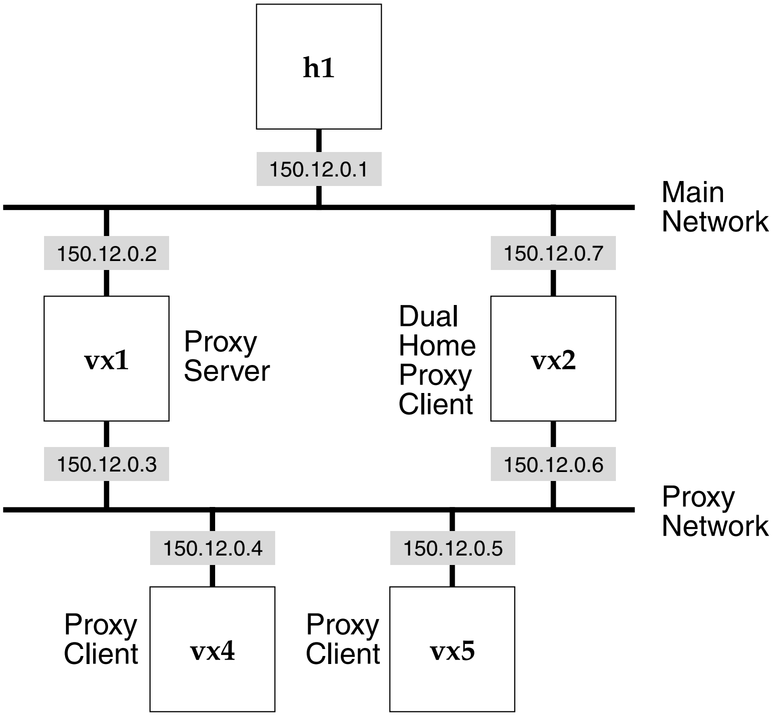
proxyPortShow( )
This routine displays the ports currently enabled.

Routing Configuration for Multi-Homed Proxy Clients

If a proxy client also has an interface to the main network, some additional configuration is required for optimal communications. The proxy client's routing tables must have network-specific routes with netmask **0xFFFFFFFF** for nodes on the proxy network, and a network-specific route for the main network. Otherwise, traffic travels an extra unnecessary hop through the proxy server.

In the example shown in Figure 4-9, **vx1** is the proxy server and **vx2** is a proxy client with an interface on the main network. You must configure **vx2** to store network-specific routes to each of the other proxy clients (**vx4** and **vx5**) and the main network. In addition, these routes must use a mask of **0xFFFFFFFF**. Otherwise, any traffic from **vx2** to **vx4** (or **vx5**) unnecessarily travels over the main network through the proxy server (**vx1**).

Figure 4-9 **Routing Example**



The following is an example of **vx2**'s routing table. The routing table is manipulated using **mRouteAdd( )** and **mRouteDelete( )**. For more information, see the reference entry for **routeLib**.

```
Destination                                             Gateway
150.12.0.4 (network with netmask   0xffffffff)         150.12.0.6
150.12.0.5 (network with netmask   0xffffffff)         150.12.0.6
150.12.0.0 (network)                                   150.12.0.7
```

Broadcasts Configuration for Multi-Homed Proxy Clients

A proxy client that also has an interface connected to the main network must disable broadcast packets from the proxy interface. Otherwise, it receives duplicate copies of broadcast datagrams (one from Ethernet and one from the shared-memory network).

4.9 Using Unnumbered Interfaces

Typically, each IP host or router needs its own IP address. However, when an interface is the local end of a point-to-point link, it is possible for that interface to borrow the IP address of a local interface that joins the router to a larger network. This scheme, called unnumbered interfaces, is described in section 2.2.7 of RFC 1812. VxWorks provides support for unnumbered interfaces using **ifUnnumberedSet( )**, an **ifLib** routine.

As input, **ifUnnumberedSet( )** expects the name of the local unnumbered interface and the router IDs that identify the ends of the point-to-point link. Both router IDs are "borrowed" IP addresses. For example, Figure 4-10 shows 147.38.11.150 serving as the IP address of both fei0 and fei1 on Vx A. Similarly, 138.12.12.12 serves as the IP address for both fei0 and fei1 on Vx B.

To use unnumbered interfaces to connect two VxWorks hosts:

1. Call **ipAttach( )** to assign a name to the local unnumbered interface on a point-to-point link.

2. Call **ifUnnumberedSet( )** to assign a borrowed IP address to source interface and associate that interface with a destination interface. For example, on Vx A in Figure 4-10, the system manager calls:

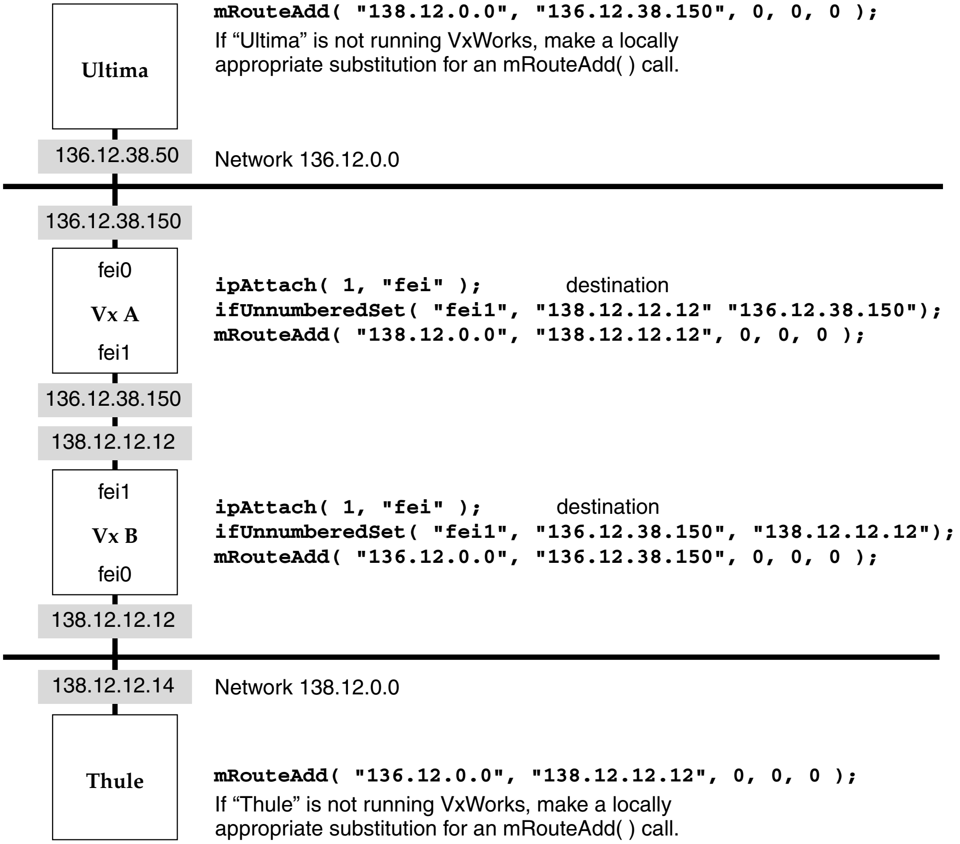
   ```
   ifUnnumberedSet( "fei1", "138.12.12.12", "136.12.38.150" );
   ```

   This reuses 136.12.38.150 for the local interface and tells Vx A that 138.12.12.12 is the interface on the other side of the point-to-point link.

   Similarly on Vx B, the system manager calls:

   ```
   ifUnnumberedSet( "fei1", "138.12.12.12", "136.12.38.150" );
   ```

Figure 4-10 **Unnumbered Interface Setup**



```
mRouteAdd( "138.12.0.0", "136.12.38.150", 0, 0, 0 );
```

If “Ultima” is not running VxWorks, make a locally appropriate substitution for an mRouteAdd( ) call.

```
ipAttach( 1, "fei" );
ifUnnumberedSet( "fei1", "138.12.12.12" "136.12.38.150");
mRouteAdd( "138.12.0.0", "138.12.12.12", 0, 0, 0 );
```

```
ipAttach( 1, "fei" );
ifUnnumberedSet( "fei1", "136.12.38.150", "138.12.12.12");
mRouteAdd( "136.12.0.0", "136.12.38.150", 0, 0, 0 );
```

```
mRouteAdd( "136.12.0.0", "138.12.12.12", 0, 0, 0 );
```

If “Thule” is not running VxWorks, make a locally appropriate substitution for an mRouteAdd( ) call.

3. Use **mRouteAdd( )** to create appropriate network routes on all machines participating in the system. For example, on Vx A in Figure 4-10, the system manager calls:

```
mRouteAdd( "138.12.0.0", "138.12.12.12", 0, 0, 0 );
```

This tells Vx A that it can reach network 138.12.0.0 by forwarding to the router at 138.12.12.12. The previous **ifUnnumberedSet( )** call told it that it could reach 138.12.12.12 by broadcasting on fei1.

4.10 Network Byte Order

A single network can contain CPUs using different internal architectures. The numeric representation schemes of these architectures can differ: some use *big-endian* numbers, and some use *little-endian* numbers. To permit exchanging numeric data over a network, some overall convention is necessary. *Network byte order* is the convention that governs exchange of numeric data related to the network itself, such as socket addresses or shared-semaphore IDs. Numbers in network byte order are big-endian.

The routines in Table 4-7 convert longs and shorts between host and network byte order. To minimize overhead, macro implementations (which have no effect on architectures where no conversion is needed) are also available, in **h/netinet/in.h**.

Table 4-7 **Network Address Conversion Macros**

Macro	Description
htonl	Convert a **long** from host to network byte ordering.
htons	Convert a **short** from host to network byte ordering.
ntohl	Convert a **long** from network to host byte ordering.
ntohs	Convert a **short** from network to host byte ordering.

To avoid macro-expansion side effects, do not apply these macros directly to an expression. The following increments **pBuf** four times (on little-endian architectures):

```
pBufHostLong = ntohl (*pBuf++);     /* UNSAFE */
```

It is safer to increment separately from the macro call. The following increments **pBuf** only once, whether the architecture is big- or little-endian:

```
pBufHostLong = ntohl (*pBuf);
pBuf++;
```

4.11 Assigning Host Names to IP Addresses

On a VxWorks host, you can use the functions of the **ifLib** library to associate Internet addresses with network interfaces and addresses. For a listing of these configuration functions, see the reference entry for **ifLib**. To add host names, use the functions supplied in **hostLib**. For a listing of these configuration functions, see the reference entry for **hostLib**.

Associating Internet Addresses with Host Names

The underlying Internet protocol uses the 32-bit Internet addresses of systems on the network. People, however, prefer to use system names that are more meaningful to them. Thus VxWorks and most host development systems maintain their own maps between system names and Internet addresses.

On UNIX systems, **/etc/hosts** contains the mapping between system names and Internet addresses. Each line consists of an Internet address and the assigned name(s) for that address:

```
150.12.0.1 vx1
```

There must be an entry in this file for each UNIX system and for each VxWorks host on the network. For more information on **/etc/hosts**, see your UNIX system reference entry **hosts**(5).

On a VxWorks host, call **hostAdd( )** to associate system names with Internet addresses. Make one call to **hostAdd( )** for each system with which the VxWorks host communicates. For example:

```
hostAdd ("vx1", "150.12.0.1");
```

NOTE: In addition to **hostAdd( )**, VxWorks also includes DNS. You can use DNS to create and automatically maintain host-name/address associations for a VxWorks host. See *9. DNS and SNTP*.

To associate more than one name with an Internet address, **hostAdd( )** can be called several times with different host names and the same Internet address. The routine **hostShow( )** displays the current system name and Internet address associations. In the following example, 150.12.0.1 is accessible using the names **host**, **myHost**, and **widget**:

```
-> hostShow
value = 0 = 0x0
```

The standard output device displays the following output:

```
hostname         inet address     aliases
--------          -----------     -------
localhost        127.0.0.1
host             150.12.0.1

-> hostAdd "myHost", "150.12.0.1"
value = 0 = 0x0
-> hostAdd "widget", "150.12.0.1"
value = 0 = 0x0
-> hostShow
value = 0 = 0x0
```

Now standard output displays the following:[1]

```
hostname         inet address     aliases
--------          -----------     -------
localhost        127.0.0.1
vx1             150.12.0.1       myHost widget
value = 0 = 0x0
```

The VxWorks startup routine, **usrNetInit( )** in **usrNetwork.c**, automatically adds the name of the host from which the target booted, using the host name specified in the boot parameters.

[1] Internally, **hostShow( )** uses the resolver library to access DNS to get the information it needs to respond to a query.

5
Network Configuration Protocols

5.1 Introduction

This chapter describes the protocols used for retrieving network configuration information. These protocols are:

- DHCP (Dynamic Host Configuration Protocol)
- BOOTP (Bootstrap Protocol)
- SNMP (Simple Network Management Protocol)

DHCP uses the BSD Packet Filter (BPF) in its boot scheme. For this reason, this chapter also discusses BPF.

Both a DHCP server and a BOOTP server can supply an Internet host with an IP address and related configuration information. When a BOOTP server assigns an IP address to an Internet host, the address is permanently assigned.

A DHCP server is more flexible. It assigns an IP address on either a permanent or leased basis. Leased IP addresses are an advantage in environments where large numbers of Internet hosts join the network for sessions of limited duration. Predicting the duration of such sessions is usually not possible at the time the leases are assigned.

Fortunately, a DHCP client has the ability to contact its server again and renegotiate the lease on an IP address (or request a replacement address). Unlike a BOOTP client, a DHCP client must remain active for as long as the target needs a current lease on an IP address.

Also included at the end of this section is a brief description of SNMP, an optional networking product that is compatible with VxWorks and purchased separately.

For detailed usage information on SNMP, see the *WindNet SNMP* VxWorks *Optional Product Supplement*.

DHCP and BOOTP Work Only on Broadcast-capable MUX Devices

Both the DHCP and BOOTP clients use broadcasts to discover an appropriate server. Both protocols require network drivers that are capable of implementing some sort of broadcast. In addition, these drivers must be implemented using the MUX interface (in other words, as ENDs or NPT drivers).

5.2 BOOTP, Bootstrap Protocol

BOOTP is a basic bootstrap protocol implemented on top of the Internet User Datagram Protocol (UDP). The BOOTP client provided with VxWorks lets a target retrieve a single set of configuration parameters from a BOOTP server. Included among these configuration parameters is a permanently assigned IP address and a filename specifying a bootable image. To retrieve the boot file, the target can use a file transfer program, such as TFTP, FTP, or RSH.[1]

NOTE: For many applications, the DHCP protocol can function as an alternative to BOOTP.

BOOTP offers centralized management of target boot parameters on the host system. Using BOOTP, the target can retrieve the boot parameters stored on a host system. This lets you set up systems that can automatically reboot without the need to enter the configuration parameters manually.

A BOOTP server must be running (with **inetd** on a UNIX system) on the boot host, and the boot parameters for the target must be entered into the BOOTP database (**bootptab**). The format of this database is server specific. An example **bootptab** file is described in *The BOOTP Database*, p.97.

BOOTP is a simple protocol based on single-packet exchanges. The client transmits a BOOTP request message on the network. The server gets the message, and looks

[1] For the complete BOOTP protocol specification, see *RFC 951* "Bootstrap Protocol (BOOTP)," *RFC 1542* "Clarifications and Extensions for BOOTP," and *RFC 1048* "BOOTP Vendor Information Extensions."

up the client in the database. It searches for the client's IP address if that field is specified; if not, it searches for the client's hardware address.

If the server finds the client's entry in the database, it performs name translation on the boot file, and checks for the presence (and accessibility) of that file. If the file exists and is readable, the server sends a reply message to the client.

5.2.1 BOOTP Configuration

Using the BOOTP server to supply boot parameters requires that you edit the server's BOOTP database file, **bootptab**. However, the specifics of how to do this can vary from server to server. Refer to the manuals for your host's BOOTP server. If the host does not provide a BOOTP server as part of the operating system, a copy of the publicly available CMU BOOTP server is provided in **target/unsupported/bootp2.1**.

The following discussion of how to modify **bootptab** applies to the CMU BOOTP server.

The BOOTP Database

To register a VxWorks target with the BOOTP server, you must enter the target parameters in the host's BOOTP database (**/etc/bootptab**). The following is an example **bootptab** for the CMU version of the BOOTP server:

```
# /etc/bootptab: database for bootp server (/etc/bootpd)
# Last update Mon 11/7/88 18:03
# Blank lines and lines beginning with '#' are ignored.
#
# Legend:
#
#    first field -- hostname
#             (may be full domain name and probably should be)
#
#    hd -- home directory
#    bf -- boot file
#    cs -- cookie servers
#    ds -- domain name servers
#    gw -- gateways
#    ha -- hardware address
#    ht -- hardware type
#    im -- impress servers
#    ip -- host IP address
#    lg -- log servers
#    lp -- LPR servers
#    ns -- IEN-116 name servers
#    rl -- resource location protocol servers
#    sm -- subnet mask
#    tc -- template host (points to similar host entry)
#    to -- time offset (seconds)
#    ts -- time servers
#
# Be careful to include backslashes where they are needed. Weird (bad)
# things can happen when a backslash is omitted where one is intended.
#
# First, we define a global entry which specifies what every host uses.

global.dummy:\
:sm=255.255.255.0:\
:hd=/usr/wind/target/vxBoot:\
:bf=vxWorks:

vx240:ht=ethernet:ha=00DD00CB1E05:ip=150.12.1.240:tc=global.dummy
vx241:ht=ethernet:ha=00DD00FE2D01:ip=150.12.1.241:tc=global.dummy
vx242:ht=ethernet:ha=00DD00CB1E02:ip=150.12.1.242:tc=global.dummy
vx243:ht=ethernet:ha=00DD00CB1E03:ip=150.12.1.243:tc=global.dummy
vx244:ht=ethernet:ha=0000530e0018:ip=150.12.1.244:tc=global.dummy
```

Note that common data is described in the entry **global.dummy**. Any target entries that want to use the common data use **tc=global.dummy**. Any target-specific information is listed separately on the target line. For example, in the previous file, the entry for the target **vx244** specifies only its Ethernet address (0000530e0018) and IP address (150.12.1.244). The subnet mask (255.255.255.0), home directory (**/usr/wind/target/vxBoot**), and boot file (VxWorks) are taken from the common entry **global.dummy**.

Editing the BOOTP Database to Register a Target

To register a target with the BOOTP server, log onto the host machine, edit the BOOTP database file to include an entry that specifies the target address (**ha=**), IP address (**ip=**), and boot file (**bf=**). For example, to add a target called **vx245**, with Ethernet address 00:00:4B:0B:B3:A8, IP address 150.12.1.245, and boot file **/usr/wind/target/vxBoot/vxWorks**, you would add the following line to the file:

```
vx245:ht=ethernet:ha=00004B0BB3A8:ip=150.12.1.245:tc=global.dummy
```

Note that you do not need to specify the boot filename explicitly. The home directory (**hd**) and the boot file (**bf**) are taken from **global.dummy**.

When performing the boot filename translation, the BOOTP server uses the value specified in the boot file field of the client request message as well as the **bf** (boot file) and the **hd** (home directory) field in the database. If the form of the filename calls for it (for example, if it is relative), the server prepends the home directory to the filename. The server checks for the existence of the file; if the file is not found, it sends no reply. For more information, see **bootpd** in the manual for your host.

When the server checks for the existence of the file, it also checks whether its read-access bit is set to public, because this is required by **tftpd**(8) to permit the file transfer. All filenames are first tried as *filename.hostname* and then as *filename*, thus providing for individual per-host boot files.

In the previous example, the server first searches for the file **/usr/wind/target/vxBoot/vxWorks.vx245**. If the file does not exist, the server looks for **/usr/wind/target/vxBoot/vxWorks**.

5.3 DHCP, Dynamic Host Configuration Protocol

DHCP, an extension of BOOTP, is designed to supply clients with all of the Internet configuration parameters defined in the Host Requirements documents (RFCs 1122 and 1123) without manual intervention. Like BOOTP, DHCP allows the permanent allocation of configuration parameters to specific clients. However, DHCP also supports the assignment of a network address for a finite lease period. This feature allows the serial reassignment of network addresses to different clients. This feature is useful when IP addresses are limited and the clients connect to the network for limited periods, such as is usually the case with clients that connect to the network over a modem.

The DHCP implementation provided with VxWorks conforms to the Internet standard *RFC 2131*.

VxWorks DHCP Components

VxWorks includes a DHCP client, server, and relay agent. The DHCP client can retrieve one or more sets of configuration parameters from either a DHCP or BOOTP server. The VxWorks DHCP client also maintains any leases it has retrieved. Likewise, the VxWorks DHCP server can process both BOOTP and DHCP messages. Both the client and server implementations support all options described in *RFC 2132*. The DHCP relay agent provides forwarding of DHCP and BOOTP messages across subnet boundaries.

Interface Settings Retrieved Using DHCP

If the server is configured to provide them, a lease can include configuration parameters in addition to an assigned IP address. To minimize network traffic, the DHCP client sets configuration values to the defaults specified in the Host Requirements documents (RFCs 1122 and 1123) if the server does not specify values for the corresponding parameters.

Unlike the configuration parameters supplied by BOOTP, the DHCP-assigned configuration parameters can expire. Although the DHCP server can duplicate BOOTP behavior and issue a permanent IP address to the client, the lease granted is usually temporary. To continue using the assigned parameters, the client must periodically contact the issuing server to renew the lease.

WARNING: The Tornado tools do not currently have any way to discover or respond to a change in the target's IP address. Such a change breaks the network connection. In response, you must manually reconnect the Tornado tools to the target's new IP address. During development, this is rarely a serious problem, and you can avoid it by having the DHCP server issue an infinite lease on the target's IP address.

5.3.1 Including DHCP Components in an Image

The VxWorks DHCP implementation includes a server, a client, and a relay agent. You can configure your image to include all, two, one, or none of these components using the following configuration parameters:

INCLUDE_DHCPS
Includes the DHCP server.

INCLUDE_DHCPC
Includes the DHCPv4 run-time client. You need this code if you want the target to boot using DHCP.

INCLUDE_DHCPR
Includes the DHCP relay agent. Include the DHCP relay agent if the target must relay information from a DHCP server on a different subnet.

After setting any of the above configuration parameters, rebuild VxWorks.

NOTE: If you are building a non-AE VxWorks image from the command line (that is, using a BSP without the project facility), you can add network components such as the DHCP client and server by editing the **config.h** header file for the BSP. For example, to include the DHCP client, you could edit the BSP **config.h** to include the line:

```
#define INCLUDE_DHCPC
```

However, the location of this line in the file is critical. The **config.h** file also includes **configAll.h**, which contains statements that defines other constants depending upon whether **INCLUDE_DHCP** is defined. Therefore, the **INCLUDE_DHCP** statement must come before the inclusion of **configAll.h**. Otherwise, the BSP build will likely fail.

5.3.2 Configuring the DHCP Client

The following configuration parameters are set by default for the DHCP client:

DHCPC_SPORT—DHCP Client Target Port
Port monitored by DHCP servers. Default: 67.

DHCPC_CPORT—DHCP Client Host Port
Port monitored by DHCP clients. Default: 68.

DHCPC_MAX_LEASES—DHCP Client Maximum Leases
Maximum number of simultaneous leases. Default: 4.

DHCPC_OFFER_TIMEOUT—DHCP Client Timeout Value
Seconds to wait for multiple offers. Default: 5.

DHCPC_DEFAULT_LEASE—DHCP Client Default Lease
Desired lease length in seconds. Default: 3600.

DHCPC_MIN_LEASE—DHCP Client Minimum Lease
Minimum allowable lease length (seconds). Default: 30.

DHCPC_MAX_MSGSIZE—DHCP Client Maximum Message Size
Maximum size (in bytes) for a DHCP message. Default: 590.
The default value is the minimum DHCP message in an Ethernet frame.

When setting values for these parameters, keep in mind that the DHCP client rejects all offers whose duration is less than the minimum lease. Therefore, setting the DHCP Client Minimum Lease value too high could prevent the retrieval of any configuration parameters. In addition, if the DHCP client is used at boot time, the values for DHCP Client Target Port and DHCP Client Host Port used in the boot program and run-time image must match.

Finally, the DHCP Client Maximum Leases limit on multiple concurrent leases includes a lease established at boot time. For example, if this limit has a value of four, and if a boot-time DHCP client retrieves a lease, the run-time DHCP client is limited to three additional sets of configuration parameters (until the boot-time lease expires).

NOTE: In addition to setting values for the defines mentioned above, asynchronous use of DHCP requires that you provide an event hook routine to handle lease events. For more information, see the **dhcpcEventHookAdd( )** reference entry.

5.3.3 Configuring DHCP Servers

Configuring the DHCP server requires that you create a pool of configuration parameter sets. Each parameter set must include an IP address. When a DHCP client makes a request of the server, the server can then assign a parameter set to the client (either permanently or on a leased basis). To store and maintain this pool of configuration parameter sets, some DHCP servers use one or more files. This approach is analogous to the use of the **bootptab** file associated with SunOS BOOTP servers. The unsupported DHCP server distributed with VxWorks takes this approach.

However, some VxWorks targets do not include a file system. The supported target-resident DHCP server does not use a file-based mechanism for parameter storage. Instead, the target-resident server maintains configuration parameters in memory-resident structures. To control the contents of these memory-resident structures, you can add entries using **dhcpLeaseEntryAdd( )**, or you can modify the source code that defines these structures.

The following sections describe how to configure the supported DHCP server. Also included are pointers to reference information on configuring the unsupported DHCP server. If you decide to use a third-party DHCP server, consult the configuration information in the vendor-supplied documentation.

Configuring the Supported DHCP Server

Configuring the supported (target-resident) DHCP server involves setting appropriate values for certain configuration parameters. For more information on configuring VxWorks, see the *Tornado User's Guide: Customizing VxWorks AE*. The relevant configuration parameters follow:

DHCPS_LEASE_HOOK — DHCP Server Lease Storage Routine
Default: None. This constant specifies the name of the routine that handles non-volatile storage of the active leases. For more information, see *Storing and Retrieving Active Network Configurations*, p.106.

DHCPS_ADDRESS_HOOK — DHCP Server Address Storage Routine
Default: None. This constant specifies the name of an optional storage routine. For more information, see *Storing and Retrieving Active Network Configurations*, p.106.

DHCPS_DEFAULT_LEASE — DHCP Server Standard Lease Length
Default: 3600. This constant specifies the default lease length in seconds. This value applies if no explicit value is set in the address pool.

DHCP_MAX_HOPS — DHCP Server/Relay Agent Network Radius
Default: 4. This value limits the number of subnets that a DHCP message can cross (prevents network flooding). The maximum valid value is 16.

DHCP_SPORT — DHCP Server/Relay Agent Host Port
Default: 67. This value specifies the port monitored by DHCP servers.

DHCPS_CPORT — DHCP Server/Relay Agent Target Port
Default: 68. This value specifies the port monitored by DHCP clients.

DHCPS_MAX_MSGSIZE — DHCP Server/Relay Agent Maximum Message Size
Maximum size (in bytes) for a DHCP message. Default: 590.
The default value is the minimum DHCP message in an Ethernet frame.

Configuring the Lease Table in the Supported DHCP Server

To determine its initial configuration data, the supported DHCP server uses the **dhcpsLeaseTbl[ ]** defined in **target/config/comps/src/net/usrNetDhcpsCfg.c**. This table describes the server's pool of network configuration parameter sets. It has the following format:

```
DHCPS_LEASE_DESC dhcpsLeaseTbl [] =
    {
    /* {"Name", "Start IP", "End IP", "parameters"} */

    {"dflt", NULL, NULL, DHCPS_DEFAULT_ENTRY},

    /* Sample database entries. */

/*  {"ent1", "90.11.42.24", "90.11.42.24",
     "clid=\"1:0x08003D21FE90\":maxl=90:dfl l=60"},   */
/*  {"ent2", "90.11.42.25", "90.11.42.26",
     "snmk=255.255.255.0:maxl=90:dfl1=70:file=/vxWorks"},*/

/*  {"ent3", "90.11.42.27", "90.11.42.27",
     "maxl=0xffffffff:file=/vxWorks"}, */

/*  {"entry4", "90.11.42.28", "90.11.42.29",
     "albp=true:file=/vxWorks"} */

    };
```

Each entry in this lease table must include a unique entry name of up to eight characters and an IP address range for assignment to requesting clients. The parameters field contains a colon-separated list of optional parameters for inclusion in the DHCP server's response. If subnetting is in effect, a critical entry in the parameters field is the subnet mask (**snmk**). The server does not issue addresses to clients that would change their current subnet. The address pool must specify a correct subnet mask if the default class-based mask is not valid.

A complete description of the parameters field is found in the manual pages for the DHCP server. Any parameters not specified take default values according to the Host Requirements Documents (*RFC 1122* and *1123*). The server can also read additional entries from an optional storage hook (*Storing and Retrieving Active Network Configurations*, p.106). The most commonly used lease table parameters are:

clid Indicates that this is a manual lease. Such a lease is issued only to the client with the matching **type:id** pair. The address range for these entries must specify a single IP address. The sample shown for "ent1" uses the hardware address that the supported DHCP client uses for an identifier.

maxl Indicates that this lease is dynamic. This parameter specifies the maximum lease duration granted to any requesting client. The automatic lease illustrated in the third sample entry is implied by the assignment of an infinite value for **maxl**.

albp Indicates a special type of automatic lease. Setting the **albp** parameter to true in the fourth entry marks this lease as suitable for BOOTP clients that contact this DHCP server.

siad Specifies the (boot) server Internet address, the IP address of the boot host.

Of the parameters shown above, the first three, **clid**, **max1**, and **albp**, indicate lease types. The server uses the lease type to select one of the three supported mechanisms for IP address allocation. With manual allocation, DHCP simply conveys the related manual lease to the client. If dynamic allocation is used, the protocol assigns one of the dynamic leases to the client for a finite period.

Automatic allocation assigns a permanent IP address from the corresponding automatic leases.

Dynamic allocation is the only method that allows reuse of addresses. The allocation type defines the priority for assigning an IP address to a DHCP client. Manual allocations have the highest priority, and automatic allocations the lowest. Among automatic leases, the preferred configurations are those available only to DHCP clients.

Configuring the Relay Agent Table in the Supported DHCP Server

If the DHCP server expects messages from relay agents, you must list those agents (identified by IP address and subnet number) in the **dhcpsRelayTbl[ ]**. This table acts as an authorization list. If messages arrive from relay agents not listed in the table, the messages are ignored.

```
DHCPS_RELAY_DESC dhcpsRelayTbl [] =
    {
    /*
     IP address of agent              Subnet Number
     --------------------             -------------
     */
    /* {"90.11.42.254",               "90.11.42.0"}, */
    };
```

Adding Entries to the Database of a Running DHCP Server

After the server has started, use the following routine to add new entries to the lease database:

```
STATUS dhcpsLeaseEntryAdd
    (
    char *      pName,        /* Name of lease entry */
    char *      pStartIp,     /* First IP address to assign */
    char *      pEndIp,       /* Last IP address in assignment range */
    char *      pParams       /* Formatted string of lease parameters */
    )
```

As input, **dhcpsLeaseEntryAdd( )** expects to receive an entry name, starting and ending IP addresses for assignment to clients, and a formatted string containing lease parameters. If the entry is added successfully, the routine returns **OK** or **ERROR** otherwise. This routine allows the expansion of the address pool without rebuilding the image whenever new entries are needed. If you provide an appropriate storage hook, these entries are preserved across server restarts.

Storing and Retrieving Active Network Configurations

To store and retrieve network configuration information, you need to implement an address storage routine and a lease storage routine. The lease storage routine uses the prototype:

```
STATUS dhcpsLeaseStorageHook
    (
    int    op,            /* requested storage operation */
    char * pBuffer,       /* memory location for record of active lease */
    int    dataLen        /* amount of lease record data */
    )
```

⚠ **CAUTION:** Not providing the storage routine could cause DHCP to fail.

Your lease storage routine must store and retrieve active network configurations. To install the routine you created, set **DHCPS_LEASE_HOOK** (a configuration parameter) to a string containing the routine name. The address storage routine uses the following prototype:

```
STATUS dhcpsAddressStorageHook
    (
    int         op,             /* requested storage operation */
    char *      pName,          /* name of address pool entry */
    char *      pStartIp,       /* first IP address in range */
    char *      pEndIp,         /* last IP address in range */
    char *      pParams         /* lease parameters for each address */
    )
```

Your address storage routine (optional) stores and retrieves additional address-pool entries created using **dhcpsLeaseEntryAdd( )**. To preserve these entries, set **DHCPS_ADDRESS_HOOK** (a configuration parameter) to the name of your storage routine. If you don't do this, any active leases using alternate entries are not renewed when the server is restarted.

The **op** parameters of both storage routines expect one of the following values:[2]

DHCPS_STORAGE_START
Tells your storage routine to perform any necessary initialization. Your storage routine should "reset" and thus prepare to return or replace any previously stored data.

[2] These symbolic constants are defined in **dhcpsLib.h**.

DHCPS_STORAGE_STOP
Tells your storage routine to perform any necessary cleanup. After a stop, the storage routine should not perform any reads or writes until after the next start.

DHCPS_STORAGE_WRITE
Tells the routine to store network configurations. Each write must store the data to some form of permanent storage.

The write functionality of your lease storage routine is critical. It is required to preserve the integrity of the protocol and prevent assignment of IP addresses to multiple clients. If the server is unable to store and retrieve the active network configurations, the results are unpredictable. The write functionality of the lease storage routine must accept a sequence of bytes of the indicated length.

The write functionality of the address storage routine must accept **NULL**-terminated strings containing the entry name, starting and ending addresses, and additional parameters.

If a write completes successfully, the routine must return **OK**.

DHCPS_STORAGE_READ
Tells your storage routine to retrieve network configurations. Each read must copy the data (stored by earlier writes) into the buffers provided. The returned information must be of the same format provided to the write operation.

If a read completes successfully, your routine must return **OK**. If earlier reads have retrieved all available data, or no data is available, your routine must return **ERROR**. The server calls your routine with read requests until **ERROR** is returned.

DHCPS_STORAGE_CLEAR
Used only in calls to your lease storage routine. This value tells your routine that any data currently stored is no longer needed. Following this operation, reads should return error until after the next write.

NOTE: Under VxWorks AE, all code you want to include in a target requires a component description file. This is true also for the code you write for your storage hooks. A **.cdf** file for an address storage hook would look as follows:

```
/* 00comp_dhcpstest.cdf - Component configuration file */
/* Copyright 1984 - 2000 Wind River Systems, Inc. */
/*
modification history
--------------------
01a,18jul00,spm  written
*/

Component INCLUDE_DHCPS_STORE
        {
        NAME            DHCP dynamic address pool hook
        SYNOPSIS        Provides permanent storage for potential DHCP leases
        MODULES         dhcpsTestHook.o
        PREF_DOMAIN     KERNEL
        }

Module dhcpsTestHook.o {
        ENTRY_POINTS    sampleAddressStorageHook
}

EntryPoint sampleAddressStorageHook {
        SYNOPSIS        Dummy routine for displaying storage operations
}
```

Configuring the Unsupported DHCP Server

The files in **target/unsupported/dhcp-1.3beta/server** contain a port of a public domain server available from the WIDE project. This port modifies the original code so that it supports Solaris as well as SunOS.

Unlike the supported VxWorks DHCP server, the unsupported server uses files to store the databases that track the IP addresses and the other configuration parameters that it distributes.

You can specify the names of these files in the **dhcps** command that you use to start the DHCP server. If you do not specify the configuration files by name, the server uses the following defaults: **/etc/dhcpdb.pool**, and **/etc/dhcpdb.bind** (or **/var/db/dhcpdb.bind** for BSD/OS). If the server supports a relay agent, it also maintains an extra database with the default name of **/etc/dhcpdb.relay**. The server also creates other files as needed in the **/etc** directory, but you do not need to edit these files to configure the server.

For the specifics of how you should edit these files, see the **DHCPS(5)**, **DHCPDB.POOL(5)**, and **DHCPDB.RELAY(5)** man pages included with the source code for the unsupported DHCP server.

5.3.4 Configuring the Supported DHCP Relay Agent

To include the VxWorks DHCP relay agent in an image, use the **INCLUDE_DHCPR** configuration parameter. The relay agent uses some of the same configuration parameters as the DHCP server:

DHCP_MAX_HOPS — DHCP Server/Relay Agent Network Radius
Default: 4. Hops before discard, up to 16.

DHCPS_SPORT—DHCP Server/Relay Agent Host Port
Default: 67. Port monitored by DHCP servers.

DHCPS_CPORT—DHCP Server/Relay Agent Target Port
Default: 68. Port monitored by DHCP clients.

DHCPS_MAX_MSGSIZE—DHCP Server/Relay Agent Maximum Message Size
Maximum size (in bytes) for a DHCP message. Default: 590.

To find other DHCP relay agents or servers, the relay agent reads the **dhcpTargetTbl[ ]** table defined in **usrNetDhcprCfg.c**. This table is of the form:

```
DHCP_TARGET_DESC dhcpTargetTbl [] =
    {
    /*
     IP address of DHCP target servers
     --------------------------------
     */
     /* {"90.11.42.2"}, */
     };
```

Each entry in the table must specify a valid IP address for a DHCP server on a different subnet than the relay agent. The relay agent transmits a copy of all DHCP messages sent by clients to each of the specified addresses. The agent does *not* set the IP routing tables so that the specified target addresses are reachable.

The relay agent forwards DHCP client messages through only a limited number of targets: the DHCP Server/Relay Agent Network Radius. When the configured value is exceeded, the message is silently discarded. This value is increased only when a DHCP agent forwards the message. It is completely independent of the similar value used by IP routers. *RFC 1542* specifies the maximum value of 16 for this constant. The default hops value is four.

Beyond providing the list of target addresses, and optionally changing the maximum number of hops permitted, no further action is necessary. The DHCP relay agent executes automatically whenever it is included in the image.

5.3.5 DHCP within an Application

The target-resident DHCP client can retrieve multiple sets of configuration information. These retrieval requests can execute either synchronously or asynchronously. In addition, the retrieved network configuration information can be applied directly to the underlying network interface or used for some other purpose. The following example demonstrates the asynchronous execution of a DHCP request for a lease with a 30-minute duration in which the retrieved configuration parameters are applied to the network interface used to contact the DHCP server.[3]

```
pIf = ifunit ("net0");     /* Access network device. */

/* Initialize lease variables for automatic configuration. */

pLeaseCookie = dhcpcInit (pIf, TRUE);
if (pLeaseCookie == NULL)
    return (ERROR);

/* Set any lease options here. */

dhcpcOptionAdd (pLeaseCookie, _DHCP_LEASE_TIME_TAG, 4, 1800);

result = dhcpcBind (pLeaseCookie, FALSE);  /* Asynchronous execution. */
if (result != OK)
    return (ERROR);
```

In the code above, the **dhcpcInit( )** call used a value of **TRUE** for the **autoconfig** parameter. This automatically includes a request for a subnet mask and broadcast address in the cookie (**pLeaseCookie**). To request additional options for this lease the code makes a call to **dhcpcOptionSet( )**. Because the DHCP protocol requires that all requested parameters be specified before a lease is established, both the **dhcpcOptionSet( )** and **dhcpcOptionAdd( )** calls must precede the asynchronous **dhcpcBind( )** call that establishes the lease.

Although it is omitted from the example, you can use a **dhcpcLeaseHookAdd( )** call to associate a lease event hook routine with this lease. That way, you can note

[3] The limit on the number of concurrent leases is the "DHCP Client Maximum Leases" value set during configuration (Configuration parameter: **DHCPC_MAX_LEASES**). When setting this value, remember to count the lease (if any) that the client retrieved at boot time.

the **DHCPC_LEASE_NEW** event that occurs when the asynchronous **dhcpcBind( )** completes its negotiations with the DHCP server.

To query the local DHCP client for a parameter value from the lease information it has retrieved, call **dhcpcOptionGet( )**. This routine checks whether the lease associated with a particular lease cookie is valid and whether the server provided the requested parameter. If so, **dhcpcOptionGet( )** copies the parameter value into a buffer. Otherwise, it returns **ERROR**. A call to **dhcpcOptionGet( )** generates no network traffic; it queries the local DHCP client for the information it needs. The following sample demonstrates the use of this routine:

```
inet_addr webServer;
STATUS result;
int lenght=4;
...
result = dhcpcOptionGet (pLeaseCookie, _DHCP_DFLT_WWW_SERVER_TAG,
                        &length, &webServer);
if (result == OK)
    printf("Primary web server: %s", inet_ntoa (webServer));
...
```

NOTE: To check on configuration parameters associated with a lease established at boot time, use the **pDhcpcBootCookie** global variable as the lease cookie in a call to **dhcpcOptionGet( )**.

In addition to **dhcpcOptionGet( )**, you can use **dhcpcParamsGet( )** to retrieve multiple lease parameter values simultaneously. The DHCP client library also provides other routines that you can use to get the values of particular parameters (such as the lease timers) without supplying their option tags.

For more information on DHCP client features, see the **dhcpcLib** manual pages.

NOTE: If you already have an IP address and do not want another but want to query the server for any other information it has for you, call **dhcpInformGet( )**.

5.4 Boot Parameters for DHCP, BOOTP, and Network Initialization

Before the boot program can use a DHCP or BOOTP client to retrieve additional boot parameters from a remote server, the boot program needs appropriate values for **bootDev**, **unitNum**, **procNum**, and **flags**. See Table 5-1. Because the boot program does not yet have network access, the target must be able to find these parameter values in the default boot line, a user-provided boot line, or NVRAM boot line.[4]

Table 5-1 **Boot Parameters Needed for DHCP, BOOTP, and Network Device Initialization**

Parameter Name from BOOT_PARAMS	Parameter Name from p Command Listing (followed by description)
bootDev	**boot device** Contains the name of the network device from which to boot. For example, **ln** specifies the Lance driver. Which device you specify determines the physical medium over which the boot program attempts a networked boot. To add support for another medium, write a MUX-based driver for the new network and include the driver in your boot program. For more information on writing a driver that uses the MUX interface, see the *10. Integrating a New Network Interface Driver*.
unitNum	**unit number** Contains the unit number for the network device. In boot prompts that reference the network device, the target appends this to the **bootDev**. For example, if you see an "ln0", the "ln" refers to the Lance driver, and the "0" is the network device unit number. If you do not specify a unit number, the boot program defaults to using 0.
procNum	**processor number** Contains the backplane processor number of the target CPU. This value is critical to the shared-memory network. The shared memory master must be processor number zero.
flags	**flags (f)** Contains a value composed of flags (**OR**ed in values) that configure the boot process. The predefined significance of each bit is as follows: **0x01** Disables system controller for processor 0 (not supported on all boards).

[4] If the target has NVRAM, and the user specified these parameters in a previous boot session, the boot program knows to save these parameters to an NVRAM boot line for the use of the next boot session.

Table 5-1 **Boot Parameters Needed for DHCP, BOOTP, and Network Device Initialization** *(Continued)*

Parameter Name from BOOT_PARAMS	**Parameter Name from p Command Listing (followed by description)**
	0x02 Loads the local symbols as well as the global symbols into the target-based symbol table. This has consequences for tools such as the target shell. If the target-based symbol contains local variables, the target shell has access to both locally and globally declared symbols. Setting this bit means you must also reconfigure VxWorks to include a downloaded symbol table. The relevant configuration parameter is **INCLUDE_NET_SYM_TBL**. The VxWorks startup code assumes that the file containing the symbol table is resident on the same host as the boot image. The VxWorks startup code also assumes that the name of the symbol table file is the boot file name with an appended **.sym** suffix. When reading the **.sym** file, the VxWorks image has the option of loading local symbols as well as global symbols into its target-resident symbol table.
	0x04 Prevents autoboot.
	0x08 Enables quick autoboot (no countdown).
	0x20 Disables login security.
	0x40 Specifies automatic configuration using BOOTP or DHCP. VxWorks tries first to use a DHCP client. If the boot ROM does not include the DHCP client, then the target uses the BOOTP client to retrieve information. When the 0x40 flag is set, **e** or **ead** (**inet on ethernet**) should be blank.
	0x80 Tells the target to use TFTP to get VxWorks image. Otherwise, the target uses either RSH or FTP. The target uses FTP if you enter a non-empty value for the **passwd** parameter. Otherwise, the target uses RSH.
	0x100 Makes target register as a Proxy ARP client.

5.4.1 Boot Parameters Returned from DHCP or BOOTP

If the 0x40 bit in the **flags** parameter is set, the boot program uses either DHCP or BOOTP client to retrieve the following parameters: **ead** (from which the boot program also derives a value for **bad**), **had**, **gad**, and **bootFile**.[5] See Table 5-2.

Table 5-2 **Boot Parameters Returned from DHCP or BOOTP**

Parameter Name from BOOT_PARAMS	Parameter Name from p Command Listing (followed by description)
ead	**inet on ethernet (e)** This value is the unique Internet address of this target on the Ethernet or, if you are booting from SLIP, the local end of a SLIP connection. You can also specify a subnet mask (as described in *4.5.2 Assigning an IP Address and Network Mask to an Interface*, p.61). If no mask is specified, the standard mask for the address class will be used. If **ead** is empty, the target does not attach the Ethernet interface. This is acceptable if the target is booting over the backplane.
bad	**inet on backplane (b)** Actually, neither BOOTP nor DHCP supply this value directly, the backplane Internet address. If this parameter contains a non-empty value, the target attaches the backplane interface. Typically, the boot program uses sequential and proxy default addressing conventions to derive a **bad** value from the **ead** parameter (which BOOTP can provide) and the CPU number. However, the use of sequential addressing makes booting from the shared-memory backplane incompatible with DHCP. This parameter should be empty if no shared-memory network is required. To specify a subnet mask for **bad**, see *4.5.2 Assigning an IP Address and Network Mask to an Interface*, p.61).
had	**host inet (h)** The Internet address of the host from which to retrieve the boot file.
gad	**gateway inet (g)** The Internet address of the gateway through which to boot if the host is not on the same network as the target. If **gad** has a non-empty value, a routing entry is added indicating that the address is a gateway to the network of the specified boot host. NOTE: do *not* use this field to enter the default gateway (the router's address) if the host and the target are on the same subnet. Instead, use **routeAdd( )** in your application startup code or startup script.

[5] If you accidentally include both a DHCP and BOOTP client in a boot program, the program uses the DHCP client. If neither is present and 0x40 is set, booting fails.

Table 5-2 **Boot Parameters Returned from DHCP or BOOTP** *(Continued)*

Parameter Name from BOOT_PARAMS	Parameter Name from p Command Listing (followed by description)
bootFile	**file name** The full path name of the file containing the VxWorks run-time image.

WARNING: If you decide to change the subnet mask for the target's address (**ead**), you must be careful to call the appropriate functions in the correct order:

1. **ifRouteDelete( )** – to delete the existing network route for the device
2. **ifMaskSet( )** – to enter a new mask
3. **ifAddrSet( )** – to set the IP address, whereupon the new network route is created

Failing to call these functions in the correct order results in a misconfigured route table.

5.5 SNMP, Simple Network Management Protocol

The Simple Network Management Protocol (SNMP) lets you use a Network Management Station (NMS) to remotely configure, monitor, and manage any networked device that runs an SNMP agent. The SNMP protocol is based on an exchange of messages that set or get the value of variables in the agent's Management Information Base (MIB). Standard MIBs exist for well known protocols and network devices. By adding variables to an agent's MIB, you can accommodate new network devices and protocols as necessary.

