VxWorks™ Network
Programmer's Guide

5.4

Edition 1

WindRiver®
SYSTEMS

Corporate Headquarters	Europe	Japan
Wind River Systems, Inc.	Wind River Systems, S.A.R.L.	Wind River Systems K.K.
500 Wind River Way	19, Avenue de Norvège	Ebisu Prime Square Tower 5th Fl.
Alameda, CA 94501-1153	Immeuble B4, Bâtiment 3	1-1-39 Hiroo
USA	Z.A. de Courtaboeuf 1	Shibuya-ku
	91953 Les Ulis Cédex	Tokyo 150-0012
	FRANCE	JAPAN
toll free (US