SNMP is a Separately Purchasable Option

The VxWorks stack does not bundle in SNMP support. If you need SNMP, purchase one of the following options:

- **Envoy SNMP Source Code**

  This option provides a complete SNMP v1/v2c solution. You have the option of enhancing Envoy with Envoy+3 and Envoy+X.

- **Envoy+3**

  This option provides SNMP v3 functionality. SNMP v3 enhances the security of the management protocol through the addition of encryption and authentication. SNMPv3 uses industry-standard protocols such as MD5 (Message Digest 5), SHA (Secure Hash Algorithm), and DES (Data Encryption Standard) to secure the communications channel. Envoy's SNMP v3 implementation also provides support for the Target and Notify MIBs.

- **Envoy+X**

  This option provides AgentX functionality for implementing SNMP master agent/subagent systems. AgentX lets users create dynamically extensible SNMP agents. AgentX master agents communicate directly with network management stations. Subagents dynamically register with a master agent. Such functionality allows developers to design a modularly extensible system, such as a multi-blade switching chassis, that can be managed as a single device.

- **WindNet SNMP**

  This option provides a binary version of the Envoy portable source code product.

For more information, see the *Envoy Programmer's Guide* and the *Envoy Porting Guide*.

6
Dynamic Routing Protocols

6.1 Introduction

When a networking utility needs routing information, it searches the system's routing table. You can set up and manage this table manually (from the command line). However, if the network environment is constantly in flux, the information in a static routing table could quickly become obsolete. To update the routing table dynamically, VxWorks supports RIP (Routing Information Protocol).

RIP comes bundled with VxWorks and is intended for small to medium-sized networks. RIP is a *distance-vector protocol*, which means that it contains a vector of distances (a hop count). Each router uses these distance-vectors to update its routing tables.

6.2 RIP, Routing Information Protocol

RIP maintains routing information within small internetworks. You can use RIP only in networks where the largest number of hops is 15. Although 15 hops can encompass a very large network, many networks already exceed this limit.[1]

[1] A packet takes a *hop* every time it crosses a subnet. If a packet leaves machine Q and must pass through two subnet routers before it reaches its destination on machine N, the number of hops is two.

RIP is based on work done in the Internet community, and its algorithmic base goes back to the ARPANET circa 1969. It is based on the distance-vector algorithm, also called Bellman-Ford, which is described in "Dynamic Programming," from Princeton University by R. E. Bellman. This paper was published in 1957.

The RIP server provided with VxWorks is based on the BSD 4.4 **routed** program. There are several relevant RFCs; the two most important are RFC 1058, in which RIP version 1 was first documented, and RFC 1388, in which the version 2 extensions are documented.

The VxWorks RIP server supports three modes of operation:

- Version 1 RIP

  This mode of operation follows RFC 1058. It uses subnet broadcasting to communicate with other routers and sends out only a gateway and metric for each subnet.

- Version 2 RIP with Broadcasting

  This mode is the same as Version 2 RIP with multicasting (see below), except that it uses broadcasting instead of multicasting. This mode is backward compatible with RIP Version 1 and is the mode recommended in RFC 1388.

- Version 2 RIP with Multicasting

  In this mode, the server not only knows about routers but can also describe routes based on their subnet mask and can designate a gateway that is not the router that sends the updates. Thus, the machine that hosts the RIP server does not necessarily have to be the gateway. Because this mode uses multicasting to communicate, only interested nodes in the network see routing information and updates.

6.2.1 VxWorks Debugging Routines for RIP

The RIP server provides several routines that make debugging easier. The most often used is **ripLogLevelBump( )**, which enables tracing of packets and routing changes. Keep in mind that bumping the log level several times prints large amounts of data to the console. Another routine is **ripRouteShow( )**, which prints the router's internal tables to the console. The printed message provides the following information:

- the route being advertised
- the router that routes the packets
- a subnet mask

- the time out on the route (in seconds)[2]
- the flags value (see Table 6-1)

Table 6-1 **Flag Constants for ripRouteShow( )**

Constant	Meaning
RTS_CHANGED	Route has changed recently (within the last 30 seconds).
RTS_EXTERNAL	Route should not propagate to other routers.
RTS_INTERNAL	Route is internal, used to implement border gateway filtering.
RTS_PASSIVE	Route is on a passive interface (loopback).
RTS_INTERFACE	Route is on a directly connected interface.
RTS_REMOTE	Route is on a point to point link.
RTS_SUBNET	Route is to a subnet (not a host).
RTS_OTHER	Route belongs to some other (non-RIP) protocol.
RTS_PRIMARY	Route is a primary route. If this flag is set, the **RTS_OTHER** flag must also be set.

RIP periodically pushes routing information into the VxWorks routing table. Between updates, the two tables can diverge, but updating only periodically avoids route thrashing (pushing transient routes into the system route table but then removing them immediately).

6.2.2 Configuring RIP

To include the RIP server, reconfigure the image. The relevant configuration parameter is **INCLUDE_RIP**.

NOTE: If you exclude RIP, but include SNMP, a separately purchasable option, you might want to edit **snmpMib2.mib** to exclude RIP MIB objects. Including these objects does no harm, but it makes the image larger unnecessarily.

[2] The time out is the length of time for which the route remains current. If a route is not updated within 3 minutes, it is flushed from the routing table.

Compile-Time Configuration

The RIP server starts up when the network initialization code calls **ripLibInit( )**. This routine takes several parameters. You set the value of these parameters by adjusting the following configuration parameters:

BSD 4.3 Compatible Sockets — **BSD43_COMPATIBLE**
Although **BSD43_COMPATIBLE** is not a RIP-specific configuration parameter, you must turn it off if you want to use VxWorks RIP. By default, this parameter is already set. **BSD43_COMPATIBLE** is also automatically defined if VxWorks is configured to use sockets, **INCLUDE_BSD_SOCKET**.

RIP Supplier Flag — **RIP_SUPPLIER**, default: 0
Set to 1, **RIP_SUPPLIER** tells the RIP server to send out routing information and updates no matter how many physical interfaces are attached to it. Setting this constant to 0 turns off this feature.

RIP Gateway Flag — **RIP_GATEWAY**, default: 0
Set to 1, **RIP_GATEWAY** tells the server that the router is a default gateway to all hosts external to the routing domain. If this is not the case, set this constant to 0.

WARNING: Do not set **RIP_GATEWAY** to 1 unless this really is the general gateway. Setting this to 1 configures the RIP server to ignore any advertisements for default routers received from other RIP peers.

RIP Multicast Flag — **RIP_MULTICAST**, default: 0
Set to 1, **RIP_MULTICAST** tells the server to use the RIP multicast address (224.0.0.9) instead of using broadcasts. This mode lowers the load on the network generated by the routing updates. Unfortunately, not all RIP server implementations (for example, BSD and SunOS **routed**) can handle multicasting.

RIP Version Number — **RIP_VERSION**, default: 1
Set to 1, **RIP_VERSION** tells the server to run just as a version 1 RIP router (as described in RFC 1058). Such a server ignores all version 2 packets as well as malformed version 1 packets. Set this constant to 2 to tell the server that it should send out version 2 packets and that it should listen for and process both version 1 and version 2 packets. If you set this constant to 2 and set the RIP Multicast Flag, **RIP_MULTICAST**, to 1, you put the server in full version-2 mode.

RIP Timer Rate — **RIP_TIMER_RATE**, default: 1 second
RIP_TIMER_RATE tells RIP how often it should examine the routing table for changes and expired routes.

RIP Supply Interval — **RIP_SUPPLY_INTERVAL**, default: 30 seconds
RIP_SUPPLY_INTERVAL tells RIP how frequently it should transmit route updates over every known interface. This value must be set to a multiple of the RIP Timer Rate.

RIP Expire Time — **RIP_EXPIRE_TIME**, default: 180 seconds
RIP_EXPIRE_TIME tells RIP the maximum time between updates before a route is invalidated and removed from the kernel table.

RIP Garbage Time — **RIP_GARBAGE_TIME**, default: 120 seconds
RIP_GARBAGE_TIME tells RIP specifies the amount of time to wait before a route removed from the kernel table is also deleted from the internal routing table. This wait time does not apply until after the **RIP_EXPIRE_TIME** expires. Thus, when a **RIP_GARBAGE_TIME** applies, the total wait time is **RIP_GARBAGE_TIME** plus **RIP_EXPIRE_TIME**. By default, that delay would be 300 seconds (120 plus 180).

RIP Authentication Type — **RIP_AUTH_TYPE**, default: 1 (no authentication)
RIP_AUTH_TYPE tells RIP which authentication method (if any) that it should use. Valid values for **RIP_AUTH_TYPE** are:

1 — no authentication
2 — simple password authentication
3 — MDS authentication

⚠ **CAUTION:** The RIP server does not support separate routing domains. Only routing domain 0, the default, is supported.

Run-Time Configuration

In addition to setting the defines shown above, there are two alternate methods you can use to configure RIP:

- Use the **m2Rip** routines to configure RIP. These routines are documented in the reference entries. The parameters to these routines are also described in RFC-1389.

- Use an SNMP agent to configure RIP.

RIP Task Priority and User Protection Domains

RIP tasks run at a priority of 100 and 101. If you are using your own protection domain, make sure that the task priority range for the protection domain spans the above two values.

6.2.3 Creating an Interface Exclusion List for RIP

By default, RIP runs on all interfaces active when RIP is initialized. If you do not want to run RIP on a particular interface, you can name the interface on a RIP exclusion list. However, you must put the interface on the exclusion list before RIP is initialized. If that is not possible, you can add the interface to the list and then call **ripIfReset( )**.

NOTE: Calling **ripIfReset( )** clears the RIP interface settings for the interface. This means you lose the interface hooks, the interface MIB2 settings, and all other such information associated with the interface.

If RIP is already running on an interface, simply putting an interface on the exclusion list does not automatically shut down RIP on that interface.

To manage an interface exclusion list, **ripLib** provides the following functions:

ripIfExcludeListAdd( ) — add an interface to the RIP exclusion list

ripIfExcludeListDelete( ) — remove an interface from the RIP exclusion list

ripIfExcludeListShow( ) — show the interfaces on the RIP exclusion list

For more information on these functions, see the relevant **ripLib** reference entries.

7
Sockets under VxWorks

7.1 Introduction

This chapter describes how to use the standard BSD socket interface for stream sockets and datagram sockets on a VxWorks target. It also describes how to use zbuf sockets, an alternative set of socket calls based on a data abstraction called the *zbuf*. These zbuf calls let you share data buffers (or portions of data buffers) between separate software modules.

Using sockets, processes can communicate within a single CPU, across a backplane, across an Ethernet, or across any connected combination of networks. Socket communications can occur between VxWorks tasks and host system processes in any combination. In all cases, the communications appear identical to the application—except, of course, for the speed of the communications.

One of the biggest advantages of socket communication is that it is a homogeneous mechanism: socket communications among processes are the same, regardless of the location of the processes in the network or the operating system where they run. This is true even if you use zbuf sockets, which are fully interoperable with standard BSD sockets.

NOTE: This chapter focuses on how to use a socket connection between processes. If you are interested in learning how to add socket-support code to a new network service or protocol, see *11.4 Adding a Socket Interface to Your Service*, p.232.

For additional information on the socket interface, see the **sockLib** reference entry.

7.2 BSD Sockets

A socket is a communications end-point that is *bound* to a UDP or TCP port within the node. Under VxWorks, your application can use the *sockets* interface to access features of the Internet Protocol suite (features such as multicasting). Depending on the bound port type, a socket is referred to either as a stream socket or a datagram socket. VxWorks sockets are UNIX BSD 4.4 compatible. However, VxWorks does not support signal functionality for VxWorks sockets.

S*tream sockets* use TCP to bind to a particular port number. Another process, on any host in the network, can then create another stream socket and request that it be connected to the first socket by specifying its host Internet address and port number. After the two TCP sockets are connected, there is a *virtual circuit* set up between them, allowing reliable socket-to-socket communications. This style of communication is conversational.

*Datagram socket*s use UDP to bind to a particular port number. Other processes, on any host in the network, can then send messages to that socket by specifying the host Internet address and the port number. Compared to TCP, UDP provides a simpler but less robust communication method. In a UDP communication, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams*. There is no sense of conversation with a datagram socket. The communication is in the style of a letter. Each packet carries the address of both the destination and the sender. Compared to TCP, UDP is unreliable. Like the mail, packets that are lost or out-of-sequence are not reported.

NOTE: The complexities of socket programming are beyond the scope of this document. For additional information, consult a socket-programming book, such as those mentioned in the introduction to this manual.

7.2.1 VxWorks-Specific Socket Dependencies

Although the socket interface is compatible with VxWorks, the environment does affect how you use sockets. Specifically, the globally accessible file descriptors available in the task-independent address space of VxWorks require that you take extra precautions when closing a file descriptor.

You must make sure that one task does not close the file descriptor on which another task is pending during an **accept( )**. Although the **accept( )** on the closed file descriptor sometimes returns with an error, the **accept( )** can also fail to return at all. Thus, if you need to be able to close a socket connection's file descriptor asynchronously, you may need to set up a semaphore-based locking mechanism that prevents the close while an **accept( )** is pending on the file descriptor.

7.2.2 Datagram Sockets (UDP)

You can use datagram (UDP) sockets to implement a simple client-server communication system. You can also use UDP sockets to handle multicasting.

Using a Datagram Socket to Implement a Client-Server Communication System

The following code example uses a client-server communication model. The server communicates with clients using datagram-oriented (UDP) sockets. The main server loop, in **udpServer( )**, reads requests and optionally displays the client's message. The client builds the request by prompting the user for input. Note that this code assumes that it executes on machines that have the same data sizes and alignment.

Example 7-1 **Datagram Sockets (UDP)**

```
/* udpExample.h - header used by both UDP server and client examples */

#define SERVER_PORT_NUM         5002    /* server's port number for bind() */
#define REQUEST_MSG_SIZE        1024    /* max size of request message */

/* structure used for client's request */

struct request
    {
    int  display;                       /* TRUE = display message */
    char message[REQUEST_MSG_SIZE];     /* message buffer */
    };
```

```
/* udpClient.c - UDP client example */

/*
DESCRIPTION
This file contains the client-side of the vxWorks UDP example code.
The example code demonstrates the usage of several BSD 4.4-style
socket routine calls.
*/


/* includes */
```

```
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "udpExample.h"

/****************************************************************************
*
* udpClient - send a message to a server over a UDP socket
*
* This routine sends a user-provided message to a server over a UDP socket.
* Optionally, this routine can request that the server display the message.
* This routine may be invoked as follows:
*       -> udpClient "remoteSystem"
*       Message to send:
*       Greetings from UDP client
*       Would you like server to display your message (Y or N):
*       y
*       value = 0 = 0x0
*
* RETURNS: OK, or ERROR if the message could not be sent to the server.
*/

STATUS udpClient
    (
    char *              serverName     /* name or IP address of server */
    )
    {
    struct request      myRequest;     /* request to send to server */
    struct sockaddr_in  serverAddr;    /* server's socket address */
    char                display;       /* if TRUE, server prints message */
    int                 sockAddrSize;  /* size of socket address structure */
    int                 sFd;           /* socket file descriptor */
    int                 mlen;          /* length of message */

    /* create client's socket */

    if ((sFd = socket (AF_INET, SOCK_DGRAM, 0)) == ERROR)
        {
        perror ("socket");
        return (ERROR);
        }

    /* explicit bind not required - local port number is dynamic */

    /* build server socket address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_len = (u_char) sockAddrSize;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons (SERVER_PORT_NUM);
    if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
        ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
        {
        perror ("unknown server name");
        close (sFd);
        return (ERROR);
        }

    /* build request, prompting user for message */

    printf ("Message to send: \n");
    mlen = read (STD_IN, myRequest.message, REQUEST_MSG_SIZE);
    myRequest.message[mlen - 1] = '\0';

    printf ("Would you like the server to display your message (Y or N): \n");
    read (STD_IN, &display, 1);
    switch (display)
        {
        case 'y':
        case 'Y': myRequest.display = TRUE;
            break;
        default: myRequest.display = FALSE;
            break;
        }

    /* send request to server */

    if (sendto (sFd, (caddr_t) &myRequest, sizeof (myRequest), 0,
        (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
        {
        perror ("sendto");
        close (sFd);
        return (ERROR);
        }

    close (sFd);
    return (OK);
    }
```

```
/* udpServer.c - UDP server example */

/*
DESCRIPTION
This file contains the server-side of the vxWorks UDP example code.
The example code demonstrates the usage of several BSD 4.4-style
socket routine calls.
*/

/* includes */
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
```

```
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "udpExample.h"


/*********************************************************************
*
* udpServer - read from UDP socket and display client's message if requested
*
* Example of vxWorks UDP server:
*       -> sp udpServer
*       task spawned: id = 0x3a1f6c, name = t2
*       value = 3809132 = 0x3a1f6c
*       -> MESSAGE FROM CLIENT (Internet Address 150.12.0.11, port 1028):
*       Greetings from UDP client
*
* RETURNS: Never, or ERROR if a resources could not be allocated.
*/

STATUS udpServer (void)
    {
    struct sockaddr_in  serverAddr;    /* server's socket address */
    struct sockaddr_in  clientAddr;    /* client's socket address */
    struct request      clientRequest; /* request/Message from client */
    int                 sockAddrSize;  /* size of socket address structure */
    int                 sFd;           /* socket file descriptor */
    char                inetAddr[INET_ADDR_LEN];
                                       /* buffer for client's inet addr */

    /* set up the local address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_len = (u_char) sockAddrSize;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons (SERVER_PORT_NUM);
    serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

    /* create a UDP-based socket */

    if ((sFd = socket (AF_INET, SOCK_DGRAM, 0)) == ERROR)
        {
        perror ("socket");
        return (ERROR);
        }

    /* bind socket to local address */

    if (bind (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
        {
        perror ("bind");
        close (sFd);
        return (ERROR);
        }
    /* read data from a socket and satisfy requests */

    FOREVER
        {
        if (recvfrom (sFd, (char *) &clientRequest, sizeof (clientRequest), 0,
            (struct sockaddr *) &clientAddr, &sockAddrSize) == ERROR)
            {
            perror ("recvfrom");
            close (sFd);
            return (ERROR);
            }

        /* if client requested that message be displayed, print it */

        if (clientRequest.display)
            {
            /* convert inet address to dot notation */

            inet_ntoa_b (clientAddr.sin_addr, inetAddr);
            printf ("MSG FROM CLIENT (Internet Address %s, port %d):\n%s\n",
                inetAddr, ntohs (clientAddr.sin_port),                     cl
ientRequest.message);
            }
        }
    }
```

Using a Datagram (UDP) Socket to Access IP Multicasting

Multicasting is the delivery of the same packets to multiple IP addresses. Typical multicasting applications include audio and video conferencing, resource discovery tools, and shared white boards. Multicasting is a feature of the IP layer, but to access this function, an application uses a UDP socket.

A VxWorks process must multicast on a network interface driver that supports multicasting (many do not). To review the capabilities of all attached network drivers, use **ifShow( )**. If a network interface supports multicasting, **IFF_MULTICAST** is listed among the flags for that network interface.

Multicast IP addresses range from 224.0.0.0 to 239.255.255.255. These addresses are also called class D addresses or multicast groups. A datagram with a class D destination address is delivered to every process that has joined the corresponding multicast group.

To multicast a packet, a VxWorks process need do nothing special. The process just sends to the appropriate multicast address. The process can use any normal UDP socket. To set the route to the destination multicast address, use **mRouteAdd( )**.

To receive a multicast packet, a VxWorks process must join a multicast group. To do this, the VxWorks process must set the appropriate socket options on the socket (see Table 7-1).

Table 7-1 **Multicasting Socket Options***

Command	Argument	Description
IP_MULTICAST_IF	**struct in_addr**	Select default interface for outgoing multicasts.
IP_MULTICAST_TTL	**char**	Select default time to live (TTL) for outgoing multicast packets.
IP_MULTICAST_LOOP	**char**	Enable or disable loopback of outgoing multicasts.
IP_ADD_MEMBERSHIP	**struct ip_mreq**	Join a multicast group.
IP_DROP_MEMBERSHIP	**struct ip_mreq**	Leave a multicast group.

* For more on multicasting socket options, see the **setsockopt( )** reference entry.

When choosing an address upon which to multicast, remember that certain addresses and address ranges are already registered to specific uses and protocols. For example, 244.0.0.1 multicasts to all systems on the local subnet. The Internet Assigned Numbers Authority (IANA) maintains a list of registered IP multicast groups. The current list can be found in RFC 1700. For more information about the IANA, see *RFC 1700*. Table 7-2 lists some of the well known multicast groups.

Table 7-2 **Well Known Multicast Groups**

Group	Constant	Description
224.0.0.0	**INADDR_UNSPEC_GROUP**	Reserved for protocols that implement IP unicast and multicast routing mechanisms. Datagrams sent to any of these groups are not forwarded beyond the local network by multicast routers.
224.0.0.1	**INADDR_ALLHOSTS_GROUP**	All systems on this subnet. This value is automatically added to all network drivers at initialization.
224.0.0.2		All routers on this subnet.
224.0.0.3		Unassigned.

Table 7-2 **Well Known Multicast Groups** *(Continued)*

Group	Constant	Description
224.0.0.4		DVMRP routers.
224.0.0.5		OSPF routers.
224.0.0.6		OSPF designated routers.
224.0.0.9		All RIP routers.
224.0.0.255	**INADDR_MAX_LOCAL_GROUP**	Unassigned.
224.0.1.1		NTP (Network Time Protocol).

The following code samples define two routines, **mcastSend( )** and **mcastRcv( )**. These routines demonstrate how to use UDP sockets for sending and receiving multicast traffic.

mcastSend( ) transmits a buffer to the specified multicast address. As input, this routine expects a multicast destination, a port number, a buffer pointer, and a buffer length. For example:

```
status = mcastSend ("224.1.0.1", 7777, bufPtr, 100);
```

mcastRcv( ) receives any packet sent to a specified multicast address. As input, this routine expects the interface address from which the packet came, a multicast address, a port number, and the number of bytes to read from the packet. The returned value of the function is a pointer a buffer containing the read bytes. For example:

```
buf = mcastRcv (ifAddress, "224.1.0.1", 7777, 100) ;
```

Example 7-2 **Datagram Sockets (UDP) and Multicasting**

```
/* includes */
#include "vxWorks.h"
#include "taskLib.h"
#include "socket.h"
#include "netinet/in.h"
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "sockLib.h"
#include "inetLib.h"
#include "ioLib.h"
```

```
#include "routeLib.h"

/* defines */
/* globals */
/* forward declarations */


STATUS mcastSend (char * mcastAddr, USHORT mcastPort, char * sendBuf,
                  int sendLen);
char * mcastRcv (char * ifAddr, char * mcastAddr, USHORT mcastPort,
                 int numRead);

/************************************************************************
* mcastSend - send a message to the multicast address
* This function sends a message to the multicast address
* The multicast group address to send, the port number, the pointer to the
* send buffer and the send buffer length are given as input parameters.
* RETURNS: OK if successful or ERROR
*/

STATUS mcastSend
    (
    char *      mcastAddr,      /* multicast address  */
    USHORT      mcastPort,      /* udp port number */
    char *      sendBuf,        /* send Buffer */
    int         sendLen         /* length of send buffer */
    )
    {
    struct sockaddr_in  sin;
    struct sockaddr_in  toAddr;
    int                 toAddrLen;
    int                 sockDesc;
    char *              bufPtr;
    int                 len;

    /* create a send and recv socket */

    if ((sockDesc = socket (AF_INET, SOCK_DGRAM, 0)) < 0 )
        {
        printf (" cannot open send socket\n");
        return (ERROR);
        }

    /* zero out the structures */
    bzero ((char *)&sin, sizeof (sin));
    bzero ((char *)&toAddr, sizeof (toAddr));

    sin.sin_len = (u_char) sizeof(sin);
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(0);

    if (bind(sockDesc, (struct sockaddr *)&sin, sizeof(sin)) != 0)
        {
        perror("bind");
        if (sockDesc) close (sockDesc);
        return (ERROR);
        }

    toAddrLen = sizeof(struct sockaddr_in);
    toAddr.sin_len      = (u_char) toAddrLen;
    toAddr.sin_family      = AF_INET;


    /* initialize the address to the send */
    toAddr.sin_addr.s_addr = inet_addr (mcastAddr);

    /* initialize the port to send */
    toAddr.sin_port        = htons(mcastPort);

    bufPtr = sendBuf;           /* initialize the buffer pointer */

    /* send the buffer */
    while (sendLen > 0)
        {
        if ((len = sendto (sockDesc, bufPtr, sendLen, 0,
                           (struct sockaddr *)&toAddr, toAddrLen)) < 0 )
            {
            printf("mcastSend sendto errno:0x%x\n", errno );
            break;
            }

        sendLen -= len;
        bufPtr += len;

        taskDelay (1);          /* give  a taskDelay */
        }

    close (sockDesc);

    return (OK);
    }

/************************************************************************
 * mcastRcv - receive a message from a  multicast address
 * This function receives a message from a  multicast address
 * The interface address from which to receive the multicast packet,
 * the multicast address to recv from, the port number and the number of
 * bytes to read are given as input parameters to this routine.
 * RETURNS: Pointer to the Buffer or NULL if error.
 */

char * mcastRcv
    (
    char *      ifAddr,         /* interface address to recv mcast packets */
    char *      mcastAddr,      /* multicast address  */
    USHORT      mcastPort,      /* udp port number to recv */
    int         numRead         /* number of bytes to read */
    )
    {
    struct sockaddr_in  fromAddr;
    struct sockaddr_in  sin;
```

```
int                 fromLen;
struct ip_mreq      ipMreq;
int                 recvLen;
int                 sockDesc;
char *              bufPtr;
int                 status  = OK;
char *              recvBuf = NULL;

if ((sockDesc = socket (AF_INET, SOCK_DGRAM, 0)) < 0)
    {
    printf (" cannot open recv socket\n");
    return (NULL);
    }

bzero ((char *)&sin, sizeof (sin));
bzero ((char *) &fromAddr, sizeof(fromAddr));
fromLen = sizeof(fromAddr);

if ((recvBuf = calloc (numRead, sizeof (char))) == NULL)
    {
    printf (" calloc error, cannot allocate memory\n");
    status = ERROR;
    goto cleanUp;
    }

sin.sin_len = (u_char) sizeof(sin);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;

/* UDP port number to match for the received packets */
sin.sin_port = htons (mcastPort);

/* bind a port number to the socket */
if (bind(sockDesc, (struct sockaddr *)&sin, sizeof(sin)) != 0)
    {
    perror("bind");
    status = ERROR;
    goto cleanUp;
    }

/* fill in the argument structure to join the multicast group */
/* initialize the multicast address to join */

ipMreq.imr_multiaddr.s_addr = inet_addr (mcastAddr);

/* unicast interface addr from which to receive the multicast packets */
ipMreq.imr_interface.s_addr = inet_addr (ifAddr);

/* set the socket option to join the MULTICAST group */
if (setsockopt (sockDesc, IPPROTO_IP, IP_ADD_MEMBERSHIP,
            (char *)&ipMreq,
            sizeof (ipMreq)) < 0)
    {
    printf ("setsockopt IP_ADD_MEMBERSHIP error:\n");
    status = ERROR;
    goto cleanUp;
```

```
        }

    /* get the data destined to the above multicast group */
    bufPtr = recvBuf;



    while (numRead > 0)
      {
      if ((recvLen = recvfrom (sockDesc, bufPtr, numRead, 0,
                              (struct sockaddr *)&fromAddr, &fromLen)) < 0)
          {
          perror("recvfrom");
          status = ERROR;
          break;
          }
      numRead -= recvLen;       /* decrement number of bytes to read */
      bufPtr  += recvLen;       /* increment the buffer pointer */
      }

    /* set the socket option to leave the MULTICAST group */
    if (setsockopt (sockDesc, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                    (char *)&ipMreq,
                    sizeof (ipMreq)) < 0)
        printf ("setsockopt IP_DROP_MEMBERSHIP error:\n");

    cleanUp:
        close (sockDesc);
    if ((status != OK) && (recvBuf != NULL))
        {
        free (recvBuf);
        recvBuf = NULL;
        }
     return (recvBuf);
    }
```

7.2.3 Stream Sockets (TCP)

The Transmission Control Protocol (TCP) provides reliable, two-way transmission of data. In a TCP communication, two sockets are *connected*, allowing a reliable byte-stream to flow between them in either direction. TCP is referred to as a *virtual circuit* protocol, because it behaves as though a circuit is created between the two sockets.

A good analogy for TCP communications is a telephone system. Connecting two sockets is similar to calling from one telephone to another. After the connection is established, you can write and read data (talk and listen).

Table 7-3 shows the steps in establishing socket communications with TCP, and the analogy of each step with telephone communications.

Table 7-3 **TCP Analogy to Telephone Communication**

Task 1 Waits	Task 2 Calls	Function	Analogy
socket( )	**socket( )**	Create sockets.	Hook up telephones.
bind( )		Assign address to socket.	Assign telephone number.
listen( )		Allow others to connect to socket.	Allow others to call.
	connect( )	Request connection to another socket.	Dial another telephone 's number.
accept( )		Complete connection between sockets.	Answer telephone and establish connection.
write( )	**write( )**	Send data to other socket.	Talk.
read( )	**read( )**	Receive data from other socket.	Listen.
close( )	**close( )**	Close sockets.	Hang up.

Example 7-3 **Stream Sockets (TCP)**

The following code example uses a client-server communication model. The server communicates with clients using stream-oriented (TCP) sockets. The main server loop, in **tcpServerWorkTask( )**, reads requests, prints the client's message to the console, and, if requested, sends a reply back to the client. The client builds the request by prompting for input. It sends a message to the server and, optionally, waits for a reply to be sent back. To simplify the example, we assume that the code is executed on machines that have the same data sizes and alignment.

```
/* tcpExample.h - header used by both TCP server and client examples */

/* defines */
#define SERVER_PORT_NUM         5001   /* server's port number for bind() */
#define SERVER_WORK_PRIORITY    100    /* priority of server's work task */
#define SERVER_STACK_SIZE       10000  /* stack size of server's work task */
#define SERVER_MAX_CONNECTIONS  4      /* max clients connected at a time */
#define REQUEST_MSG_SIZE        1024   /* max size of request message */
#define REPLY_MSG_SIZE          500    /* max size of reply message */

/* structure for requests from clients to server */
struct request
    {
    int reply;                         /* TRUE = request reply from server */

    int msgLen;                        /* length of message text */
    char message[REQUEST_MSG_SIZE];    /* message buffer */
    };
```

```
/* tcpClient.c - TCP client example */

/*
DESCRIPTION
This file contains the client-side of the VxWorks TCP example code.
The example code demonstrates the usage of several BSD 4.4-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "tcpExample.h"

/****************************************************************************
*
* tcpClient - send requests to server over a TCP socket
*
* This routine connects over a TCP socket to a server, and sends a
* user-provided message to the server.  Optionally, this routine
* waits for the server's reply message.
*
* This routine may be invoked as follows:
*       -> tcpClient "remoteSystem"
*       Message to send:
*       Hello out there
*       Would you like a reply (Y or N):
*       y
*       value = 0 = 0x0
*       -> MESSAGE FROM SERVER:
*       Server received your message
*
* RETURNS: OK, or ERROR if the message could not be sent to the server.
*/

STATUS tcpClient
    (
    char *              serverName     /* name or IP address of server */
    )
    {
    struct request      myRequest;     /* request to send to server */
    struct sockaddr_in  serverAddr;    /* server's socket address */
    char                replyBuf[REPLY_MSG_SIZE]; /* buffer for reply */
    char                reply;         /* if TRUE, expect reply back */
    int                 sockAddrSize;  /* size of socket address structure */
    int                 sFd;           /* socket file descriptor */
    int                 mlen;          /* length of message */
```

```
/* create client's socket */
if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
    {
    perror ("socket");
    return (ERROR);
    }

/* bind not required - port number is dynamic */
/* build server socket address */
sockAddrSize = sizeof (struct sockaddr_in);
bzero ((char *) &serverAddr, sockAddrSize);
serverAddr.sin_family = AF_INET;
serverAddr.sin_len = (u_char) sockAddrSize;
serverAddr.sin_port = htons (SERVER_PORT_NUM);

if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
    ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
    {
    perror ("unknown server name");
    close (sFd);
    return (ERROR);
    }

/* connect to server */
if (connect (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
    {
    perror ("connect");
    close (sFd);
    return (ERROR);
    }

/* build request, prompting user for message */
printf ("Message to send: \n");
mlen = read (STD_IN, myRequest.message, REQUEST_MSG_SIZE);
myRequest.msgLen = mlen;
myRequest.message[mlen - 1] = '\0';
printf ("Would you like a reply (Y or N): \n");
read (STD_IN, &reply, 1);
switch (reply)
    {
    case 'y':
    case 'Y': myRequest.reply = TRUE;
        break;
    default: myRequest.reply = FALSE;
        break;
    }

/* send request to server */


if (write (sFd, (char *) &myRequest, sizeof (myRequest)) == ERROR)
    {
    perror ("write");
    close (sFd);
    return (ERROR);
```

```
        }

    if (myRequest.reply)        /* if expecting reply, read and display it */

        {
        if (read (sFd, replyBuf, REPLY_MSG_SIZE) < 0)
            {
            perror ("read");
            close (sFd);
            return (ERROR);
            }

        printf ("MESSAGE FROM SERVER:\n%s\n", replyBuf);
        }

    close (sFd);
    return (OK);
    }
```

```
/* tcpServer.c - TCP server example */

/*
DESCRIPTION
This file contains the server-side of the VxWorks TCP example code.
The example code demonstrates the usage of several BSD 4.4-style
socket routine calls.
*/

/* includes */
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "taskLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "tcpExample.h"

/* function declarations */

VOID tcpServerWorkTask (int sFd, char * address, u_short port);

/****************************************************************************
*
* tcpServer - accept and process requests over a TCP socket
*
* This routine creates a TCP socket, and accepts connections over the socket
* from clients.  Each client connection is handled by spawning a separate
* task to handle client requests.
*
* This routine may be invoked as follows:
*       -> sp tcpServer
*       task spawned: id = 0x3a6f1c, name = t1
*       value = 3829532 = 0x3a6f1c
*       -> MESSAGE FROM CLIENT (Internet Address 150.12.0.10, port 1027):
*       Hello out there
*
* RETURNS: Never, or ERROR if a resources could not be allocated.
*/

STATUS tcpServer (void)
    {
    struct sockaddr_in  serverAddr;    /* server's socket address */
    struct sockaddr_in  clientAddr;    /* client's socket address */
    int                 sockAddrSize;  /* size of socket address structure */
    int                 sFd;           /* socket file descriptor */
    int                 newFd;         /* socket descriptor from accept */
    int                 ix = 0;        /* counter for work task names */
    char                workName[16];  /* name of work task */

    /* set up the local address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len = (u_char) sockAddrSize;
    serverAddr.sin_port = htons (SERVER_PORT_NUM);
    serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);

    /* create a TCP-based socket */

    if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
        {
        perror ("socket");
        return (ERROR);
        }

    /* bind socket to local address */

    if (bind (sFd, (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
        {
        perror ("bind");
        close (sFd);
        return (ERROR);
        }

    /* create queue for client connection requests */

    if (listen (sFd, SERVER_MAX_CONNECTIONS) == ERROR)
        {
        perror ("listen");
        close (sFd);
        return (ERROR);
        }
    /* accept new connect requests and spawn tasks to process them */

    FOREVER
        {
        if ((newFd = accept (sFd, (struct sockaddr *) &clientAddr,
            &sockAddrSize)) == ERROR)
            {
            perror ("accept");
            close (sFd);
            return (ERROR);
            }

        sprintf (workName, "tTcpWork%d", ix++);
        if (taskSpawn(workName, SERVER_WORK_PRIORITY, 0, SERVER_STACK_SIZE,
            (FUNCPTR) tcpServerWorkTask, newFd,
            (int) inet_ntoa (clientAddr.sin_addr), ntohs                (clien
tAddr.sin_port),
            0, 0, 0, 0, 0, 0, 0) == ERROR)
            {
            /* if taskSpawn fails, close fd and return to top of loop */

            perror ("taskSpawn");
            close (newFd);
            }
        }
    }

/*****************************************************************************
*
* tcpServerWorkTask - process client requests
*
* This routine reads from the server's socket, and processes client
* requests.  If the client requests a reply message, this routine
* will send a reply to the client.
*
* RETURNS: N/A.
*/

VOID tcpServerWorkTask
    (
    int                 sFd,            /* server's socket fd */
    char *              address,        /* client's socket address */
    u_short             port            /* client's socket port */
    )
    {
    struct request      clientRequest;  /* request/message from client */
    int                 nRead;          /* number of bytes read */
    static char         replyMsg[] = "Server received your message";

    /* read client request, display message */

    while ((nRead = fioRead (sFd, (char *) &clientRequest,
        sizeof (clientRequest))) > 0)
        {
        printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n%s\n",
                 address, port, clientRequest.message);
        free (address);                 /* free malloc from inet_ntoa() */

        if (clientRequest.reply)
            if (write (sFd, replyMsg, sizeof (replyMsg)) == ERROR)
                perror ("write");
        }

    if (nRead == ERROR)                 /* error from read() */
        perror ("read");

    close (sFd);                        /* close server socket connection */
    }
```

7.3 Zbuf Sockets

VxWorks includes an alternative set of socket calls based on a data abstraction called a *zbuf*, a zero-copy buffer. Using the zbuf socket interface, applications can read and write UNIX BSD sockets without copying data between application buffers and network buffers. You can use zbufs with either UDP or TCP applications. The TCP subset of the zbuf interface is sometimes called *zero-copy TCP*.

Zbuf-based socket calls are *interoperable* with the standard BSD socket interface: the other end of a socket has no way of telling whether your end is using zbuf-based calls or traditional calls. However, zbuf-based socket calls are *not source-compatible* with the standard BSD socket interface: you must call different socket functions to use the zbuf interface. Applications that use the zbuf interface are thus less portable.

⚠ **WARNING:** The send socket buffer size must exceed that of any zbufs sent over the socket. To set the send socket buffer size, use either the **TCP_SND_SIZE_DFLT** or **UDP_SND_SIZE_DFLT** configuration parameter.

To include zbuf functionality in your image, use the **INCLUDE_ZBUF_SOCK** configuration parameter.

7.3.1 Zbuf Sockets and Protection Domains

If you are using zbufs within the VxWorks AE protection domain model, you must do so within the kernel domain.

7.3.2 Zbuf Calls to Send Existing Data Buffers

The simplest way to use zbuf sockets is to call either **zbufSockBufSend( )** (in place of **send( )** for a TCP connection) or **zbufSockBufSendto( )** (in place of **sendto( )** for a UDP datagram). In either case, you supply a pointer to your application's data buffer containing the data or message to send, and the network protocol uses that same buffer rather than copying the data out of it.

⚠ **WARNING:** Using zbufs allows different modules to share the same buffers. This lets your application avoid the performance hit associated with copying the buffer. To make this work, your application must not modify (let alone free!) the data buffer while network software is still using it. Instead of freeing your buffer explicitly, you can supply a free-routine callback: a pointer to a routine that knows how to free the buffer. The zbuf library keeps track of how many zbufs point to a data buffer and calls the free routine when the data buffer is no longer in use.

To receive socket data using zbufs, see the following sections. *7.3.3 Manipulating the Zbuf Data Structure*, p.143 describes the routines to create and manage zbufs, and *7.3.4 Zbuf Socket Calls*, p.152 introduces the remaining zbuf-specific socket routines. See also the reference entries for **zbufLib** and **zbufSockLib**.

7.3.3 Manipulating the Zbuf Data Structure

A zbuf has three essential properties:

- A zbuf holds a sequence of bytes.
- The data in a zbuf is organized into one or more *segments* of contiguous data. Successive zbuf segments are not usually contiguous to each other.
- Zbuf segments refer to data buffers through pointers. The underlying data buffers can be shared by more than one zbuf segment.

Zbuf segments are at the heart of how zbufs minimize data copying; if you have a data buffer, you can incorporate it (by reference, so that only pointers and lengths move around) into a new zbuf segment. Conversely, you can get pointers to the data in zbuf segments, and examine the data there directly.

Zbuf Byte Locations

You can address the contents of a zbuf by *byte locations*. A zbuf byte location has two parts, an *offset* and a *segment ID*.

An *offset* is a signed integer (type **int**): the distance in bytes to a portion of data in the zbuf, relative to the beginning of a particular segment. Zero refers to the first byte in a segment; negative integers refer to bytes in previous segments; and positive integers refer to bytes after the start of the current segment.

A *segment ID* is an arbitrary integer (type **ZBUF_SEG**) that identifies a particular segment of a zbuf. You can always use **NULL** to refer to the first segment of a zbuf.

Figure 7-1 shows a simple zbuf with data organized into two segments. The offsets are relative to the first segment. This is the most efficient addressing scheme to use to refer to bytes a, b, or c in the figure.

Figure 7-1 **Zbuf Addressing Relative to First Segment (NULL)**

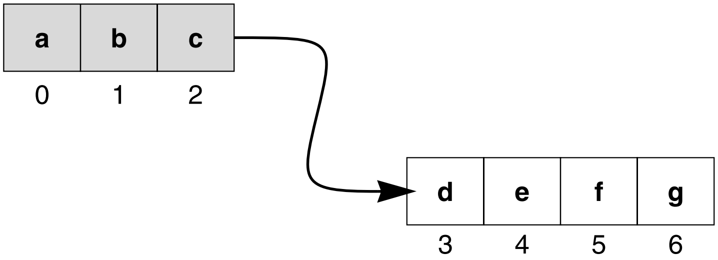
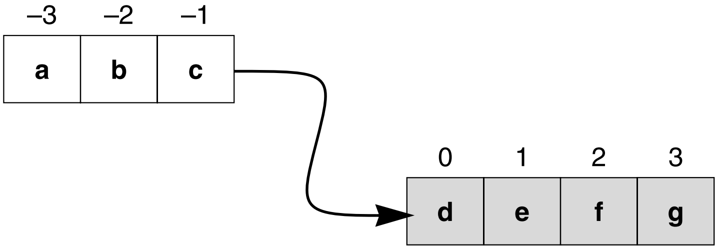


Figure 7-2 shows the same zbuf, but it is labeled with offsets relative to the second segment. This is the most efficient addressing scheme to refer to bytes d, e, f, or g in the figure.

Figure 7-2 **Zbuf Addressing Relative to Second Segment**



Two special shortcuts give the fastest access to either the beginning or the end of a zbuf. The constant **ZBUF_END** refers to the position after all existing bytes in the zbuf. Similarly, **ZBUF_BEGIN** refers to the position before all existing bytes. These constants are the only offsets with meanings not relative to a particular segment.

When you insert data in a zbuf, the new data is always inserted *before* the byte location you specify in the call to an insertion routine. That is, the byte location you specify becomes the address of the newly inserted data.

Creating and Destroying Zbufs

To create a new zbuf, call **zbufCreate( )**. The routine takes no arguments, and returns a zbuf identifier (type **ZBUF_ID**) for a zbuf containing no segments. After you have the zbuf ID, you can attach segments or otherwise insert data. While the zbuf is empty, **NULL** is the only valid segment ID, and 0 the only valid offset.

When you no longer need a particular zbuf, call **zbufDelete( )**. Its single argument is the ID for the zbuf to delete. The **zbufDelete( )** routine calls the free routine associated with each segment in the zbuf, for segments that are not shared by other zbufs. After you delete a zbuf, its zbuf ID is meaningless; any reference to a deleted zbuf ID is an error.

Table 7-4 **Zbuf Creation and Deletion Routines**

Call	Description
zbufCreate( )	Create an empty zbuf.
zbufDelete( )	Delete a zbuf and free any associated segments.

Getting Data In and Out of Zbufs

The usual way to place data in a zbuf is to call **zbufInsertBuf( )**. This routine builds a zbuf segment pointing to an existing data buffer, and inserts the new segment at whatever byte location you specify in a zbuf. You can also supply a callback pointer to a free routine, which the zbuf library calls when no zbuf segments point to that data buffer.

Because the purpose of the zbuf socket interface is to avoid data copying, the need to actually copy data into a zbuf (rather than designating its location as a shareable buffer) occurs much less frequently. When that need does arise, however, the routine **zbufInsertCopy( )** is available. This routine does not require a callback pointer to a free routine, because the original source of the data is not shared.

Similarly, the most efficient way to examine data in zbufs is to read it in place, rather than to copy it to another location. However, if you must copy some of the data out of a zbuf (for example, to guarantee the data is contiguous, or to place it in a data structure required by another interface), call **zbufExtractCopy( )**. Within the call, specify what to copy (zbuf ID, byte location, and the number of bytes) and where to put it (an application buffer).

Table 7-5 **Zbuf Data Copying Routines**

Call	Description
zbufInsertBuf( )	Create a zbuf segment from a buffer and insert into a zbuf.
zbufInsertCopy( )	Copy buffer data into a zbuf.
zbufExtractCopy( )	Copy data from a zbuf to a buffer.

Operations on Zbufs

The routines listed in Table 7-6 perform several fundamental operations on zbufs.

Table 7-6 **Zbuf Operations**

Call	Description
zbufLength( )	Determine the length of a zbuf, in bytes.
zbufDup( )	Duplicate a zbuf.
zbufInsert( )	Insert a zbuf into another zbuf.
zbufSplit( )	Split a zbuf into two separate zbufs.
zbufCut( )	Delete bytes from a zbuf.

The routine **zbufLength( )** reports how many bytes are in a zbuf.

The routine **zbufDup( )** provides the simplest mechanism for sharing segments between zbufs: it produces a new zbuf ID that refers to some or all of the data in the original zbuf. You can exploit this sort of sharing to get two different views of the same data. For example, after duplicating a zbuf, you can insert another zbuf into one of the two duplicates, with **zbufInsert( )**. None of the data in the original zbuf segments moves, yet after some byte location (the byte location where you inserted data) addressing the two zbufs gives completely different data.

The **zbufSplit( )** routine divides one zbuf into two; you specify the byte location for the split, and the result of the routine is a new zbuf ID. The new zbuf's data begins after the specified byte location. The original zbuf ID also has a modified view of the data: it is truncated to the byte location of the split. However, none of the data in the underlying segments moves through all this. If you duplicate the original zbuf before splitting it, three zbuf IDs share segments. The duplicate permits you to view the entire original range of data, another zbuf contains a leading fragment, and the third zbuf holds the trailing fragment.

Similarly, if you call **zbufCut( )** to remove some range of bytes from within a zbuf, the effects are visible only to callers who view the data through the same zbuf ID you used for the deletion. Other zbuf segments can still address the original data through a shared buffer.

For the most part, these routines do not free data buffers or delete zbufs, but there are two exceptions:

- **zbufInsert( )** deletes the zbuf ID it inserts. No segments are freed, because they now form part of the larger zbuf.
- If the bytes you remove with **zbufCut( )** span one or more complete segments, the free routines for those segments can be called (if no other zbuf segment refers to the same data).

The data-buffer free routine runs only when *none* of the data in a segment is part of any zbuf. To avoid data copying, zbuf manipulation routines such as **zbufCut( )** record which parts of a segment are currently in a zbuf, postponing the deletion of a segment until no part of its data is in use.

Segments of Zbufs

The routines in Table 7-7 give your applications access to the underlying segments in a zbuf.

Table 7-7 **Zbuf Segment Routines**

Call	Description
zbufSegFind( )	Find the zbuf segment containing a specified byte location.
zbufSegNext( )	Get the next segment in a zbuf.

Table 7-7 **Zbuf Segment Routines** *(Continued)*

Call	Description
zbufSegPrev( )	Get the previous segment in a zbuf.
zbufSegData( )	Determine the location of data in a zbuf segment.
zbufSegLength( )	Determine the length of a zbuf segment.

By specifying a **NULL** segment ID, you can address the entire contents of a zbuf as offsets from its very first data byte. However, it is always more efficient to address data in a zbuf relative to the closest segment. Use **zbufSegFind( )** to translate any zbuf byte location into the most local form.

The pair **zbufSegNext( )** and **zbufSegPrev( )** are useful for going through the segments of a zbuf in order, perhaps in conjunction with **zbufSegLength( )**.

Finally, **zbufSegData( )** allows the most direct access to the data in zbufs: it gives your application the address where a segment's data begins. If you manage segment data directly using this pointer, bear the following restrictions in mind:

- Do not change data if any other zbuf segment is sharing it.
- As with any other direct memory access, it is up to your own code to restrict itself to meaningful data: remember that the next segment in a zbuf is usually not contiguous. Use **zbufSegLength( )** as a limit, and **zbufSegNext( )** when you exceed that limit.

Example: Manipulating Zbuf Structure

The following interaction illustrates the use of some of the previously described **zbufLib** routines, and their effect on zbuf segments and data sharing. To keep the example manageable, the zbuf data used is artificially small, and the execution environment is the Tornado shell (for details on this shell, see the *Tornado User's Guide: Shell*).

To begin with, we create a zbuf, and use its ID **zId** to verify that a newly created zbuf contains no data; **zbufLength( )** returns a result of 0.

```
-> zId = zbufCreate()
new symbol "zId" added to symbol table.
zId = 0x3b58e8: value = 3886816 = 0x3b4ee0
-> zbufLength (zId)
value = 0 = 0x0
```

Next, we create a data buffer **buf1**, insert it into zbuf **zId**, and verify that **zbufLength( )** now reports a positive length. To keep the example simple, **buf1** is a literal string, and therefore we do not supply a free-routine callback argument to **zbufInsertBuf( )**.

```
-> buf1 = "I cannot repeat enough!"
new symbol "buf1" added to symbol table.
buf1 = 0x3b5898: value = 3889320 = 0x3b58a8 = buf1 + 0x10
-> zbufInsertBuf (zId, 0, 0, buf1, strlen(buf1), 0, 0)
value = 3850240 = 0x3ac000
-> zbufLength (zId)
value = 23 = 0x17
```

To examine the effect of other zbuf operations, it is useful to have a zbuf-display routine. The remainder of this example uses a routine called **zbufDisplay( )** for that purpose; for the complete source code, see Example 7-4.

For each zbuf segment, **zbufDisplay( )** shows the segment ID, the start-of-data address, the offset from that address, the length of the segment, and the data in the segment as a character string. The following display of **zId** illustrates that the underlying data in its only segment is still at the **buf1** address (0x3b58a8), because **zbufInsertBuf( )** incorporates its buffer argument into the zbuf without copying data.

```
-> ld </usr/jane/zbuf-examples/zbufDisplay.o
value = 3890416 = 0x3b5cf0 = zbufDisplay.o_bss + 0x8
-> zbufDisplay zId
segID 0x3ac000 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!
value = 0 = 0x0
```

When we copy the zbuf, the copy has its own IDs, but still uses the same data address:

```
-> zId2 = zbufDup (zId,0,0,23)
new symbol "zId2" added to symbol table.
zId2 = 0x3b5ff0: value = 3886824 = 0x3b4ee8
-> zbufDisplay zId2
segID 0x3abf80 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!
value = 0 = 0x0
```

If we insert a second buffer into the middle of the existing data in **zId**, there is still no data copying. Inserting the new buffer gives us a zbuf made up of three segments—but notice that the address of the first segment is still the start of **buf1**, and the third segment points into the middle of **buf1**:

```
-> buf2 = " this"
new symbol "buf2" added to symbol table.
buf2 = 0x3b5fb0: value = 3891136 = 0x3b5fc0 = buf2 + 0x10
-> zbufInsertBuf (zId, 0, 15, buf2, strlen(buf2), 0, 0)
value = 3849984 = 0x3abf00
```

```
-> zbufDisplay zId
segID 0x3ac000 at 0x3b58a8 + 0x0 (15 bytes): I cannot repeat
segID 0x3abf00 at 0x3b5fc0 + 0x0 ( 5 bytes):  this
segID 0x3abe80 at 0x3b58b7 + 0x0 ( 8 bytes):  enough!
value = 0 = 0x0
```

Because the underlying buffer is not modified, both **buf1** and the duplicate zbuf **zId2** still contain the original string, rather than the modified one now in **zId**:

```
-> printf ("%s\n", buf1)
I cannot repeat enough!
value = 24 = 0x18
-> zbufDisplay zId2
segID 0x3abf80 at 0x3b58a8 + 0x0 (23 bytes): I cannot repeat enough!
value = 0 = 0x0
```

The **zbufDup( )** routine can also select part of a zbuf without copying, for instance to incorporate some of the same data into another zbuf—or even into the same zbuf, as in the following example:

```
-> zTmp = zbufDup (zId, 0, 15, 5)
new symbol "zTmp" added to symbol table.
zTmp = 0x3b5f70: value = 3886832 = 0x3b4ef0

-> zbufInsert (zId, 0, 15, zTmp)
value = 3849728 = 0x3abe00
-> zbufDisplay zId
segID 0x3ac000 at 0x3b58a8 + 0x0 (15 bytes): I cannot repeat
segID 0x3abe00 at 0x3b5fc0 + 0x0 ( 5 bytes):  this
segID 0x3abf00 at 0x3b5fc0 + 0x0 ( 5 bytes):  this
segID 0x3abe80 at 0x3b58b7 + 0x0 ( 8 bytes):  enough!
value = 0 = 0x0
```

After **zbufInsert( )** combines two zbufs, the second zbuf ID (**zTmp** in this example) is automatically deleted. Thus, **zTmp** is no longer a valid zbuf ID—for example, **zbufLength( )** returns ERROR:

```
-> zbufLength (zTmp)
value = -1 = 0xffffffff = zId2 + 0xffc4a00f
```

However, you must still delete the remaining two zbuf IDs explicitly when they are no longer needed. This releases all associated zbuf-structure storage. In a real application, with free-routine callbacks filled in, it also calls the specified free routine on the data buffers, as follows:

```
-> zbufDelete (zId)
value = 0 = 0x0
-> zbufDelete (zId2)
value = 0 = 0x0
```

Example 7-4 **Zbuf Display Routine**

The following is the complete source code for the **zbufDisplay( )** utility used in the preceding example:

```
/* zbufDisplay.c - zbuf example display routine */

/* includes */

#include "vxWorks.h"
#include "zbufLib.h"
#include "ioLib.h"
#include "stdio.h"
/********************************************************************
*
* zbufDisplay - display contents of a zbuf
*
* RETURNS: OK, or ERROR if the specified data could not be displayed.
*/
STATUS zbufDisplay
    (
    ZBUF_ID     zbufId,         /* zbuf to display */
    ZBUF_SEG    zbufSeg,        /* zbuf segment base for <offset> */
    int         offset,         /* relative byte offset */
    int         len,            /* number of bytes to display */
    BOOL        silent          /* do not print out debug info */
    )
    {
    int         lenData;
    char *      pData;
    /* find the most-local byte location */
    if ((zbufSeg = zbufSegFind (zbufId, zbufSeg, &offset)) == NULL)
        return (ERROR);
    if (len <= 0)
        len = ZBUF_END;
    while ((len != 0) && (zbufSeg != NULL))
        {
        /* find location and data length of zbuf segment */
        pData = zbufSegData (zbufId, zbufSeg) + offset;
        lenData = zbufSegLength (zbufId, zbufSeg) - offset;
        lenData = min (len, lenData);     /* print all of seg ? */
        if (!silent)
            printf ("segID 0x%x at 0x%x + 0x%x (%2d bytes): ",
                (int) zbufSeg, (int) pData, offset, lenData);
        write (STD_OUT, pData, lenData);     /* display data */
        if (!silent)
            printf ("\n");
        zbufSeg = zbufSegNext (zbufId, zbufSeg); /* update segment */
        len -= lenData;                      /* update length */
        offset = 0;                          /* no more offset */
        }
    return (OK);
    }
```

Limitations of the Zbuf Implementation

The following zbuf limitations are due to the current implementation; they are not inherent to the data abstraction. They are described because they can have an impact on application performance.

- With the current implementation, references to data in zbuf segments before a particular location (whether with **zbufSegPrev( )**, or with a negative offset in a byte location) are significantly slower than references to data after a particular location.
- The data in small zbuf segments (less than 512 bytes) is sometimes copied, rather than having references propagated to it.

7.3.4 Zbuf Socket Calls

The zbuf socket calls listed in Table 7-8 are named to emphasize parallels with the standard BSD socket calls: thus, **zbufSockSend( )** is the zbuf version of **send( )**, and **zbufSockRecvfrom( )** is the zbuf version of **recvfrom( )**. The arguments also correspond directly to those of the standard socket calls.

Table 7-8 **Zbuf Socket Library Routines**

Call	Description
zbufSockLibInit( )	Initialize the socket libraries (called automatically if the configuration has zbuf sockets enabled. The relevant configuration parameter is **INCLUDE_SOCK_ZBUF**).
zbufSockSend( )	Send zbuf data through a TCP socket.
zbufSockSendto( )	Send a zbuf message through a UDP socket.
zbufSockBufSend( )	Create a zbuf and send it as TCP socket data.
zbufSockBufSendto( )	Create a zbuf and send it as a UDP socket message.
zbufSockRecv( )	Receive data in a zbuf from a TCP socket.
zbufSockRecvfrom( )	Receive a message in a zbuf from a UDP socket.

For a detailed description of each routine, see the corresponding reference entry.

Standard Socket Calls and Zbuf Socket Calls

The zbuf socket calls are particularly useful when large data transfer is a significant part of your socket application. For example, many socket applications contain sections of code like the following fragment:

```
pBuffer = malloc (BUFLEN);
while ((readLen = read (fdDevice, pBuffer, BUFLEN))  > 0)
 write (fdSock, pBuffer, readLen);
```

You can eliminate the overhead of copying from the application buffer **pBuffer** into the internal socket buffers by changing the code to use zbuf socket calls. For example, the following fragment is a zbuf version of the preceding loop:

```
pBuffer = malloc (BUFLEN * BUFNUM);           /* allocate memory */
for (ix = 0; ix < (BUFNUM - 1); ix++, pBuffer += BUFLEN)
    appBufRetn (pBuffer);                    /* fill list of free bufs */

while ((readLen = read (fdDevice, pBuffer, BUFLEN)) > 0)
    {
    zId = zbufCreate ();                     /* insert into new zbuf */
    zbufInsertBuf (zId, NULL, 0, pBuffer, readLen, appBufRetn, 0);
    zbufSockSend (fdSock, zId, readLen, 0);   /* send zbuf */
    pBuffer = appBufGet (WAIT_FOREVER);       /* get a fresh buffer */
    }
```

The **appBufGet( )** and **appBufRetn( )** references in the preceding code fragment stand for application-specific buffer management routines, analogous to **malloc( )** and **free( )**. In many applications, these routines do nothing more than manipulate a linked list of free fixed-length buffers.

Example 7-5 **The TCP Example Server Using Zbufs**

For a small but complete example that illustrates the mechanics of using the zbuf socket library, consider the conversion of the client-server example in Example 7-3 to use zbuf socket calls.

No conversion is needed for the client side of the example; the client operates the same regardless of whether or not the server uses zbufs. The next example illustrates the following changes to convert the server side to use zbufs:

- Instead of including the header file **sockLib.h**, include **zbufSockLib.h**.
- The data processing component must be capable of dealing with potentially non-contiguous data in successive zbuf segments. In the TCP example, this component displays a message using **printf( )**; we can use the **zbufDisplay( )** routine from Example 7-4 instead.

– The original TCP example exploits **fioRead( )** to collect the complete message, rather than calling **recv( )** directly. To achieve the same end while avoiding data copying by using zbufs, the following example defines a **zbufFioSockRecv( )** subroutine to call **zbufSockRecv( )** repeatedly until the complete message is received.

– A new version of the worker routine **tcpServerWorkTask( )** must tie together these separate modifications, and must explicitly extract the **reply** and **msgLen** fields from the client's transmission to do so. When using zbufs, these fields cannot be extracted by reference to the C structure in **tcpExample.h** because of the possibility that the data is not contiguous.

The following example shows the auxiliary **zbufFioSockRecv( )** routine and the zbuf version of **tcpServerWorkTask( )**. To run this code:

1. Start with **tcpServer.c** as defined in Example 7-3.
2. Include the header file **zbufSockLib.h**.
3. Insert the **zbufDisplay( )** routine from Example 7-4.
4. Replace the **tcpServerWorkTask( )** definition with the following two routines:

```
/*******************************************************************
*
* zbufFioSockRecv - receive <len> bytes from a socket into a zbuf
*
* This routine receives a specified amount of data from a socket into a
* zbuf, by repeatedly calling zbufSockRecv() until <len> bytes
* are read.
*
* RETURNS:
* The ID of the zbuf containing <len> bytes of data,
* or NULL if there is an error during the zbufSockRecv() operation.
*
* SEE ALSO: zbufSockRecv()
*/
ZBUF_ID zbufFioSockRecv
    (
    int         fd,             /* file descriptor of file to read */
    int         len             /* maximum number of bytes to read */
    )
    {
    BOOL        first = TRUE;           /* first time thru ? */
    ZBUF_ID     zRecvTotal = NULL;      /* zbuf to return */
    ZBUF_ID     zRecv;                  /* zbuf read from sock */
    int         nbytes;                 /* number of recv bytes */
    for (; len > 0; len -= nbytes)
        {
        nbytes = len;                   /* set number of bytes wanted */
        /* read a zbuf from the socket */
        if (((zRecv = zbufSockRecv (fd, 0, &nbytes)) == NULL) ||
            (nbytes <= 0))
            {
            if (zRecvTotal != NULL)
                zbufDelete (zRecvTotal);
            return (NULL);
            }
        /* append recv'ed zbuf onto end of zRecvTotal */
        if (first)
            zRecvTotal = zRecv;         /* cannot append to empty zbuf */
            first = FALSE;              /* can append now... */
        else if (zbufInsert (zRecvTotal, NULL, ZBUF_END, zRecv) == NULL)
            {
            zbufDelete (zRecv);
            zbufDelete (zRecvTotal);
            return (NULL);
            }
        }
    return (zRecvTotal);
    }
/************************************************************************
*
* tcpServerWorkTask - process client requests
*
* This routine reads from the server's socket, and processes client
* requests.  If the client requests a reply message, this routine
* sends a reply to the client.
*
* RETURNS: N/A.
*/
VOID tcpServerWorkTask
    (
    int                 sFd,            /* server's socket fd */
    char *              address,        /* client's socket address */
    u_short             port            /* client's socket port */
    )
    {
    static char         replyMsg[] = "Server received your message";
    ZBUF_ID             zReplyOrig;     /* original reply msg */
    ZBUF_ID             zReplyDup;      /* duplicate reply msg */
    ZBUF_ID             zRequest;       /* request msg from client */
    int                 msgLen;         /* request msg length */
    int                 reply;          /* reply requested ? */
    /* create original reply message zbuf */
    if ((zReplyOrig = zbufCreate ()) == NULL)
        {
        perror ("zbuf create");
        free (address);                 /* free malloc from inet_ntoa() */
        return;
        }
    /* insert reply message into zbuf */
    if (zbufInsertBuf (zReplyOrig, NULL, 0, replyMsg,
        sizeof (replyMsg), NULL, 0) == NULL)
        {
        perror ("zbuf insert");
        zbufDelete (zReplyOrig);
        free (address);                 /* free malloc from inet_ntoa() */
        return;
        }
    /* read client request, display message */
    while ((zRequest = zbufFioSockRecv (sFd, sizeof(struct request))) != NULL)
        {
        /* extract reply field into <reply> */
        (void) zbufExtractCopy (zRequest, NULL, 0,
            (char *) &reply, sizeof (reply));
        (void) zbufCut (zRequest, NULL, 0, sizeof (reply));
        /* extract msgLen field into <msgLen> */
        (void) zbufExtractCopy (zRequest, NULL, 0,
            (char *) &msgLen, sizeof (msgLen));
        (void) zbufCut (zRequest, NULL, 0, sizeof (msgLen));
        /* duplicate reply message zbuf, preserving original */
        if ((zReplyDup = zbufDup (zReplyOrig, NULL, 0, ZBUF_END)) == NULL)
            {
            perror ("zbuf duplicate");
            zbufDelete (zRequest);
            break;
            }
        printf ("MESSAGE FROM CLIENT (Internet Address %s, port %d):\n",
            address, port);

        /* display request message zbuf */
        (void) zbufDisplay (zRequest, NULL, 0, msgLen, TRUE);
        printf ("\n");
        if (reply)
            {
            if (zbufSockSend (sFd, zReplyDup, sizeof (replyMsg), 0) < 0)
                perror ("zbufSockSend");
            }
        /* finished with request message zbuf */
        zbufDelete (zRequest);
        }
    free (address);                     /* free malloc from inet_ntoa() */
    zbufDelete (zReplyOrig);
    close (sFd);
    }
```

⚠ **CAUTION:** In the interests of brevity, the **STATUS** return values for several zbuf socket calls are discarded with casts to **void**. In a real application, check these return values for possible errors.

8
Remote Access Applications

8.1 Introduction

This chapter discusses the applications that provide remote network access. VxWorks supports the following:[1]

- RPC (Remote Procedure Call, for distributed processing)
- RSH (Remote Shell, for remote file access)
- FTP (File Transfer Protocol, for remote file access)
- NFS (Network File System, for remote file access)
- TFTP (Trivial File Transfer Protocol, for remote file access)
- **rlogin** (for remote login)
- **telnet** (for remote login)

In addition to the simple implementation of the protocols listed above, VxWorks also includes the drivers:

- **netDrv** — for downloading files using either FTP or RSH
- **nfsDrv** — for locally mounting remote file systems using NFS

[1] If you are developing on a Windows host, check your Windows and networking software documentation for information on which of these protocols are supported under Windows and how to use them.

8.2 RSH, FTP, and netDrv

VxWorks provides an implementation of the client side (but not the server side) of the RSH protocol. VxWorks also provides an implementation of both the client and the server sides of the FTP protocol.

Using RSH, a VxWorks application can run commands on a remote system and receive the command results on standard output and standard error over socket connections. To execute commands remotely, RSH requires that the remote system supports the server side of RSH and that the remote system grant access privileges to the user specified in the RSH request. On a UNIX system, RSH server support is implemented using the **rshd** shell daemon, and access privileges are controlled by a **.rhosts** file. On a VxWorks host, there is no equivalent to **rshd**. Thus, remote systems cannot use RSH to run commands on a VxWorks host.

You can use both RSH and FTP directly, but you can also use them indirectly to download files through the mediation of the **netDrv** library. Using **netDrv** in this way is especially convenient when a target needs to download a run-time image at boot time.

That **netDrv** can use FTP to download a file is not surprising, given that FTP is a protocol designed specifically for file transfer. Specifically, **netDrv** uses the FTP **RETR** and **STOR** commands to retrieve and store the entire requested file. That **netDrv** can use RSH to download a file is less obvious. RSH has no built-in commands dedicated to file transfer. However, **netDrv** executes a remote **cat** on the file it wants to download.

Setting the User ID for Remote File Access with RSH or FTP

All FTP and RSH requests to a remote system include the user name. All FTP requests include a password as well as a user name. From VxWorks, you can specify the user name and password for remote requests by calling **iam( )**:

```
iam ("username", "password")
```

The first argument to **iam( )** is the user name that identifies you when you access remote systems. The second argument is the FTP password. This is ignored if RSH is being used, and can be specified as **NULL** or 0 (zero).

For example, the following command tells VxWorks that all accesses to remote systems with RSH or FTP are through user *darger*, and if FTP is used, the password is *unreal*:

```
-> iam "darger", "unreal"
```

NOTE: When a VxWorks boot program downloads a run-time image from a remote network source using a **netDrv** instance, it relies upon either the FTP or RSH protocols. To determine its user name and password (if any) for use with these protocols, the boot program relies upon the values specified for these parameters in the boot line.

Setting File Permissions on the Remote System

For a VxWorks system to have access to a particular file on a host, you must set up permissions on the host system appropriately. The user name seen from the host must have permission to read that file (and write it, if necessary). That user name must also have permission to access all directories in the path. The easiest way to check this is to log in to the host with the user name VxWorks uses, and try to read or write the file in question. If you cannot do this, neither can the VxWorks system.

8.2.1 RSH

Using the VxWorks RSH implementation, a VxWorks target can execute commands on remote systems that run an **rshd** shell daemon. The command results return on *standard output* and *standard error* over socket connections.

To include RSH in VxWorks, use the configuration component:

INCLUDE_NETWRS_REMLIB — include the remote command (RSH) library
Associated with this component is the configuration parameter:

RSH_STDERR_SETUP_TIMEOUT
Use this parameter to specify how long an **rcmd( )** call should wait for a return from its internal call to select. The default value for this parameter is -1, which is no timeout or **WAIT_FOREVER**.

For more information on how to use RSH under VxWorks, see the **remLib** reference entries.

NOTE: The VxWorks RSH implementation does not include an equivalent to the **rshd** daemon. Thus, remote systems cannot use RSH to run commands remotely on a VxWorks host.

Configuring the Remote Host to Allow Access to an RSH User

Included in an RSH request is the name of the requesting user. The receiving host can then honor or ignore the request based on the user name and the site from which the request originates. How you set up a receiving system to allow access to particular users and sites depends on the specifics of the receiving system's OS and networking software.

For Windows hosts, support for RSH is determined by your version of Windows and the networking software you are using. See that documentation for details.

For UNIX hosts, an RSH request is honored only if it originated on a known system by a user with local login privileges. The list of known systems is specified in either of two locations. The first location, the **/etc/hosts.equiv** file, maintains a list of all systems from which remote access is allowed for all users that have local accounts. The second location, a ~*userName*/**.rhosts** file, maintains a list of systems from which remote access is allowed for that particular user, *userName*.

Which location you use depends on your security needs. In most environments, adding system names to the **/etc/hosts.equiv** file is considered too dangerous. Thus, for most environments, the preferred method is to add system names to a ~*userName*/**.rhosts** file. The format for this file is one system name per line.

The FTP protocol, unlike RSH, specifies both the user name and password on every request. Therefore, when using FTP, the UNIX system does not use the **.rhosts** or **/etc/hosts.equiv** files to authorize remote access.

8.2.2 FTP

To include the VxWorks FTP implementation, use the following configuration components:

INCLUDE_FTP — includes the FTP client, **ftpLib**.

INCLUDE_FTP_SERVER — includes the FTP server, **ftpdLib**.

INCLUDE_FTPD_SECURITY — includes FTP server security functionality.

For information on how to use the VxWorks FTP client and server implementations directly, see the reference entries for **ftpLib** and **ftpdLib**.

NOTE: The VxWorks FTP implementation does not support the **REST** (restart) command.

8.2.3 Using netDrv

Although you can use **netDrv** at boot time to download a run-time image, **netDrv** is not limited to boot time or run-time images. It is a generic I/O device that you can use to access files on a remote networks system. To include **netDrv** in VxWorks, use the configuration component:

INCLUDE_NET_DRV — **netDrv** I/O library for accessing files on a remote host

To use **netDrv**, you must create a **netDrv** device for each system on which you want to access files. You can then use this device in standard VxWorks I/O device calls such as **open( )**, **read( )**, **write( )**, and **close( )**. To create a **netDrv** device, call **netDevCreate( )**:

```
netDevCreate ("devName", "host", protocol)
```

Its arguments are:

devName
The name of the device to be created.

host
The Internet address of the host in dot notation, or the name of the remote system as specified in a previous call to **hostAdd( )**. Most typically, one composes the device name using the host name followed by a colon.

protocol
The file transfer protocol: 0 for RSH or 1 for FTP.

For example, the following call creates a new I/O device on VxWorks called **mars:**, which accesses files on the host system **mars** using RSH:

```
-> netDevCreate "mars:", "mars", 0
```

After a network device is created, files on that host are accessible by appending the host pathname to the device name. For example, the filename **mars:/usr/darger/myfile** refers to the file **/usr/darger/myfile** on the **mars** system. You can read or write to this file as if it were a local file. For example, the following Tornado shell command opens that file for I/O access:

```
-> fd = open ("mars:/usr/darger/myfile", 2)
```

Using netDrv to Download Run-Time Images

The **usrNetInit( )** call in a VxWorks boot program can automatically create a **netDrv** instance for the host name specified in the VxWorks boot parameters. The boot program then uses this device to download an image from the host specified in the boot parameters. Whether the **netDrv** instance uses FTP or RSH to download the image depends on whether the boot parameters include an FTP password. When the FTP password is present, **netDrv** uses FTP. Otherwise, **netDrv** uses RSH.[2]

For most single-processor stand-alone VxWorks targets, using FTP or RSH to download an image is useful while developing an application but rare in the deployed system. However, for devices that consist of multiple VxWorks targets linked together by a shared memory backplane, using FTP to download run-time images from a central server CPU to its client CPUs is convenient and common. As the clients on the backplane boot, they create instances of **netDrv** that they then use to download their run-time images from the server.

Consider the system shown in Figure 8-1. Using the shared memory backplane, CPU 1 can use FTP to download its run-time image from the storage device (in this case, a SCSI disk) accessible through CPU 0. Note that the client must include a non-empty **ftp password** field in its boot parameter:

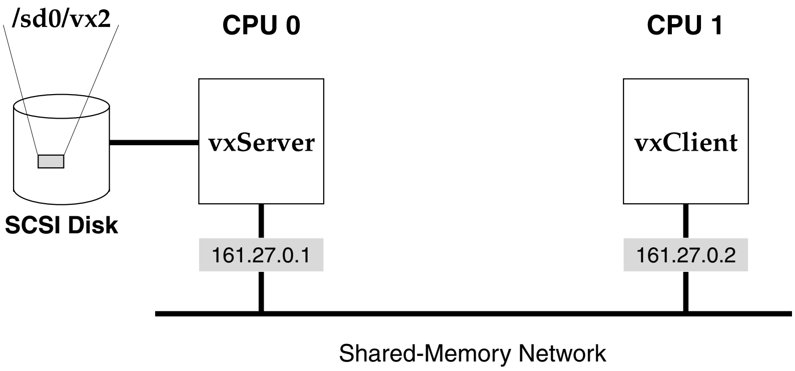
```
boot device                         : sm=0x800000
processor number                    : 1
host name                           : vxServer
file name                           : /sd0/vx2
inet on backplane (b)               : 161.27.0.2
host inet (h)                       : 161.27.0.1
user (u)                            : caraboo
ftp password (pw) (blank=use rsh)   : ignored
```

Including an FTP password tells **netDrv** to use FTP. Whether the FTP server on CPU 0 checks the validity of the password depends on whether the FTP server on CPU 0 has been configured with security turned off (the default) or on. The relevant configuration parameter is **INCLUDE_FTPD_SECURITY**.

The FTP server daemon is initialized on the VxWorks server based on the configuration. The relevant configuration parameter is **INCLUDE_FTP_SERVER**. See also the reference entry for **ftpdLib**.

[2] When creating a boot program that must download an image using an RSH or and FTP client, you must make sure that you include those components in the program.

Figure 8-1 **FTP Boot Example**



8.3 NFS and nfsDrv

The VxWorks NFS implementation supports both the client and server side of the protocol. The relevant configuration components are:

INCLUDE_NFS — the NFS (version 2) client, **nfsLib** and **nfsDrv**

The basic NFS client support functions are provided in **nfsLib**. Using the **nfsLib** functionality, **nfsDrv** completes the implementation of an NFS client compatible with VxWorks. Associated with **INCLUDE_NFS** are the following parameters:

NFS_CLIENT_NAME — local host name for NFS access, defaults to the target name specified in the system boot parameters.

NFS_GROUP_ID — group identifier for NFS access, defaults to 100

NFS_USER_ID — user identifier for NFS access, defaults to 2001

INCLUDE_NFS_MOUNT_ALL — mount file systems exported by the remote host

Including this component tells the VxWorks NFS client to automatically mount all file systems exported from the remote host that you named in the boot parameters. If you include **INCLUDE_NFS_MOUNT_ALL**, the VxWorks boot procedure automatically calls **nfsMountAll( )** using the host name and client (target) name in the boot parameters.

INCLUDE_NFS_SERVER — the NFS (version 2) server, **mountLib** and **nfsdLib**
The Mount Protocol Library, **mountLib**, provides the functionality needed to manage file export under VxWorks. The NFS Server library, **nfsdLib**, implements the server side of the NFS protocol. This NFS server is designed for use with dosFs file systems.

The VxWorks NFS libraries are implemented using RPC.

8.3.1 VxWorks NFS Clients

To mount a remote file system on a VxWorks host, you must make sure that the remote system runs an NFS server and that the remote directory has been made available for export to your client. Then, on the target running a VxWorks NFS client, you must set your NFS client name, user ID, and group ID. Finally, you can call **nfsMount( )**, which creates an **nfsDrv** instance that mounts and manages the mounted remote file system.

Exporting File Systems from the Remote NFS Server

For a UNIX NFS server, the **/etc/exports** file specifies which of the server's file systems are exported for mounting by remote NFS clients. For example, if **/etc/exports** contains the line:

```
/usr
```

The server exports **/usr** without restriction. If you want to limit access to this directory, you can include additional parameters on the line. For more information on these parameters, consult your UNIX system documentation. If a file system on a UNIX NFS server is not listed in **/etc/exports**, the file system is not exported, which means other machines cannot use NFS to mount it.

Windows systems also support NFS. Thus, it is possible to configure a directory on a Windows system so that it is exported over NFS. However, the exact procedures for doing so depend upon the particular network package you purchased. For more information, consult the documentation included with your Windows networking package.

Setting Your NFS Client Name, User ID, and Group ID

Internally, NFS depends upon RPC to handle the remote execution of the commands (open, read, write, and others) that access the data in the remote file system. Associated with the RPC protocol is an authentication system known as **AUTH_UNIX**. This authentication system requires RPC peers to provide a user name, a user ID, and a group name. The recipient of an RPC message uses this information to decide whether to honor or ignore the RPC request.

On a VxWorks host, you can set the NFS user name, user ID, and group name using the **NFS_CLIENT_NAME**, **NFS_GROUP_ID**, and **NFS_USER_ID** parameters included in the **INCLUDE_NFS** configuration component. You can also set these values by calling **nfsAuthUnixSet( )** or **nfsAuthUnixPrompt( )**. For example, to use **nfsAuthUnixSet( )** to set the NFS user ID to 1000 and the NFS group ID to 200 for the machine **mars**, you would call **nfsAuthUnixSet( )** as follows:

```
-> nfsAuthUnixSet "mars", 1000, 200, 0
```

The **nfsAuthUnixPrompt( )** routine provides a more interactive way of setting the NFS authentication parameters from the Tornado shell.

On UNIX systems, a user ID is specified in the file **/etc/passwd**. A list of groups that a user belongs to is specified in the file **/etc/group**. To configure a default user ID and group ID, set **NFS_USER_ID** and **NFS_GROUP_ID**. The NFS authentication parameters will take on these values at system startup. If NFS file access is unsuccessful, make sure that the configuration is correct.

Mounting a Remote File System

After setting your NFS client name, user ID, and group ID, you are ready to call **nfsMount( )** to mount any file system exported by a known host. To add a system to the list of hosts known to a VxWorks system, call **hostAdd( )**:

```
hostAdd ("host", "IPaddress" )
```

This function associates a host name with an IP address. Thus, if you wanted to mount a file system exported by a system called “mars,” you would need to have already called **hostAdd( )** for “mars.” For, example, if “mars” were at 150.12.0.1, you would need to call **hostAdd( )** as follows:

```
hostAdd ("mars", "150.12.0.1" )
```

If “mars” exports a file system called **/usr**, you can now use a call to **nfsMount( )** to create a local mount of that remotely exported file system. The syntax of an **nfsMount( )** call is as follows:

```
nfsMount ("hostName", "hostFileSys", "localName")
```

Its arguments are:

hostName
The host name of the NFS server that export the file system you want to mount.

hostFileSys
The name of the host file system or subdirectory as it is known on the exporting NFS server system.

localName
The local name to assign to the file system.

For example, the following call mounts **/usr** of the host **mars** as **/vwusr** locally:

```
-> nfsMount "mars", "/usr", "/vwusr"
```

If the call above completes successfully, it creates a local I/O device called **/vwusr**. This device refers to the mounted file system. You can open, read, write, and close **/vwusr** just like any other VxWorks I/O device. Further, a reference on the VxWorks target to a file with the name **/vwusr/darger/myfile** refers to the file **/usr/darger/myfile** on the host **mars** as if it were local to the VxWorks system.

If you do not need to mount the remote file system under a new name, you should consider using **nfsMountAll( )** instead of **nfsMount( )**. A call to **nfsMountAll( )** mounts all file systems that are exported from the remote system and that are accessible to the specified client. The syntax of **nfsMountAll( )** is as follows:

```
nfsMountAll( "hostName", "clientName", quietFlag )
```

hostName
The name of the host from which you want to mount all exported file systems.

clientName
Your NFS client name.

quietFlag
A boolean value that tells **nfsMountAll( )** whether to execute in verbose or silent mode. **FALSE** indicates verbose mode, and **TRUE** indicates quiet mode.

8.3.2 VxWorks NFS Servers

To include the VxWorks NFS server in an image, use the configuration component **INCLUDE_NFS_SERVER**. Using the VxWorks NFS server, it is possible for a VxWorks target to act as a file server for any system that runs an NFS client. Consistent with the NFS protocol, the VxWorks NFS server exports only those file systems explicitly marked for NFS export. Under VxWorks, preparing a file system for export is a two step procedure. First, when you initialize the file system, you must initialize it to allow NFS export. Then you must register the file system with the NFS server by calling **nfsExport( )**.

Initializing a File System for NFS Export

On a VxWorks target, you can export a file system over NFS only if you have initialized that file system. The following steps initialize a DOS file system called **/goodstuff** on a SCSI drive. You can use any block device instead of SCSI. Your BSP can also support other suitable device drivers; see your BSP's documentation.

1. Initialize the block device containing your file system.

   For example, you can use a SCSI drive as follows:

   ```
   scsiAutoConfig (NULL);
   pPhysDev = scsiPhysDevIdGet (NULL, 1, 0);
   pBlkDev = scsiBlkDevCreate (pPhysDev, 0, 0);
   ```

   Calling **scsiAutoConfig( )** configures all SCSI devices connected to the default system controller. (Real applications often use **scsiPhysDevCreate( )** instead, to specify an explicit configuration for particular devices.) The **scsiPhysDevIdGet( )** call identifies the SCSI drive by specifying the SCSI controller (**NULL** specifies the default controller), the bus ID (1), and the Logical Unit Number (0). The call to **scsiBlkDevCreate( )** initializes the data structures to manage that particular drive.

2. Initialize the file system. For example, if the device already has a valid dosFs file system on it, initialize it as follows:

   ```
   dosFsDevInit ("/goodstuff", pBlkDev, NULL);
   ```

   Otherwise, specify a pointer to a **DOS_VOL_CONFIG** structure rather than **NULL** as the third argument to **dosFsDevInit( )** (see the **dosFsLib** reference entry for details).

⚠ **CAUTION:** For NFS-exportable file systems, the device name must *not* end in a slash.

Exporting a File System through NFS

After you have an exportable file system, call **nfsExport( )** to make it available to NFS clients on your network. Then mount the file system from the remote NFS client, using the facilities of that system. The following example shows how to export the new file system from a VxWorks platform called **vxTarget**, and how to mount it from a typical UNIX system.

1. After the file system (**/goodstuff** in this example) is initialized, the following function call specifies it as a file system to be exported with NFS:

   ```
   nfsExport ("/goodstuff", 0, FALSE, 0);
   ```

   The first three arguments specify the name of the file system to export; the VxWorks NFS export ID (0 means to assign one automatically); and whether to export the file system as read-only. The last argument is a placeholder for future extensions.

2. To mount the file system from another machine, see the system documentation for that machine. Specify the name of the VxWorks system that exports the file system, and the name of the desired file system. You can also specify a different name for the file system as seen on the NFS client.

⚠ **CAUTION:** On UNIX systems, you need root access to mount file systems.

For example, on a typical UNIX system, the following command (executed with root privilege) mounts the **/goodstuff** file system from the VxWorks system **vxTarget**, using the name **/mnt** for it on UNIX:

```
# /etc/mount vxTarget:/goodstuff /mnt
```

Limitations of the VxWorks NFS Server

When exporting dosFs file systems that do not provide file permissions, the VxWorks NFS Server does not normally provide authentication services for NFS requests. To authenticate incoming requests, write your own authentication functions and arrange to call them when needed. See the reference entries for **nfsdInit( )** and **mountdInit( )** for information on authorization hooks.

About leofs

A file system exported from a VxWorks target always contains a file called **leofs**. This file is essential to the normal operation of the VxWorks NFS server. Do not delete it.

8.4 TFTP

The Trivial File Transfer Protocol (TFTP) is implemented on top of the Internet User Datagram Protocol (UDP) and conforms to the RFC 1350 recommendations for packet format. VxWorks provides both a TFTP client and a TFTP server. The TFTP client is useful at boot time, when you can use it to download a VxWorks image from the boot host. The TFTP server is useful if you want to boot an X-Terminal from VxWorks. It is also useful if you want to boot another VxWorks system from a local disk.

To include the VxWorks TFTP client or server, use the following configuration components:

INCLUDE_TFTP_CLIENT — include the TFTP client, **tftpLib**.

INCLUDE_TFTP_SERVER — include the TFTP server, **tftpdLib**.

Unlike FTP and RSH, TFTP requires no authentication; that is, the remote system does not require an account or password. The TFTP server allows only publicly readable files to be accessed. Files can be written only if they already exist and are publicly writable.

8.4.1 Host TFTP Server

Typically, the host-resident Internet daemon starts the TFTP server. For added security, some hosts (for example, Sun hosts) default to starting the TFTP server with the secure (**-s**) option enabled. If **-s** is specified, the server restricts host access by limiting all TFTP requests to the specified directory (for example, **/tftpboot**).

For example, if the secure option was set with -**s /tftpboot**, a TFTP request for the file **/vxBoot/vxWorks** is satisfied by the file **/tftpboot/vxBoot/vxWorks** rather than the expected file **/vxBoot/vxWorks**.

To disable the secure option on the TFTP server, edit **/etc/inetd.conf** and remove the **-s** option from the **tftpd** entry.

8.4.2 VxWorks TFTP Server

The TFTP server daemon is initialized by default when VxWorks is appropriately configured. The relevant configuration parameter is **INCLUDE_TFTP_SERVER**. See the reference entry for **tftpdLib**.

8.4.3 VxWorks TFTP Client

Include the VxWorks TFTP client side by reconfiguring VxWorks. The relevant configuration parameter is **INCLUDE_TFTP_CLIENT**. To boot using TFTP, specify 0x80 in the boot flags parameters. To transfer files from the TFTP host and the VxWorks client, two high-level interfaces are provided, **tftpXfer( )** and **tftpCopy( )**. See the reference entry for **tftpLib**.

8.5 RPC Remote Procedure Calls

The Remote Procedure Call (RPC) protocol implements a client-server model of task interaction. In this model, client tasks request services of server tasks and then wait for replies. RPC formalizes this model and provides a standard protocol for passing requests and returning replies.

Internally, RPC uses sockets as the underlying communication mechanism. RPC, in turn, is used in the implementation of several higher-level facilities, including the Network File System (NFS) and remote source-level debugging. In addition, RPC includes utilities to help generate the client interface routines and the server skeleton.

The VxWorks implementation of RPC is task-specific. Each task must call **rpcTaskInit( )** before making any RPC-related calls.

The VxWorks RPC implementation is based on a public domain implementation that originated at Sun Microsystems. RPC is equivalent to the Open Networking Computing (ONC) RPC standard. For more information, see the public domain RPC documentation and the reference entry for **rpcLib**.

To include VxWorks RPC, use the **INCLUDE_RPC** configuration component.

8.6 rlogin

You can log in to a host system from a VxWorks terminal using **rlogin( )**. For more information on the VxWorks implementation of **rlogin( )**, see the reference entry for **rlogLib**. To include VxWorks **rlogin( )** in an image, use the **INCLUDE_RLOGIN** configuration component.

When connecting with a Windows host system, VxWorks's ability to remotely login depends on your version of Windows and the networking software you are using. See that documentation for details.

When connecting with a UNIX host system, access permission must be granted to the VxWorks system by entering its system name either in the **.rhosts** file (in your home directory) or in the **/etc/hosts.equiv** file. For more information, see *Configuring the Remote Host to Allow Access to an RSH User*, p. 160.

8.7 telnet

Like **rlogin**, **telnet** is another remote login utility. However, **telnet** does not require any previous setup of a **.rhosts** file or its equivalent under non-UNIX systems, but it does require that the remote system is configured to grant login privileges to the user specified for a **telnet** session.

For more information on how to use the **telnet** server with a VxWorks target, see the reference entry for **telnetdLib**.

NOTE: VxWorks does not support a **telnet** client. As a result, you cannot use **telnet** to connect *from* a machine running VxWorks.

To include the VxWorks telnet server, use the **INCLUDE_TELNET** configuration component:

INCLUDE_TELNET — include the telnet server
Associated with this component are the following parameters:

TELNETD_MAX_CLIENTS — maximum number of simultaneous client sessions allowed. The default value is 1, which is the only possible value using the default **TELNETD_PARSER_HOOK**.

TELNETD_TASKFLAG — permission for the server to create tasks before establishing connections, defaults to **FALSE**.

TELNETD_PORT — the port monitored by the telnet server. The default is 23.

TELNETD_PARSER_HOOK — name of the function that implements the command interpreter. This function connects clients to the parser. The default routine, **shellParserControl**, accesses the VxWorks target shell. This function, defined in **target/src/ostool/remShellLib.[c,o]**, is not normally exposed to users and requires **INCLUDE_SHELL**. However, if you have written or ported an application that requires that you to replace **shellParserControl( )** with your own implementation, you must do so using a function with the same API:

```
STATUS shellParserControl
    (
    int    remoteEvent,  /* Starting or stopping a connection? */
    UINT32 sessionId,    /* Unique identifier for each session */
    int *  pInFd,  /* Input to command interpreter (written by socket) */
    int *  pOutFd  /* Output from command interpreter (read by socket) */
    )
```

The internal (not printed) reference entry for **shellParserControl( )** is as follows:

shellParserControl - handle connecting and disconnecting remote users

This routine configures the shell to connect new **telnet** or **rlogin** sessions to the command interpreter by redirecting standard input and standard output and restores the original settings when those sessions exit. This routine is the default parser control installed as part of the **INCLUDE_SHELL_REMOTE** component. The default **TELNETD_PARSER_HOOK** setting for the **INCLUDE_TELNET** component accesses this routine. It only supports a single remote session at a time, which determines the default value of the **TELNETD_MAX_CLIENTS** parameter.

Returns: **OK** or **ERROR**.

9
DNS and SNTP

9.1 Introduction

This chapter provides brief descriptions of the VxWorks implementations of DNS and SNTP.

DNS is a distributed database that most TCP/IP applications can use to translate host names to IP addresses and back. DNS uses a client/server architecture. The client side is known as the *resolver*. The server side is called the *name server*. VxWorks provides the resolver functionality in **resolvLib**. For detailed information on DNS, see *RFC 1034* and *RFC 1035*.

SNTP is a Simple Network Protocol for Time. Using an SNTP client, a target can maintain the accuracy of its internal clock based on time values reported by one or more remote sources. Using an SNTP server, the target can provide time information to other systems.

9.2 DNS: Domain Name System

Most TCP/IP applications use Internet host names instead of IP addresses when they must refer to locations in the network. One reason for this is that host names are a friendlier human interface than IP addresses. In addition, when a host-name/IP-address pair changes, the services associated with that site typically follow the host name and not the IP address. Most applications should probably refer to network locations using host names instead of IP addresses. To make this possible, the applications need a way to translate between host names and IP addresses.

On a small isolated network, a hand-edited table is a viable solution to the look-up problem. Such a table contains entries that pair up host names with their corresponding IP addresses. If you copy this table to each host on the network, you give the applications running on those hosts the ability to translate host names to IP addresses. However, as hosts are added to the network, you must update this table and then redistribute it to all the hosts in the network. This can quickly become an overwhelming task if you must manage it manually.

As networks grow, they develop a hierarchy whose structure changes with the growth. Such restructuring can change the network addresses of almost every machine on the network. In addition, these changes are not necessarily made from a single central location. Network users at different locations can add or remove machines at will. This gives rise to a decentralized network with a dynamically changing structure. Trying to track such a structure using a static centralized table is impractical. One response to this need is the Domain Name System (DNS).

9.2.1 Domain Names

DNS is modeled after a tree architecture. The root of the tree is unnamed. Below the root comes a group of nodes. Each of these nodes represents a domain within the network. Associated with each node is a unique label, a domain name of up to 63 characters. The domain names are managed by the NIC (Network Information Center), which delegates control of the top-level domains to countries, universities, governments, and organizations.

An example of a domain name is "com", the commercial domain. Wind River Systems is a commercial organization, thus it fits under the commercial domain. The NIC has given Wind River the authority to manage the name space under "windriver.com". Wind River uses this space to name all the hosts in its network.

9.2.2 The VxWorks Resolver

The VxWorks implementation of the resolver closely follows the 4.4 BSD resolver implementation. However, the VxWorks implementation differs in the way it handles the **hostent** structure. The 4.4 BSD resolver is a library that links with each process. It uses static structures to exchange data with the process.

This is not possible under VxWorks, which uses a single copy of the library that it shares among all the tasks in the system. All applications using the resolver library must provide their own buffers. Thus, the functions **resolvGetHostByName( )** and **resolvGetHostByAddr( )** require two extra parameters (for a detailed description of the interface, see the reference entries for these routines).

The VxWorks resolver library uses UDP to send requests to the configured name servers. The resolver also expects the server to handle any recursion necessary to perform the name resolution. You can configure the resolver at initialization or at run-time.[1]

The resolver can also query multiple servers if you need to add redundancy to name resolution in your system. Additionally, you can configure how the resolver library responds to a failed name server query. Either the resolver looks in the static host configuration table immediately after the failed query, or the resolver ignores the static table.[2] The default behavior of the resolver is to query only the name server and ignore the static table.

Resolver Integration

The resolver has been fully integrated into VxWorks. Existing applications can benefit from the resolver without needing to make any code changes. This is because the code internal to **hostGetByName( )** and **hostGetByAddr( )** has been updated to use the resolver.[3] Thus, the only thing you need do to take advantage of the resolver is to include it in your VxWorks stack.

Resolver Configuration

The resolver library is not included by default in the VxWorks stack. Thus, to include the resolver in your VxWorks stack, you must set configuration parameters as follows:

1. Reconfigure VxWorks with the DNS resolver on. The relevant configuration parameter is **INCLUDE_DNS_RESOLVER**.
2. Establish the IP address of the Domain Name Server by changing the default value for the configuration parameter **RESOLVER_DOMAIN_SERVER** to the IP address of the server. The IP address of the server needs to be in dotted decimal notation (for example, 90.0.0.3).

[1] For initialization, call **resolvParamsGet( )** and **resolvParamsSet( )**. See the reference entries for these routines.

[2] The static host name table is maintained by **hostLib**.

[3] Both **hostGetByName( )** and **hostGetByAddr( )** are **hostLib** functions.

3. Make sure that a route to the Domain Name Server exists before you try to access the resolver library. To do this, you can use **mRouteAdd( )** to add the route to the routing table. However, if you have included a routing protocol such as RIP in your VxWorks stack, it will add the route for you.
4. Define the domain to which the resolver belongs by changing the default Resolver Domain (defined by the **RESOLVER_DOMAIN** configuration parameter).

   You must change this domain name to the domain name to which your organization belongs. The resolver uses this domain name when it tries to query the domain server for the name of the host machine for its organization.

The resolver library supports a debug option, the DNS Debug Messages parameter: **DNS_DEBUG**. Using this parameter causes a log of the resolver queries to be printed to the console. The use of this feature is limited to a single task. If you have multiple tasks running, the output to the console will be garbled.

9.3 SNTP: A Time Protocol

VxWorks supports a client and server for the Simple Network Time Protocol (SNTP). You can use the client to maintain the accuracy of your system's internal clock based on time values reported by one or more remote sources. You can use the server to provide time information to other systems.

9.3.1 Using the SNTP Client

To include the SNTP client, reconfigure your VxWorks stack. The relevant configuration parameter is **INCLUDE_SNTPC**. To retrieve the current time from a remote source, call **sntpcTimeGet( )**. This routine retrieves the time reported by a remote source and converts that value for POSIX-compliant clocks. To get time information, **sntpcTimeGet( )** either sends a request and extracts the time from the reply, or it waits until a message is received from an SNTP/NTP server executing in broadcast mode. See the **sntpcTimeGet( )** reference entry.

Summary of SNTP Client Build-Time Configuration Parameters

If you include **INCLUDE_SNTPC**, you can configure the SNTP Server at build time using the following parameter:

SNTPC_PORT — SNTP Client Port
Port Used for SNTP Communication
Default: 123
Valid Values: a **uint**, a port number

9.3.2 Using the SNTP Server

To include the SNTP server, reconfigure your VxWorks stack. The relevant configuration parameter is **INCLUDE_SNTPS**. If the image includes the SNTP server, it automatically calls **sntpsInit( )** during system startup.

Depending on the value of the SNTP Server Mode Selection (set by the configuration parameter **SNTPS_MODE**), the server executes either in **SNTP_PASSIVE** or **SNTP_ACTIVE** mode. In **SNTP_PASSIVE** mode, the server waits for requests from clients and sends replies containing an NTP timestamp. In **SNTP_ACTIVE** mode, the server periodically transmits NTP timestamp information at fixed intervals.

When executing in active mode, the SNTP server determines the target IP address and broadcast interval using the configuration parameters:

- **SNTPS_DSTADDR** — SNTP Server Destination Address
- **SNTPS_INTERVAL** — SNTP Server Update Interval

By default, the server transmits the timestamp information to the local subnet broadcast address every 64 seconds. To change these settings after system startup, call the **sntpsConfigSet( )** routine. The SNTP server operating in active mode can also respond to client requests as they arrive.

The SNTP Client/Server Port (configuration parameter **SNTP_PORT**) assigns the source and destination UDP port. The default port setting is 123 as specified by the RFC 1769.

Finally, the SNTP server requires access to a reliable external time source. To do this, you must provide a routine of the form:

```
STATUS sntpsClockHook (int request, void *pBuffer);
```

Until this routine is hooked into SNTP, the server cannot provide timestamp information. There are two ways to hook this routine into the SNTP server. The first is to configure VxWorks with the SNTPS Time Hook (configuration parameter **SNTPS_TIME_HOOK**) set to the appropriate routine name. You can also call **sntpsClockSet( )**. For more information, see the **sntpsClockSet( )** reference entry.

⚠ **CAUTION:** In the VxWorks AE protection domain model, hook routines like **sntpsClockHook( )** must be within the kernel domain.

Summary of SNTP Server Build-Time Configuration Parameters

If you include **INCLUDE_SNTPS**, you can configure the SNTP Server at build time using the following parameters:

SNTPS_PORT—SNTP Server Port
Port used for SNTP communication.
Default: 123 (as specified by the RFC 1769)
Valid Values: a **uint**, a port number

SNTPS_DSTADDR—STNP Server Destination Address
Recipient for active mode updates (NULL for broadcasts).
Default: NULL
Valid Values: a **string**, an IP address expressed in dot notation

SNTPS_INTERVAL—SNTP Server Update Interval
For active mode, update interval in seconds.
Default: 64
Valid Values: a **uint**, a count of seconds

SNPTS_MODE—SNTP Server Mode Selection
Determines whether server sends unsolicited update messages.
Default: **SNTP_ACTIVE**
Valid Values: a **uint**, either **SNTP_ACTIVE** or **SNTP_PASSIVE**

SNTPS_TIME_HOOK—SNTP Time Hook Function
Name of required clock access routine.
Default: NULL
Valid Values: a **FUNCPTR**, the name of a clock access routine

10
Integrating a New Network Interface Driver

10.1 Introduction

The purpose of the MUX is to provide an interface that insulates network services from the particulars of network interface drivers, and vice versa. Currently, the MUX supports two network driver interface styles, the Enhanced Network Driver (END) interface and the Network Protocol Toolkit (NPT) driver interface.

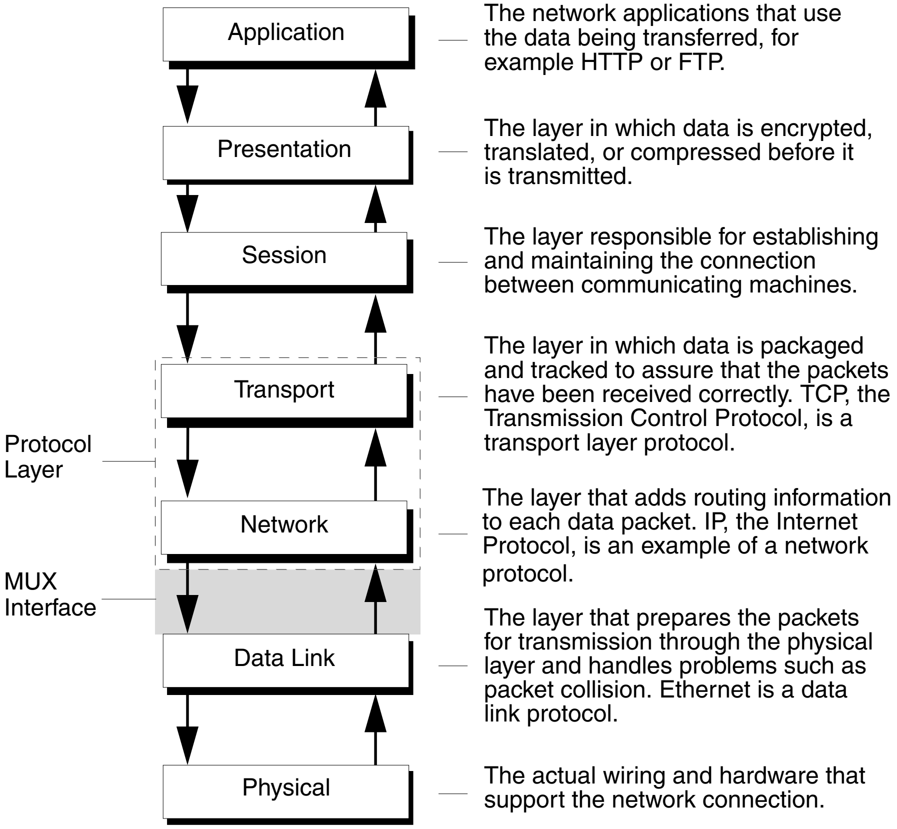
The END is the original MUX network driver interface. ENDs are frame-oriented drivers that exchange frames with the MUX. The NPT-style drivers are packet-oriented drivers that exchange packets with the MUX. These packets are stripped of all datalink layer information. Currently, all network drivers supplied by Wind River are ENDs, as is the generic driver template defined in **templateEnd.c**. There is no generic template for an NPT driver.

Support for the NPT-style driver is part of a set of MUX extensions designed to facilitate the implementation of MUX-compatible network services. These extensions include a registration mechanism for an alternative address resolution utility and support for back ends that let you extend the sockets API so that applications can use sockets to access a new network service. Both of these features are described in *11. Integrating a New Network Service*.

10.1.1 The MUX and the OSI Network Model

The OSI Network Model describes seven layers through which data passes when it is transmitted from an application on one machine to a peer on a remote machine reachable through a network. Transmitted data passes through these layers in the order shown in Figure 10-1.

Figure 10-1 **The OSI Network Model and the MUX**


Application
Presentation
Session
Transport
Network
Data Link
Physical
Protocol Layer
MUX Interface
The network applications that use the data being transferred, for example HTTP or FTP.
The layer in which data is encrypted, translated, or compressed before it is transmitted.
The layer responsible for establishing and maintaining the connection between communicating machines.
The layer in which data is packaged and tracked to assure that the packets have been received correctly. TCP, the Transmission Control Protocol, is a transport layer protocol.
The layer that adds routing information to each data packet. IP, the Internet Protocol, is an example of a network protocol.
The layer that prepares the packets for transmission through the physical layer and handles problems such as packet collision. Ethernet is a data link protocol.
The actual wiring and hardware that support the network connection.

Starting in the application layer, data passes down through each layer of the stack to the physical layer, which handles the physical transmission to the remote machine. After arriving on the remote machine, data passes up through each layer from the physical to the application.

In the abstract, each layer in the stack is independent of the other layers. A protocol in one layer exchanges messages with peers in the same layer on remote machines by passing the message to the layer immediately below it. Whether the message passes down through other layers is not its concern. Ideally, the protocol is insulated from such details.

In practice, network stacks that implement each layer with perfect independence are rare. Within TCP/IP, the protocols that manage the Transport and Network layer functions are sufficiently coupled that they effectively comprise a single layer, which this manual sometimes refers to as the protocol layer.

The MUX is an interface between the datalink and this protocol layer. However, the MUX is not a new layer. There are no MUX-level protocols that communicate with peers in the MUX of a remote machine. The MUX concerns itself solely with standardizing communication between the protocol and datalink layers of a single stack. Because of the MUX, no protocol or network driver needs direct knowledge of the other's internals.

For example, when a network driver needs to pass up a received packet, the driver does not directly access any structure or function within the destination network service. Instead, when the driver is ready to pass data to the service, the driver calls a MUX function that handles the details. The MUX does this by calling the receive function that the network service registered with the MUX. This design lets any MUX-compatible network service use any MUX-compatible network driver.

10.1.2 The Protocol-to-MUX Interface

To interact with the MUX, a protocol or service calls either **muxBind( )** or **muxTkBind( )**. These routines bind the protocol or service to at least one network driver through the MUX. Within the bind call, the network protocol or service supplies pointers to functions that the MUX uses to:

- shut down the service
- pass an error message up to the service
- pass a packet up to the service
- restart the service

The exact prototypes for these functions vary slightly depending on whether you use **muxBind( )** or **muxTkBind( )**. After the protocol or service has bound itself to a driver through the MUX, it can then call MUX-supplied functions, such as **muxSend( )** or **muxTkSend( )**, to transmit a packet or request other MUX services.

When working with the default VxWorks stack, you do not need to make a direct call to **muxBind( )**. This is handled for you by the internals of **ipAttach( )**. For the boot device, the built-in TCP/IP stack initialization code automatically calls **ipAttach( )**. To bind the VxWorks stack to additional interfaces, you must make an explicit **ipAttach( )** call for each additional interface. Again, the internals of **ipAttach( )** handle the **muxBind( )** call.

To free a network service from a binding to a driver in the MUX, you can call **muxUnbind( )**. Although, if the binding was created using **ipAttach( )**, you should call **ipDetach( )**, which handles the **muxUnbind( )** call and other details for you.

10.1.3 The Datalink-to-MUX Interface

To add an END or NPT device to the MUX, call **muxDevLoad( )** for each driver you want to add. This is done for you automatically in the standard network initialization code if you name the driver's load function in the **endDevTbl[ ]**. The stack initialization code uses this function name as input to a **muxDevLoad( )** call.

Internally, the driver's load function must allocate and partially populate an **END_OBJ** structure and a **NET_FUNCS** structure. The **END_OBJ** structure provides the MUX with a description of the device, and the **NET_FUNCS** structure provides the MUX with pointers to the driver's standard MUX interface functions, *x***Start( )**, *x***Stop( )**, *x***Receive( )**, *x***Ioctl( )**, and so on.

After a driver is loaded in to the MUX and has been bound to a protocol, it can pass received packets up to the MUX by calling **muxReceive( )**, if it is an END, or **muxTkReceive( )**, in NPT drivers.[1]

NOTE: The standard VxWorks stack expects to borrow the buffers it receives and thus avoid data copying. If a device cannot transfer incoming data directly into clusters, the driver must explicitly copy the data from private memory into a cluster in sharable memory before passing it in an **mBlk** up to the MUX.

The receive function calls the *stack***RcvRtn( )** function registered by the network service to which the driver wants to send a packet. If the **mux[Tk]Receive( )** call returns **OK**, the driver can consider the data delivered. After a packet is delivered, the driver must free or schedule a free (pending a transmit interrupt) of the buffers it allocated for the packet.

To remove a network interface from the MUX, call **muxDevUnload( )**.

10.1.4 How ENDs and NPT Drivers Differ

The NPT driver is a packet-oriented equivalent to the frame-oriented END. Both the NPT driver and the END are organized around the **END_OBJ** and the

[1] The **mux[Tk]Receive( )** calls in the shipped ENDs and the template END are hard to identify as such when casually reading the code. When passing a packet up to the MUX, each of these drivers uses the function pointer referenced in the **receiveRtn** member of its **END_OBJ**. An earlier call to **muxDevLoad( )** set the **receiveRtn** member to **muxReceive( )** or **muxTkReceive( )**, whichever was appropriate.

NET_FUNC structures, and both driver styles require many of the same entry points:

*x***Load( )** – load a device into the MUX and associate a driver with the device

*x***Unload( )** – release a device, or a port on a device, from the MUX

*x***Send( )** – accept data from the MUX and send it on towards the physical layer

*x***MCastAddrDel( )** – delete a multicast address registered for a device

*x***MCastAddrGet( )** – get a list of multicast addresses registered for a device

*x***MCastAddrAdd( )** – add a multicast address to those registered for a device

*x***PollSend( )** – send packets in polled mode rather than interrupt-driven mode

*x***PollReceive( )** – receive frames in polled rather than interrupt-driven mode

*x***Start( )** – connect device interrupts and activate the interface

*x***Stop( )** – stop or deactivate a network device or interface

*x***Ioctl( )** – support various *ioctl* commands[2]

*x***Bind( )** – exchange data with the protocol layer at bind time (optional)[3]

For the most part, the prototypes for these entry points are identical. The exceptions are the send and receive entry points.

- NPT send entry points take these additional parameters:
  - a MAC address character pointer
  - a networks service type value
  - a **void*** pointer for any network service data the driver might need in order to prepare the packet for transmission on the physical layer
- NPT receive entry points likewise take additional parameters:
  - a frame type
  - a pointer to the start of the network frame
  - a **void*** pointer for any addition network service data that is important to the protocol layer

[2] Although the API for both the END and the NPT *x***Ioctl( )** are identical, the NPT *x***Ioctl( )** must support two extra *ioctl* commands, **EIOCGNPT** and **EIOCGMIB2233**.

[3] The *x***Bind( )** entry point was not a part of the original END design. It was added with the NPT enhancements, but the MUX supports its use in an END.

The three END entry points not included in an NPT driver are:[4]

*x***AddressForm( )** – add addressing information to a packet

*x***AddrGet( )** – extract the addressing information from a packet

*x***PacketDataGet( )** – separate the addressing information and data in a packet

The above functions were removed from the NPT driver because they are frame-oriented and so irrelevant to a packet-oriented driver.

The following registration interface lets you manage an address resolution function for a protocol/interface pair.

muxAddrResFuncAdd( ) – add an address resolution function

muxAddrResFuncGet( ) – get the address resolution function for ifType/protocol

muxAddrResFuncDel( ) – delete an address resolution function

For Ethernet devices, the standard VxWorks implementation automatically assigns **arpresolve( )** as the address resolution function. If you are writing an END that does not run over Ethernet, you also need to implement the *x***AddressForm( )**, *x***AddrGet( )**, and *x***PacketDataGet( )** entry points explicitly. ENDs running over Ethernet typically use the **endLib** implementations of these functions.

10.1.5 Managing Memory for Network Drivers and Services

The default VxWorks stack uses **netBufLib** to manage its internal system and data memory pools. Similarly, almost all the shipped ENDs use **netBufLib** to manage memory pools for their receive buffers. If the standard **netBufLib** implementation is not suitable to your needs, you can replace it. If you are careful to preserve the API of the current **netBufLib** routines, the standard VxWorks stack and the shipped ENDs should be able to use your new memory allocation routines without modification.

NOTE: The standard VxWorks stack expects to borrow the buffers it receives and thus avoid data copying. If a device cannot transfer incoming data directly into clusters, the driver must explicitly copy the data from private memory into a cluster in sharable memory before passing it in an **mBlk** up to the MUX.

[4] For Ethernet, these functions are implemented in **endLib**. Thus, if your driver runs over Ethernet (using either 802.3 or DIX header formats), you can reference the existing functions and do not need to implement them.

10.1.6 Supporting Scatter-Gather in Your Driver

Some devices support breaking up a single network packet into separate chunks of memory. This makes it possible to handle the outgoing network packets as a chain of **mBlk**/**clBlk**/cluster constructs without any copying.

When a driver gets a chain of **mBlk**s, it can decide how to transmit the clusters in the chain. If it is able to do a gather-write, it does not need to do any data copying. If it cannot, then it must collect all of the data from the chain into a single memory area before transmitting it.

10.1.7 Early Link-Level Header Allocation in an NPT Driver

The NPT driver and its MUX support functions are automatically configured to allocate extra room at the beginning of an outgoing packet to hold the network- and transport-layer header information. This allows those layers to copy in their headers without additional overhead in the form of memory allocation and buffer chaining.

You can configure the NPT-supporting MUX functions to also allocate extra room for the data-link-layer header by setting the **USR_MAX_LINK_HDR** configuration parameter or **#define** in the **configNet.h** file for your BSP. You should set this value to the largest of the data-link-layer header sizes used by the drivers in your system. For instance, if you add the following line:

```
#define USR_MAX_LINK_HDR 16
```

sixteen extra bytes will be prepended to outgoing packets, and drivers with data-link-layer headers of 16 bytes or fewer will have that space available in the packet without having to prepend a new **mBlk** during their **endAddressForm( )** or *npt***Send( )** functions.

NOTE: The **USR_MAX_LINK_HDR** configuration parameter only applies to header information within the **AF_INET** addressing family.

When using the **M_PREPEND( )** macro to add a header to a packet, this extra space will be automatically used (or, if the space has not been pre-allocated, a new **mBlk** will automatically be generated and prepended). See *A.6 Macros for Buffer Manipulation*, p.255.

10.1.8 Buffer Alignment

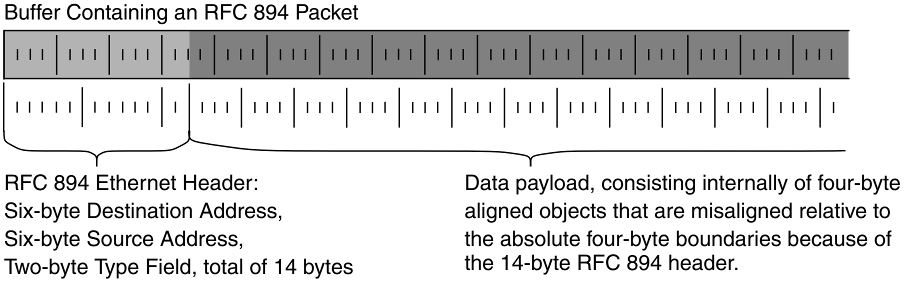
Some microprocessors, most notably those from MIPS and ARM, restrict data access for long words (32-bit values) to the four-byte boundary. Accessing the data in a long word at any other point, such as at a two-byte boundary, results in a segmentation fault. When this restriction is applied to buffers, it requires that all long-word fields within the buffer must align on absolute four-byte boundaries.[5]

Many protocols (IP, TCP, and the like) specify four-byte fields in their headers. Conveniently, these protocol packets are usually set up so that these four-byte fields align on four-byte boundaries relative to the start of the protocol packet. Thus, if your network driver passes up a buffer in which the payload data (for example, an IP packet) always starts at an absolute four-byte boundary, the fields within the data should align correctly.

⚠ **WARNING:** When working with alignment-sensitive hardware, if the payload data contains fields that the stack accesses as long words, those fields must align on absolute four-byte boundaries. Otherwise, the stack crashes when it tries to access those long word fields.

Unfortunately, when a packet first arrives, its datalink layer header might not end on an absolute four-byte boundary. For example, the RFC 894 Ethernet packet (see Figure 10-2) has a 14-byte header that precedes the payload data. If you receive this type of Ethernet packet on a long word boundary, any four-byte aligned payload data (such as an IP packet) following such a header would be misaligned because the payload data would start on a two-byte boundary. If you pass this misaligned data up to the MUX, the stack crashes.

Figure 10-2 **RFC 894 Ethernet Packets Are Problematic for Alignment-Sensitive Hardware**


Buffer Containing an RFC 894 Packet
RFC 894 Ethernet Header:
Six-byte Destination Address,
Six-byte Source Address,
Two-byte Type Field, total of 14 bytes
Data payload, consisting internally of four-byte
aligned objects that are misaligned relative to
the absolute four-byte boundaries because of
the 14-byte RFC 894 header.

[5] If the buffer contains a four-byte region that you access as an array of 8-bit or 16-bit values, there is no alignment restriction. The restriction applies only when you access long words.

To guarantee correctly aligned payload data, your driver can:

- copy the data (the worst-case solution)
- offset the receive buffers (often impossible on hardware that needs it)
- use a scatter-gather receive (the most elegant solution)

Copying the Data

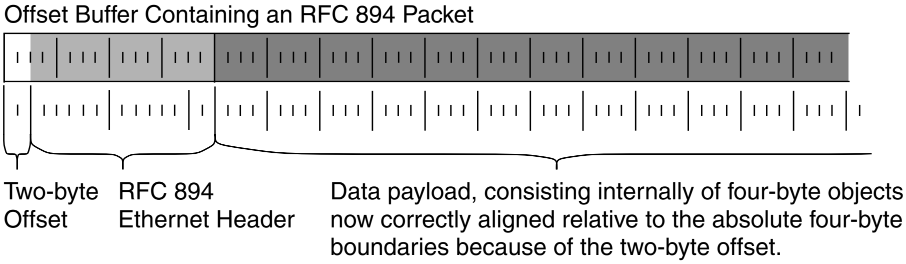
The simplest solution to the alignment problem is for the driver to copy each packet upon reception such that the data aligns properly. The driver can then safely pass the data up to the MUX. Because such a copy is time consuming, it is a worst-case solution.

Offsetting the Receive Buffers

On some hardware, the driver can offset the receive buffers that it gives to the hardware such that the protocol headers within the received packets start at a four-byte boundary. This misaligns the MAC header, but this is not usually a problem. Neither the stack nor the MUX ever access Ethernet header data as anything but eight-bit and 16-bit quantities. Thus, the access restriction that applies to long words only does not apply.

To succeed, this approach relies on the ability to DMA data into a misaligned (non four-byte aligned) buffer. Unfortunately, some DMA controllers do not allow misaligned buffers. In fact, on boards that have CPUs with alignment issues, these issues are usually pushed down into the rest of the hardware infrastructure, including the DMA engines. Thus, this approach is often impossible on the very hardware that requires it.

Figure 10-3 **Receiving an RFC 894 Packet into an Offset Buffer**

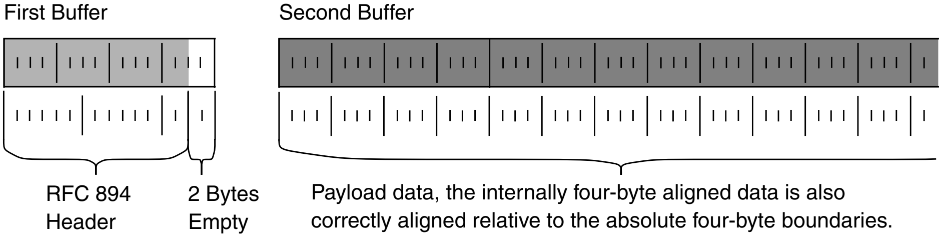

Offset Buffer Containing an RFC 894 Packet
Two-byte Offset
RFC 894 Ethernet Header
Data payload, consisting internally of four-byte objects now correctly aligned relative to the absolute four-byte boundaries because of the two-byte offset.

In Figure 10-3, the Ethernet header no longer starts on a four-byte boundary. Therefore, neither the destination address nor the type fields in the Ethernet header fall on four-byte boundaries. However, because the stack does not access either of these fields as long words, no segmentation fault occurs.

Using a Scatter-Gather Receive

If the hardware can handle scatter-gather lists, your driver can use this feature to handle the alignment problem elegantly. For each packet, your driver allocates two buffers: one buffer for the MAC layer header data (Ethernet header), and a second buffer for the payload data (such as an IP datagram). See Figure 10-4.

Figure 10-4 **Receiving an RFC 894 Packet into Two Linked Scatter-Gather Buffers**


First Buffer
Second Buffer
RFC 894 Header
2 Bytes Empty
Payload data, the internally four-byte aligned data is also correctly aligned relative to the absolute four-byte boundaries.

On reception, the device places the MAC header into one buffer and the payload data into the other. The driver receive function then links these together and processes them as a buffer chain in which each component buffer automatically starts on a four-byte boundary. Therefore, when the driver hands the buffer chain up to the MUX, the payload data is correctly aligned.

10.2 END Implementation

This section presents an overview of how an END operates followed by implementation recommendations for the standard END entry points. If you compare this section with *10.3 NPT Driver Implementation*, p.203, you will notice a strong parallelism. This is because the NPT is a generalized extension of the END. Thus, the implementation recommendations for both driver styles are nearly identical. However, the few differences that do exist are critical.

If you are writing an END, you are writing a driver that passes its frames up to the standard VxWorks implementation. As a starting point for your END, you should use the generic END in **templateEnd.c**.

10.2.1 END Operation

This subsection presents an overview of the following END operations:

- adding an END to VxWorks
- launching an END
- responding to a service bind event
- receiving frames

NOTE: For instructions on starting additional drivers at run time, see *Manually Starting Additional Network Interfaces at Run-Time*, p.67.

For the most part, the NPT and the END handle these operations identically. The major exception is in the area of receiving frames. NPT drivers pass up the received frames stripped of all datalink header information. ENDs include the frame header information in the packets they pass up to the MUX.

Adding an END to VxWorks

Adding your driver to the target VxWorks system is much like adding any other component. The first step is to compile and include the driver code in the VxWorks image. A description of the general procedures can be found in the *Tornado User's Guide*.

Because VxWorks allows you to create more than one network device, you must set up a table that groups the **#define** statements that configure these devices into device-specific groups. This table, **endDevTbl[ ]**,is defined in the **configNet.h** file in your **target/src/config/***bspname* directory, where *bspname* is the name of your board support package, such as **mv162** or **pc486**. For example, to add an ln7990 END, you would edit **configNet.h** to contain lines such as:

```
/* Parameters for loading ln7990 END supporting buffer loaning. */
#define LOAD_FUNC_0 ln7990EndLoad
#define LOAD_STRING_0 "0xfffffe0:0xffffffe2:0:1:1"
#define BSP_0 NULL
```

You should define three constants, like those shown above, for each of the devices you want to add. To set appropriate values for these constants, consider the following:

LOAD_FUNC
Specifies the name of your driver's *end***Load( )** entry point. For example, if your driver's *end***Load( )** entry point were **ln7990EndLoad( )**, you would edit **configNet.h** to include the line:[6]

```
#define LOAD_FUNC_n ln7990EndLoad
```

LOAD_STRING
Specifies the initialization string passed into **muxDevLoad( )** during network initialization as the *initString* parameter. This string is passed along blindly to the *end***Load( )** function of the driver, and its contents depend on what the driver expects.

You must also edit the definition of the **endDevTbl[ ]** (a table in **configNet.h** that specifies the ENDs included in the image) to include entries for each of the devices to be loaded:

```
END_TBL_ENTRY endDevTbl [] =
{
{ 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, NULL, FALSE },
{ 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, NULL, FALSE },
{ 0, END_TBL_END, NULL, 0, NULL, FALSE },
};
```

The first number in each table entry specifies the unit number for the device. The first entry in the example above specifies a unit number of 0. Thus, the device it loads is *deviceName***0**. The **FALSE** at the end of each entry indicates that the entry has not been processed. After the system has successfully loaded a driver, it changes this value to **TRUE** in the run-time version of this table. To prevent the system from automatically loading your driver, set this value to **TRUE**.

At this point, you are ready to rebuild VxWorks to include your new drivers. When you boot this rebuilt image, the system calls **muxDevLoad( )** for each device specified in the table in the order listed.

NOTE: The **endDevTbl[ ]** can contain a mix of NPT drivers and ENDs.

[6] Do not confuse END entry points, indicated as *end***Load( )**, *end***Start( )**, and so on, with functions in **endLib**, an END support library that defines functions such as **endTxSemTake( )** and **endTxSemGive( )**.

Launching the Driver

At system startup, the VxWorks kernel spawns the user root task, which initializes the network. This task calls **muxDevLoad( )**, which calls the *end***Load( )** function in your driver. This *end***Load( )** function creates and partially populates an **END_OBJ** structure that describes the driver. Among the information that *end***Load( )** must supply in the **END_OBJ** is a reference to a **NET_FUNCS** structure that *end***Load( )** has allocated and populated with references to the driver entry points.

After **muxDevLoad( )** loads your driver, a **muxDevStart( )** call executes the *end***Start( )** function in your driver. The *end***Start( )** function should activate the driver and register an interrupt service routine for the driver with the appropriate interrupt connect routine for your architecture and BSP.

Binding to a Service

An END typically does not react when a service uses binds to a device. However, an END may take advantage of the same facility used by an NPT driver to exchange data with a service during the bind phase. See *Responding to Network Service Bind Calls*, p.205, for more complete information on this process.

Receiving Frames in Interrupt Mode

When an interrupt is received, VxWorks invokes the interrupt service routine (ISR) that was registered by the *end***Start( )** function. This ISR should do the minimum amount of work necessary to transfer the frame from the local hardware into accessible memory (ideally a cluster: see *10.1.3 The Datalink-to-MUX Interface*, p.182).

To minimize interrupt lockout time, your ISR should handle directly (at interrupt level) only those actions that require minimum execution time, such as error checking or device status change. The ISR should queue all time-consuming work for processing at task level.

To queue frame reception work for processing at the task level, your ISR can use **netJobAdd( )**. This function takes a function pointer and up to five additional arguments (representing parameters to the function referenced by the function pointer)[7].

[7] You cannot call **netJobAdd( )** from outside of the kernel protection domain.

The **netJobAdd( )** function prototype is:

```
STATUS netJobAdd
    (
    FUNCPTR  routine,
    int      param1,
    int      param2,
    int      param3,
    int      param4,
    int      param5
    )
```

The *routine* in this case should be the function in your driver that performs frame processing at the task level. The **netJobAdd( )** function puts that function on **tNetTask**'s work queue and gives a semaphore that awakens **tNetTask**.

Upon awakening, **tNetTask** dequeues function calls and associated arguments from its work queue. It then executes these functions in its context until the queue is empty.

Your task-level frame reception function should do whatever is necessary to construct an **mBlk** chain containing the frame to hand off to the MUX, such as assuring data coherency. This function might also use a level of indirection in order to check for and avoid race conditions before it attempts to do any processing on the received data. This routine should also set **M_MCAST** and **M_BCAST** flags in the **mBlk** header, if appropriate. When all is ready, your driver passes the frame up to the MUX by calling the function referenced in the **receiveRtn** member of the **END_OBJ** structure that represents your device (see *B.3.3 END_OBJ*, p.285).

10.2.2 The END Interface to the MUX

This subsection describes the driver entry points and the shared data structures that comprise an END's interface to the MUX.

Data Structures Shared by the END and the MUX

The core data structure for an END is the END object, or **END_OBJ**. This structure is defined in **target/h/end.h** (see also *B.3.3 END_OBJ*, p.285). The driver's load function returns a pointer to an **END_OBJ** that it allocated and partially populated. This structure supplies the MUX with information that describes the driver as well as a pointer to a **NET_FUNCS** structure populated with references to the END's standard entry points.

Although the driver's load function is responsible for populating much of the **END_OBJ** structure, some of its members are set within the MUX when a protocol binds to the device. Specifically, the MUX sets the **END_OBJ**'s **receiveRtn** member so that it contains a reference to the bound protocol's receive routine. The driver calls this referenced function when it needs to send a packet up to the protocol. The driver could access this function reference with a call to **muxReceive( )** or **muxTkReceive( )**.

END Entry Points Exported to the MUX

Table 10-1 lists the standard driver entry points that the **NET_FUNCS** structure exports to the MUX. In this manual, the functions use a generic "*end*" prefix, but in practice this prefix is usually replaced with a driver-specific identifier, such as "**ln7990**" for the Lance Ethernet driver.

Table 10-1 **END Functions**

Function	Description
*end***Load( )**	Load a device into the MUX and associate a driver with the device.
*end***Unload( )**	Release a device, or a port on a device, from the MUX.
*end***Send( )**	Accept data from the MUX and send it on to the physical layer.
*end***MCastAddrDel( )**	Remove a multicast address from those registered for the device.
*end***MCastAddrGet( )**	Retrieve a list of multicast addresses registered for a device.
*end***MCastAddrAdd( )**	Add a multicast address to the list of those registered for the device.
*end***PollSend( )**	Send frames in polled mode rather than interrupt-driven mode.
*end***PollReceive( )**	Receive frames in polled mode rather than interrupt-driven mode.
*end***Start( )**	Connect device interrupts and activate the interface.
*end***Stop( )**	Stop or deactivate a network device or interface.
*end***AddressForm( )**	Add addressing information to a packet.
*end***AddrGet( )**	Extract the addressing information from a packet.
*end***PacketDataGet( )**	Separate the addressing information and data in a packet.
*end***Ioctl( )**	Support various *ioctl* commands.

*end***Load( )**

Before a network interface can be used to send and receive frames, the appropriate device must be loaded into the MUX and configured. The **muxDevLoad( )** function calls your driver's *end***Load( )**.

This function takes an initialization string, the contents of which are user-defined but generally include such items as the unit number identifying the physical interface[8], an interrupt vector number, and the address of memory mapped registers.

The *end***Load( )** function must be written as a two-pass algorithm. The MUX calls it twice during the load procedure. In the first pass, the initialization string is blank (all zeros). The *end***Load( )** routine is expected to check for the blank string and return with the name of the device copied into the string. A second call is then made to *end***Load( )** with the actual initialization string that was supplied to **muxDevLoad( )**. The *end***Load( )** then must return a pointer to the **END_OBJ** that it allocates or a **NULL** if the load fails.

Typically, the *end***Load( )** function, in its second pass, will:

- Initialize the device and interface.
- Allocate and fill the **END_OBJ** structure.
- Initialize any necessary private structures.
- Parse and process the initialization string.
- Create and populate the MIB II interface table.
- Create and initialize a private pool of memory using the API in **netBufLib**.
- Allocate one or more network buffer pools using **netBufLib**.
- Fill the **NET_FUNCS** table referenced by **pNetFuncs** in the **END_OBJ** structure.

The *end***Load( )** function is based on the following skeleton:

```
END_OBJ * endLoad
    (
    char *   initString,  /* defined in endTbl */
    void *   pBsp         /* BSP-specific information (optional) */
    )
{
    END_OBJ * newEndObj;

    if( !initString )             /* initString is NULL, error condition */
    {
     /* set errno perhaps */
```

[8] Although a driver is only loaded once, the driver's *end***Load( )** routine will be called for each port to be activated within the driver. This is so the MUX may allocate an entry for each port. The MUX interface does not impose restrictions on how drivers handle multiple ports as long as a separate **END_OBJ** is allocated for each port.

```
        return( (END_OBJ *) 0 );
        }
        else if( initString[0] == 0 )  /* initString[0] is NULL, pass one */
        {
        strcpy( initString, "foo" );
        return( (END_OBJ *) 0 );
        }
        else                           /* initString is not NULL, pass two */
        {
        /* initialize device */
        newEndObj = (END_OBJ *) malloc( sizeof(END_OBJ) );
        /* fill newEndObj and newEndObj->pFuncTable */
        /* create and populate the MIB2 interface table */
        /* initialize any needed private structures */
        /* parse and process initString, and pBsp if necessary */
        /* create a private pool of memory using netBufLib API */
        /* create network buffer pools using netBufLib API */
        return( newEndObj );
        }
    }
```

*end*Unload( )

An *end***Unload( )** function is invoked when **muxDevUnload( )** is called by the system application. In this routine, the driver is responsible for doing whatever is necessary to "release" the device.

This function is called for each port that has been activated by a call to *end***Load( )**. If the device has multiple ports loaded, the driver must not free up any shared resources until an unload request has been received for each of the loaded ports.

The *end***Unload( )** routine does not need to notify services about unloading the device. Before calling *end***Unload( )**, the MUX sends a shutdown notice to each service attached to the device.

The *end***Unload( )** prototype is:

```
STATUS endUnload
    (
    END_OBJ *  pEND /* END object */
    )
```

*end*Send( )

The network driver send routine is referenced from the **NET_FUNCS** table that is created during the load process. The MUX calls this function when the network service issues a send request. The send routine is supplied a reference to an **mBlk** chain representing the link-level frame to be sent.

The *end***Send( )** prototype is:

```
STATUS endSend
    (
    END_OBJ *  pEND,     /* END object */
    M_BLK_ID   pPkt      /* mBlk chain containing the frame */
    )
```

This function should return a status of:

- **OK** , if the send was successful.
- **END_ERR_BLOCK**, if the send could not be completed because of a transient problem such as insufficient resources.
- **ERROR**, in this case, **errno** should be set appropriately.

*end*MCastAddrAdd( )

This function registers a physical-layer multicast address with the device. It takes as arguments a pointer to the **END_OBJ** returned by *end***Load( )** and a string containing the physical address to be added.

This routine should reconfigure the interface in a hardware-specific way that lets the driver receive frames from the specified address.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *end***MCastAddrAdd( )** prototype is:

```
STATUS endMCastAddrAdd
    (
    END_OBJ *  pEND,      /* END object */
    char *     pAddress   /* physical address or a reference thereto */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*MCastAddrDel( )

This function removes a previously registered multicast address from the list maintained for a device. It takes as arguments a pointer to the **END_OBJ** returned by *end***Load( )**, and a string containing the physical address to be removed.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *end***MCastAddrDel( )** prototype is:

```
STATUS endMCastAddrDel
    (
    END_OBJ *  pEND,     /* END object */
    char *     pAddress  /* physical address, or a reference thereto */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*MCastAddrGet( )

This function retrieves a list of all multicast addresses that are currently active on the device. It takes as arguments a pointer to the **END_OBJ** returned by *end***Load( )**, and a pointer to a **MULTI_TABLE** structure into which the list will be put.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *end***MCastAddrGet( )** prototype is:

```
STATUS endMCastAddrGet
    (
    END_OBJ *      pEND,        /* driver's control structure */
    MULTI_TABLE *  pMultiTable  /* container for address list */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*PollSend( )

This routine provides a polled-mode equivalent to the driver's interrupt-driven send routine. It must either transfer a frame directly to the underlying device, or it must exit immediately if the device is busy or if resources are unavailable.

NOTE: When the system calls your *end***PollSend( )** routine, it is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine must not block or delay because the entire system might halt.

Within your *end***PollSend( )** routine, verify that the device has been set to polled-mode (by a previous *end***Ioctl( )** call). Your *end***PollSend( )** routine should then put the outgoing frame directly onto the network, without queuing the frame on any output queue.

This routine takes as arguments a pointer to the **END_OBJ** structure returned by *end***Load( )** and a reference to an **mBlk** or **mBlk** chain containing the outgoing frame. It should return a status **OK** or **ERROR** (in which case, **errno** should be set).

The *end***PollSend( )** prototype is:

```
STATUS endPollSend
    (
    END_OBJ *  pEND,     /* END object*/
    M_BLK_ID   pMblk     /* mBlk chain: data to be sent */
    )
```

*end*PollReceive( )

This function receives frames using polling instead of an interrupt-driven model. The routine retrieves the frame directly from the network and copies it into the **mBlk**s passed to the routine. If no frame is available, the function returns **ERROR**.

NOTE: When the system calls your *end***PollReceive( )** routine, it is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine must not block or delay because the entire system might halt.

Within the *end***PollReceive( )** routine, verify that the device has been set to polled-mode (by a previous *end***Ioctl( )** call). The routine should then retrieve the frame directly from the network and copy it into the **mBlk** passed in to the routine.

It takes as arguments a pointer to the **END_OBJ** structure returned by *end***Load( )** and a reference to an **mBlk** or **mBlk** chain into which the incoming data should be put.

The *end***PollReceive( )** prototype is:

```
STATUS endPollReceive
    (
    END_OBJ *  pEND,     /* returned from endLoad() */
    M_BLK_ID   pPkt      /* mBlk chain: data being received */
    )
```

This function should return **OK** or an error value of **EAGAIN** if the received data is too large to fit in the provided **mBlk**, or if no data is available.

*end*Start( )

The driver's *end***Start( )** function connects device interrupts and makes the interface active and available. This function takes as its argument the unique interface identifier returned from the *end***Load( )** call. As with *end***Load( )**, this call is made for each port that is to be activated within the driver.

The *end***Start( )** prototype is:

```
STATUS endStart
    (
    END_OBJ *  pEND,  /* END object */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*Stop( )

The driver's *end***Stop( )** function halts a network device, typically by disconnecting the appropriate interrupt. It does not remove the device by releasing the allocated data structures. This function takes as its argument the unique interface identifier returned from the *end***Load( )** call.

The *end***Stop( )** prototype is:

```
STATUS endStop
    (
    END_OBJ *  pEND,  /* END object */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*AddressForm( )

The *end***AddressForm( )** routine generates a frame-specific header which it prepends to the **mBlk** chain containing outgoing data. After adding the address segment to the **mBlk**, the routine should adjust the **mBlk.mBlkHdr.mLen** and **mBlk.mBlkHdr.mData** members accordingly.

If the incoming **mBlk** is not large enough to contain the added address information, an additional **mBlk**/**clBlk** cluster must be created for this purpose, and inserted at the beginning of the **mBlk** chain. For information on how to prevent this extra allocation and chaining, see *10.1.7 Early Link-Level Header Allocation in an NPT Driver*, p.185.

The network protocol type can be found in the **pDst.mBlkHdr.reserved** field.

A reference to the new **mBlk** chain head is returned from *end***AddressForm( )**.

The *end***AddressForm( )** prototype is:

```
M_BLK_ID endAddressForm
    (
    M_BLK_ID  pData        /* mBlk chain containing outgoing data */
    M_BLK_ID  pSrc,        /* source address, in an mBlk */
    M_BLK_ID  pDst,        /* destination address, in an mBlk */
    BOOL      bcastFlag    /* use link-level broadcast ? */
    )
```

NOTE: The **endLib** library contains an address formation routine that generates and prepends Ethernet frame headers. Thus, you probably do not need to implement this function if you are running over Ethernet.

*end*AddrGet( )

This routine retrieves the address values for an incoming frame provided in an **mBlk** chain. If the additional **mBlk** parameters are not **NULL**, it sets the **mBlk.mBlkHdr.mData** and **mBlk.mBlkHdr.mLen** fields to indicate the location and size of the corresponding data-link layer addresses. The additional **mBlk** parameters are:

pSrc
The local source address of the frame.

pDst
The local destination of the frame.

pESrc
The original link-level source address, or the *pSrc* settings if none.

pEDst
The final link-level destination address, or the *pDst* settings if none.

NOTE: The **endLib** library contains a routine for retrieving address values from Ethernet frame headers. Thus, you probably do not need to implement this function if your driver runs over Ethernet.

The *end*AddrGet( ) prototype is:

```
STATUS endAddrGet
    (
    M_BLK_ID  pData,  /* mBlk chain containing frame */
    M_BLK_ID  pSrc,   /* local source address, in an mBlk */
    M_BLK_ID  pDest,  /* local destination address, in an mBlk */
    M_BLK_ID  pESrc,  /* actual source address, in an mBlk */
    M_BLK_ID  pEDest  /* actual destination address, in an mBlk */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*PacketDataGet( )

This routine parses an incoming frame provided in an **mBlk** chain. It places the address information, including the size and offset of the address information, the offset of the frame payload, and the frame type, in the **LL_HDR_INFO** structure that is passed in as a parameter.

NOTE: The **endLib** library contains a routine for retrieving the address and data offsets and sizes and the type information from Ethernet frames. Thus, you probably do not need to implement this function if your driver runs over Ethernet.

The *end***PacketDataGet( )** prototype is:

```
STATUS endPacketDataGet
    (
    M_BLK_ID       pData,   /* mBlk chain containing packet */
    LL_HDR_INFO *  pHeader  /* structure to hold header info */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*end*Ioctl( )

An END may need to support *ioctl* commands, particularly if it is to be used with the existing IP network service sublayer. See Table 10-4 for a list of commonly used *ioctl* commands.

WARNING: If you are porting a driver from the BSD 4.3 model, you might be tempted to use the existing *xx***Ioctl( )** routine as your *end***Ioctl( )** routine, skipping the creation of separate routines for the multicast address table maintenance functions. Do not do this! Your driver *must* implement the multicast address table maintenance routines.

The *end***Ioctl( )** function takes the following arguments:

- the unique interface identifier returned from the **muxDevLoad( )** call
- the *ioctl* command being issued
- a buffer for additional data given in the command or for data to be returned on completion of the command (defined as a **caddr_t** structure)

Table 10-2 **ioctl Commands and Data Types**

Command	Function	Data Type
EIOCSFLAGS	Set device flags. See **flags** in *B.3.3 END_OBJ*, p.285.	**int**
EIOCGFLAGS	Get device flags.	**int**
EIOCSADDR	Set device address.	**char ***
EIOCGADDR	Get device address.	**char ***
EIOCMULTIADD	Add multicast address.	**char ***
EIOCMULTIDEL	Delete multicast address.	**char ***
EIOCMULTIGET	Get multicast list.	**MULTI_TABLE ***
EIOCPOLLSTART	Put device in polling mode.	**NULL**
EIOCPOLLSTOP	Put device in interrupt mode.	**NULL**
EIOCGFBUF	Get minimum first buffer for chaining.	**int**
EIOCQUERY	Retrieve the bind function.	**END_QUERY ***
EIOCGHDRLEN	Get the size of the datalink header.	**int**
EIOCGMIB2	Get MIB-II counters from the driver.	**M2_INTERFACETBL ***

NOTE: Wind River reserves command constants (such as **EIOCGFBUF**) that are in the range of 0-128. If you want to use your own custom *ioctl* commands, define their command constants to be equivalent to numbers outside this range.

The *end***Ioctl( )** prototype is:

```
int endIoctl
    (
    END_OBJ *  pEND,      /* END Object */
    int        command,   /* ioctl command */
    caddr_t    buffer     /* holds response from command */
    )
```

This function returns 0 (zero) if successful, an appropriate error value otherwise, or **EINVAL** if the command is not supported.

10.3 NPT Driver Implementation

This section presents an overview of how an NPT driver operates followed by implementation recommendations for the standard NPT driver entry points. If you compare this section with *10.2 END Implementation*, p.188, you will notice a strong parallelism. This arises from the fact that the NPT is a packet-oriented equivalent to the END. Thus, the implementation recommendations for both NPT drivers and ENDs are nearly identical. However, the few differences that do exist are critical.

Currently, VxWorks does not include an NPT driver implementation, only END implementations. As a starting point for an NPT driver, you might want to use the generic END defined in **templateEnd.c**.

NOTE: The design situations that require an NPT driver instead of an END are rare. If you are writing a new driver, think first of implementing it as an END, for which there exists a template as well as working driver implementations that you can study. However, if porting packet-oriented driver, feel free to port it as an NPT driver. The VxWorks stack supports mixing the two driver styles.

10.3.1 NPT Driver Operation

This subsection presents an overview of the following NPT driver operations:

- adding an NPT Driver to VxWorks
- launching an NPT Driver
- responding to a service bind event
- receiving frames

For the most part, the NPT and the END handle these operations identically. The major exception is in the area of receiving frames. NPT drivers pass up the received frames stripped of all datalink header information. ENDs include the Ethernet header information in the packets they pass up to the MUX.

NOTE: For instructions on starting additional drivers at run time, see *Manually Starting Additional Network Interfaces at Run-Time*, p.67.

Adding an NPT Driver to VxWorks

Adding your driver to the target VxWorks system is much like adding any other component. The first step is to compile and include the driver code in the VxWorks image. A description of this procedure is found in the *Tornado User's Guide*.

Because VxWorks allows you to create more than one network device, you must set up a table that groups the **#define** configuration statements for these devices into device-specific groups. This table, **endDevTbl[ ]**, is defined in the **configNet.h** file in your **target/src/config/***bspname* directory where *bspname* is the name of your board support package, such as **mv162** or **pc486**. For example, to add an ln7990 NPT driver, you would edit **configNet.h** to contain lines such as:

```
/* Parameters for loading ln7990 NPT driver supporting buffer loaning. */
#define LOAD_FUNC_0 ln7990nptLoad
#define LOAD_STRING_0 "0xfffffe0:0xffffffe2:0:1:1"
#define BSP_0 NULL
```

Define three constants, like those shown above, for each of the devices you want to add. To set appropriate values for these constants, consider the following:

LOAD_FUNC
Specifies the name of your driver's *npt***Load( )** entry point. For example, if your driver's *npt***Load( )** entry point were **ln7990nptLoad( )**, you would edit **configNet.h** to include the line:

```
#define LOAD_FUNC_n ln7990nptLoad
```

LOAD_STRING
Specifies the initialization string passed into **muxDevLoad( )** as the *initString* parameter. This string contains information that is passed along blindly to the *npt***Load( )** function of the driver, and its contents depend on what the driver expects.

You must also edit the definition of the **endDevTbl[ ]** (a table in **configNet.h** that specifies the drivers included in the image) to list the devices to be loaded:

```
END_TBL_ENTRY endDevTbl [] =
{
{ 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, NULL, FALSE },
{ 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, NULL, FALSE },
...
{ 0, END_TBL_END, NULL, 0, NULL, FALSE },
};
```

The first number in each table entry specifies the unit number for the device. The first entry in the example above specifies a unit number of 0. Thus, the device it loads is *deviceName***0**. The **FALSE** at the end of each entry indicates that the entry has not been processed. After the system successfully loads a driver, it changes this value to **TRUE** in the run-time version of this table. To prevent the system from automatically loading your driver, set this value to **TRUE**.

After constructing these entries in **configNet.h**, you are ready to rebuild VxWorks to include your new drivers. When you boot this rebuilt image, the system calls **muxDevLoad( )** for each device specified in the table in the order listed.

NOTE: The **endDevTbl[ ]** can contain a mix of NPT drivers and ENDs.

Launching the Driver

At system startup, the VxWorks kernel spawns the user root task to initialize the network. The task calls **muxDevLoad( )** which in turn calls the *npt***Load( )** function in your driver. The *npt***Load( )** function creates and partially populates an **END_OBJ** structure and a **NET_FUNCS** structure. The **END_OBJ** structure describes the driver to the MUX. The **NET_FUNCS** structure provides the MUX with references to the MUX-callable driver functions.

After **muxDevLoad( )** loads your driver, a **muxDevStart( )** call executes the *npt***Start( )** function in your driver. The *npt***Start( )** function should activate the driver and register an interrupt service routine for the driver with the appropriate interrupt connect routine for your architecture and BSP.

Responding to Network Service Bind Calls

A driver is not required to respond when a service binds to a device. However, if you want your driver to respond to a bind event, your driver can support an *npt***Bind( )** function.

When a service or protocol binds to an interface controlled by your driver, the MUX uses the driver's *npt***Ioctl( )** function to retrieve a pointer to that driver's *npt***Bind( )** function (if any). The MUX then executes that function before continuing with the bind.

To get a pointer to a driver's *npt***Bind( )**, the MUX issues an **EIOCQUERY** command to the driver's *npt***Ioctl( )** function. As input, the call supplies an **END_QUERY** structure whose members are used as follows:

query
Set by MUX to **END_BIND_QUERY**.

queryLen
Set by MUX to the expected size of the data in **queryData**.

queryData
Set by your driver's *npt***Ioctl( )** to point to the drivers *npt***Bind( )**.

Receiving Frames in Interrupt Mode

When an interrupt is received, VxWorks invokes the interrupt service routine that was registered by the *npt***Start( )** function. This interrupt service routine should do the minimum amount of work necessary to transfer the frame from the local hardware into accessible memory (ideally a cluster: see *10.1.3 The Datalink-to-MUX Interface*, p.182).

To minimize interrupt lockout time, the routine should handle at interrupt level only those tasks that require minimum execution time, such as error checking or device status change. The routine should queue all time-consuming work for processing at task level.

To queue frame reception work for processing at the task level, use the **netJobAdd( )** function. This function takes a function pointer and up to five additional arguments (representing parameters to the function referenced by the function pointer). The **netJobAdd( )** function prototype is: [9]

```
STATUS netJobAdd
    (
    FUNCPTR  routine,
    int      param1,
    int      param2,
    int      param3,
    int      param4,
    int      param5
    )
```

The *routine* in this case should be the function in your driver that completes frame processing at the task level. The **netJobAdd( )** function puts the job request on **tNetTask**'s work queue and gives the appropriate semaphore that awakens **tNetTask**.

Upon awakening, **tNetTask** dequeues function calls and associated arguments from its work queue. It then executes these functions in its context until the queue is empty.

[9] You cannot call **netJobAdd( )** from outside of the kernel protection domain.

Your task-level frame reception function should do whatever is necessary to construct an **mBlk** chain containing the frame to hand off to the MUX, such as assuring data coherency. This routine should also set **M_MCAST** and **M_BCAST** flags in the **mBlk** header if appropriate. When all is ready, your driver passes the frame up to the MUX by calling the function referenced as the **receiveRtn** member of the **END_OBJ** structure representing your device.

10.3.2 NPT Driver Interface to the MUX

This subsection describes the driver entry points and the shared data structures that comprise the NPT driver's interface to the MUX.

Data Structures Used by the Driver

The core data structure for an NPT driver is the END object, or **END_OBJ**. The structure is defined in **target/h/end.h** (see also *B.3.3 END_OBJ*, p.285). The driver's load function returns a pointer to an **END_OBJ** that it allocated and partially populated. This structure supplies the MUX with information that describes the driver as well as a pointer to a **NET_FUNCS** structure populated with references to the NPT's standard entry points.

Although the driver's load function is responsible for populating much of the **END_OBJ** structure, some of its members are set within the MUX when a protocol binds to the device. Specifically, the MUX sets the **END_OBJ**'s **receiveRtn** member so that it contains a reference to the bound protocol's receive routine. The driver calls this referenced function when it needs to send a packet up to the protocol. The driver could access this function reference with a call to **muxReceive( )** or **muxTkReceive( )**.

NPT Driver Entry Points Exported to the MUX

Table 10-3 lists the standard driver entry points that the **NET_FUNCS** structure exports to the MUX. In this manual, the functions use a generic "*npt*" prefix, but in practice this prefix is usually replaced with a driver-specific identifier, such as "**ln7990**" for the Lance Ethernet driver.

*npt***Load( )**

Before you can use a network interface to send and receive frames, you must load the appropriate device driver into the MUX and then configure that driver for the

Table 10-3 **NPT Driver Functions**

Function	Description
*npt***Load( )**	Load a device into the MUX and associate a driver with the device.
*npt***Unload( )**	Release a device, or a port on a device, from the MUX.
*npt***Bind( )**	Exchange data with the protocol layer at bind time. (Optional)
*npt***Send( )**	Accept data from the MUX and send it on to the physical layer.
*npt***MCastAddrDel( )**	Remove a multicast address from those registered for the device.
*npt***MCastAddrGet( )**	Retrieve a list of multicast addresses registered for a device.
*npt***MCastAddrAdd( )**	Add a multicast address to the list of those registered for the device.
*npt***PollSend( )**	Send packets in polled mode rather than interrupt-driven mode.
*npt***PollReceive( )**	Receive frames in polled mode rather than interrupt-driven mode.
*npt***Start( )**	Connect device interrupts and activate the interface.
*npt***Stop( )**	Stop or deactivate a network device or interface.
*npt***Ioctl( )**	Support various *ioctl* commands.

interface. The user root task loads network device drivers into the MUX by calling **muxDevLoad( )** for all the drivers referenced in **endDevTbl[ ]**. The entries in this table provide all the information needed to call **muxDevLoad( )**. This includes the actual name of your driver's *npt***Load( )** function.

As input, the *npt***Load( )** function takes an initialization string. The contents this string are user-defined but generally include such items as the unit number identifying the physical interface[10], an interrupt vector number, and the address of memory mapped registers.

You must write your *npt***Load( )** function as a two-pass algorithm. The MUX calls it twice during the load procedure. In the first pass, it calls your *npt***Load( )** function using a blank (all zeros) initialization string. Your *npt***Load( )** routine is expected to check for the blank string and return with the name of the device copied into the string.

[10] Although a driver is only loaded once, the driver's *npt***Load( )** routine is called for each port to be activated within the driver. This lets the MUX allocate an entry for each port. The MUX interface does not impose restrictions on how drivers handle multiple ports as long as the driver allocates a separate **END_OBJ** for each port.

The MUX then calls your *npt***Load( )** a second time using the actual initialization string that was supplied to **muxDevLoad( )**. Your *npt***Load( )** routine must then return a pointer to the **END_OBJ** that it allocates, or a **NULL** if the load fails.

Typically, the *npt***Load( )** function, in its second pass, does the following:

- Initialize the device and interface.
- Allocate and fill the **END_OBJ** structure.
- Initialize any necessary private structures.
- Parse and process the initialization string.
- Create and initialize a private pool of memory using the API in **netBufLib**.
- Allocate one or more network buffer pools using **netBufLib**.
- Create and populate the MIB II interface table.
- Fill the **NET_FUNCS** table referenced by **pNetFuncs** in the **END_OBJ** structure.

The *npt***Load( )** function is based on the following skeleton:

```
END_OBJ * nptLoad
    (
    char *   initString,  /* defined in endTbl */
    void *   pBsp         /* BSP-specific information (optional) */
    )
{
    END_OBJ * newEndObj;

    if( !initString )              /* initString is NULL, error condition */
    {
     /* set errno perhaps */
     return( (void *) 0 );
    }
    else if( initString[0] == 0 )  /* initString[0] is NULL, pass one */
    {
     strcpy( initString, "foo" );
     return( (void *) 0 );
    }
    else                           /* initString is not NULL, pass two */
    {
     /* initialize device */
     newEndObj = (END_OBJ *) malloc( sizeof(END_OBJ) );
     /* fill newEndObj and newEndObj->pFuncTable */
     /* create and populate a MIB2 interface table */
     /* initialize any needed private structures */
     /* parse and process initString, and pBsp if necessary */
     /* create a private pool of memory using netBufLib API */
     /* create network buffer pools using netBufLib API */
     return( newEndObj );
    }
}
```

*npt*Unload( )

The MUX calls your driver's *npt***Unload( )** when a system application calls **muxDevUnload( )**. In its *npt***Unload( )**, your driver is responsible for doing whatever it takes to "release" the device. It should also free the memory allocated for the END object.

The MUX calls your driver's *npt***Unload( )** for each port that has been activated by a call to the driver's *npt***Load( )**. If the device has loaded multiple ports, the driver's *npt***Load( )** must not free up any shared resources until an unload request has been received for each of the loaded ports.

The *npt***Unload( )** routine does not need to notify services about unloading the device. Before calling *npt***Unload( )**, the MUX first sends a shutdown notice to each service attached to the device.

The *npt***Unload( )** prototype is:

```
STATUS nptUnload
    (
    END_OBJ *  pEND  /* END object */
    )
```

*npt*Bind( )

The *npt***Bind( )** function is an optional driver function that gives your driver the ability to respond to bind events. Using *npt***Bind( )**, your driver can support the exchange of information between a service and a driver whenever the service binds to a device through that driver.

The MUX calls your driver's *npt***Bind( )** function (if any), in response to bind events through that driver. To get an executable reference to a driver's *npt***Bind( )** function, the MUX uses the driver's *npt***Ioctl( )** function.

The MUX calls *npt***Bind( )** while processing a bind request from the network service. Arguments passed to *npt***Bind( )** include:

- a reference to information supplied by the network service
  (although the network service may choose to supply no information at all)
- a reference to a template for network driver information
  (if your driver cares to provide any)
- the network service type

The function is expected to return **OK** if the bind request is accepted. The driver may also reject the bind request by returning **ERROR**, in which case the MUX will deny the bind request to the network service.

The *npt***Bind( )** prototype is:

```
STATUS nptBind
    (
    END_OBJ *   pEND,            /* END object */
    void *      pNetSvcInfo,     /* info provided by the network service */
    void *      pNetDrvInfo,     /* template for network driver info */
    long        type             /* of network service attempting to bind */
    )
```

*npt***Send( )**

The network driver send routine is referenced from the **NET_FUNCS** table. The MUX calls this function when the network service issues a send request. The send routine is supplied a reference to an **mBlk** chain representing the packet to be sent. This routine should prepend the link-level header to the packet (for information on making this more efficient, see *10.1.7 Early Link-Level Header Allocation in an NPT Driver*, p.185).

The *npt***Send( )** prototype is:

```
STATUS nptSend
    (
    END_OBJ *   pEND,         /* END object */
    M_BLK_ID    pMblk,        /* network packet to transmit */
    char *      dstAddr,      /* destination MAC address */
    int         netSvcType,   /* network service type */
    void *      pSpare        /* optional network service data */
    )
```

This function should return a status **OK** if the send was successful, **i** if the send could not be completed because of a transient problem such as insufficient resources, or **ERROR** (in which case, **errno** should be set appropriately).

*npt***MCastAddrAdd( )**

This routine registers a physical-layer multicast address with the device. As arguments, it takes:

– a pointer to the **END_OBJ** structure returned by *npt***Load( )**

– a string containing the physical address to be added
(or a reference to the address to be added)

This routine should reconfigure the interface in a hardware-specific way that lets the driver receive frames from the specified address and pass those frames along.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *npt***MCastAddrAdd( )** prototype is:

```
STATUS nptMCastAddrAdd
    (
    END_OBJ *  pEND,        /* driver's control structure */
    char *     pAddress     /* physical address or a reference thereto */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*npt*MCastAddrDel( )

This routine removes a previously registered multicast address from the list maintained for a device. It takes as arguments a pointer to the **END_OBJ** structure returned by *npt***Load( )**, and a string containing the physical address to be removed or a reference to that address.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *npt***MCastAddrDel( )** prototype is:

```
STATUS nptMCastAddrDel
    (
    END_OBJ *  pEND,        /* END object */
    char *     pAddress     /* physical address, or a reference thereto */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*npt*MCastAddrGet( )

This routine retrieves a list of all multicast addresses that have been registered with the device. It takes as arguments a pointer to the **END_OBJ** structure returned by *npt***Load( )**, and a pointer to a **MULTI_TABLE** structure into which the list will be put.

NOTE: To help you manage a list of Ethernet multicast addresses, VxWorks provides the **etherMultiLib** library.

The *npt***MCastAddrGet( )** prototype is:

```
STATUS nptMCastAddrGet
    (
    END_OBJ *      pEND,        /* END object */
    MULTI_TABLE *  pMultiTable  /* container for address list */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*npt*PollSend( )

This routine provides a polled-mode equivalent to the driver's interrupt-driven send routine. Either it must transfer a frame directly to the underlying device, or it must exit immediately if the device is busy or resources are unavailable.

NOTE: When the system calls your *npt***PollSend( )** routine, it is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine should not block or delay because the entire system might halt.

Within your *npt***PollSend( )** routine, verify that the device has been set to polled-mode (by a previous *npt***Ioctl( )** call). Your *npt***PollSend( )** routine should then put the outgoing packet directly onto the network, without queuing the packet on any output queue.

This routine takes as arguments a pointer to the **END_OBJ** structure returned by *npt***Load( )** and a reference to an **mBlk** or **mBlk** chain containing the outgoing frame.

The *npt***PollSend( )** prototype is:

```
STATUS nptPollSend
    (
    END_OBJ *  pEND,        /* END object */
    M_BLK_ID   pPkt,        /* network packet to transmit */
    char *     dstAddr,     /* destination MAC address */
    long       netType,     /* network service type */
    void *     pSpareData   /* optional network service data */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set).

*npt***PollReceive( )**

This routine receives frames using polling instead of an interrupt-driven model. The routine retrieves the frame directly from the network and copies it into the **mBlk** passed to the routine. If no frame is available, it returns **ERROR**.

NOTE: When the system calls your *npt***PollReceive( )** routine, it is probably in a mode that cannot service kernel calls. Therefore, this routine must not perform any kernel operations, such as taking a semaphore or allocating memory. Likewise, this routine must not block or delay because the entire system might halt.

Within the *npt***PollReceive( )** routine, verify that the device has been set to polled-mode (by a previous *npt***Ioctl( )** call). The routine should then retrieve the frame directly from the network and copy it into the **mBlk** passed in to the routine.

It takes as arguments a pointer to the **END_OBJ** structure returned by *npt***Load( )** and a reference to an **mBlk** or **mBlk** chain into which the incoming data should be put, as well as information about the frame type and the offset within the packet to the network frame.

The *npt***PollReceive( )** prototype is:

```
STATUS nptPollReceive
    (
    END_OBJ * pEND,       /* END object */
    M_BLK_ID  pMblk,      /* received frame */
    long *    pNetSvc,    /* payload/network frame type */
    long *    pNetOffset, /* offset to network frame */
    void *    pSpareData  /* optional network service data */
    )
```

This function should return **OK** or an error value of **EAGAIN** if the received data is too large to fit in the provided **mBlk**, or if no data is available.

*npt***Start( )**

The driver's *npt***Start( )** function connects device interrupts and makes the interface active and available. This function takes as its argument the **END_OBJ** structure pointer returned from the *npt***Load( )** call. As with *npt***Load( )**, this call is made for each port that is to be activated within the driver.

The *npt***Start( )** prototype is:

```
STATUS nptStart
    (
    END_OBJ * pEND,  /* END object */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*npt***Stop( )**

The driver's *npt***Stop( )** routine halts a network device, typically by disconnecting the appropriate interrupt. It does not remove the device by releasing the allocated data structures. It takes as its argument the **END_OBJ** structure pointer returned from the *npt***Load( )** call.

The *npt***Stop( )** prototype is:

```
STATUS nptStop
    (
    END_OBJ *  pEND,  /* END object */
    )
```

This function should return a status **OK** or **ERROR** (in which case, **errno** should be set appropriately).

*npt***Ioctl( )**

An NPT driver must support the *ioctl* command defined as **EIOCGNPT**. This command is used to determine if the driver is of the NPT variety. All the driver needs to do upon receiving this command is to return 0 (zero), indicating success. For other drivers, this ioctl command is undefined and therefore returns **EINVAL**, indicating that it is not an NPT driver.

An NPT driver must also support the **EIOCGMIB2** command, which returns a populated MIB2 interface table.

An NPT driver may also need to support other *ioctl* commands, particularly if it is to be used with the existing IP network service sublayer. See Table 10-4 for a list of commonly used *ioctl* commands.

⚠ **WARNING:** If you are porting a driver from the BSD 4.3 model, you might be tempted to use the existing *xx***Ioctl( )** routine as your *npt***Ioctl( )** routine, skipping the creation of separate routines for the multicast address table maintenance functions. Do not do this! Your driver *must* implement the multicast address table maintenance routines.

The *npt***Ioctl( )** function takes three arguments:

– the **END_OBJ** pointer returned from the *npt***Load( )** call

– the *ioctl* command being issued

– a buffer for additional data given in the command or for data to be returned on completion of the command (defined as a **caddr_t** structure).

Table 10-4 *Ioctl* **Commands and Data Types**

Command	Function	Data Type
EIOCSFLAGS	Set device flags. See **flags** in *B.3.3 END_OBJ*, p.285.	**int**
EIOCGFLAGS	Get device flags.	**int**
EIOCSADDR	Set device address.	**char ***
EIOCGADDR	Get device address.	**char ***
EIOCMULTIADD	Add multicast address.	**char ***
EIOCMULTIDEL	Delete multicast address.	**char ***
EIOCMULTIGET	Get multicast list.	**MULTI_TABLE ***
EIOCPOLLSTART	Set device into polling mode.	**NULL**
EIOCPOLLSTOP	Set device into interrupt mode.	**NULL**
EIOCGFBUF	Get minimum first buffer for chaining.	**int**
EIOCGNPT	Indicate NPT-compliance.	**void**
EIOCQUERY	Retrieve the bind function.	**END_QUERY ***
EIOCGHDRLEN	Get the size of the datalink header.	**int**
EIOCGMIB2	Retrieve *RFC 1213* MIB-II table.	**M2_INTERFACETBL ***
EIOCGMIB2233	Retrieve *RFC 2233* MIB-II table	**M2_ID ***

NOTE: Command constants (such as **EIOGGFBUF**) in the range of 0-128 are reserved for use by Wind River. If you want to use your own custom *ioctl* commands, you should define their command constants to be equivalent to numbers outside of this range.

The *npt***Ioctl( )** prototype is:

```
int nptIoctl
    (
    END_OBJ *  pEND,      /* END object */
    int        command,   /* ioctl command */
    caddr_t    buffer     /* holds response from command */
    )
```

This function returns 0 (zero) if successful, an appropriate error value otherwise, or **EINVAL** if the command is not supported.

In the case where *command* is set to **EIOCQUERY**, the *buffer* will be set to point to an **END_QUERY** structure. The **query** member of this structure will be set to the type of query (for instance, **END_BIND_QUERY**), and the **queryLen** member will be set to the size of the **queryData** buffer. Upon receipt of an **EIOCQUERY** command, you should respond either by copying data into this **queryData** buffer, or by returning an error value such as **EINVAL** from *npt***Ioctl( )**.

Your driver is not required to support **EIOCQUERY** commands, but you may find this a useful way of communicating with the protocol layer.

10.4 Porting a BSD Driver to the MUX

To convert a BSD driver that communicates directly with the protocol layer into one that communicates with the protocol layer through the MUX, you need to make the following changes:

- remove unit number references
- create an END Object to represent the device
- implement the standard END or NPT entry points

When deciding whether to implement an END or NPT driver, choose the interface style that is most convenient for the driver you are porting. If you are porting a frame-oriented driver, the END is likely to be the more convenient driver style.

10.4.1 Remove Unit Number References

Under the MUX, each device is independent. Your BSD model may consider each device to be part of an array of devices, each with a unit number. BSD driver routines are sometimes written to take unit numbers as parameters, and to distinguish between devices based on these unit numbers. In the MUX model, the END Object is the distinguishing feature of devices, and MUX routines distinguish between devices based on the END Object pointer that is passed in to the routines.

10.4.2 Create an END Object to Represent the Device

The head of your driver control object should be an **END_OBJ** structure that includes all hardware- and driver-specific elements.

10.4.3 Implementing the Standard END or NPT Entry Points

The END and NPT models for network interface drivers contain standard entry points that are not present in the BSD model. Table 10-5 shows some of the analogies. You should be able to reuse much of the code from the BSD driver.

Table 10-5 **Required Driver Entry Points and their Derivations**

NPT or END Entry Points	BSD 4.3 Style Entry Points
*x***Load( )**	*xx***attach( )**
*x***Unload( )**	None—see *endUnload( )*, p.195, or *nptUnload( )*, p.210, and **templateEnd.c**.
N/A	*xx***Receive( )**
*x***Send( )**	*xx***Output( )**
*x***Ioctl( )**	*xx***Ioctl( )**
*x***MCastAddrAdd( )**	None—see *endMCastAddrAdd( )*, p.196, or *nptMCastAddrAdd( )*, p.211, and **templateEnd.c**.
*x***MCastAddrDel( )**	None—see *endMCastAddrDel( )*, p.196, or *nptMCastAddrDel( )*, p.212, and **templateEnd.c**.
*x***MCastAddrGet( )**	None—see *endMCastAddrGet( )*, p.197, or *nptMCastAddrGet( )*, p.212, and **templateEnd.c**.
*x***PollSend( )**	N/A—see *endPollSend( )*, p.197, or *nptPollSend( )*, p.213, and **templateEnd.c**.
*x***PollReceive( )**	N/A—see *endPollReceive( )*, p.198, or *nptPollReceive( )*, p.214.

Table 10-5 **Required Driver Entry Points and their Derivations** *(Continued)*

NPT or END Entry Points	BSD 4.3 Style Entry Points
*x***Start( )**	N/A—see *endStart( )*, p.198, or *nptStart( )*, p.214.
*x***Stop( )**	N/A—see *endStop( )*, p.199, or *nptStop( )*, p.215.
*end***AddressForm( )**[*]	N/A—see also *endAddressForm( )*, p.199.
*end***AddrGet( )**[*]	N/A—see also *endAddrGet( )*, p.200.
*end***PacketDataGet( )**[*]	N/A—see also *endPacketDataGet( )*, p.201.

[*] These functions are implemented for Ethernet in **endLib**. If porting the BSD driver to run over Ethernet, you probably do not need to implement these functions.

⚠ **CAUTION:** When porting a BSD network driver to the MUX, you must replace all calls into the protocol with appropriate calls into the MUX. In addition, you must remove all code that implements or uses **etherInputHook( )** or **etherOutputHook( )** routines.

Rewrite xxattach( ) to Use an npt/endLoad( ) Interface

Rewrite the interface of your *xx***attach( )** to match the *npt/end***Load( )** function described in *nptLoad( )*, p.207, or *endLoad( )*, p.194.

Much of the code that handles the specifics of hardware initialization should be the same. However, when allocating the memory for packet reception buffers that are passed up to the service, you should use the MUX buffer management utilities. See *10.1.5 Managing Memory for Network Drivers and Services*, p.184, *A. Using netBufLib* as well as the reference entry for **muxBufInit( )**.

Remove any code your *xx***attach( )** included to support the implementation of the **etherInputHook( )** and **etherOutputHook( )** routines. Etherhooks are no longer supported. Similar functionality is now provided using BPF (see *3.2.1 BPF, the BSD Packet Filter*, p.20).

You may also need to add code that clears out the MIB2 variables in the END Object's **mib2Tbl** structure.

The xxReceive( ) Routine Still Handles Task-Level Packets

Because the MUX does not directly call the driver's packet reception code, there is no *npt/end***Receive( )** entry point. However, your driver still needs to handle packet reception at the task level. Unfortunately, most of the code in this driver routine will require extensive revision. Instead of calling the service directly, this routine uses a MUX-supplied function to pass a packet up to the service. Likewise, your receive routine should use a MUX-managed memory pool as its receive buffer area.

Rewrite xxOutput( ) to Use an npt/endSend( ) Interface

Rewrite the interface of your output routine to match the *npt/end***Send( )** entry point described in *nptSend( )*, p.211 or *endSend( )*, p.195.

Much of the code that dealt directly with putting the packet on the hardware should need little if any revision. However, you should change your code to use **mBlk** chains allocated out of an **netBufLib**-managed memory pool. See the reference entry for **netBufLib** for details.

The xxIoctl( ) Routine is the Basis of npt/endIoctl( )

Rewrite the interface of your *xx***Ioctl( )** to match the *npt/end***Ioctl( )** function described in *nptIoctl( )*, p.215 or *endIoctl( )*, p.201. If your driver used *xx***Ioctl( )** to implement multicasting, you must break that functionality out into the separate *npt/end***MCastAddrAdd( )**, *npt/end***MCastAddrDel( )**, and *npt/end***MCastAddrGet( )** entry points.

Implement All Remaining Required END or NPT Entry Points

Table 10-5 lists a handful of driver points unique to ENDs and NPT drivers. Both an END and an NPT require you to implement the *x***Send( )**, *x***Start( )**, and *x***Stop( )** entry points. There are no BSD equivalents for these entry points. In addition, if you are implementing an END, you must implement entry points for *end***AddressForm( )**, *end***AddrGet( )**, and *end***PacketDataGet( )**. However, these functions are already implemented for Ethernet in **endLib**. If your driver will run over Ethernet, you may use the functions supplied in **endLib**.

10.5 Supporting Multiple Network Interface Drivers

The VxWorks network stack allows you to use multiple network interface cards simultaneously. You can use multiple cards of the same variety, or different types of cards, with a combination of END and NPT drivers.[11]

10.5.1 Configuring VxWorks for Multiple Drivers

To configure VxWorks to support multiple drivers, make sure that the drivers are compiled into your VxWorks image. Follow the directions in *Adding an NPT Driver to VxWorks*, p.204 (for an NPT driver) or *Adding an END to VxWorks*, p.189 (for an END). You may also need to increase the value of the configuration parameters **IP_MAX_UNITS** and **MUX_MAX_BINDS**.

10.5.2 Starting Additional Drivers at Run-Time

To start additional network drivers at run-time:

1. Use **muxAddrResFuncAdd( )** to install an address resolution function if necessary. If your driver does not register itself as an Ethernet driver, and if the link layer requires hardware address resolution, you need to install an address resolution function. See *B.2.1 muxAddrResFuncAdd( )*, p.270.
2. Use **muxBind( )** or **muxTkBind( )** to bind the driver to the service. In the case of IP, the binding is done in the **ipAttach( )** routine.
3. Configure the interface. In the case of IP, this is done with calls to **ifMaskSet( )** and **ifAddrSet( )**.

10.6 Avoiding Memory Leaks

Your driver implementation may allocate a semaphore during its initialization phase and store a reference to it in the END object's **txSem** member. For example, **endObjInit( )**, which is commonly used to initialize ENDs, does this.[12]

[11] Some BSPs and drivers may have their own limitations on the number of interfaces and units they support.

If this semaphore is not deleted when the driver is unloaded, a memory leak will result equal to the size of the semaphore data structure plus any memory allocation overhead.

This may not be an issue for your application, since the amount of memory that leaks is small, but if drivers are loaded and unloaded frequently, this could add up and become a problem.

[12] Please note that **endObjInit( )** is an actual function defined in **endLib.c**. It is not an END entry point, which would have been indicated as *end***ObjInit( )**.

11
Integrating a New Network Service

11.1 Introduction

A network service, such as a network protocol, is an implementation of the network and transport layers of the OSI network model. As shown in Figure 10-1, network services communicate with the data link layer through the MUX interface. Everything specific to the network interface is handled in the drivers of the data link layer, which are described in *10. Integrating a New Network Interface Driver*.

11.2 Writing a Network Service Sublayer

A network service sublayer allows a network service to send and receive packets through the MUX. It may be written as part of a network service, or as a separate element that the service uses. The minimum requirements of a network service sublayer are an initialization routine and routines that support packet transfer and error reporting. Support for flow control is optional.

11.2.1 Interface Initialization

When the system constructs the network stack, it activates network interfaces that include a network service and a network driver. The activation routine that you provide in your network service sublayer, by convention, is named *foo***Attach( )**, where *foo* is replaced by an abbreviation for the network service. This naming convention is a generalization based on the name of **ipAttach( )**. Your *foo***Attach( )** function typically allocates and initializes data structures that represent the interface being attached to, determines the network driver paradigm (END or NPT), and binds to the driver interface through the MUX.

Determining the Driver Paradigm

To determine a driver's operating paradigm, use the **muxTkDrvCheck( )** function (see *B.2.17 muxTkDrvCheck( )*, p.280). If your network service supports only NPT devices, its *foo***Attach( )** routine should return an error if **muxTkDrvCheck( )** returns **FALSE**.

The Bind Phase

The network service must bind to a driver before it can send and receive packets through it. Binding to a network driver is accomplished by calling the **muxTkBind( )** function (see *B.2.16 muxTkBind( )*, p.278).[1]

The protocol type supplied to the bind function is used by the MUX to prioritize the services, and determines which service sees which packets. A **MUX_PROTO_SNARF** type service sees all the packets that are processed by any driver to which it is bound. A **MUX_PROTO_PROMISC** type service sees a packet only after all other services bound to a driver have had a chance to consume it. A **MUX_PROTO_OUTPUT** type service sees outgoing rather than incoming packets. Any other type value configures the service to see only packets of the specified type.

After the bind operation has complete successfully, the sublayer should determine which (if any) registered service address mapping functions are relevant to the network interface. The sublayer should obtain references to those that apply and retain these references for later use within the interface.

Network Address Resolution Function

The address resolution function translates a network service address to a network driver (physical layer) address. To find the address resolution function that applies to a specific network service/network driver pair, use the **muxAddrResFuncGet( )** function (see *B.2.3 muxAddrResFuncGet( )*, p.272).

Typically, the network service performs address resolution when it sends a packet to the network driver. The sublayer uses the address resolution function obtained

[1] The **muxBind( )** function may also be used, but only with ENDs. The **muxBind( )** function also requires you to implement a slightly different set of network service sublayer functions than those that are used with **muxTkBind( )**.

in this step to resolve the address. If an address resolution function does not exist for this combination of service and driver, the network service is not expected to initiate the address resolution. This allows flexibility for those services that prefer to have the address resolution performed by the network driver.

If the address resolution mechanism qualifies as an address resolution protocol (for instance, ARP) or network service in itself, it should bind to the MUX as a distinct service. Typically, it would bind itself to the same interface to which the corresponding network service is bound. The address resolution mechanism could even share the same callback functions of the network service if it knows how to distinguish between the two services.

The network address resolution function also supports multicast mapping, and maps a network service multicast address to a network driver (physical layer) multicast address.

11.2.2 Data Structures and Resources

The network service sublayer may allocate buffer pools in the form of **mBlk** clusters for receiving and sending packets. The network buffer management library **netBufLib** facilitates the implementation of an effective scheme for this purpose. The sublayer may also make use of the system **mBlk** pools. See *10.1.5 Managing Memory for Network Drivers and Services*, p.184, and *A. Using netBufLib* for more information on network buffer management.

Other resources commonly used by network services include receive/transmit queues and data structures that represent each active network interface controlled by the sublayer. Data that should be collected and maintained for each interface includes:

- the network driver's paradigm
- the state of the interface (for instance: attached, stopped, flow-controlled)
- the *cookie* supplied by the MUX, this cookie identifies the binding instance
- references to the service address mapping functions for the interface

11.2.3 Sublayer Routines

The subsections provides an overview of how a network service sublayer handles:

- sending packets
- receiving packets
- shutting down an interface

- reporting errors
- flow control
- device control

Sending Packets

Network layer packets are sent down through the MUX by using the **muxTkSend( )** routine (see *B.2.21 muxTkSend( )*, p.282). Data to be sent arrives from an upper layer in the form of an **mBlk** chain, and is modified by your network service before being sent.

The **muxTkSend( )** routine may return an error indicating that the driver is out of resources and cannot transmit the packet. A network service can use this error to establish a flow control mechanism (see *Flow Control*, p.228).

Sending Packets through an END

Sending packets through an END requires that the **mBlk** chain being sent contains fully formed physical layer frames. If necessary, a protocol must use the address resolution function registered for the interface to determine the destination address, and then use the **muxAddressForm( )** routine to add the necessary frame header to the packet.

Sending Packets through an NPT driver

When sending packets through an NPT driver, if the interface over which packets are being sent has a registered address resolution function, it should be called at send-time and the resolved address should be passed into **muxTkSend( )**.

Receiving Packets

The MUX forwards incoming packets to the appropriate network service by invoking the *stack***RcvRtn( )** callback that was installed by the sublayer during the bind phase (see *stackRcvRtn( )*, p.229).

If a service binds to a network interface using **muxTkBind( )**, it typically receives packets (not frames) from the MUX. This is true whether the network interface is managed using an END or an NPT driver. If a service needs to receive both packets as well as the physical layer header, it can use the optional "piggy-back" facility provided for in the *stack***RcvRtn( )** argument list. The only absolute exception to this behavior occurs when the service binds to the MUX as a **SNARF** protocol. Such a service always receives frames not packets. If the service binds as a **PROMISC** protocol, it typically sees frames. However, it can see a packet if some other service that registered for packets does not consume its packet. If a service binds to an END using **muxBind( )**, the service always receives frames not packets.

If your *stack***RcvRtn( )** returns **TRUE** (except if your service is of type **MUX_PROTO_PROMISC**), the packet is consumed and will not be available to lower priority services listening to the same driver. If, on the other hand, your routine returns **FALSE** (or is of type **MUX_PROTO_PROMISC**), the packet will remain available to other services.

If your service has been registered as of type **MUX_PROTO_OUTPUT**, its *stack***RcvRtn( )** callback routine will be called for all packets going out over the driver to which your service is bound. If the *stack***RcvRtn( )** of such an output protocol returns **TRUE**, the packet is consumed by the protocol and will not go out over the driver. If it returns **FALSE**, the outgoing packet will continue on to the driver. Only one service at a time may bind with type **MUX_PROTO_OUTPUT** to any given driver.

Shutting Down an Interface

The MUX initiates a shutdown in response to a **muxDevUnload( )** call from the system. Before unloading the network driver, the MUX issues a shutdown message to every network service bound to that driver by calling the *stack***ShutdownRtn( )** callbacks that were registered for those driver/service interfaces at bind time (see *stackShutdownRtn( )*, p.229).

Within this shutdown routine, the network service must take the necessary steps to close the interface, including a call to **muxUnbind( )** to unbind the network service from the device (see *B.2.23 muxUnbind( )*, p.283).

Error Reporting

Your network service may want to be notified of errors encountered in lower layers of the stack. Error conditions encountered by a network driver are passed up to the MUX. The MUX forwards each error to your network service sublayer if a *stack***ErrorRtn( )** callback is registered at bind time (see *stackErrorRtn( )*, p.230).

Flow Control

The **muxTkSend( )** routine may return an error, **END_ERR_BLOCK**, indicating that the network driver has insufficient resources to transmit data[2]. The network service sublayer can use this feedback to establish a flow control mechanism by holding off on making any further calls to **muxTkSend( )** until the device is ready to restart transmission. At that time, the MUX calls the *stack***RestartRtn( )** that you registered for the interface at bind time (see *stackRestartRtn( )*, p.230).

Device Control

A driver may be written to respond to specific *ioctl* commands. These commands can be issued from your network service by calling **muxIoctl( )** (see *B.2.12 muxIoctl( )*, p.276).

11.3 Interfacing with the MUX

When a network service registers with the MUX, it must provide references to functions that the MUX can call to handle the following:

- shutting down the network service
- passing a packet into the service
- passing a error message into the service
- restarting the service after a pause

The prototypes of the functions you specify to handle these functions differ depending on whether you use **muxTkBind( )** or **muxBind( )** to bind the service to a network interface in the MUX. If you are implementing a new network service, you should use the **muxTkBind( )** interface. This chapter includes the older **muxBind( )** interface solely to assist people maintaining network services that were designed to work with ENDs before the development of the NPT.

[2] Some less-carefully written drivers may simply return **ERROR** in this case, and it would not be possible for your service to determine whether this was due to a temporary problem such as insufficient resources or a more serious problem.

11.3.1 Service Functions Registered Using muxTkBind( )

This section describes the four service functions referenced in a **muxTkBind( )** call.

*stack***ShutdownRtn( )**

This routine is typically called by the MUX when it has received a call to **muxDevUnload( )** for a specific network device. Before unloading the driver, every network service bound to that device is issued a shutdown message by calling the *stack***ShutdownRtn( )** function registered for that service.

Within this routine, the network service must call **muxUnbind( )** to release itself from the device, and it should bring itself to an orderly halt.

The *stack***ShutdownRtn( )** prototype is:

```
STATUS stackShutdownRtn
    (
    void *  netCallbackId  /* the handle/ID installed at bind time */
    )
```

*stack***RcvRtn( )**

The MUX forwards packets received by the driver to the protocol layer by using the *stack***RcvRtn( )** callback that was installed with **muxTkBind( )**. This routine receives:

- a pointer to a **mBlk** chain containing the incoming packet
- the callback ID specific to the binding instance of the service/driver pair
- a network service type
- a pointer to additional data, the format of which, and the need for which, depends on the requirements of the driver.

The *stack***RcvRtn( )** prototype is:

```
BOOL stackRcvRtn
    (
    void *    netCallbackId,  /* the handle/ID installed at bind time */
    long      type,           /* network service type */
    M_BLK_ID  pNetBuf,        /* network service datagram */
    void *    pSpareData      /* pointer to optional data from driver */
    )
```

If a network protocol accepts the frame by returning **TRUE**, it must free the given **mBlk** chain when processing is complete.

*stack*ErrorRtn( )

Error conditions encountered by an driver are passed to the network service when the MUX calls *stack***ErrorRtn( )**. It is up to the network service to take the necessary action upon receiving the error.

This function takes two arguments: the callback identifier supplied to the service at bind-time, and a pointer to an **END_ERR** structure that describes the error (see *B.3.2 END_ERR*, p.285).

The *stack***ErrorRtn( )** prototype is:

```
void stackErrorRtn
    (
    void *     netCallbackId,  /* the handle/ID installed at bind time */
    END_ERR *  pError          /* pointer to structure containing error */
    )
```

*stack*RestartRtn( )

This routine is called by the MUX to restart network services that had previously stopped, perhaps because **muxTkSend( )** had returned an error indicating that the network service should wait before transmitting more packets.

When the device has determined that it has enough resources to resume transmission, it will indicate this to the MUX, which will then call *stack***RestartRtn( )**.

This function takes a single argument: the identifier specific to the service/driver pair that was supplied at bind-time.

The *stack***RestartRtn( )** prototype is:

```
STATUS stackRestartRtn
    (
    void *  netCallbackId  /* the handle/ID installed at bind time */
    )
```

11.3.2 Service Functions Registered Using muxBind( )

These function prototypes are included in this chapter to help people maintaining network services that were designed to work with **muxBind( )**, which predated the NPT. If you are designing a new network service, implement the routines associated with **muxTkBind( )**.

*stack*ENDShutdownRtn( )

This routine is typically called by the MUX when it has received a call to **muxDevUnload( )** for the specified network interface. Before unloading the END, every network service bound to that END's device is issued a shutdown message by calling the *stack***ENDShutdownRtn( )** function registered for the interface at bind time.

Within this routine, the network service must call **muxUnbind( )** to release itself from the device, and it should bring itself to an orderly halt.

The *stack***ENDShutdownRtn( )** prototype is:

```
STATUS stackENDShutdownRtn
    (
    void *  pEND,     /* END_OBJ from the driver's load routine */
    void *  pSpare    /* defined on a per-service basis */
    )
```

*stack*ENDRcvRtn( )

The MUX forwards packets received by the END to the protocol layer by using the *stack***ENDRcvRtn( )** callback that is installed with **muxBind( )**. The **pNetBuff** contains the entire driver-level frame, and **pLinkHdr** contains information about offsets to the network payload in the frame.

The *stack***ENDRcvRtn( )** prototype is:

```
BOOL stackENDRcvRtn
    (
    void *         pCookie,    /* returned by muxBind() */
    long           type,       /* from RFC1700, or user-defined */
    M_BLK_ID       pNetBuff,   /* packet with link-level info */
    LL_HDR_INFO *  pLinkHdr,   /* link-level header info structure */
    void *         pCallbackId /* registered by the network svc with MUX */
    )
```

If a network protocol accepts the frame by returning **TRUE**, it must free the given **mBlk** chain when processing is complete.

*stack*ENDErrorRtn( )

Error conditions encountered by an END are passed to the network service when the MUX calls *stack***ENDErrorRtn( )**. It is up to the network service to take the necessary action upon receiving the error.

This function takes three arguments: the callback identifier supplied to the service at bind-time, a pointer to an **END_ERR** structure, and the spare data, if any, defined for the service during the bind phase.

The *stack***ENDErrorRtn( )** prototype is:

```
void stackENDErrorRtn
    (
    void *     pEND,    /* END_OBJ passed to the MUX by the driver */
    END_ERR *  pError,  /* holds error information */
    void *     pSpare   /* defined on a per-service basis */
    )
```

*stack*ENDRestartRtn( )

This routine is called by the MUX to restart network services that had previously stopped, perhaps because **muxTkSend( )** had returned an error indicating that the network service should wait before transmitting more packets.

When the device has determined that it has enough resources to resume transmission, it will indicate this to the MUX, which will then call *stack***ENDRestartRtn( )**.

This function takes a single argument: the identifier specific to the service/driver pair that was supplied at bind-time.

The *stack***ENDRestartRtn( )** prototype is:

```
STATUS stackENDRestartRtn
    (
    void *  pEND,     /* END_OBJ passed to the MUX by the driver */
    void *  pSpare,   /* defined on a per-service basis */
    )
```

11.4 Adding a Socket Interface to Your Service

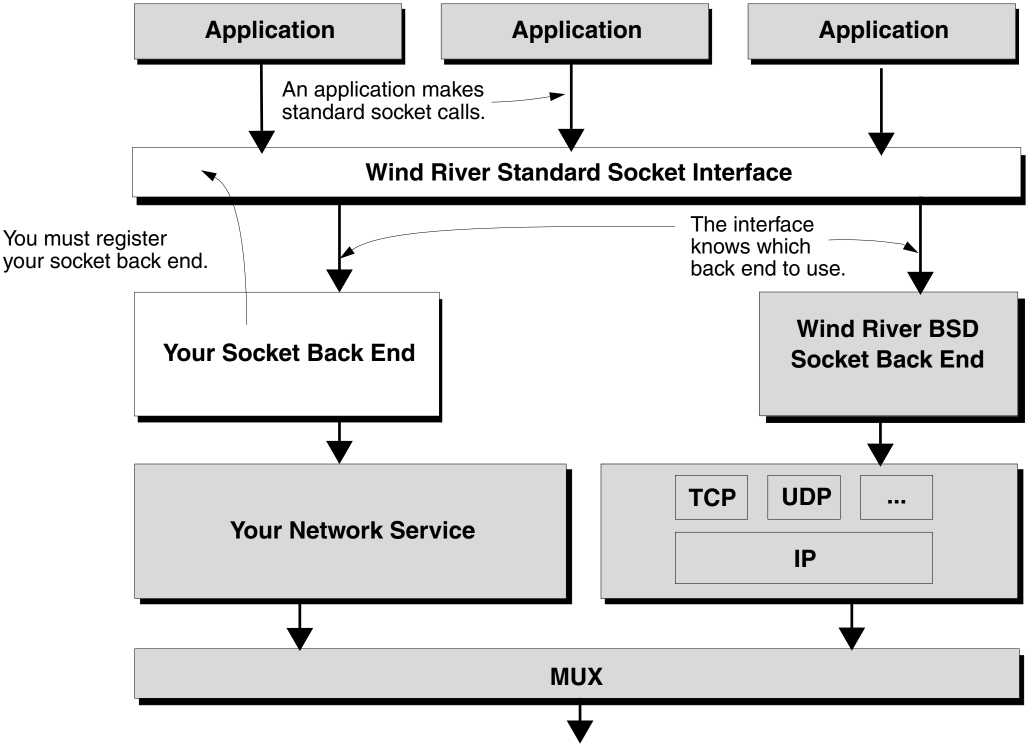
One way to give applications easy access to your network service is to add socket support to the service. In order to make it easier for you to write a network service that includes sockets support, the VxWorks stack includes a standard socket interface.

With the standard socket interface, you can add new socket back ends to access the network stack through your protocol layer implementation. This allows developers who are already familiar with the standard socket API to more easily use your service.

The standard socket interface is designed so that you can use socket back ends simultaneously for multiple protocol layer implementations. A layered architecture makes this possible. The Wind River standard socket interface is a layer above your back end socket layer, as shown in Figure 11-1.

This chapter introduces the process of implementing a socket back end.

Figure 11-1 **The Standard Socket Interface**


Application
Application
Application
An application makes standard socket calls.
Wind River Standard Socket Interface
You must register your socket back end.
The interface knows which back end to use.
Your Socket Back End
Wind River BSD Socket Back End
Your Network Service
TCP
UDP
...
IP
MUX

Process Overview

Socket calls made by an application are directed to the correct underlying socket back end based on the *domain* parameter that is passed to the **socket( )** call when the application creates the socket. If this parameter matches the *domainMap* parameter that you use when you add your new socket back end with the **sockLibAdd( )** routine, the socket calls are directed to your back end.

When you register your socket back end, you give the system a table that is filled with references to socket functions that you have created to support your implementation (see *The Socket Functional Interface*, p.234). The system is then able to support standard socket calls that are made using your back end.

11.4.1 Implementing a Socket Back End

To provide a socket back end requires that you implement socket functionality for your service and that you make the system aware of this new sockets implementation. The following sections show how this is done.

The Socket Functional Interface

The socket functional interface is the set of implementations of standard socket functions that are supported by a particular socket back end. There are two steps involved in creating a socket functional interface, both of which must be completed before the network is initialized.

The first step in creating a socket functional interface is to create a unique constant identifying the back end (for example, the BSD-specific **INET** back end is identified by the constant **AF_INET_BSD**). Add this constant to the list found in **/target/h/sys/socket.h**.

Then, create an initialization function that returns a reference to a **SOCK_FUNC** table filled with references to all of the functions that your socket back end will support, and call **sockLibAdd( )** to have this function invoked by the system. (For details about this initialization function, see *usrSockLibInit( )*, p.237).

NOTE: You are not required to support all of the possible socket functions. A call made by a user application to a non-supported function in your socket back end (indicated by a **NULL** pointer in the **SOCK_FUNC** table) results in an error returned to the calling application, with **errno** set to **ENOTSUP**.

Populating the SOCK_FUNC Table

The **SOCK_FUNC** table is a structure containing references to 19 implementations of functions common to sockets — functions such as **bind( )**, **recvfrom( )**, and **setsockopt( )** (see *11.4.3 Implementing Socket Functions*, p.236). A new network service that wants to be socket-accessible should implement service-specific versions of as many of these functions as it intends to support. To support the registration of these functions, the service should set up its *usr***SockLibInit( )** routine to return a **SOCK_FUNC** table that is populated with references to these functions.

Adding the Socket Library

Use the **sockLibAdd( )** function to add new socket functionality to the system's list of socket implementations. As input, this function expects a reference to the *usr***SockLibInit( )** function and the domain and service for which this socket implementation is to be registered.

The sockLibAdd( ) Function

The **sockLibAdd( )** function is used to make available a specific implementation of sockets for a particular domain. This function takes three parameters:

sockLibInitRtn
The parameter refers to a socket library initialization function that is invoked when **sockLibAdd( )** is called (for details about this function, see *usrSockLibInit( )*, p. 237). This function, among other things, passes the table of socket function implementations to the system.

domainMap
This parameter should be set to a protocol family identifier uniquely defining the sockets implementation (such as **AF_INET_BSD**).

domainReal
This parameter passes in a constant uniquely defining the address domain (such as **AF_INET**) for which the implementation is being added.

The following is an example of how **sockLibAdd( )** might be called to add the BSD sockets back end:

```
sockLibAdd ((FUNCPTR) bsdSockLibInit, AF_INET_BSD, AF_INET);
```

The **sockLibAdd( )** function is defined as follows:

```
STATUS sockLibAdd
    (
    FUNCPTR  sockLibInitRtn,  /* back end's initialization routine */
    int      domainMap,       /* AF_FOO_BAR, identifying # of back end */
    int      domainReal,      /* AF_FOO, identifying # of domain */
    )
```

The function returns **OK**, or **ERROR** if the socket back end could not be added.

11.4.2 Enabling Zbuf Support Within a Socket Back End

Zbufs (zero-copy buffers) are an enhancement that reduces the overhead involved in copying data between buffers as it passes through the layers of a network stack. A socket back end does not have to support zbufs, but may achieve significant performance gains by doing so.

The Wind River implementation of zbufs relies on the flags parameter in the socket send and receive routines. One of the flags that may be set in this parameter is **MSG_MBUF**. If this flag is set, this indicates that the data is in zbuf format — in other words, the buffer is not a true buffer, but a pointer to an mbuf chain.

⚠ **WARNING:** Zbuf sockets can only operate within a single protection domain. You may *not* pass data from one protection domain to another using zbuf sockets.

Your socket back end must implement the function *usr***SockZbufRtn( )** which indicates whether the back end supports zbufs.

If your socket back end has been written to support zbufs, it should check for the **MSG_MBUF** flag, and if it is present, it should treat the buffers as mbuf chains. See the zbuf section of *7. Sockets under VxWorks* for a more complete description of zero-copy buffers.

If your socket back end has not been written to support zbufs, then applications that try to use zbufs with your socket back end will fail with **errno** set to **ENOTSUP**. However, these applications will be able to use the sockets interface without zbufs.

11.4.3 Implementing Socket Functions

This subsection provides implementation recommendations of the functions referenced in a **SOCK_FUNC** table followed by implementation recommendations to the functions referenced in a **iosDrvInstall( )** call.

Implementation Recommendations for the Elements of a SOCK_FUNC Table

This subsection provides implementation details for the functions you must supply in a **SOCK_FUNC** structure:

```
typedef struct sockFunc                 /* SOCK_FUNC */
{
    FUNCPTR     libInitRtn;             /* sockLibInit()        */
    FUNCPTR     acceptRtn;              /* accept()             */
    FUNCPTR     bindRtn;                /* bind()               */
FUNCPTR     connectRtn;                /* connect()            */
FUNCPTR     connectWithTimeoutRtn;     /* connectWithTimeout() */
FUNCPTR     getpeernameRtn;            /* getpeername()        */
FUNCPTR     getsocknameRtn;            /* getsockname()        */
FUNCPTR     listenRtn;                 /* listen()             */
FUNCPTR     recvRtn;                   /* recv()               */
FUNCPTR     recvfromRtn;               /* recvfrom()           */
FUNCPTR     recvmsgRtn;                /* recvmsg()            */
FUNCPTR     sendRtn;                   /* send()               */
FUNCPTR     sendtoRtn;                 /* sendto()             */
FUNCPTR     sendmsgRtn;                /* sendmsg()            */
FUNCPTR     shutdownRtn;               /* shutdown()           */
FUNCPTR     socketRtn;                 /* socket()             */
FUNCPTR     getsockoptRtn;             /* getsockopt()         */
FUNCPTR     setsockoptRtn;             /* setsockopt()         */
FUNCPTR     zbufRtn;                   /* ZBUF support         */
} SOCK_FUNC;
```

*usr***SockLibInit( )**

The *usr***SockLibInit( )** function should install the socket back end as a driver within the VxWorks I/O system by calling **iosDrvInstall( )**, and then should return a pointer to a **SOCK_FUNC** structure.

The **iosDrvInstall( )** routine takes pointers to seven I/O functions, four of which must be supported by a socket back end: *usr***SockClose( )**, *usr***SockRead( )**, *usr***SockWrite( )** and *usr***SockIoctl( )**[3]. This routine returns a driver number, which should be stored by the socket back end.

Your *usr***SockLibInit( )** routine, which should be declared public, is based on the following skeleton:

```
SOCK_FUNC * usrSockLibInit (void)
{
    /* install driver for socket */
    int driverNum = iosDrvInstall( (FUNCPTR) NULL, (FUNCPTR) NULL,
                   (FUNCPTR) NULL, (FUNCPTR) usrSockClose,
                   (FUNCPTR) usrSockRead, (FUNCPTR) usrSockWrite),
                   (FUNCPTR) usrSockIoctl) );
    if( driverNum == ERROR ) return( (SOCK_FUNC *) NULL );
    /* Store driverNum somewhere convnient for future reference */
    /* Initialize SOCK_FUNC table */
    SOCK_FUNC * usrSockFuncs = (SOCK_FUNC *) malloc(sizeof(SOCK_FUNC));
    if( !usrSockFuncs )
    {
       errno = ENOMEM;
       return( (SOCK_FUNC *) NULL );
    }
    usrSockFuncs->libInitRtn    = (FUNCPTR) usrSockLibInit;
```

[3] The others may be skipped as **NULL** pointers. For more information on these essential functions, see *Socket Functions Passed to iosDrvInstall( )*, p.246.

```
    usrSockFuncs->acceptRtn     = (FUNCPTR) usrSockAccept;
    /* and so forth... */
    usrSockFuncs->setsockoptRtn = (FUNCPTR) usrSockSetSockOpt;
    usrSockFuncs->zbufRtn       = (FUNCPTR) usrSockZbufRtn;
    return (usrSockFuncs);
}
```

*usr***Socket( )**

When a **socket( )** call is issued, the standard socket interface searches for a back end that corresponds to the *domain* parameter passed to the **socket( )** routine. This *domain* parameter may be the actual domain name, *domainReal*. Alternatively, it could be a domain name that maps to the actual domain, *domainMap*. A back end is registered (using **sockLibAdd( )**) both with its actual domain name (*domainReal*) and with *domainMap*.

If a back end is found for the domain, the *usr***Socket( )** function from the **SOCK_FUNC** structure that was registered for that domain is called. This function is called with the real socket domain (*domainReal*) passed as the *domain* parameter, regardless of whether the *domainReal* or *domainMap* parameters was passed to the original **socket( )** call. The *type* and *protocol* entries are passed unchanged.

The *usr***Socket( )** routine should create a socket structure and then generate a new file descriptor representing the socket by calling **iosFdNew( )** with the address of the new socket structure. Other back end functions will receive this file descriptor as a reference, and will use it to retrieve the associated socket structure by calling **iosFdValue( )** with the file descriptor as an argument.

The *usr***Socket( )** routine is of the form:

```
int usrSocket
    (
    int domain,      /* socket domain or address family number */
    int type,        /* used to further define socket's nature */
    int protocol     /* the protocol variety of the socket */
    )
```

The *domain* argument refers to the socket domain or address family the socket belongs to (a particular back end may potentially be invoked for more than one domain). The *type* argument can be used to further define the nature of the socket (examples of types in the **AF_INET** domain include **SOCK_STREAM**, **SOCK_RAW** and **SOCK_DGRAM**). The *protocol* argument refers to the protocol variety of the socket (in the **AF_INET_BSD** back end, an example of a protocol variety is **IPPROTO_TCP**).

The *usr***Socket( )** function returns the file descriptor that was generated for the socket, or **ERROR** if it was unable to open a socket.

*usr***SockAccept( )**

This routine accepts a connection on a socket and returns a file descriptor representing the new socket created for the connection. Typically for this function to succeed, the socket represented by *fd* must have been previously bound to an address with *usr***SockBind( )** and enabled for connections by a call to *usr***SockListen( )**. When *usr***SockAccept( )** is called, *addr* should be an available buffer, and *addrlen* should indicate the size of the buffer.

The *usr***SockAccept( )** function will block the caller until a connection is present, unless the socket has been explicitly marked as non-blocking.

The *usr***SockAccept( )** routine is of the form:

```
int usrSockAccept
    (
    int                 fd,      /* file descriptor for socket */
    struct sockaddr *   addr,    /* network address */
    int *               addrlen  /* length of address structure */
    )
```

This function returns a file descriptor representing a new socket with the same properties as the one represented by *fd* (or **ERROR** if the accept fails). In addition, on a successful return, *addr* should be filled with the address of the machine making the connection, and *addrlen* should be set to the length of that address.

*usr***SockBind( )**

This routine associates a network address (referred to by *name*) with a specified socket so that other processes can connect or send to it.

The *usr***SockBind( )** routine is of the form:

```
STATUS usrSockBind
    (
    int                 fd,      /* file descriptor representing socket */
    struct sockaddr *   name,    /* network address */
    int                 namelen  /* length of network address */
    )
```

This function returns **OK**, or **ERROR** if the bind fails.

*usr***SockConnect( )**

This routine initiates a connection between a socket, *fd*, and another socket which is specified by *name*.

The *usr***SockConnect( )** routine is of the form:

```
STATUS usrSockConnect
    (
    int                 fd,      /* file descriptor representing socket */
    struct sockaddr *   name,    /* network address */
    int                 namelen  /* length of network address */
    )
```

This function returns **OK**, or **ERROR** if the connection fails.

*usr*SockConnectWithTimeout( )

This routine attempts to initiate a connection between a socket, *fd*, and another socket specified by *name*, for a duration specified by *timeout*, reporting an error if it cannot do so in the time required. If *timeout* is **NULL**, this function should act exactly like *usr***SockConnect( )**.

The *usr***SockConnectWithTimeout( )** routine is of the form:

```
STATUS usrSockConnectWithTimeout
    (
    int                fd,       /* file descriptor representing socket */
    struct sockadr *   name,     /* network address */
    int                namelen,  /* length of address */
    struct timeval *   timeout   /* maximum duration of connect attempt */
    )
```

This function returns **OK**, or **ERROR** if it cannot make the connection in the specified time.

*usr*SockGetpeername( )

This routine gets the name of the peer connected to the socket *fd*, placing this name in the **sockaddr** structure of length *namelen* that was passed in.

The *usr***SockGetpeername( )** routine is of the form:

```
STATUS usrSockGetpeername
    (
    int                 fd,      /* file descriptor representing socket */
    struct sockaddr *   name,    /* structure to hold peer name */
    int *               namelen  /* length of returned peer name */
    )
```

This function should place the name of the peer in *name* and set *namelen* to the size of this name in bytes, then return **OK**, or **ERROR** if a name could not be retrieved for the specified socket.

*usr*SockGetsockname( )

This routine gets the current name for the socket *fd*, placing this name in the available **sockaddr** structure of size *namelen* that was passed in.

The *usr***SockGetsockname( )** routine is of the form:

```
STATUS usrSockGetsockname
    (
    int                fd,      /* file descriptor representing socket */
    struct sockaddr *  name,    /* structure to hold socket name */
    int *              namelen  /* length of returned socket name */
    )
```

This function places the name of the socket in *name* and set *namelen* to the size of this name in bytes, then returns **OK**, or **ERROR** if a name could not be retrieved for the specified socket.

*usr*SockListen( )

This routine enables connections to a socket. The *backlog* parameter specifies the maximum number of unaccepted connections that can be pending at any given time. After enabling connections with *usr***SockListen( )**, connections are actually accepted by *usr***SockAccept( )**.

The *usr***SockListen( )** routine is of the form:

```
STATUS usrSockListen
    (
    int fd,         /* file descriptor representing the socket */
    int backlog     /* max number of unaccepted pending connections */
    )
```

This function returns **OK**, or **ERROR** if the listen request fails.

*usr*SockRecv( )

This routine receives data from a connection-based (stream) socket. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockRecv( )** routine is of the form:

```
int usrSockRecv
    (
    int     fd,       /* file descriptor represents the socket */
    char *  buf,      /* buffer holding the received data */
    int     bufLen,   /* length of the buffer */
    int     flags,    /* flags describe the nature of the data */
    )
```

NOTE: If the **MSG_MBUF** flag is set in the *flags* parameter, this means that a zero-copy buffer (zbuf) is being received. In this case, the *buf* parameter is a pointer to a **NULL** mbuf pointer and not the **char** * specified in the parameter list. In other words, ***buf == (struct mbuf *) NULL**. The *usr***SockRecv( )** routine should set *buf* to point to the mbuf chain holding the incoming data.

This function returns the number of bytes received, or **ERROR** if the receive fails.

*usr*SockRecvFrom( )

This routine receives data from a datagram socket, regardless of whether it is connected. When this function is called, *from* will either be **NULL** or will be an available buffer of size *pFromLen* designed to hold the address from which the data is coming. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockRecvFrom( )** routine is of the form:

```
int usrSockRecvFrom
    (
    int                 fd,       /* file descriptor represents the socket */
    char *              buf,      /* buffer holding the received data */
    int                 bufLen,   /* length of the buffer */
    int                 flags,    /* flags describe the nature of the data */
    struct sockaddr *   from,     /* address of the sending agent */
    int *               pFromLen  /* length of the from structure */
    )
```

NOTE: If the **MSG_MBUF** flag is set in the *flags* parameter, this means that a zero-copy buffer (zbuf) is being received. In this case, the *buf* parameter is a pointer to a **NULL** mbuf pointer and not the **char** * specified in the parameter list. In other words, ***buf == (struct mbuf *) NULL**. The *usr***SockRecvFrom( )** routine should set *buf* to point to the mbuf chain holding the incoming data.

If *from* is not **NULL**, the address of the sending socket is copied into it, and *pFromLen* is set to the length of this address. This function returns the number of bytes received, or **ERROR** if the receive fails.

*usr*SockRecvMsg( )

This routine receives a message from a datagram socket. It may be used in place of *usr***SockRecvFrom( )** to decrease the overhead of breaking down the message-header structure in each message. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockRecvMsg( )** routine is of the form:

```
int usrSockRecvMsg
    (
    int              fd,       /* file descriptor for the socket */
    struct msghdr *  pMsgHdr,  /* the message header */
    int              flags     /* flags describing nature of the data */
    )
```

This function returns the number of bytes received, or **ERROR** if the receive fails.

*usr*SockSend( )

This routine transmits data from the socket *fd* to a previously-established connection-based (stream) socket. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockSend( )** routine is of the form:

```
int usrSockSend
    (
    int     fd,      /* file descriptor representing the socket */
    char *  buf,     /* buffer containing data to be sent */
    int     bufLen,  /* length of the buffer */
    int     flags    /* flags describing the nature of the data */
    )
```

NOTE: If the **MSG_MBUF** flag is set in the *flags* parameter, this means that a zero-copy buffer (zbuf) is being sent. In this case, the *buf* parameter is a pointer to a **NULL** mbuf pointer and not the **char** * specified in the parameter list. In other words, ***buf == (struct mbuf *) NULL**. The *usr***SockSend( )** routine should set *buf* to point to the mbuf chain holding the data.

This function returns the number of bytes sent, or **ERROR** if the send fails.

*usr*SockSendto( )

This routine transmits data from the socket specified by *fd* to the datagram socket specified by *to*. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockSendto( )** routine is of the form:

```
int usrSockSendto
    (
    int                 fd,      /* file descriptor representing the socket */
    caddr_t             buf,     /* buffer containing message to be sent */
    int                 bufLen,  /* length of this buffer */
    int                 flags,   /* flags describing the nature of the data */
    struct sockaddr *   to,      /* recipient's address */
    int                 tolen    /* length of the to structure */
    )
```

NOTE: If the **MSG_MBUF** flag is set in the *flags* parameter, this means that a zero-copy buffer (zbuf) is being sent. In this case, the *buf* parameter is a pointer to a **NULL** mbuf pointer and not the **char** * specified in the parameter list. In other words, ***buf == (struct mbuf *) NULL**. The *usr***SockSendto( )** routine should set *buf* to point to the mbuf chain holding the data.

This function returns the number of bytes sent, or **ERROR** if the send fails.

*usr*SockSendMsg( )

This routine transmits a message to a datagram socket specified by *fd*. It may be used in place of *usr***SockSendto( )** to decrease the overhead of reconstructing the message header structure for each message. How the *flags* parameter should be set depends on the nature of the sockets involved and the requirements of the connection. This will differ for different socket and service implementations.

The *usr***SockSendMsg( )** routine is of the form:

```
int usrSockSendMsg
    (
    int               fd,       /* file descriptor for the socket */
    struct msghdr *   pMsgHdr,  /* message header */
    int               flags     /* flags describing nature of the data */
    )
```

This function returns the number of bytes sent, or **ERROR** if the send fails.

*usr*SockShutdown( )

This routine shuts down all, or part, of the connection-based socket *fd*. The *how* value allows for some control over how this shutdown takes place if sends and receives are still pending.

The *usr***SockShutdown( )** routine is of the form:

```
STATUS usrSockShutdown
    (
    int fd, /* file descriptor representing the socket */
    int how /* directs how shutdown proceeds when activity is pending */
    )
```

This function returns **OK**, or **ERROR** if the specified socket was invalid or could not be shut down.

*usr*GetSockOpt( )

This routine retrieves socket option values[4] associated with a specified socket. To find options set at the socket level, *level* is set to **SOL_SOCKET**. To find options set for a particular service, *level* is set to the identifying number of that service. The *optval* parameter points to an available buffer of size *optlen*. The buffer itself, although passed in as a **char ***, is treated as a pointer to whatever data type or structure is appropriate to the option being referenced.

The *usr***GetSockOpt( )** routine is of the form:

```
STATUS usrGetSockOpt
    (
    int     fd,       /* file descriptor representing the socket */
    int     level,    /* scope of option being retrieved */
    int     optname,  /* name of option being retrieved */
    char *  optval,   /* holds the value of the option */
    int *   optlen    /* indicates the length of optval */
    )
```

This function fills *optval* with the setting of the specified option, and sets *optlen* to the actual size of this value. The function returns **OK**, or **ERROR** if it was unable to retrieve a value for this option given these parameters.

*usr*SetSockOpt( )

This routine sets the options associated with a socket[5]. To manipulate options at the socket level, *level* is set to **SOL_SOCKET**. Otherwise, *level* is set to the service number of the service for which the option is being set.

[4] For instance, in TCP, socket options include **SO_KEEPALIVE**, **SO_LINGER** and **TCP_NODELAY**.

[5] For instance, in UDP, socket options include **SO_BROADCAST**, **IP_ADD_MEMBERSHIP** and **IP_MULTICAST_IF**.

The *usr***SetSockOpt( )** routine is of the form:

```
STATUS usrSetSockOpt
    (
    int     fd,       /* file descriptor representing the socket */
    int     level,    /* scope of option being set */
    int     optname,  /* name of option being set */
    char *  optval,   /* value the option is being set to */
    int     optlen    /* length of the value field */
    )
```

This function returns **OK**, or **ERROR** if the request to set the socket option for the specified socket fails.

*usr*SockZbufRtn( )

This routine returns **TRUE** if the back end supports the zero-copy interface (zbufs), otherwise it returns **FALSE**. This function is of the form:

```
STATUS usrSockZbufRtn()
```

Socket Functions Passed to iosDrvInstall( )

An **iosDrvInstall( )** call expects pointers to function to handle:

- closing the socket
- reading the socket
- writing to the socket
- I/O control for the socket

*usr*SockClose( )

This routine is called by the I/O system to close a socket.

The *usr***SockClose( )** routine is of the form:

```
int usrSockClose
    (
    [socket structure] * so      /* socket being closed */
    )
```

The *socket structure* corresponds to whatever data structure describes the socket object that was returned from **iosFdValue( )**. This might be a **struct socket**, or it may be some other structure.

This function returns 0 on success, or -1 on failure.

*usr*SockRead( )

The *usr***SockRead( )** routine is of the form:

```
int usrSockRead
    (
    [socket structure] *   so,     /* socket being read from */
    char *                buf,    /* buffer to contain incoming data */
    int                   bufLen  /* length of this buffer */
    )
```

The *socket structure* corresponds to whatever data structure describes the socket object that was returned from **iosFdValue( )**. This might be a **struct socket**, or it may be some other structure.

This function returns the number of bytes read, or -1 if the read fails.

*usr*SockWrite( )

The *usr***SockWrite( )** routine is of the form:

```
int usrSockWrite
    (
    [socket structure] *   so,     /* socket being written to */
    char *                buf,    /* buffer containing outgoing data */
    int                   bufLen  /* length of this buffer */
    )
```

The *socket structure* corresponds to whatever data structure describes the socket object that was returned from **iosFdValue( )**. This might be a **struct socket**, or it may be some other structure.

This function returns the number of bytes written, or -1 if the write fails.

*usr*SockIoctl( )

The *usr***SockIoctl( )** routine is of the form:

```
int usrSockIoctl
    (
    int fd,         /* file descriptor representing the socket */
    int function,   /* the ioctl function being called */
    int arg         /* the argument to this ioctl function */
    )
```

This function returns a positive number whose value depends on the ioctl function being invoked, or -1 in the case of an error.

A
Using netBufLib

A.1 Introduction

Using **netBufLib**, you can set up and manage a memory pool specialized to the needs of network interface drivers and network services (protocols). In a **netBufLib** memory pool, data is held in clusters. Elements of the network stack exchange references to these clusters using data structures called **mBlk**s and **clBlk**s.

The VxWorks stack uses **netBufLib** to manage its internal system and data memory pools. Likewise, the supplied network drivers use **netBufLib** to manage their memory pools. This facilitates their sharing buffers with the MUX, which expects buffers organized using **mBlk** and **clBlk** structures.

To include **netBufLib** in VxWorks, include **INCLUDE_NETWORK**.

Replacing netBufLib

Because the buffer management code in the VxWorks stack access only the published **netBufLib** API, you can safely replace the shipped **netBufLib** implementation with your own. If your implementation supports all the public **netBufLib** function calls and structures, the current stack and network interface drivers should continue to function normally. However, the replacements must conform to the API of the existing routines. Otherwise, the network stack will be unable to access memory.

A.2 How a netBufLib Pool Organizes Memory

The memory in a **netBufLib** memory pool is organized using pools of **mBlk** structures, pools of **clBlk** structures, and pools of cluster buffers (simple character arrays). A **netPoolInit( )** call joins all these sub-pools into a single pool that is organized around **M_CL_CONFIG** and **CL_DESC** structures.

To reserve a buffer from a **netBufLib** pool, call **netTupleGet( )**. This call returns a tuple, a construct consisting of an **mBlk** structure, a **clBlk** structure, and a cluster buffer. The **mBlk** and **clBlk** structures provide information necessary to support buffer loaning and buffer chaining for the data that is stored in the clusters. The clusters come in sizes determined by the **CL_DESC** table that describes the memory pool.

Clusters

Valid cluster sizes are within bounds set by powers of two to up to 64KB (65536) – see Figure A-5. Exactly which cluster sizes are valid within a particular memory pool depends on the contents of the **CL_DESC** table submitted to the **netPoolInit( )** call that initializes the pool.

mBlks and clBlks

To support buffer (cluster) loaning, **netBufLib** tracks clusters using **mBlk** and **clBlk** structures. For each cluster in a memory pool, there needs to be a corresponding **clBlk** structure. The **clBlk** structure tracks how many **mBlk**s share its underlying cluster. Above the **clBlk**, is the **mBlk** structure. This structure stores a link to a **clBlk** and can store a link to another **mBlk**. By chaining **mBlk**s, you can reference an arbitrarily large amount of data, such as a packet chain (see Figure A-1).

The **mBlk** structure is the primary object you use to access the data that resides in a memory pool. Because an **mBlk** is only a reference to the data available through a **clBlk**, network layers can exchange data without copying between internal buffers. Each **mBlk** structure stores separate links for the data within a packet and for the data that starts a new packet.

Because the **mBlk** references the cluster data through a **clBlk**, duplicating an **mBlk** need not copy the cluster data. For example, **mBlk A** in Figure A-2 is a duplicate of **mBlk 1**. Creating this duplicate did not require the copying of the underlying cluster. However, the duplication did require incrementing the external reference count stored in the **clBlk** that manages the underlying cluster. This is critical when it comes time to free an **mBlk**.

Figure A-1 **Presentation of Two Packets in One mBlk Chain**

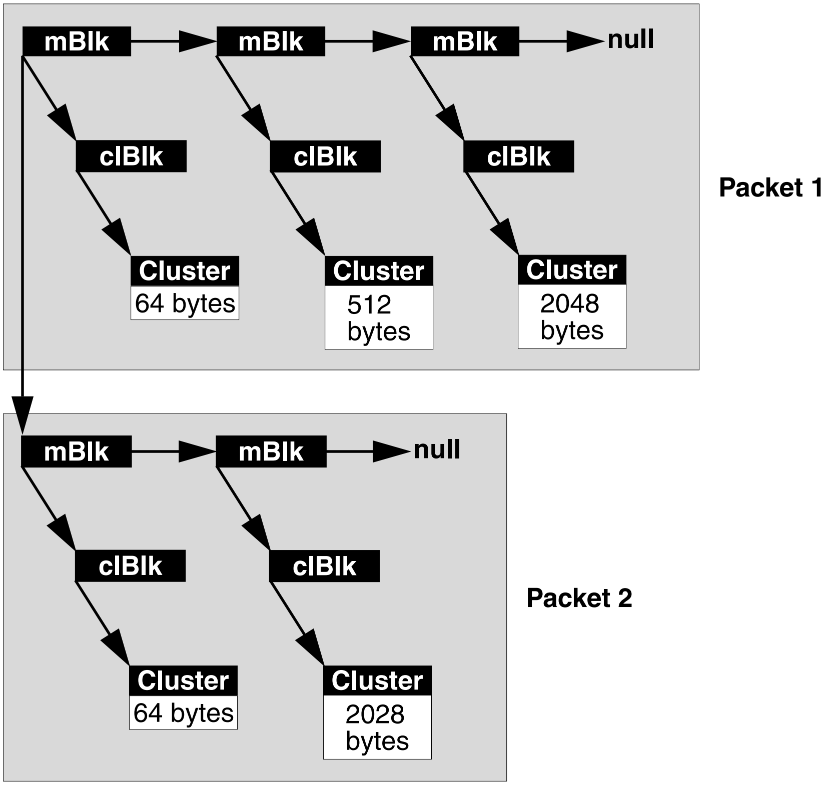
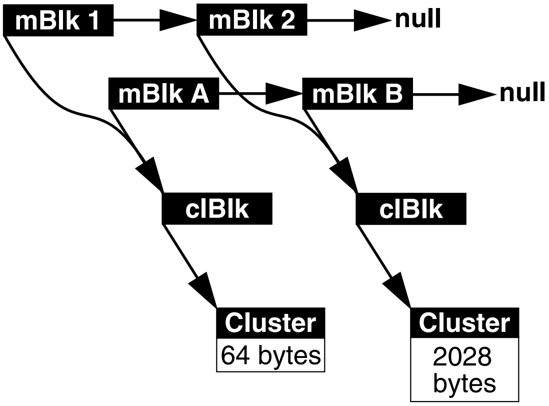


Figure A-2 **Different mBlks Can Share the Same Cluster**



If you use **netBufLib** to free the **mBlk**, the **mBlk** is freed back to the pool and the reference count in the underlying **clBlk** is decremented. If this reference count drops to zero (indicating that no **mBlk**s are referencing the cluster), the **clBlk** and cluster are also freed back to the memory pool.

A.3 Setting Up a Memory Pool

Setting up a memory pool culminates in a call to **netPoolInit( )**. Before calling **netPoolInit( )**, you must have allocated all the memory you want to include in the pool. You then reference that memory in the **CL_DESC** and **M_CL_CONFIG** structures that you submit to **netPoolInit( )**. The **CL_DESC** and **M_CL_CONFIG** structures supply **netPoolInit( )** with the memory pointers, structure counts, buffer sizes, and buffer counts that define the memory pool.

- **M_CL_CONFIG**

  An **M_CL_CONFIG** table contains only one row. The elements in the row specify the pool's **clBlk** count, its **mBlk** count, and a previously allocated memory area identified by a starting address and a size. The count values are analogous to the **NET_NUM_BLK** and **NET_CL_BLKS** values you specified for the network system memory table.

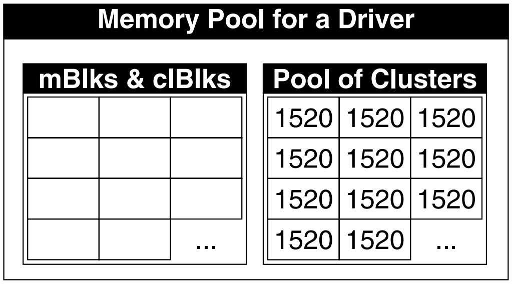
  The **clBlk** count you specify must be equal to the number of clusters in the memory pool. The **mBlk** count must be at least as large, but could be larger, depending on how you use the memory pool. The allocated memory referenced in the supplied pointer should be large enough to contain all the **clBlk** and **mBlk** structures specified by the supplied counts.

- **CL_DESC**

  A **CL_DESC** table associates a cluster size with a cluster count and a memory area identified by a starting address and a size. Although you did not need to provide the memory area starting address and size in **clDescTbl[ ]** and **sysClDescTbl[ ]**, you do need to provide this information in any **CL_DESC** you create. In other words, you must make the memory allocation calls that reserve memory for the pool.

  If you are setting up a **CL_DESC** table for a network driver pool, your table will likely need only one row. This row specifies a cluster size close to the MTU of the underlying device. For example, the Lance Ethernet driver uses a cluster size is 1520 bytes, which is the Ethernet MTU plus some slack (see Figure A-3). The **CL_DESC** table for a network protocol will likely contain several rows. This is because protocols typically require buffers of several different sizes (see Figure A-4).

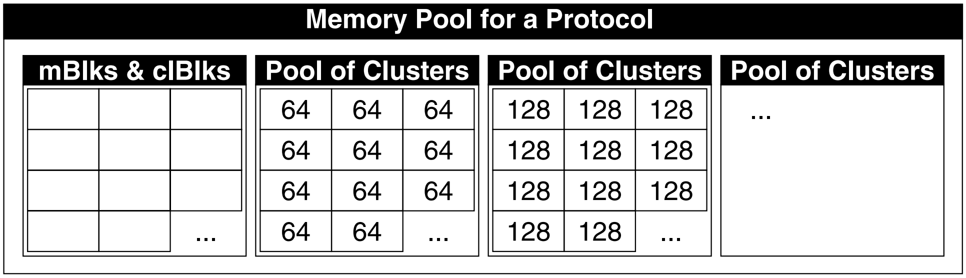
Figure A-3 **A Driver Memory Pool**



The number of **mBlks** and **clBlks** available to pool and their location in memory is set in the **M_CL_CONFIG** table you supply to **netPoolInit( )**.

The number of cluster pools, the size of the clusters in each pool, and their location in memory is set in the **CL_DESC** that you supply to **netPoolInit( )**. The **CL_DESC** for this pool would have only one row.

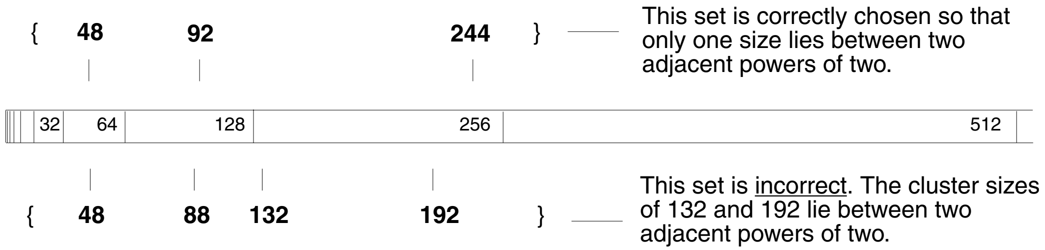
Figure A-4 **A Protocol Memory Pool**



When deciding on cluster sizes, choose a size that is alone within a power of two boundary (see Figure A-5) and that is large enough to contain the buffer you want to store in the cluster. In addition, no two rows in the table can share the use the same cluster size.

Figure A-5 shows two examples of sets of cluster sizes. The first, {48, 92, 244}, is correct, because there is at least one power of two between the different sizes. The second, {48, 88, 132, 192}, is incorrect, because the cluster sizes of 132 and 192 both fall between the adjacent powers of two of 128 and 256.

Figure A-5 **Choosing Correct Cluster Sizes**


{ 48 92 244 }
This set is correctly chosen so that only one size lies between two adjacent powers of two.
32
64
128
256
512
{ 48 88 132 192 }
This set is incorrect. The cluster sizes of 132 and 192 lie between two adjacent powers of two.

A.4 Storing and Using Data in Clusters

When your driver or protocol needs a buffer for a packet, reserve an appropriately sized cluster buffer from an established memory pool. Then associate that cluster with an **mBlk** and a **clBlk** structure from the pool.

To create an **mBlk**/**clBlk**/cluster construct one step at a time:

1. Call **netClusterGet( )** to reserve a cluster buffer for your data.
2. Call **netClBlkGet( )** to reserve a **clBlk** structure.
3. Call **netClBlkJoin( )** to join the **clBlk** to the cluster containing the packet.
4. Call **netMblkGet( )** to reserve a **mBlk** structure.
5. Call **netMblkClJoin( )** to join the **mBlk** structure to the **clBlk**/cluster construct.

Alternatively, you can reduce all of the above allocations and joins to a single **netTupleGet( )** call (see the **netBufLib** reference entry).

Now that the data is contained within an **mBlk**/**clBlk**/cluster construct, you can use various **netBufLib** routines to adjust or inspect this data. For example, to read the data in the construct, you can call **netMblkToBufCopy( )**. In addition, you can use the **mBlk** chaining feature to prepend or append data to the packet.

NOTE: The various **net*Get( )** routines reserve memory from a pre-allocated pool. Internally, they do not use semaphores. Thus, they are safe to call when a call to **malloc( )** would be unsafe or impossible.

A.5 Freeing mBlks, clBlks, and Clusters

When you want to return an **mBlk**/**clBlk**/cluster chain to its memory pool, call **netMblkClChainFree( )**. This frees all **mBlk**s in the chain. It also decrements the reference counters in all the **clBlk**s in the chain. If the reference counter for a **clBlk** drops to zero, that **clBlk** and its associated cluster are also freed back to the pool. To free a single **mBlk**/**clBlk**/cluster back to the memory pool, use **netMblkClFree( )**.

A.6 Macros for Buffer Manipulation

M_PREPEND(*m*, *plen*, *how***)**

This macro prepends *plen* bytes at the beginning of buffer *m*. You can set *how* to **M_WAIT** or **M_DONTWAIT**. This corresponds to the *canWait* parameter in the **netMblkGet( )** call. It specifies the desired behavior if there is not enough space in *m* for *plen* bytes and a new buffer must be allocated and added to the buffer chain to contain these prepended bytes.

For information on pre-allocating space at the beginning of a buffer, which increases the speed of the **M_PREPEND( )** operation, see *10.1.7 Early Link-Level Header Allocation in an NPT Driver*, p.185.

M_ALIGN(*m*, *len***)**

This macro appends an area of *len* bytes to the buffer *m*, and aligns this area on a long word boundary. It does not verify that this amount of space is available in the buffer, so if you are in doubt, use **M_TRAILINGSPACE( )** to verify this before performing the **M_ALIGN( )** operation.

M_LEADINGSPACE(*m***)**

This macro reports the size of the leading space that comes before the data held in the cluster belonging to buffer *m*. Knowing the size of this leading space is useful before calling **M_PREPEND( )**. If you find that there is not enough space, you can allocate a new buffer.

M_TRAILINGSPACE(*m*)

This macro computes the amount of space past the data held in the cluster belonging to mbuf *m*. This can be used to estimate how large a data block can be added with an **M_ALIGN( )** operation on that mbuf.

A.7 The netBufLib Library

This library contains routines that you can use to organize and maintain a memory pool consisting of pools of **mBlk** structures, pools of **clBlk** structures, and pools of clusters. The **mBlk** and **clBlk** structures are used to manage the clusters. The clusters are containers for the actual data.

These structures and the functions in this library constitute a buffering API that has been designed to meet the needs both of network services and network drivers.

The **mBlk** structure is the primary vehicle for passing data between a network driver and a service. However, the **mBlk** structure must first be properly joined with a **clBlk** structure that was previously joined with a data cluster. The actual vehicle for passing data is not merely an **mBlk** structure but an **mBlk**/**clBlk/**cluster construct.

To include **netBufLib** in VxWorks, build your image with basic network support by including **INCLUDE_NETWORK**.

netBufLibInit( )

This function initializes the **netBufLib**. If **INCLUDE_NETWORK** is included in the configuration of your VxWorks image, VxWorks automatically calls **netBufLibInit( )**.

```
STATUS netBufLibInit (void)
```

The function returns **OK** on successful initialization or **ERROR** otherwise.

netClBlkFree( )

This function frees the **clBlk**/cluster construct from an **mBlk**. If there are no other **mBlk**s that refer to the construct, both the **clBlk** and its associated cluster are freed back into the specified memory pool.

```
void netClBlkFree
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    CL_BLK_ID    pClBlk     /* pointer to the clBlk to be freed */
    )
```

netClBlkGet( )

This function gets a **clBlk** from the specified memory pool. Set the *canWait* parameter either to **M_WAIT** or **M_DONTWAIT**. If *canWait* is set to **M_WAIT**, and if a **clBlk** is not initially available, **netClBlkGet( )** attempts garbage collection before failing. If *canWait* is set to **M_DONTWAIT** and if no **clBlk** is available, **netClBlkGet( )** returns a **NULL** value immediately.

```
CL_BLK_ID netClBlkGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    int          canWait    /* M_WAIT/M_DONTWAIT */
    )
```

This function returns a **clBlk** identifier, or **NULL** if none are available and *canWait* is set to **M_DONTWAIT**.

netClBlkJoin( )

This function joins the previously reserved cluster specified by *pClBuf* to the previously reserved **clBlk** structure specified by *pClBlk*. The *size* parameter indicates the size of the cluster referenced in *pClBuf*. The arguments *pFreeRtn*, *arg1*, *arg2*, and *arg3* set the values of the **pCLFreeRtn**, **clFreeArg1**, **clFreeArg2**, and **clFreeArg1** members of the specified **clBlk** structure.

```
CL_BLK_ID netClBlkJoin
    (
    CL_BLK_ID  pClBlk,    /* pointer to a cluster Blk */
    char *     pClBuf,    /* pointer to a cluster buffer */
    int        size,      /* size of the cluster buffer */
    FUNCPTR    pFreeRtn,  /* pointer to the free routine */
    int        arg1,      /* argument 1 of the free routine */
    int        arg2,      /* argument 2 of the free routine */
    int        arg3       /* argument 3 of the free routine */
    )
```

This function returns a **CL_BLK_ID** corresponding to the new *pClBlk*, or **NULL** if it was unable to perform the join.

netClFree( )

This function returns the specified cluster back to the specified memory pool.

```
void netClFree
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    UCHAR *      pClBuf     /* pointer to the cluster buffer */
    )
```

netClPoolIdGet( )

This function returns a **CL_POOL_ID** corresponding to a cluster pool containing clusters that match the specified *bufSize*. If *bestFit* is **TRUE**, this routine returns a **CL_POOL_ID** for a pool that contains clusters greater than or equal to *bufSize*. If *bestFit* is **FALSE**, this routine returns a **CL_POOL_ID** for a cluster from whatever cluster pool is available. If the memory pool specified by *pNetPool* contains only one cluster pool, *bestFit* should always be set to **FALSE**.

```
CL_POOL_ID netClPoolIdGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    int          bufSize,   /* size of the buffer */
    BOOL         bestFit    /* TRUE/FALSE */
    )
```

This function returns either a **CL_POOL_ID** corresponding to a particular cluster pool, or **NULL** if no such cluster pool could be found.

netClusterGet( )

This function retrieves a cluster from the specified cluster pool *pClPool* within the specified memory pool *pNetPool*.

```
char * netClusterGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    CL_POOL_ID   pClPool    /* ptr to the cluster pool */
    )
```

The function returns a character pointer referring to a cluster buffer, or **NULL** if no buffer is available in the specified pool.

netMblkChainDup( )

This function copies an **mBlk** chain or a portion of an **mBlk** chain, and returns a new **M_BLK_ID** referring to a new **mBlk** allocated from *pNetPool* that holds the copy. The copy starts at *offset* bytes from the beginning of the chain specified by *pMblk*, and continues for *len* bytes, or, if *len* is set to **M_COPYALL**, for the remainder of the **mBlk** chain.

```
M_BLK_ID netMblkChainDup
    (
    NET_POOL_ID  pNetPool,  /* pointer to the pool          */
    M_BLK_ID     pMblk,     /* pointer to source mBlk chai */
    int          offset,    /* offset to duplicate from     */
    int          len,       /* length to copy               */
    int          canWait    /* M_DONTWAIT/M_WAIT            */
    )
```

This routine copies the references from a source *pMblk* chain to a newly allocated **mBlk** chain. This lets the two **mBlk** chains share the same **clBlk**/cluster constructs. This routine also increments the reference count in the shared **clBlk**.

The *canWait* parameter expects either **M_WAIT** or **M_DONTWAIT**. If *canWait* is **M_WAIT**, this routine will make an attempt at garbage collection before failing if an **mBlk** is not initially available. If *canWait* is **M_DONTWAIT** and no **mBlk** is immediately available, this routine returns immediately with a **NULL** value.

This function returns an **M_BLK_ID** referring to the newly allocated **mBlk** chain, or **NULL** if the copy fails. If **NULL** is returned, **errno** is set to either of the following:

– **S_netBufLib_INVALID_ARGUMENT**, indicating that one of the parameters is set so that a copy could not take place

– **S_netBufLib_NO_POOL_MEMORY**, indicating that *pNetPool* does not have enough room for the copied **mBlk** chain

netMblkClChainFree( )

This function frees a chain of **mBlk** structures back into its memory pool, and decrements the reference count in the associated **clBlk** structures. If no other **mBlk**s reference these **clBlk** structures, they are also freed along with their associated clusters.

```
void netMblkClChainFree
    (
    M_BLK_ID  pMblk  /* pointer to the mBlk */
    )
```

If *pMblk* does not refer to a valid **M_BLK_ID**, this function sets **errno** to **S_netBufLib_MBLK_INVALID**.

netMblkClFree( )

This function frees the **mBlk** specified back into its memory pool, and decrements the reference count in the associated **clBlk** structure. If no other **mBlk**s reference this **clBlk** structure, it is also freed along with its associated cluster.

```
M_BLK_ID netMblkClFree
    (
    M_BLK_ID  pMblk  /* pointer to the mBlk */
    )
```

If the specified **mBlk** is part of an **mBlk** chain, this routine returns an **M_BLK_ID** referring to the next **mBlk** in the chain, otherwise it returns **NULL**. If *pMblk* does not refer to a valid **M_BLK_ID**, this function sets **errno** to **S_netBufLib_MBLK_INVALID**.

netMblkClGet( )

This function retrieves a **clBlk**/cluster construct of size *bufSize* from the specified net pool and joins it to the specified **mBlk**.

```
STATUS netMblkClGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    M_BLK_ID     pMblk,     /* mBlk to embed the cluster in */
    int          bufSize,   /* size of the buffer to get */
    int          canWait,   /* wait or dontwait */
    BOOL         bestFit    /* TRUE/FALSE */
    )
```

If *canWait* is set to **M_WAIT**, this routine performs garbage collection and makes a second attempt if a **clBlk**/cluster construct is not initially available. If *canWait* is set to **M_DONTWAIT** and no **clBlk**/cluster construct is available, this routine returns **ERROR** immediately.

If *bestFit* is set to **TRUE** and a cluster of the exact size is unavailable, this routine tries to retrieve a larger cluster instead. If *bestFit* is set to **FALSE** and an exactly matching size is unavailable, this routine tries to retrieve either a smaller or larger cluster (depending on what is available). For memory pools containing only one cluster size, *bestFit* should always be set to **FALSE**.

This function returns **OK** if the requested **clBlk**/cluster was retrieved and joined to the specified mBlk, or **ERROR** if the attempt fails. If the **M_BLK_ID** specified does not refer to a valid **mBlk**, **errno** is set to **S_netBufLib_MBLK_INVALID**.

netMblkClJoin( )

This function joins a specified **mBlk** to a specified **clBlk**/cluster construct. Internally, this routine sets the **M_EXT** flag in **mBlk.mBlkHdr.mFlags**. It also sets the **mBlk.mBlkHdr.mData** to point to the start of the data in the cluster.

```
M_BLK_ID netMblkClJoin
    (
    M_BLK_ID   pMblk,  /* pointer to an mBlk */
    CL_BLK_ID  pClBlk  /* pointer to a cluster Blk */
    )
```

This function returns an **M_BLK_ID** corresponding to the modified *pMblk*, or **NULL** if either the *pMblk* or *pClBlk* arguments are invalid.

netMblkDup( )

This function copies the **clBlk**/cluster references from the source **mBlk** into the destination **mBlk** and increments the reference count in the shared **clBlk**. This allows the two **mBlk** structures to share the same **clBlk**/cluster construct.

```
M_BLK_ID netMblkDup
    (
    M_BLK_ID  pSrcMblk,  /* pointer to source mBlk */
    M_BLK_ID  pDestMblk  /* pointer to the destination mBlk */
    )
```

This function returns a pointer to the destination **mBlk**, or **NULL** if the source mBlk is not part of a valid **mBlk**/**clBlk**/cluster construct.

netMblkFree( )

This function frees a specified **mBlk** back into its memory pool.

```
void netMblkFree
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    M_BLK_ID     pMblk      /* mBlk to free */
    )
```

netMblkGet( )

This function retrieves an **mBlk** from the specified net pool. If *canWait* is set to **M_WAIT**, this routine performs garbage collection and makes a second attempt if an **mBlk** is not initially available. If *canWait* is set to **M_DONTWAIT** and no **mBlk** is immediately available, this routine returns immediately with a **NULL** value. The *type* parameter indicates the type value to be associated with the retrieved **mBlk**.

```
M_BLK_ID netMblkGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool */
    int          canWait,   /* M_WAIT/M_DONTWAIT */
    UCHAR        type       /* mBlk type */
    )
```

This routine returns an **M_BLK_ID** referring to the newly retrieved **mBlk**, or **NULL** if no **mBlk** is available.

netMblkToBufCopy( )

This function copies data from the **mBlk** chain specified to the buffer referenced in *pBuf*. It is assumed that *pBuf* points to enough memory to contain all the data in the entire **mBlk** chain. The argument *pCopyRtn* expects either a **NULL** or a function pointer to a copy routine. The arguments passed to the copy routine are source pointer, destination pointer and the length of data to copy. If *pCopyRtn* is **NULL**, **netMblkToBufCopy( )** uses a default routine to extract the data from the chain.

```
int netMblkToBufCopy
    (
    M_BLK_ID  pMblk,    /* pointer to an mBlk */
    char *    pBuf,     /* pointer to the buffer to copy */
    FUNCPTR   pCopyRtn  /* function pointer for copy routine */
    )
```

This function returns the length of the data copied, or 0 (zero) if no copy takes place.

netPoolDelete( )

This function frees the specified memory pool.

```
STATUS netPoolDelete
    (
    NET_POOL_ID  pNetPool  /* pointer to a net pool */
    )
```

This routine returns **OK** if the pool is freed, or **ERROR** if it is unable to free the specified pool. If *pNetPool* does not refer to a valid net pool, **errno** is set to **S_netBufLib_NETPOOL_INVALID**.

netPoolInit( )

This function initializes a **netBufLib**-managed memory pool, organizing several sub-pools within it for **mBlk** structures, **clBlk** structures and cluster-size-specific sub-pools. Set *pNetPool* to an identifier referring to a previously allocated **NET_POOL** structure.

```
STATUS netPoolInit
    (
    NET_POOL_ID    pNetPool,          /* pointer to a net pool */
    M_CL_CONFIG *  pMclBlkConfig,     /* pointer to a mBlk configuration */
    CL_DESC *      pClDescTbl,        /* pointer to cluster desc table */
    int            clDescTblNumEnt,   /* number of cluster desc entries */
    POOL_FUNC *    pFuncTbl           /* pointer to pool function table */
    )
```

Set *pMclBlkConfig* to point to a previously allocated and initialized **M_CL_CONFIG** structure that specifies the requested subdivision of the memory pool. Within this structure, you must provide the following values:

– *mBlkNum*, a count of **mBlk** structures
– *clBlkNum*, a count of **clBlk** structures
– *memArea*, a pointer to memory that can contain the **mBlk** and **clBlk** structures
– *memSize*, the size of that memory area

For example, you can set up an **M_CL_CONFIG** structure as follows:

```
M_CL_CONFIG mClBlkConfig = /* mBlk, clBlk configuration table */
    {
 /* mBlkNum     clBlkNum        memArea         memSize */
 /* ----------  ----            -------         ------- */
    400,        245,            0xfe000000,     21260
    };
```

You can calculate the *memArea* and *memSize* values. Such code can first define a table as shown above, but set both *memArea* and *memSize* as follows:

```
mClBlkConfig.memSize =
    (mClBlkConfig.mBlkNum * (M_BLK_SZ + sizeof(long)))
    + (mClBlkConfig.clBlkNum * CL_BLK_SZ);
```

You can set the *memArea* value to a pointer to private memory, or you can reserve the memory with a call to **malloc( )**. For example:

```
    mClBlkConfig.memArea = malloc(mClBlkConfig.memSize);
```

The **netBufLib.h** file defines **M_BLK_SZ** as:

```
    sizeof(struct mBlk)
```

Currently, this evaluates to 32 bytes. Likewise, this file defines **CL_BLK_SZ** as:

```
    sizeof(struct clBlk)
```

Currently, this evaluates to 32 bytes.

When choosing values for *mBlkNum* and *clBlkNum*, remember that you need as many **clBlk** structures as you have clusters (data buffers). You also need at least as many **mBlk** structures as you have **clBlk** structures, but you will most likely need more. That is because **netBufLib** shares buffers by letting multiple **mBlk** structures join to the same **clBlk** and thus to its underlying cluster. The **clBlk** keeps a count of the number of **mBlk** structures that reference it.

The *pClDescTbl* argument should be set to point to a table of previously allocated and initialized **CL_DESC** structures. Each structure in this table describes a single cluster pool. You need a dedicated cluster pool for each cluster size you want to support. Within each **CL_DESC** structure, you must provide the following values:

– *clusterSize*, the size of a cluster in this cluster pool
– *num*, the number of clusters in this cluster pool
– *memArea*, a pointer to an area of memory that can contain all the clusters
– *memSize*, the size of that memory area

Thus, if you need to support six different cluster sizes, this parameter must point to a table containing six **CL_DESC** structures. For example, consider the following:

```
CL_DESC clDescTbl [] =   /* cluster descriptor table */
    {
 /* clusterSize        num     memArea         memSize */
 /* ----------         ----    -------         ------- */
    {64,               100,    0x10000,        6800},
    {128,              50,     0x20000,        6600},
    {256,              50,     0x30000,        13000},
    {512,              25,     0x40000,        12900},
    {1024,             10,     0x50000,        10280},
    {2048,             10,     0x60000,        20520}
    };
```

As with the *memArea* and *memSize* members in the **M_CL_CONFIG** structure, you can set these members of the **CL_DESC** structures by calculation after you create the table. The formula would be as follows:

```
    clDescTbl[n].memSize =
       (clDescTbl[n].num * (clDescTbl[n].clusterSize + sizeof(long)));
```

The *memArea* member can point to a private memory area that you know to be available for storing clusters, or you can use **malloc( )**.

```
    clDescTbl[n].memArea =  malloc( clDescTbl[n].memSize );
```

Valid cluster sizes range from 64 bytes to 65536 bytes. If there are multiple cluster pools, valid sizes are further restricted by being separated by powers of two. See *A.3 Setting Up a Memory Pool*, p.252 for further discussion of this restriction. A typical buffer size for Ethernet devices is 1514 bytes. However, because a cluster size requires a 4-byte alignment, the cluster size for this Ethernet buffer has to be increased to at least 1516 bytes.

The *clDescTblNumEnt* argument should be set to the number of elements in the *pClDescTbl* table. This is a count of the number of cluster pools. You can get this value by using the **NELEMENTS** macro defined in **vxWorks.h**. For example:

```
int clDescTblNumEnt = (NELEMENTS(clDescTbl));
```

The *pFuncTbl* argument should be set either to **NULL** or to refer to a function table containing pointers to the functions used to manage the buffers in this memory pool. Using a **NULL** for this parameter tells **netBufLib** to use its default function table. If you opt for the default function table, every **mBlk** and every cluster is prepended by a 4-byte header (which is why the size calculations above for clusters and **mBlk** structures contained an extra **sizeof(long)**). However, users need not concern themselves with this header when accessing these buffers. The returned pointers from functions such as **netClusterGet( )** return pointers to the start of data, which is just after the header.

Assuming you have set up the configuration tables as shown above, a typical call to **netPoolInit( )** is as follows:

```
int clDescTblNumEnt = (NELEMENTS(clDescTbl));
NET_POOL      netPool;
NET_POOL_ID   pNetPool = &netPool;
if (netPoolInit (pNetPool, &mClBlkConfig, &clDescTbl [0],
    clDescTblNumEnt, NULL) != OK) return (ERROR);
```

This function returns **OK** if the initialization succeeds, or **ERROR** otherwise, in which case **errno** is set to the following, depending on the reason for the failure:

- **S_netBufLib_MEMSIZE_INVALID**
- **S_netBufLib_CLSIZE_INVALID**
- **S_netBufLib_NO_SYSTEM_MEMORY**
- **S_netBufLib_MEM_UNALIGNED**
- **S_netBufLib_MEMSIZE_UNALIGNED**
- **S_netBufLib_MEMAREA_INVALID**

netPoolShow( )

This function displays the distribution of **mBlk**s and clusters in a given network pool, specified by *pNetPool*.

```
void netPoolShow
    (
    NET_POOL_ID  pNetPool
    )
```

netShowInit( )

This function initializes the network show routines. These routines are included in VxWorks automatically if network show routines are built into the VxWorks image by including **INCLUDE_NET_SHOW**.

```
void netShowInit (void)
```

netStackDataPoolShow( )

This function displays the distribution of **mBlk**s and clusters in the network data pool. This pool is used only for data transfer through the network stack.

```
void netStackDataPoolShow (void)
```

netStackSysPoolShow( )

This function displays the distribution of **mBlk**s and clusters in the network system pool. This pool is used only for system structures such as sockets, routes, interface addresses, protocol control blocks, multicast addresses, and multicast route entries.

```
void netStackSysPoolShow (void)
```

netTupleGet( )

This function gets a **mBlk**/**clBlk**/cluster construct from the memory pool specified by *pNetPool*. This construct is used to pass data across the layers of the network stack. The *bufSize* parameter indicates the number of bytes in the cluster.

```
M_BLK_ID netTupleGet
    (
    NET_POOL_ID  pNetPool,  /* pointer to the net pool   */
    int          bufSize,   /* size of the buffer to get */
    int          canWait,   /* wait or dontwait          */
    UCHAR        type,      /* type of data              */
    BOOL         bestFit    /* TRUE/FALSE                */
    )
```

If *canWait* is **M_WAIT**, this routine performs garbage collection and makes a second attempt if an **mBlk**/**clBlk**/cluster construct is not initially available. If *canWait* is **M_DONTWAIT** and no **mBlk**/**clBlk**/cluster construct is immediately available, this routine returns immediately with a **NULL** value.

The *type* parameter indicates the type of data in the cluster, for example **MT_DATA** or **MT_HEADER**. The acceptable type values are defined in **netBufLib.h**.

If *bestFit* is **TRUE** and a cluster of the exact size is unavailable, this routine gets a larger cluster (if available). If *bestFit* is **FALSE** and an exactly matching size is unavailable, this routine gets either a smaller or a larger cluster (depending on what is available). Otherwise, it returns immediately with a **NULL** value. For memory pools containing only one cluster size, *bestFit* should always be set to **FALSE**.

This function returns an **M_BLK_ID** corresponding to an **mBlk**/**clBlk/**cluster construct, or **NULL** if it is unable to get such a construct from the specified net pool. In case of an error, **errno** may be set to **S_netBufLib_MBLK_INVALID**, **S_netBufLib_CLSIZE_INVALID** or **S_netBufLib_NETPOOL_INVALID**.

B

MUX/NPT Routines and Data Structures

B.1 Introduction

This appendix describes the routines and data structures that comprise the MUX/NPT API.

B.2 MUX Routines

This section provides descriptions of the following routines:

– **muxAddrResFuncAdd( )**
– **muxAddrResFuncDel( )**
– **muxAddrResFuncGet( )**
– **muxAddressForm( )**
– **muxBind( )**
– **muxDevExists( )**
– **muxDevLoad( )**
– **muxDevStart( )**
– **muxDevStop( )**
– **muxDevUnload( )**
– **muxError( )**
– **muxIoctl( )**
– **muxMCastAddrAdd( )**
– **muxMCastAddrDel( )**

- **muxMCastAddrGet( )**
- **muxTkBind( )**
- **muxTkDrvCheck( )**
- **muxTkPollReceive( )**
- **muxTkPollSend( )**
- **muxTkReceive( )**
- **muxTkSend( )**
- **muxTxRestart( )**
- **muxUnbind( )**

B.2.1 muxAddrResFuncAdd( )

Use **muxAddrResFuncAdd( )** to register an address resolution function for an interface-type/protocol pair. You must call **muxAddrResFuncAdd( )** before calling the protocol's *protocol***Attach( )** routine. If the driver registers itself as an Ethernet driver, you do not need to call **muxAddrResFuncAdd( )** because VxWorks automatically assigns **arpresolve( )** to registered Ethernet devices.

```
STATUS muxAddrResFuncAdd
    (
    long    ifType,        /* interface type from m2Lib.h, or driver type */
    long    protocol,      /* protocol from RFC 1700, or service type */
    FUNCPTR addrResFunc    /* the function being added */
    )
```

NOTE: If you are running IP over Ethernet, it is *very* unlikely that you will want to replace **arpresolve( )** with your own implementation.

The prototype for your address resolution function (*addrResFunc*) must conform to the following:

```
int xxxResolvRtn
    (
    FUNCPTR           ipArpCallBackRtn,
    struct mbuf *     pMbuf,
    struct sockaddr * dstIpAddr,
    struct ifnet *    ifp,
    struct rtentry *  rt,
    char *            dstBuff
    )
```

In addition, your *xxx***ResolvRtn( )** must return one upon success, which indicates that *dstIpAddr* has been updated with the necessary data-link layer information and that the IP sublayer output function can transmit the packet.

Your *xxx***ResolvRtn( )** must return zero if it cannot resolve the address immediately. In the default **arpresolve( )** implementation, resolving the address immediately means **arpresolve( )** was able to find the address in its table of results from previous ARP requests. Returning zero indicates that the table did not contain the information but that the packet has been stored and that an ARP request has been queued.

If the ARP request times out, the packet is dropped. If the ARP request completes successfully, processing that event updates the local ARP table and resubmits the packet to the IP sublayer's output function for transmission. This time, the **arpresolve( )** call will return one.

What is essential to note here is that **arpresolve( )** did not wait for the ARP request to complete before returning. If you replace the default **arpresolve( )** function, you must make sure your function returns as soon as possible and that it never blocks. Otherwise, you block the IP sublayer from transmitting other packets out through the interface for which this packet was queued. You must also make sure that your **arpresolve( )** function takes responsibility for the packet if it returns zero.

Writing an Address Resolution Function

An address resolution function designed for use with the VxWorks IP sublayer implementation must conform strictly to expectations of that sublayer. The function that you register with **muxAddrResFuncAdd( )** must be of the form:

```
int xxxResolvRtn
    (
    FUNCPTR            aCallBackRtn,
    struct mbuf *      pMbuf,
    struct sockaddr *  dstIpAddr,
    struct ifnet *     ifp,
    struct rtentry *   rt,
    char *             dstBuff
    )
```

In addition, your *xxx***ResolvRtn( )** must not block, and it must return with a function value of zero or one, where one indicates *desIpAddr* contains the resolved address and zero indicates that it does not. If your address resolution function returns a zero, it should take responsibility for the packet.

Within your *xxx***ResolvRtn( )**, you will likely implement an address resolution table or some equally speedy mechanism that lets you reuse address resolution information discovered previously. If your speedy address resolution mechanism fails, you should queue the work to an address resolution protocol and return zero. As a prototype for your function that queues the work, consider:

```
int resolveIPaddress
    (
    FUNCPTR          pCallback,  /* a retransmit callback function */
    M_BLK_ID         pIPPacket,  /* the IP packet */
    struct sockaddr  pIPAddress, /* the IP address */
    void *           passthru1,  /* reserved for internal use */
    void *           passthru2   /* reserved for internal use */
    char *           pBuffer,    /* pre-allocated, up to 128 bytes */
    )
```

This function would supply the address resolution protocol with the information needed to perform the address resolution as well as the packet destined for that address. If the address resolution succeeds, the function would add that information to its table (or equivalent), and then use *pCallback* to submit the packet (with its original parameters) to the IP layer output function for transmission. This time, your *xxx***ResolvRtn( )** replacement function should be able to return with the address resolution information.

B.2.2 muxAddrResFuncDel( )

Use **muxAddrResFuncDel( )** to undo the assignment of an address resolution function to an interface-type/protocol pair.

```
STATUS muxAddrResFuncDel
    (
    long    ifType,        /* media interface type from m2Lib.h */
    long    protocol,      /* protocol type, for instance from RFC 1700 */
    )
```

The **muxAddrResFuncDel( )** function returns **OK**, or **ERROR** if the request fails.

B.2.3 muxAddrResFuncGet( )

Use **muxAddrResFuncGet( )** to retrieve a pointer to the address resolution function registered to the specified interface-type/protocol pair.

```
FUNCPTR muxAddrResFuncGet
    (
    long  ifType,       /* interface type from m2Lib.h, or driver type*/
    long  protocol      /* protocol from RFC 1700, or service type */
    )
```

This function returns a pointer to the address resolution function, or **NULL** if no function is registered for the service.

B.2.4 muxAddressForm( )

Use **muxAddressForm( )** to form a frame with a link-layer address. A network service needs this function when working with ENDs, which are frame-oriented, but not with NPT drivers, which are packet oriented.

When given the source and destination addressing information (through *pSrcAddr* and *pDstAddr*), this function returns an *mBlk* that points to an assembled link-level header, and prepends this header to the *mBlk* chain pointed to by *pMblk*. Note that the **pDstAddr.mBlkHdr.reserved** field should be set to the network protocol type.

```
M_BLK_ID muxAddressForm
    (
    void *   pCookie,   /* the cookie returned by muxBind() */
    M_BLK_ID pMblk,     /* pointer to the packet being reformed */
    M_BLK_ID pSrcAddr,  /* pointer to the mBlk with the source address */
    M_BLK_ID pDstAddr,  /* pointer to the mBlk with the dest address */
    )
```

This function returns the head of an *mBlk* chain containing the data from *pMblk* with a prepended link-level header, or **NULL** if the attempt fails.

B.2.5 muxBind( )

You can use **muxBind( )** to bind a network service to an END. However, it is often better to use **muxTkBind( )**, which works with both ENDs and NPT drivers.

```
void * muxBind
    (
    char *  pName,         /* interface name, for example, ln, ei,... */
    int     unit,          /* unit number */
    BOOL    (* stackENDRcvRtn)(void *, long, M_BLK_ID, LL_HDR_INFO *, void *),
    STATUS  (* stackENDShutdownRtn)(void *, void *),
    STATUS  (* stackENDRestartRtn)(void *, void *),
    void    (* stackENDErrorRtn)(END_OBJ *, END_ERR *, void *),
    long    type,          /* protocol type, from RFC1700 or user-defined */
    char *  pProtoName,    /* string name for protocol  */
    void *  pSpare         /* identifies the binding instance */
    )
```

Note that the **stack*Rtn** functions in the parameter list are defined differently than in **muxTkBind( )**.

The **muxBind( )** function returns a cookie identifying the network driver to which the MUX has bound the service. The service should keep track of this cookie for use with other MUX functions.

B.2.6 muxDevExists( )

Use **muxDevExists( )** to test whether a given device has already been loaded into the network stack. As input, it expects the name and unit number of the device to be tested.

```
BOOL muxDevExists
    (
    char *  pName,  /* string containing a device name (ln, ei, ...)*/
    int     unit    /* unit number */
    )
```

This function returns **TRUE** if the device has already been loaded into the network stack, or **FALSE** otherwise.

B.2.7 muxDevLoad( )

Use **muxDevLoad( )** to load a network driver into the MUX. Internally, **muxDevLoad( )** calls the driver's *end***Load( )** or *npt***Load( )**. After the device is loaded, you must call **muxDevStart( )** to start the device.

The *pInitString* argument passes directly into the *end***Load( )** or *npt***Load( )** function. Likewise, the *pBSP* argument is passed along to the driver, which may or may not use it. This argument can be used to pass in tables of BSP-specific functions that the driver can use.

```
void * muxDevLoad
    (
    int        unit,                           /* unit number of device */
    END_OBJ *  (* endLoad)(char *, void *),    /* driver's load function */
    char *     pInitString,                    /* init string for driver */
    BOOL       loaning,                        /* unused */
    void *     pBSP                            /* BSP-specific */
    )
```

This function returns a *cookie* that can be passed to **muxDevStart( )**, or **NULL**, if the device could not be loaded, in which case **errno** is set to **S_muxLib_LOAD_FAILED**.

B.2.8 muxDevStart( )

Use **muxDevStart( )** to start a device after you have successfully loaded the device using **muxDevLoad( )**. Internally, **muxDevStart( )** activates the network interfaces for a device by calling the drivers *end***Start( )** routine.

```
STATUS muxDevStart
    (
    void * pCookie  /* the cookie returned from muxDevLoad() */
    )
```

The **muxDevStart( )** function returns **OK**, on success; **ERROR**, if the driver's *end***Start( )** routine fails; or **ENETDOWN**, if *pCookie* does not represent a valid device.

B.2.9 muxDevStop( )

Use **muxDevStop( )** to stop the specified driver. Internally, **muxDevStop( )** calls the *end***Stop( )** or *npt***Stop( )** routine registered for the driver.

```
STATUS muxDevStop
    (
    void *  pCookie  /* the cookie returned from muxDevLoad() */
    )
```

The **muxDevStart( )** function returns **OK**, on success; **ERROR**, if the *end***Stop( )** or *npt***Stop( )** routine registered for the driver fails; or **ENETDOWN**, if *pCookie* does not represent a valid device.

B.2.10 muxDevUnload( )

Use **muxDevUnload( )** to unload a device from the MUX. Unloading a device closes any network connections made through the device.

To notify protocols that the device is unloading, **muxDevUnload( )** calls the *stack***ShutdownRtn( )** function for each protocol bound to the device (internally, the *stack***ShutdownRtn( )** should call **muxUnbind( )** to detach from the device).

To free device-internal resources, **muxDevUnload( )** calls the *end***Unload( )** or *npt***Unload( )** function that the device registered with the MUX.

```
STATUS muxDevUnload
    (
    char *  pName,  /* the name of the device, for example, ln or ei */
    int     unit    /* the unit number */
    )
```

This function returns **OK**, on success; or **ERROR**, if the device could not be found, or the error returned from the device's *end***Unload( )** or *npt***Unload( )** function if that function fails.

B.2.11 muxError( )

Drivers use **muxError( )** to report an error to a network service that is bound to it through the MUX. You can find predefined errors in **end.h**. This function is passed two arguments: the END object that identifies the device that is issuing the error and a pointer to an **END_ERR** structure (see *B.3.2 END_ERR*, p.285).

```
void muxError
    (
    void *     pEnd,     /* END object pointer returned by end/nptLoad() */
    END_ERR *  pError    /* error structure */
    }
```

B.2.12 muxIoctl( )

Use **muxIoctl( )** to access the *ioctl* services that network interfaces have registered with the MUX. Typical uses of **muxIoctl( )** include starting, stopping, or resetting a network interface, or adding or configuring MAC and network addresses.

```
STATUS muxIoctl
    (
    void *   pCookie,  /* returned by muxTkBind() */
    int      cmd,      /* ioctl command */
    caddr_t  data      /* data needed to carry out the command */
    )
```

This function returns **OK** if successful; **ERROR**, if the device was unable to successfully complete the command; or **ENETDOWN**, if the cookie did not represent a valid device.

B.2.13 muxMCastAddrAdd( )

Use **muxMCastAddrAdd( )** to add an address to the table of multicast addresses maintained for a device. It expects two arguments: a cookie that was returned when **muxTkBind( )** was used to bind to the device, and a string containing the address to be added.

```
STATUS muxMCastAddrAdd
    (
    void *  pCookie,   /* returned by muxTkBind() */
    char *  pAddress   /* address to add to the table */
    )
```

This function returns **OK**, if successful; **ERROR**, if the device was unable to successfully add the address (if this is because the device does not support multicasting, **errno** will be set to **ENOTSUP**); or **ENETDOWN**, if the cookie does not represent a valid device.

B.2.14 muxMCastAddrDel( )

Use **muxMCastAddrDel( )** to remove an address from the table of multicast addresses maintained for a device. It expects two arguments: a cookie that was returned when **muxTkBind( )** was used to bind to the device, and a string containing the address to be removed.

```
STATUS muxMCastAddrDel
    (
    void *  pCookie,   /* returned by muxTkBind() */
    char *  pAddress   /* address to delete from the table */
    )
```

This function returns **OK** , if successful; **ERROR**, if the device was unable to successfully remove the address (if this is because the device does not support multicasting, **errno** will be set to **ENOTSUP**, if this is because the address was not found in the table, **errno** will be set to **EINVAL**); **ENETDOWN**, if the cookie does not represent a valid device.

B.2.15 muxMCastAddrGet( )

Use **muxMCastAddrGet( )** to retrieve the list of multicast addresses that have been registered for a driver. It expects two arguments: a cookie that was returned when **muxTkBind( )** was used to bind to the device, and a pointer to a pre-allocated

MULTI_TABLE structure into which the table contents will be written during this function call (see *B.3.8 MULTI_TABLE*, p.292).

```
int muxMCastAddrGet
    (
    void *         pCookie,  /* returned by muxTkBind() */
    MULTI_TABLE *  pTable    /* structure that will hold retrieved table */
    )
```

This function returns **OK**, if successful; **ERROR**, if the device was unable to successfully supply the list; or **ENETDOWN**, if the cookie does not represent a valid device.

B.2.16 muxTkBind( )

Use **muxTkBind( )** to bind a network service to a network interface. Before the network service can send and receive packets from the network, it must bind to one or more network drivers through which the packets will be sent and received. To specify these network drivers and bind to them, use the function **muxTkBind( )**.

In the call to **muxTkBind( )** you must provide the following information:

- the network driver to bind to (name and unit number)
- a network service type, based on RFC 1700 or user-defined
- optional data structures used to exchange information with the driver (typically used when a network service is designed to work with a particular network driver)
- a set of callback routines used by the MUX (see Table B-1)
- a key or private data structure which will be passed back when these callbacks are invoked to identify the bound interface

These callback functions are listed in Table B-1 and are described at greater length in *11.3.1 Service Functions Registered Using muxTkBind( )*, p.229.

Table B-1 **Network Service Callback Functions**

Callback Function	Description
*stack***RcvRtn( )**	Receive data from the MUX.
*stack***ErrorRtn( )**	Receive an error notification from the MUX.
*stack***ShutdownRtn( )**	Shut down the network service.

Table B-1 **Network Service Callback Functions** *(Continued)*

Callback Function	Description
*stack***RestartRtn( )**	Restart a suspended network service.

Two additional arguments (*pNetSvcInfo* and *pNetDrvInfo*) allow the sublayer to exchange additional information with the network driver, depending on requirements specific to the particular service or driver. The Wind River IP network protocol, for instance, expects a driver to pass up certain information, although it does not pass anything back down. These additional arguments may be especially helpful to those network services and network driver types that are naturally "tightly coupled."

As part of the bind phase, the network service typically retrieve the address resolution and mapping functions for each network interface that is being bound to, storing them in a private data structure allocated by the service.

The **muxTkBind( )** function returns a cookie that uniquely represents the binding instance and is used to identify that binding instance in subsequent calls. A return value of **NULL** indicates that the bind failed.

The **muxTkBind( )** function is defined as:

```
void * muxTkBind
    (
    char *  pName,              /* interface name, for example: ln, ei */
    int     unit,               /* unit number */
    BOOL    (* stackRcvRtn)(void *, long, M_BLK_ID, void *),
    STATUS  (* stackShutdownRtn)(void *),
    STATUS  (* stackRestartRtn)(void *),
    void    (* stackErrorRtn)(void *, END_ERR *),
    long    type,               /* from RFC1700 or user-defined */
    char *  pProtoName,         /* string name of service */
    void *  pNetCallBackId,     /* returned to svc sublayer during recv */
    void *  pNetSvcInfo,        /* ref to netSrvInfo structure */
    void *  pNetDrvInfo         /* ref to netDrvInfo structure */
    )
```

This function returns a cookie that uniquely represents the binding instance, or **NULL** if the bind fails.

B.2.17 muxTkDrvCheck( )

A network service sublayer uses **muxTkDrvCheck( )** to determine whether a particular driver is an NPT driver.

NOTE: Internally, **muxTkDrvCheck( )** uses an **EIOCGNPT** *ioctl* to ask the driver whether its is an NPT. An NPT returns a zero in response to an **EIOCGNPT** *ioctl*.

```
int muxTkDrvCheck
    (
    char *  pDevName  /* the name of the device being checked */
    )
```

This routine returns 1 if that device is an NPT driver, 0 (zero) otherwise, and **ERROR** (-1) if the specified device could not be found.

B.2.18 muxTkPollReceive( )

A network service sublayer uses **muxTkPollReceive( )** to poll a device for incoming data. If no data is available at the time of the call, **muxTkPollReceive( )** returns **EAGAIN**. The *pSpare* argument points to any optional spare data provided by an NPT driver. In the case of an END, *pSpare* will always be **NULL** or point to **NULL**.

```
STATUS muxTkPollReceive
    (
    void *   pCookie,   /* returned by muxTkBind() */
    M_BLK_ID pNBuff,    /* a vector of buffers passed to us */
    void *   pSpare     /* a reference to  spare data is returned here */
    )
```

This function returns **OK** on success; **EAGAIN** if no packet is available; **ENETDOWN**, if the cookie passed in does not represent a loaded device; or an error value specific to the *end/npt***pollReceive( )** routine registered for the particular driver.

B.2.19 muxTkPollSend( )

Use **muxTkPollSend( )**is to transmit packets when a driver is in polled-mode. This is the polled-mode equivalent to the interrupt-mode **muxTkSend( )**. When using **muxTkPollSend( )**, the driver does not need to call **muxAddressForm( )** to complete the packet, nor does it need to prepend an **mBlk** of type **MF_IFADDR** containing the destination address. Like **muxTkSend( )**, this function expects as arguments a cookie identifying the device and a pointer to the **mBlk** chain containing the data.

```
STATUS muxTkPollSend
    (
    void *   pCookie,     /* returned by muxTkBind()*/
    M_BLK_ID pNBuff,      /* data to be sent */
    char *   dstMacAddr,  /* destination MAC address */
    USHORT   netType,     /* network service that is calling us */
    void *   pSpareData   /* spare data passed to driver on each send */
    )
```

This function returns **OK**, on success; **ENETDOWN**, if the cookie passed in does not represent a valid device; or an error value specific to the *end/npt***pollSend( )** routine of the driver being used.

B.2.20 muxTkReceive( )

A driver uses **muxTkReceive( )** to pass validated packets up to the MUX.[1]

The **muxTkReceive( )** function forwards the data to the network service sublayer by calling the *stack***RcvRtn( )** registered for that sublayer.

Arguments to the function call include:

- a reference to the END object returned by the *end***Load( )** or *npt***Load( )** function
- an **mBlk** or **mBlk** chain that contains the received frame
- the offset value into the frame where the data field (the network service layer header) begins
- the network service type of the service for which the packet is destined[2]
- a flag (**wrongDstAddr**) which should be set to **TRUE** if the packet being received is not addressed to this device/service interface (which might happen if the driver is in **MUX_PROTO_PROMISC** mode)
- a reference to any optional data or information that a network service may expect to accompany the packet

[1] This function is registered by the MUX as the **receiveRtn** in the **END_OBJ** data structure for the device. The driver should make a call to this reference rather than calling **muxTkReceive( )** directly.

[2] Typically, this value can be found in the header of the received frame.

The MUX strips off the frame header before forwarding the packet to the network service, unless the network service is registered as **MUX_PROTO_SNARF** or **MUX_PROTO_PROMISC**, in which case it will receive the complete frame.

The function is defined as:

```
STATUS muxTkReceive
    (
    END_OBJ * pEnd,           /* returned by nptLoad() */
    M_BLK_ID  pMblk,          /* the buffer being received */
    long      netSvcOffset,   /* offset to network datagram in the packet */
    long      netSvcType,     /* network service type */
    BOOL      wrongDstAddr,   /* not addressed to this interface */
    void *    pSpareData      /* out-of-band data */
    )
```

This function returns **OK**, on success; **ERROR**, if the cookie is invalid; or **FALSE**, if no services are bound to the referenced driver.

B.2.21 muxTkSend( )

A network service sublayer uses **muxTkSend( )** to transmit packet.

```
STATUS muxTkSend
    (
    void *   pCookie,      /* returned by muxTkBind()*/
    M_BLK_ID pNBuff,       /* data to be sent */
    char *   dstMacAddr,   /* destination MAC address */
    USHORT   netType,      /* network service that is calling us */
    void *   pSpareData    /* spare data passed on each send */
    )
```

To send a packet, the caller must supply:

- the cookie obtained from bind that identifies the bound interface
- a pointer to the buffer chain (**mBlk** chain) containing the packet
- the physical layer address to which the packet is being sent
- the type of network service that is sending the packet

The data to be sent should be formed into an **mBlk** chain (if it is not already in this form). If the sublayer has a registered address resolution function for the service/device interface, it should call this function to determine the destination physical-layer address.

The network service may send fully formed physical layer frames to the device. For an END, this is the required and default behavior, but when sending to a device that uses an NPT driver this requires that you set the **M_L2HDR** flag in the **mBlk** header.

The **muxTkSend( )** routine may return an error indicating that the driver is out of resources for transmitting the packet. You can use this error to establish a flow control mechanism if desired. The sublayer typically waits to send any more packets until the MUX calls the **stackRestartRtn( )** callback function.

This function returns **OK**, if successful; **END_ERR_BLOCK**, if the **send( )** routine of the driver is temporarily unable to complete the send due to insufficient resources or some other problem; **ERROR**, if the **send( )** routine of the driver fails; or **ENETDOWN**, if the cookie does not represent a valid device.

B.2.22 muxTxRestart( )

A network interface driver uses **muxTxRestart( )** to tell a network service that it may resume sending data. That network service is presumed to have paused itself in response to an error returned from a **muxTkSend( )** call. A driver can use **muxTxRestart( )** to implement flow control.

```
void muxTxRestart
    (
    END_OBJ *     pEnd  /* returned by endLoad() or nptLoad() */
    )
```

B.2.23 muxUnbind( )

A network service uses **muxUnbind( )** to disconnect from a device. As input, **muxUnbind( )** expects a cookie that identifies the device, the driver type that was passed during the bind, and a pointer to the **stack*RcvRtn( )** registered at bind-time.

```
STATUS muxUnbind
    (
    void *      pCookie,         /* returned from muxTkBind() */
    long        type,            /* the device type passed in at bind-time */
    FUNCPTR     stackRcvRtn      /* pointer to the service receive routine */
    )
```

This function returns **OK**, if the device is successfully unbound; or **ERROR** otherwise (if **errno** is set to **EINVAL**, the device was not bound to the service when this function was called).

B.3 Data Structures

This section provides descriptions for the following structures:

- **DEV_OBJ**
- **END_ERR**
- **END_OBJ**
- **END_QUERY**
- **LL_HDR_INFO**
- **M2_INTERFACETBL**
- **mBlk**
- **MULTI_TABLE**
- **NET_FUNCS**

B.3.1 DEV_OBJ

The MUX uses the **DEV_OBJ** structure to store the name and control structure of your device. The private control structure, held in the **pDevice** member of this structure, stores information such as memory pool addresses and other essential data. The **DEV_OBJ** structure is defined in **end.h** as:

```
typedef struct dev_obj
    {
    char   name[END_NAME_MAX];         /* device name */
    int    unit;                       /* for multiple units */
    char   description[END_DESC_MAX];  /* text description */
    void * pDevice;                    /* device control structure */
    } DEV_OBJ;
```

name
A pointer to a string specifying the name of the network device.

unit
The unit number of the device. Unit numbers start at zero and increase for each device controlled by the same driver.

description
A text description of the device driver. For example, the Lance Ethernet driver uses the **description** string of "AMD 7990 Lance Ethernet Enhanced Network Driver." This string is displayed if **muxShow( )** is called.

pDevice
A pointer to the private control structure used by the device. A device can access its own control structure by using the **devObject.pDevice** member of the **END_OBJ** that the MUX passes into driver functions.

B.3.2 END_ERR

The **END_ERR** structure is defined as:

```
typedef struct end_err
    {
    INT32   errCode;     /* Error code */
    char *  pMesg;       /* NULL-terminated error message */
    void *  pSpare;      /* Pointer to user-defined data */
    } END_ERR;
```

The **errCode** member of the **END_ERR** structure is 32 bits long. The lower 16 bits are reserved for system error messages, while the upper 16 bits may be used for custom error messages. Table B-2 lists currently defined error codes:

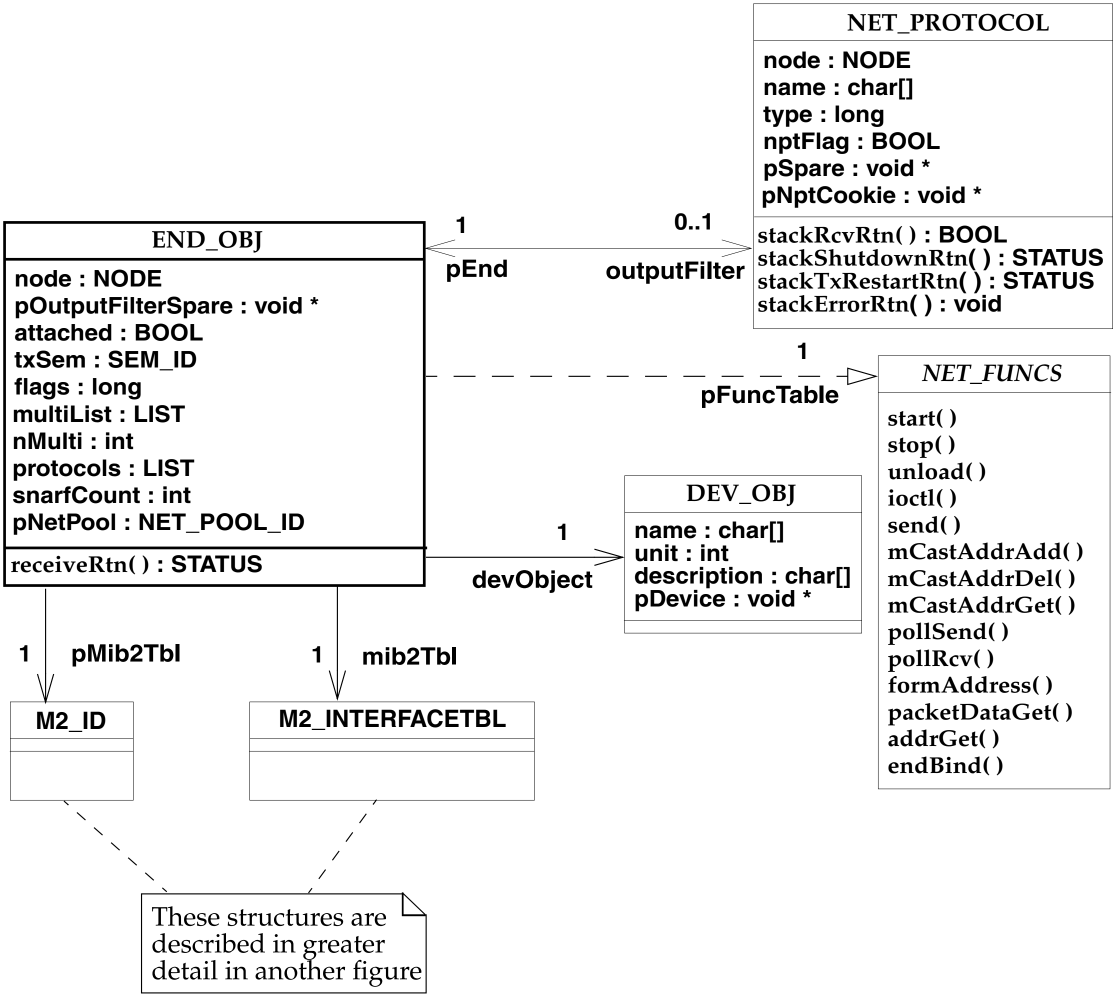
Table B-2 **END_ERR Error Codes**

Error Code	Description
END_ERR_INFO	This error is informational only.
END_ERR_WARN	A non-fatal error has occurred.
END_ERR_RESET	An error occurred that forced the device to reset itself, but the device has recovered.
END_ERR_FLAGS	The driver has changed the **flags** member of the **END_OBJ** structure.
END_ERR_DOWN	A fatal error occurred that forced the device to go down. The device can no longer send or receive packets.
END_ERR_UP	The device was down but has now come up and may again send and receive packets.

B.3.3 END_OBJ

END_OBJ is the head of the structural interface between the MUX and a network interface driver. The driver allocates this structure and initializes some of its elements within its *end***Load( )** or *npt***Load( )** function. The structure is defined in **target/h/end.h** and is diagramed in Figure B-1.

Figure B-1 **The END_OBJ Structure and Related Structures**



The MUX manages some of the elements in this structure, but the driver is responsible for setting and managing others:

node

The root of the device hierarchy. The MUX sets the value of this member. The driver should not modify the value of this item.

devObject

A pointer to the **DEV_OBJ** structure for this device (see *B.3.1 DEV_OBJ*, p.284). The driver must set this value when its **endLoad( )** or **nptLoad( )** function is called.

receiveRtn

A function pointer that references a **muxReceive( )** function. The MUX supplies this pointer when the driver is loaded. Any time after the completion of the **muxDevLoad( )** call, the driver can use this **receiveRtn( )** to pass data up to the protocol layer. The prototype for this receive routine is:

```
STATUS receiveRtn
    (
    void *   pCookie, /* The cookie passed in to endLoad() */
    M_BLK_ID pMblk    /* The packet, as an mblk chain */
    )
```

outputFilter

A function pointer that references an optional output filtering routine. This is set by the MUX to the **stackRcvRtn( )** of a network service that registers itself as of the **MUX_PROTO_OUTPUT** type, if there is such a service.

pOutputFilterSpare

The optional output filter's spare pointer, corresponding to the *pSpare* argument passed during a call to **muxBind( )**, or the *netCallbackId* argument passed during a call to **muxTkBind( )**. The MUX sets this element during the bind phase for those services registered as of the **MUX_PROTO_OUTPUT** type.

attached

A boolean indicating whether or not the device is attached. The MUX sets and manages this value.

txSem

A semaphore that controls access to the device's transmission facilities. The MUX initializes **txSem**, but the driver gives and takes the semaphore as needed.

flags

A value constructed by **OR**ing combinations of the following flags:

IFF_ALLMULTI

This device receives all multicast packets.

IFF_BROADCAST

The broadcast address is valid.

IFF_DEBUG

Debugging is on.

IFF_LINK0
A per link layer defined bit.

IFF_LINK1
A per link layer defined bit.

IFF_LINK2
A per link layer defined bit.

IFF_LOOPBACK
This is a loopback net.

IFF_MULTICAST
The device supports multicast.

IFF_NOARP
There is no address resolution protocol.

IFF_NOTRAILERS
The device must avoid using trailers.

IFF_OACTIVE
Transmission in progress.

IFF_POINTOPOINT
The interface is a point-to-point link.

IFF_PROMISC
This device receives all packets.

IFF_RUNNING
The device has successfully allocated needed resources.

IFF_SIMPLEX
The device cannot hear its own transmissions.

IFF_UP
The interface driver is up.

⚠ **WARNING:** If the driver changes the state of the **flags** element (for example, marking itself as "down" by setting the **IFF_DOWN** flag), the driver should export this change by calling **muxError( )** with the **errCode** member of the **END_ERR** structure set to **END_ERR_FLAGS**.

pFuncTable
A pointer to a **NET_FUNCS** structure (see *B.3.9 NET_FUNCS*, p.293). This structure contains pointers to driver routines for handling standard requests such as stop or send. Your driver must allocate and initialize this structure when the **endLoad( )** or **nptLoad( )** routine is called.

mib2Tbl

An **M2_INTERFACETBL** structure used to track the MIB-II variables used in the driver (see *B.3.6 M2_INTERFACETBL and M2-ID*, p.290). The driver should initialize this structure, although the elements in the structure will be used and adjusted both by the driver and by the MUX. For appropriate values for the elements in this structure, see *RFC 1158*.

multiList

A list of multicast addresses. The MUX sets and manages this list, but it uses the driver's *npt***MCastAddrAdd( )**, *npt***MCastAddrDel( )**, and *npt***MCastAddrGet( )** to do so.

nMulti

The number of addresses on the list referenced by the **multiList** member described above. The MUX sets this value using the information returned by the driver's *npt***MCastAddrGet( )** routine.

protocols

A list of services that have bound themselves to this network driver. The MUX manages this list.

snarfCount

A counter that indicates the number of snarf protocols.

pNetPool

A pointer to a **netBufLib**-managed memory pool. This pool, which is used internally by the NPT, should be initialized in the *end***Load( )** or *npt***Load( )** routine.

B.3.4 END_QUERY

This structure is designed specifically for use within the **EIOCQUERY** *ioctl* command.

```
typedef struct
    {
    int  query;           /* the query */
    int  queryLen;        /* length of expected/returned data */
    char queryData[4];    /* 4 byte minimum; 120 byte maximum */
    } END_QUERY;
```

B.3.5 LL_HDR_INFO

The MUX uses the **LL_HDR_INFO** structure to keep track of link-level header information associated with packets passed from an END to the MUX and from there up to a protocol. An **LL_HDR_INFO** structure is passed as an argument to an END's stack receive routine.

The **LL_HDR_INFO** structure is defined as:

```
typedef struct llHdrInfo
    {
    int  destAddrOffset;  /* destination address offset into mBlk */
    int  destSize;        /* size of destination address */
    int  srcAddrOffset;   /* source address offset into mBlk */
    int  srcSize;         /* size of source address */
    int  ctrlAddrOffset;  /* control info offset into mBlk */
    int  ctrlSize;        /* size of control info */
    int  pktType;         /* type of the packet */
    int  dataOffset;      /* offset into mBlk where data starts */
    } LL_HDR_INFO;
```

B.3.6 M2_INTERFACETBL and M2-ID

The **M2_INTERFACETBL** is still part of the **END_OBJ** structure for the purpose of backwards-compatibility, but a new structure, the **M2_ID** structure, has been added that encompasses and enhances the **M2_INTERFACETBL** structure.

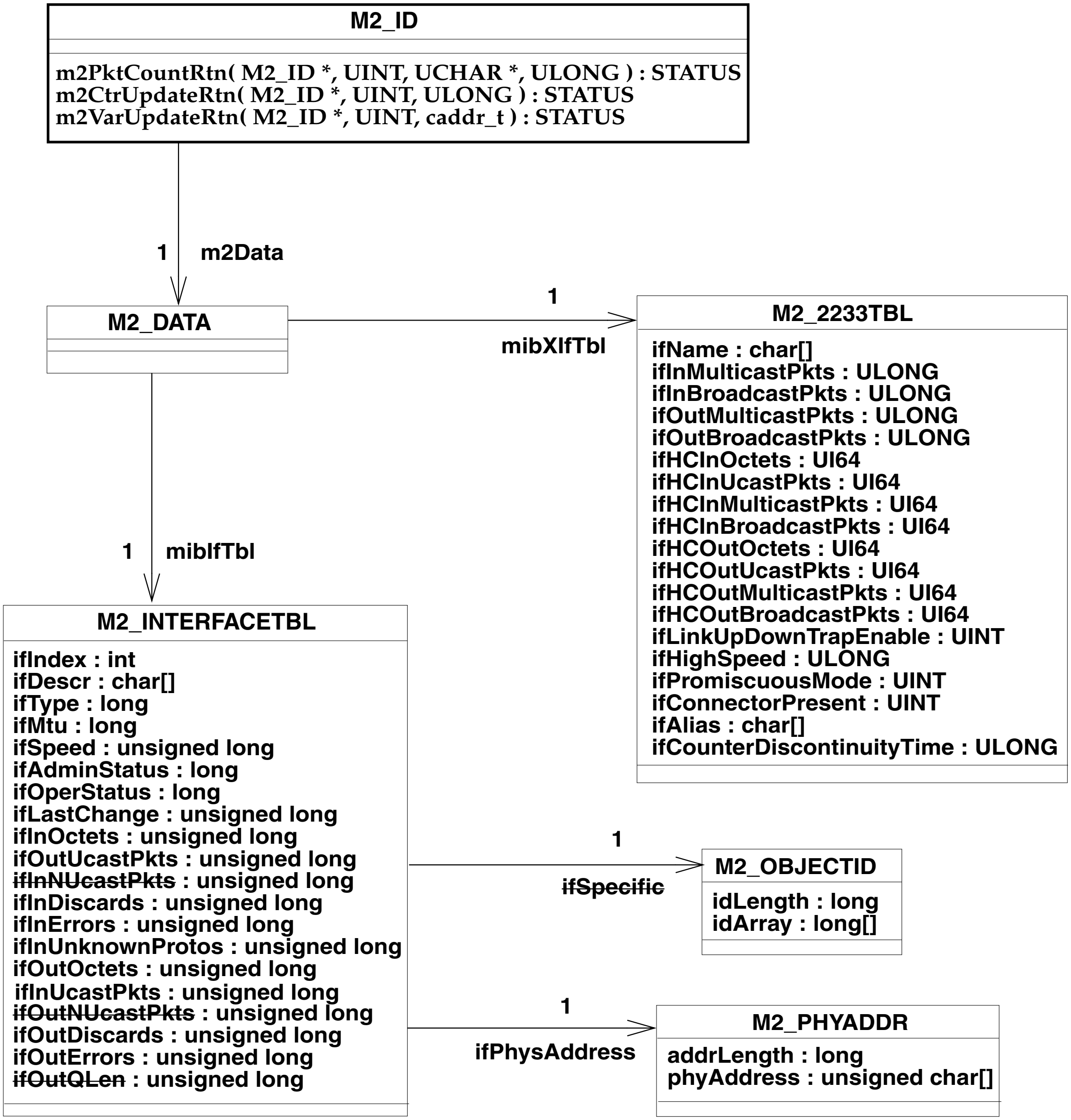
The **M2_ID** structure referenced in the **END_OBJ** includes the elements that were found in the **M2_INTERFACETBL** structure, and extends this with additional elements from *RFC 2233* (see Figure B-2).

B.3.7 mBlk

Use **mBlk** structures as a vehicle for passing packets between the driver and protocol layers. The **mBlk** structure is defined in **netBufLib.h** as:

```
typedef struct mBlk
    {
    M_BLK_HDR  mBlkHdr;     /* block header, pointer to mHdr structure */
    M_PKT_HDR  mBlkPktHdr;  /* packet header, pointer to pktHdr structure */
    CL_BLK *   pClBlk;      /* pointer to cluster block */
    } M_BLK;
```

Figure B-2 **The M2_ID Structure and Related Structures**


M2_ID
m2PktCountRtn(M2_ID *, UINT, UCHAR *, ULONG) : STATUS
m2CtrUpdateRtn(M2_ID *, UINT, ULONG) : STATUS
m2VarUpdateRtn(M2_ID *, UINT, caddr_t) : STATUS
1
m2Data
M2_DATA
1
mibXIfTbl
M2_2233TBL
ifName : char[]
ifInMulticastPkts : ULONG
ifInBroadcastPkts : ULONG
ifOutMulticastPkts : ULONG
ifOutBroadcastPkts : ULONG
ifHCInOctets : UI64
ifHCInUcastPkts : UI64
ifHCInMulticastPkts : UI64
ifHCInBroadcastPkts : UI64
ifHCOutOctets : UI64
ifHCOutUcastPkts : UI64
ifHCOutMulticastPkts : UI64
ifHCOutBroadcastPkts : UI64
ifLinkUpDownTrapEnable : UINT
ifHighSpeed : ULONG
ifPromiscuousMode : UINT
ifConnectorPresent : UINT
ifAlias : char[]
ifCounterDiscontinuityTime : ULONG
1
mibIfTbl
M2_INTERFACETBL
ifIndex : int
ifDescr : char[]
ifType : long
ifMtu : long
ifSpeed : unsigned long
ifAdminStatus : long
ifOperStatus : long
ifLastChange : unsigned long
ifInOctets : unsigned long
ifOutUcastPkts : unsigned long
~~ifInNUcastPkts~~ : unsigned long
ifInDiscards : unsigned long
ifInErrors : unsigned long
ifInUnknownProtos : unsigned long
ifOutOctets : unsigned long
ifInUcastPkts : unsigned long
~~ifOutNUcastPkts~~ : unsigned long
ifOutDiscards : unsigned long
ifOutErrors : unsigned long
~~ifOutQLen~~ : unsigned long
1
~~ifSpecific~~
M2_OBJECTID
idLength : long
idArray : long[]
1
ifPhysAddress
M2_PHYADDR
addrLength : long
phyAddress : unsigned char[]

NOTE: Text in ~~strikethrough~~ indicates elements deprecated in RFC 2233.

The elements of an **mBlk** structure are:

mBlkHdr
A pointer to an **mHdr** structure. If you chain this **mBlk** to another, you will need to set the value of **mBlkHdr.mNext** or **mBlkHdr.mNextPkt** or both. The **mNext** element is used to point to the next **mBlk** in a chain of **mBlk**s, while the **mNextPkt** element is used to point to an **mBlk** that contains the head of the next packet.

mBlkPktHdr
A pointer to a **pktHdr** structure. Drivers attached to IP using **ipAttach( )** must set **mBlkPktHdr.len** so that the IP receive routine can locate the IP header.

pClBlk
A pointer to a **clBlk** structure. If you are using the **netBufLib** routines to manage the driver's memory pool, you will have no reason to access or modify this member. If you are using your own driver memory pool management routines, you will have to change the **pClBlk.pClFreeRtn** member to point to your own memory free routine. This routine must use the same API as the **netBufLib** free routine and you will have to update the **pClBlk.pFreeArg1**, **pClBlk.pFreeArg2**, and **pClBlk.pFreeArg3** members.

Setting appropriate values for the members of an **mBlk** structure and the structures referenced by an **mBlk** structure is most easily accomplished by calling the appropriate **netBufLib** routines for the creation of an **mBlk**/**clBlk**/cluster construct.

B.3.8 MULTI_TABLE

The **MULTI_TABLE** structure is defined as follows:

```
typedef struct multi_table
    {
    long    len;       /* length of table, in bytes */
    char *  pTable;    /* pointer to entries */
    } MULTI_TABLE;
```

B.3.9 NET_FUNCS

The MUX uses this structure to reference the functions implemented for a driver. The **NET_FUNCS** structure is defined as:

```
typedef struct net_funcs
    {
    STATUS    (* start)(void *);
    STATUS    (* stop)(void *);
    STATUS    (* unload)(void *);
    int       (* ioctl)(void *, int, caddr_t);
    STATUS    (* send)(void *, M_BLK_ID);
    STATUS    (* mCastAddrAdd)(void *, char*);
    STATUS    (* mCastAddrDel)(void *, char*);
    STATUS    (* mCastAddrGet)(void *, MULTI_TABLE*);
    STATUS    (* pollSend)(void *, M_BLK_ID);
    STATUS    (* pollRcv)(void *, M_BLK_ID);
    M_BLK_ID  (* formAddress)(M_BLK_ID, M_BLK_ID, M_BLK_ID, BOOL );
    STATUS    (* packetDataGet)(M_BLK_ID, LL_HDR_INFO *);
    STATUS    (* addrGet)(M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
    int       (* endBind)(void *, void *, void *, long );
    } NET_FUNCS;
```

The driver functions referred to in this structure are described in greater detail elsewhere (see *NPT Driver Entry Points Exported to the MUX*, p. 207 and *END Entry Points Exported to the MUX*, p. 193).

C
PPP, SLIP, and CSLIP

C.1 Introduction

NOTE: The PPP, SLIP and CSLIP implementations described in this appendix are now deprecated for future use and will be removed from the next major release of Tornado. For more information on the discontinuance of these features, please contact your local Wind River account manager.

If you require a PPP solution, please ask your Wind River account manager about WindNet PPP. WindNet PPP is a reliable and manageable PPP solution built upon an extensible Remote Access Framework.

PPP Implementation Limitations

This PPP implementation is very limited. It works with a serial driver only, not with other such options as a modem or an Ethernet driver. It does not provide an API for modem support, and modem drivers are not provided for it. It is limited to sixteen connections. It also supports only standard CHAP, not the Microsoft CHAP extensions.

This implementation of PPP was originally intended for debugging purposes and to provide an additional means for downloading a boot image. This version of PPP should *not* be used for remote access applications.

C.2 Serial Driver Support

The VxWorks target can support IP communication with the host operating system over serial connections using the following protocols:

- Serial Line IP (SLIP)
- Compressed Serial Line IP (CSLIP)

SLIP and CSLIP (SLIP with compressed headers) provide a simple form of encapsulation for IP datagrams on serial lines. Using SLIP or CSLIP as a network interface driver is a straightforward way to use TCP/IP software with point-to-point configurations such as long-distance telephone lines or RS-232 serial connections between machines.

C.2.1 SLIP and CSLIP Configuration

Configuring your system for SLIP requires configuring both target and host systems. See your host system's manual for information on configuring your host.

⚠ **CAUTION:** If you choose to use CSLIP, remember to make sure your host is also using CSLIP. If your host is configured for SLIP, the VxWorks target receives packets from the host, but the host cannot correctly decode the CSLIP packets from the target. Eventually TCP resends the packets as SLIP packets, at which time the host receives and acknowledge them. However, the whole process is slow. To avoid this, configure the host and target to use the same serial protocol.

To use SLIP with your VxWorks target, make the following configuration changes (for more information on configuring VxWorks, see the *Tornado User's Guide: Projects*):

1. Reconfigure VxWorks to include SLIP support. The relevant configuration parameter is **INCLUDE_SLIP**.
2. Specify the device to be used for the SLIP connection, the SLIP Channel Identifier. The relevant configuration parameter is **SLIP_TTY**. By default this is set to **1**, which sets the serial device to **/tyCo/1**.
3. Specify the baud rate or SLIP Channel Speed (optional). The relevant configuration parameter is **SLIP_BAUDRATE**. If this is not defined, SLIP uses the baud rate defined by your serial driver.
4. Specify the SLIP Channel Capacity (optional). The relevant configuration parameter is **SLIP_MTU**. If you do not set this, the default value (576) will be used.

5. You can force the use of CSLIP when communicating with the host by setting the Transmit Header Compression Flag. The relevant configuration parameter is **CSLIP_ENABLE**.
6. Otherwise, you can allow the use of plain SLIP unless the VxWorks target receives a CSLIP packet (in which case the target also uses CSLIP) by setting the Receive Header Compression Flag. The relevant configuration parameter is **CSLIP_ALLOW**.

⚠ **CAUTION:** If you want to use VxSim for Solaris with PPP as the backend, you must configure VxWorks without BSD 4.3 compatibility. (The relevant configuration parameter is **BSD43_COMPATIBLE**). Otherwise, you get an exception in the WDB task when the target server tries to connect to the WDB agent.

C.3 PPP, the Point-to-Point Protocol for Serial Line IP

PPP provides for the encapsulation of data in frames. It also supports the following protocols:

- Link Control Protocol (LCP)
- Internet Protocol Control Protocol (IPCP)
- Password Authentication Protocol (PAP)
- Challenge-Handshake Authentication Protocol (CHAP)

This implementation of PPP includes three main components:

- A method for encapsulating multi-protocol datagrams.
- A Link Control Protocol (LCP) for establishing, configuring, and testing the data-link connection.
- A family of Network Control Protocols (NCPs) for establishing and configuring different network-layer protocols.

Reference Material on PPP

The following is a list of Requests for Comments (RFCs) associated with this unsupported PPP implementation:

RFC 1332: The PPP Internet Protocol Control Protocol (IPCP)

RFC 1334: PPP Authentication Protocols

RFC 1548: The Point-to-Point Protocol (PPP)

The USENET news group, **comp.protocols.ppp**, is dedicated to the discussion of PPP-related issues. Information presented in this forum is often of a general nature (such as equipment, setup, or troubleshooting), but technical details concerning specific PPP implementations are discussed as well.

C.3.1 PPP Configuration

Configuring your environment for PPP requires both host and target software installation and configuration. See your host's operating system manual for information on installing and configuring PPP on your host.[1]

To include the default PPP configuration, configure VxWorks with PPP support. The relevant configuration parameter is **INCLUDE_PPP**.

⚠ **CAUTION:** A VxWorks image that includes PPP sometimes fails to load. This failure is due to the static maximum size of the VxWorks image allowed by the loader. This problem can be fixed by either reducing the size of the VxWorks image (by removing unneeded options), or by burning new boot ROMs. If you receive a warning from **vxsize** when building VxWorks, or if the size of your image becomes greater than that supported by the current setting of **RAM_HIGH_ADRS**, see *Creating Bootable Applications* in the *Tornado User's Guide: Cross-Development* for information on how to resolve the problem.

You can include the optional DES cryptographic package for use with the Password Authentication Protocol (PAP). The relevant configuration parameter is **INCLUDE_PPP_CRYPT**. It is not included in the standard Tornado Release; contact your WRS Sales Representative to inquire about the availability of this optional package.

The DES package allows user passwords to be stored in encrypted form on the VxWorks target. If the package is installed, then it is useful only when the VxWorks target is acting as a PAP server, that is, when VxWorks is authenticating the PPP peer. Its absence does not preclude the use of PAP. For detailed information about using the DES package with PAP, see *Using PAP*, p.305).

[1] If your host operating system does not provide PPP facilities, you can use a publicly available implementation. One popular implementation for SunOS 4.1.*x* (and several other hosts) is the PPP version 2.1.2 implementation provided in the **unsupported/ppp-2.1.2** directory. This code is publicly available and is included only as a convenience. This code is not supported by Wind River Systems.

There are three methods of configuration:

- At compile-time, by reconfiguring VxWorks as described in the *Tornado User's Guide: Projects*. Use this method with **usrPPPInit( )**. (See *Initializing a PPP Link*, p.302.)
- At run-time, by filling in a PPP options structure. Use this method with **pppInit( )**. (See *Initializing a PPP Link*, p.302.)
- At run-time, by setting options in a PPP options file. Use this method with either **usrPPPInit( )** or **pppInit( )**. You can also use it to change the selection of PPP options previously configured by one of the other two configuration methods, although this assumes that the PPP options file is readable without using the PPP link (for example, an options file located on a target's local disk).

Each of these methods is described in a section that follows. For brief descriptions of the various PPP options, see *C.3.4 PPP Option Descriptions*, p.310.

Setting PPP Options when Configuring VxWorks

The various configuration options offered by this PPP implementation can be initialized at build-time by defining a number of configuration parameters.

NOTE: See the *Tornado User's Guide* for information on how to set configuration parameters.

First, make sure the **PPP_OPTIONS_STRUCT** configuration parameter is set (it is set by default). Unless **PPP_OPTIONS_STRUCT** configuration parameter is set, these configuration options cannot be enabled.

Then, specify the default serial interface that will be used by **usrPPPInit( )** by setting the **PPP_TTY** configuration parameter. Configuration options can be selected using configuration constants only when **usrPPPInit( )** is invoked to initialize PPP. Specify the number of seconds **usrPPPInit( )** will wait for a PPP link to be established between a target and peer by defining the **PPP_CONNECT_DELAY** configuration parameter. Table 1 lists the principal configuration parameters used with PPP.

Table 1 **PPP Configuration Parameters**

Constant	Purpose
INCLUDE_PPP	Include PPP. *

Table 1 **PPP Configuration Parameters** *(Continued)*

Constant	Purpose
INCLUDE_PPP_CRYPT	Include DES cryptographic package.†
PPP_OPTIONS_STRUCT	Enable configuration parameters.
PPP_TTY	Define default serial interface.
PPP_CONNECT_DELAY	Define time-out delay for link establishment.

* If you want to use VxSim for Solaris with PPP as the backend, you must configure VxWorks with BSD 4.3 compatibility off. The relevant configuration parameter is **BSD43_COMPATIBLE**. Otherwise, you get an exception in the WDB task when the target server tries to connect to the WDB agent.

† This option is not included in the standard Tornado Release; contact your Wind River Sales Representative to inquire about the availability of this optional package.

The full list of configuration options available with PPP appears under *C.3.4 PPP Option Descriptions*, p.310. By default, all of these options are disabled.

Setting **PPP_OPTIONS_STRUCT**, **PPP_TTY**, and **PPP_CONNECT_DELAY** as well as any additional configuration parameters, constitutes a modification to the configuration. These changes do not actually take effect until after you have recompiled VxWorks and initialized PPP. To initialize PPP, call **usrPPPInit( )**. You can make this call manually from a target shell (see *Initializing a PPP Link*, p.302).

Setting PPP Options Using an Options Structure

PPP options may be set at run-time by filling in a PPP options structure and passing the structure location to the **pppInit( )** routine. This routine is the standard entry point for initializing a PPP link (see *Initializing a PPP Link*, p.302).

The PPP options structure is **typedef**ed to **PPP_OPTIONS**, and its definition is located in **h/netinet/ppp/options.h**, which is included through **h/pppLib.h**.

The first field of the structure is an integer, **flags**, which is a bit field that holds the **OR**ed value of the **OPT_***option* macros displayed under *C.3.4 PPP Option Descriptions*, p.310. Definitions for **OPT_***option* are located in **h/netinet/ppp/options.h**. The remaining structure fields in column 2 are character pointers to the various PPP options specified by a string.

The following code fragment is one way to set configuration options using the PPP options structure. It initializes a PPP interface that uses the target's second serial port (**/tyCo/1**). The local IP address is 90.0.0.1; the IP address of the remote peer is

90.0.0.10. The baud rate is the default rate for the *tty* device. The VJ compression and authentication options have been disabled, and LCP (Link Control Protocol) echo requests have been enabled.

```
PPP_OPTIONS pppOpt;   /* PPP configuration options */

void routine ()
    {
    pppOpt.flags = OPT_PASSIVE_MODE | OPT_NO_PAP | OPT_NO_CHAP |
                    OPT_NO_VJ;
    pppOpt.lcp_echo_interval = "30";
    pppOpt.lcp_echo_failure = "10";

    pppInit (0, "/tyCo/1", "90.0.0.1", "90.0.0.10", 0, &pppOpt, NULL);
    }
```

Setting PPP Options Using an Options File

PPP options are most conveniently set using an options file. There is one restriction: the options file must be readable by the target without there being an active PPP link. Therefore the target must either have a local disk or RAM disk or an additional network connection. For more information about using file systems, see *VxWorks Programmer's Guide: Local File Systems*.

This configuration method can be used with either **usrPPPInit( )** or **pppInit( )**. It also can be used to modify the selection of PPP options previously configured using configuration parameters or the option structure **PPP_OPTION**.

When using **usrPPPInit( )** to initialize PPP, define the configuration parameter **PPP_OPTIONS_FILE** to be the absolute pathname of the options file (**NULL** by default). When using **pppInit( )**, pass in a character string that specifies the absolute pathname of the options file.

The options file format is one option per line; comment lines begin with **#**. For a description of option syntax, see the manual entry for **pppInit( )**.

The following code fragment generates the same results as the code example in *C.3.4 PPP Option Descriptions*, p.310. The difference is that the configuration options are obtained from a file rather than a structure.

```
pppFile = "mars:/tmp/ppp_options"; /* PPP config. options file */

void routine ()
    {
    pppInit (0, "/tyCo/1", "90.0.0.1", "90.0.0.10", 0, NULL, pppFile);
    }
```

In this example, **mars:/tmp/ppp_options** is a file that contains the following:

```
passive
no_pap
no_chap
no_vj
lcp_echo_interval 30
lcp_echo_failure 10
```

C.3.2 Using PPP

After it is configured and initialized, PPP attaches itself into the VxWorks TCP/IP stack at the driver (link) layer. After a PPP link has been established with the remote peer, all normal VxWorks IP networking facilities are available; the PPP connection is transparent to the user.

Initializing a PPP Link

A PPP link is initialized by calls to either **usrPPPInit( )** or **pppInit( )**. When either of these routines is invoked, the remote peer should be initialized. When a peer is running in passive mode, it must be initialized first (see *C.3.4 PPP Option Descriptions*, p.310.).

You can initialize the PPP interface by calling **usrPPPInit( )**:

– From the VxWorks shell.
– By user application code.

Use either syntax when calling **usrPPPInit( )**:

```
usrPPPInit ("bootDevice", unitNum, "localIPAddress", "remoteIPAddress")
usrPPPInit ("bootDevice", unitNum, "localHostName", "remoteHostName")
```

You can use host names in **usrPPPInit( )** provided the hosts have been previously added to the host database. For example, you can call **usrPPPInit( )** in the following way:

```
usrPPPInit ("ppp=/tyCo/1,38400", 1, "147.11.90.1", "147.11.90.199")
```

The **usrPPPInit( )** routine calls **pppInit( )**, which initializes PPP with the configuration parameters that were specified at compile-time (see *Setting PPP Options when Configuring VxWorks*, p.299). The **pppInit( )** routine can be called multiple times to initialize multiple channels.[2] The connection timeout is specified by **PPP_CONNECT_DELAY**. The return value of this routine indicates whether the link has been successfully established. If the return value is **OK**, the network connection should be fully operational.

The **pppInit( )** routine is the standard entry point for initializing a PPP link. All available PPP options can be set using parameters specified for this routine (see *C.3.4 PPP Option Descriptions*, p.310). Unlike **usrPPPInit( )**, the return value of **pppInit( )** does not indicate the status of the PPP link; it merely reports whether the link could be initialized. To check whether the link is actually established, call **pppInfoGet( )** and make sure that the state of IPCP is **OPENED**. The following code fragment demonstrates use of this mechanism for PPP unit 2:

```
PPP_INFO    pppInfo;

if ((pppInfoGet (2, &pppInfo) == OK) &&
    (pppInfo.ipcp_fsm.state == OPENED))
    return (OK);                                /* link established */
else
    return (ERROR);                             /* link down */
```

Deleting a PPP Link

There are two ways to delete a PPP link:

- By receiving a terminate request packet from the peer.
- By calling **pppDelete( )** to terminate the link.

Merely deleting the VxWorks tasks that control PPP or rebooting the target severs the link only at the TCP/IP stack, but does not delete the link on the remote peer end.

The return value of **pppDelete( )** does not indicate the status of the PPP link. To check whether the link is actually terminated, call **pppInfoGet( )** and make sure the return value is **ERROR**. The following code fragment demonstrates the usage of this mechanism for PPP unit 4:

```
    PPP_INFO    pppInfo;

    if (pppInfoGet (4, &pppInfo) == ERROR)
        return (OK);                                /* link terminated */
    else
        return (ERROR);                             /* link still up */
```

[2] The **usrPPPInit( )** routine can specify the unit number as a parameter. If this number is omitted, PPP defaults to 0.

PPP Authentication

PPP provides security through two authentication protocols: PAP and CHAP. This section introduces the use of PPP link-layer authentication and describes the format of the secrets files.

NOTE: This version of CHAP does *not* support *RFC 2433 – Microsoft PPP CHAP Extensions* ("MS-CHAP" or "CHAP-0x80"), which is used by some NT servers.

In this implementation, the default behavior of PPP is to provide authentication when requested by a peer but not to require authentication from a peer. If additional security is required, choose PAP or CHAP by enabling the corresponding option. This PPP implementation can act as a *client* (the peer authenticating itself) or a *server* (the authenticator).

Authentication for both PAP and CHAP is based on *secrets*, selected from a *secrets file* or from the secrets database built by the user (which can hold both PAP and CHAP secrets). A secret is represented by a record, which itself is composed of fields. The secrets file and the secrets database contain secrets that authenticate other clients, as well as secrets used to authenticate the VxWorks client to its peer. In the case that a VxWorks target cannot access the secrets file through the file system, use **pppSecretAdd( )** to build a secrets database.

Secrets files for PAP and CHAP use identical formats. A *secrets record* is specified in a file by a line containing at least three words that specify the contents of the fields *client*, *server*, and *secret*, in that order. For PAP, *secret* is a password that must match the password entered by the client seeking PAP authentication. For CHAP, both client and server must have identical secrets records in their secrets files; the secret consists of a string of one or more words (for example, "an unguessable secret").

Table 2 is an example of a secrets file. It could be either a PAP or CHAP secrets file, since their formats are identical.

At the time of authentication, for a given record, PPP interprets any words following *client*, *server*, and *secret* as acceptable IP addresses for the *client* and *secret* specified. If there are only three words on the line, it is assumed that any IP address is acceptable; to disallow all IP addresses, use a dash (**-**). If the secret starts with an @, what follows is assumed to be the name of a file from which to read a secret. An asterisk (*) as the client or server name matches any name. When authentication is initiated, a best-match algorithm is used to find a match to the secret, meaning that, given a client and server name, the secret returned is for the closest match found.

Table 2 **Secrets File Format**

client	*server*	*secret*	**IP address**
vxTarget	**mars**	**"vxTargetSECRET"**	
venus	**vxTarget**	**"venusSECRET"**	**147.11.44.5**
*****	**mars**	**"an unguessable secret"**	
venus	**vxTarget**	**"venusSECRET"**	**-**
vxTarget	**mars**	**@host:/etc/passwd**	

On receiving an authentication request, PPP checks for the existence of secrets either in an internal secrets database or in a secrets file. If PPP does not find the secrets information, the connection is terminated.

The secrets file contains secrets records used to authenticate the peer, and those used to authenticate the VxWorks client to the peer. Selection of a record is based on the local and remote names. By default, the local name is the host name of the VxWorks target, unless otherwise set to a different name by the option **local_auth_name** in the options file. The remote name is set to a **NULL** string by default, unless otherwise set to a name specified by the option **remote_auth_name** in the options file. (Both **local_auth_name** and **remote_auth_name** can be specified in two other forms, as can other configuration options listed under *C.3.4 PPP Option Descriptions*, p.310.)

Using PAP. The default behavior of PPP is to authenticate itself if requested by a peer but not to require authentication from a peer. For PPP to authenticate itself in response to a server's PAP authentication request, it only requires access to the secrets. For PPP to act as an authenticator, you must turn on the PAP configuration option.

Secrets can be declared in a file or built into a database. The secrets file for PAP can be specified in one of the following ways:

- By reconfiguring VxWorks with the PSP file specified. The relevant configuration parameter is **PPP_STR_PAP_FILE**.
- By setting the **pap_file** member of the **PPP_OPTIONS** structure passed to **pppInit( )**.
- By adding the following line entry in the PPP options file specified in your configuration:

```
pap_file  /xxx/papSecrets
```

If the VxWorks target is unable to access the secrets file, call **pppSecretAdd( )** to build a secrets database.

If PPP requires the peer to authenticate itself using PAP, the necessary configuration option can be set in one of the following ways:

1. By reconfiguring VxWorks with PAP required. The relevant configuration parameter is **PPP_OPT_REQUIRE_PAP**.
2. By setting the flag **OPT_REQUIRE_PAP** in the **flags** bit field of the **PPP_OPTIONS** structure passed to **pppInit( )**;
3. By adding the following line entry in the options file.

```
require_pap
```

Secrets records are first searched in the secrets database; if none are found there, then the PAP secrets file is searched. The search proceeds as follows:

- **VxWorks as an authenticator:** PPP looks for a secrets record with a *client* field that matches the user name specified in the PAP authentication request packet and a *server* field matching the local name. If the password does not match the secrets record supplied by the secrets file or the secrets database, it is encrypted, provided the optional DES cryptographic package is installed. Then it is checked against the secrets record again. Secrets records for authenticating the peer can be stored in encrypted form if the optional DES package is used. If the login option was specified, the user name and the password specified in the PAP packet sent by the peer are checked against the system password database. This enables restricted access to certain users.
- **VxWorks as a client:** When authenticating the VxWorks target to the peer, PPP looks for the secrets record with a *client* field that matches the user name (the local name unless otherwise set by the PAP user name option in the options file) and a *server* field matching the remote name.

Using CHAP. The default behavior of PPP is to authenticate itself if requested by a peer but not to require authentication from a peer. For PPP to authenticate itself in response to a server's CHAP authentication request, it only requires access to the secrets. For PPP to act as an authenticator, you must turn on the CHAP configuration option.

CHAP authentication is instigated when the authenticator sends a challenge request packet to the peer, which responds with a challenge response. Upon receipt of the challenge response from the peer, the authenticator compares it with the expected response and thereby authenticates the peer by sending the required acknowledgment. CHAP uses the MD5 algorithm for evaluation of secrets.

The secrets file for CHAP can be specified in any of the following ways:

- By reconfiguring VxWorks with the CHAP file specified. The relevant configuration parameter is **PPP_STR_CHAP_FILE**.
- By setting the **chap_file** member of the **PPP_OPTIONS** structure passed to **pppInit( )**.
- By adding the following line entry in the options file:

```
chap_file  /xxx/chapSecrets
```

If PPP requires the peer to authenticate itself using CHAP, the necessary configuration option can be set in one of the following ways:

- By reconfiguring VxWorks with CHAP required. The relevant configuration parameter is **PPP_OPT_REQUIRE_CHAP**.
- By setting the flag **OPT_REQUIRE_CHAP** in the **flags** bit field of the **PPP_OPTIONS** structure passed to **pppInit( )**.
- By adding the following line entry in the options file:

```
require_chap
```

Secrets are first searched in the secrets database; if none are found there, then the CHAP secrets file is searched. The search proceeds as follows:

- **VxWorks as an authenticator:** When authenticating the peer, PPP looks for a secrets record with a *client* field that matches the name specified in the CHAP response packet and a *server* field matching the local name.
- **VxWorks as a client:** When authenticating the VxWorks target to the peer, PPP looks for the secrets record with a *client* field that matches the local name and a *server* field that matches the remote name.

Connect and Disconnect Hooks

PPP provides connect and disconnect hooks for use with user-specific software. Use the **pppHookAdd( )** routine to add a connect hook that executes software before initializing and establishing the PPP connection or a disconnect hook that executes software after the PPP connection has been terminated. The **pppHookDelete( )** routine deletes connect and disconnect hooks.

The routine **pppHookAdd( )** takes three arguments: the unit number, a pointer to the hook routine, and the hook type (**PPP_HOOK_CONNECT** or **PPP_HOOK_DISCONNECT**). The routine **pppHookDelete( )** takes two arguments:

the unit number and the hook type. The hook type distinguishes between the connect hook and disconnect hook routines.

Two arguments are used to call the connect and disconnect hooks: *unit*, which is the unit number of the PPP connection, and *fd,* the file descriptor associated with the PPP channel. If the user hook routines return **ERROR**, then the link is gracefully terminated and an error message is logged.

⚠ **CAUTION:** In VxWorks AE, hooks such as the connect and disconnect hooks described here *must* be in the kernel domain.

The code in Example 1 demonstrates how to hook the example routines, **connectRoutine( )** and **disconnectRoutine( )**, into the PPP connection establishment mechanism and termination mechanism, respectively.

Example 1 **Using Connect and Disconnect Hooks**

```
#include <vxWorks.h>
#include <pppLib.h>

void attachRoutine(void);
static int connectRoutine(int, int);
static int disconnectRoutine(int, int);


void attachRoutine(void)
    {
                     /* add connect hook to unit 0 */
    pppHookAdd(0, connectRoutine, PPP_HOOK_CONNECT);
                     /* add disconnect hook to unit 0 */
    pppHookAdd(0, disconnectRoutine, PPP_HOOK_DISCONNECT);
    }

static int connectRoutine
    (
    int unit,
    int fd
    )
    BOOL connectOk = FALSE;

                     /* user specific connection code */
                      {
                      .......................
                     connectOk = TRUE;
    if(connectOk)
        {
        return(OK);
    else
        return(ERROR);
    }
```

```
static int disconnectRoutine
    (
    int unit,
    int fd
    )
    {
    BOOL disconnectOk = FALSE;
                     /* user specific code */
    {
                      ..........................................
    disconnectOk = TRUE;
    }
    if(disconnectOk)
       return(OK);
    else
       return(ERROR);
    }
```

C.3.3 Troubleshooting PPP

Because of the complex nature of PPP, you may encounter problems using it in conjunction with VxWorks. Give yourself the opportunity to get familiar with running VxWorks configured with PPP by starting out using a default configuration. Additional options for the local peer should be disabled. (You can always add these options later.) Problems with PPP generally occur in either of two areas: when establishing links and when using authentication. The following sections offer checklists for troubleshooting errors that have occurred during these processes.

Link Establishment

The link is the basic operating element of PPP; a proper connection ensures the smooth functioning of PPP, as well as VxWorks. The following steps should help resolve simple problems encountered when establishing a link.

1. Make sure that the serial port is connected properly to the peer. A null modem may be required.
2. Make sure that the serial driver is correctly configured for the default baud rate of 9600, no parity, 8 DATA bits, and 1 STOP bit.
3. Make sure that there are no problems with the serial driver. PPP may not work if there is a hang up in the serial driver.
4. Start the PPP daemon on the peer in the passive mode.

5. Boot the VxWorks target and start the PPP daemon by typing:

   ```
   % usrPPPInit
   ```

   If no arguments are supplied, the target configures the default settings. If a timeout error occurs, reconfigure VxWorks with a larger connect delay time. The relevant configuration parameter is **PPP_CONNECT_DELAY**. By default, the delay is set to 15 seconds, which may not be sufficient in some environments.

6. Once the connection is established, add and test additional options.

Authentication

Authentication is one of the more robust features of this PPP implementation. The following steps may help you troubleshoot basic authentication problems.

1. Turn on the debug option for PPP. The relevant configuration parameter is **PPP_OPT_DEBUG**. You can also use the alternative options in *C.3.4 PPP Option Descriptions*, p.310. By turning on the debug option, you can witness various stages of authentication.
2. If the VxWorks target has no access to a file system, use **pppSecretAdd( )** to build the secrets database.
3. Make sure the secrets file is accessible and readable.
4. Make sure the format of the secrets file is correct.
5. PPP uses the MD5 algorithm for CHAP authentication of secrets. If the peer tries to use a different algorithm for CHAP, then the CHAP option should be turned off.
6. Turn off the VJ compression. It can be turned on after you get authentication working.

C.3.4 PPP Option Descriptions

This section lists all the configurable options supported by this PPP implementation. You can configure each of these options from three different locations:

- the VxWorks image configuration tool
  (see *Setting PPP Options when Configuring VxWorks*, p.299)

- a **PPP_OPTIONS** structure passed into **pppInit( )**
  (see *Setting PPP Options Using an Options Structure*, p.300)
- a PPP options file
  (see *Setting PPP Options Using an Options File*, p.301)

If you set the same option using more than one of the above methods, the option settings specified in the options file **PPP_OPTIONS_FILE** take precedence over any set using the VxWorks image configuration tool or by passing a **PPP_OPTIONS** structure into **pppInit( )**. For example:

- If VxWorks is configured with the use of PAP negated, a subsequent setting of **require_pap** in **PPP_OPTIONS_FILE** overrides the earlier setting enabling PAP authentication. The relevant configuration parameter is **PPP_OPT_NO_PAP**.
- If **char * netmask** has been passed in the options structure **PPP_OPTIONS** to **pppInit( )** with a value of **FFFF0000**, and **netmask FFFFFF00** is passed in **PPP_OPTIONS_FILE** to **usrPPPInit( )**, the network mask value is set to **FFFFFF00**.

Table C-1 **Configuration Options for PPP**

Set in config.h	Set Using Options Structure	Set Using Options File
PPP_OPT_DEBUG	**OPT_DEBUG**	**debug**
	Enable PPP daemon debug mode.	
PPP_OPT_DEFAULT_ROUTE	**OPT_DEFAULT_ROUTE**	**default_route**
	After IPCP negotiation is successfully completed, add a default route to the system routing tables. Use the peer as the gateway. This entry is removed when the PPP connection is broken.	
PPP_OPT_DRIVER_DEBUG	**OPT_DRIVER_DEBUG**	**driver_debug**
	Enable PPP driver debug mode.	
PPP_OPT_IPCP_ACCEPT_LOCAL	**OPT_IPCP_ACCEPT_LOCAL**	**ipcp_accept_local**
	Set PPP to accept the remote peer's idea of the target's local IP address, even if the local IP address was specified.	
PPP_OPT_IPCP_ACCEPT_REMOTE	**OPT_IPCP_ACCEPT_REMOTE**	**ipcp_accept_remote**
	Set PPP to accept the remote peer's idea of its (remote) IP address, even if the remote IP address was specified.	

Table C-1 **Configuration Options for PPP**

Set in config.h	Set Using Options Structure	Set Using Options File
PPP_OPT_LOGIN	**OPT_LOGIN**	**login**
	Use the login password database for PAP authentication of peer.	
PPP_OPT_NO_ACC	**OPT_NO_ACC**	**no_acc**
	Disable address/control compression.	
PPP_OPT_NO_ALL	**OPT_NO_ALL**	**no_all**
	Do not request/allow any options.	
PPP_OPT_NO_CHAP	**OPT_NO_CHAP**	**no_chap**
	Do not allow CHAP authentication with peer.	
PPP_OPT_NO_IP	**OPT_NO_IP**	**no_ip**
	Disable IP address negotiation in IPCP.	
PPP_OPT_NO_MN	**OPT_NO_MN**	**no_mn**
	Disable magic number negotiation.	
PPP_OPT_NO_MRU	**OPT_NO_MRU**	**no_mru**
	Disable MRU (Maximum Receive Unit) negotiation.	
PPP_OPT_NO_PAP	**OPT_NO_PAP**	**no_pap**
	Do not allow PAP authentication with peer.	
PPP_OPT_NO_PC	**OPT_NO_PC**	**no_pc**
	Disable protocol field compression.	
PPP_OPT_NO_VJ	**OPT_NO_VJ**	**no_vj**
	Disable VJ (Van Jacobson) compression.	
PPP_OPT_NO_VJCCOM	**OPT_NO_ASYNCMAP**	**no_asyncmap**
	Disable async map negotiation.	
PPP_OPT_NO_VJCCOMP	**OPT_NO_VJCCOMP**	**no_vjccomp**
	Disable VJ (Van Jacobson) connection ID compression.	
PPP_OPT_PASSIVE_MODE	**OPT_PASSIVE_MODE**	**passive_mode**
	Set PPP in passive mode so it waits for the peer to connect, after an initial attempt to connect.	
PPP_OPT_PROXYARP	**OPT_PROXY_ARP**	**proxy_arp**
	Add an entry to this system's ARP (Address Resolution Protocol) table with the IP address of the peer and the Ethernet address of this system.	

Table C-1 **Configuration Options for PPP**

Set in config.h	Set Using Options Structure	Set Using Options File
PPP_OPT_REQUIRE_CHAP	**OPT_REQUIRE_CHAP**	**require_chap**
	Require CHAP authentication with peer.	
PPP_OPT_REQUIRE_PAP	**OPT_REQUIRE_PAP**	**require_pap**
	Require PAP authentication with peer.	
PPP_OPT_SILENT_MODE	**OPT_SILENT_MODE**	**silent_mode**
	Set PPP in silent mode. PPP does not transmit LCP packets to initiate a connection until a valid LCP packet is received from the peer.	
PPP_STR_ASYNCMAP	**char * asyncmap**	**asyncmap** *value*
	Set the desired async map to the specified value.	
PPP_STR_CHAP_FILE	**char * chap_file**	**chap_file** *file*
	Get CHAP secrets from the specified file. This option is necessary if either peer requires CHAP authentication.	
PPP_STR_CHAP_INTERVAL	**char * chap_interval**	**chap_interval** *value*
	Set the interval in seconds for CHAP rechallenge to the specified value.	
PPP_STR_CHAP_RESTART	**char * chap_restart**	**chap_restart** *value*
	Set the timeout in seconds for the CHAP negotiation to the specified value.	
PPP_STR_ESACAPE_CHARS	**char * escape_chars**	**escape_chars** *value*
	Set the characters to escape on transmission to the specified values.	
PPP_STR_IPCP_MAX_CONFIGURE	**char * ipcp_max_configure**	**ipcp_max_configure** *value*
	Set the maximum number of transmissions for IPCP configuration requests to the specified value.	
PPP_STR_IPCP_MAX_FAILURE	**char * ipcp_max_failure**	**ipcp_max_failure** *value*
	Set the maximum number of IPCP configuration NAKs to the specified value.	
PPP_STR_IPCP_MAX_TERMINATE	**char * ipcp_max_terminate**	**ipcp_max_terminate** *value*
	Set the maximum number of transmissions for IPCP termination requests to the specified value.	
PPP_STR_IPCP_RESTART	**char * ipcp_restart**	**ipcp_restart** *value*
	Set the timeout in seconds for the IPCP negotiation to the specified value.	

Table C-1 **Configuration Options for PPP**

Set in config.h	Set Using Options Structure	Set Using Options File
PPP_STR_LCP_ECHO_FAILURE	**char * lcp_echo_failure**	**lcp_echo_failure** *value*
	Set the maximum consecutive LCP echo failures to the specified value.	
PPP_STR_LCP_ECHO_INTERVAL	**char * lcp_echo_interval**	**lcp_echo_interval** *value*
	Set the interval in seconds for the LCP negotiation to the specified value.	
PPP_STR_LCP_MAX_CONFIGURE	**char * lcp_max_configure**	**lcp_max_configure** *value*
	Set the maximum number of transmissions for LCP configuration requests to the specified value.	
PPP_STR_LCP_MAX_FAILURE	**char * lcp_max_failure**	**lcp_max_failure** *value*
	Set the maximum number of LCP configuration NAKs to the specified value.	
PPP_STR_LCP_MAX_TERMINATE	**char * lcp_max_terminate**	**lcp_max_terminate** *value*
	Set the maximum number of transmissions for LCP termination requests to the specified value.	
PPP_STR_LCP_RESTART	**char * lcp_restart**	**lcp_restart** *value*
	Set the timeout in seconds for the LCP negotiation to the specified value.	
PPP_STR_LOCAL_AUTH_NAME	**char * local_auth_name**	**local_auth_name** *name*
	Set the local name for authentication to the specified name.	
PPP_STR_MAX_CHALLENGE	**char * max_challenge**	**max_challenge** *value*
	Set the maximum number of transmissions for CHAP challenge requests to the specified value.	
PPP_STR_MRU	**char * mru**	**mru** *value*
	Set MRU (Maximum Receive Unit) for negotiation to the specified value.	
PPP_STR_MTU	**char * mtu**	**mtu** *value*
	Set MTU (Maximum Transmission Unit) for negotiation to the specified value.	
PPP_STR_NETMASK	**char * netmask**	**netmask** *value*
	Set the network mask value for negotiation to the specified value.	

Table C-1 **Configuration Options for PPP**

Set in config.h	Set Using Options Structure	Set Using Options File
PPP_STR_PAP_FILE	**char * pap_file**	**pap_file** *file*
	Get PAP secrets from the specified file. This option is necessary if either peer requires PAP authentication.	
PPP_STR_PAP_MAX_AUTHREQ	**char * pap_max_authreq**	**pap_max_authreq** *value*
	Set the maximum number of transmissions for PAP authentication requests to the specified value.	
PPP_STR_PAP_PASSWD	**char * pap_passwd**	**pap_passwd** *passwd*
	Set the password for PAP authentication with the peer to the specified password.	
PPP_STR_PAP_RESTART	**char * pap_restart**	**pap_restart** *value*
	Set the timeout in seconds for the PAP negotiation to the specified value.	
PPP_STR_PAP_USER_NAME	**char * pap_user_name**	**pap_user_name** *name*
	Set the user name for PAP authentication with the peer to the specified name.	
PPP_STR_REMOTE_AUTH_NAME	**char * remote_auth_name**	**remote_auth_name** *name*
	Set the remote name for authentication to the specified name.	
PPP_STR_VJ_MAX_SLOTS	**char * vj_max_slots**	**vj_max_slots** *value*
	Set the maximum number of VJ compression header slots to the specified value.	

Index

IX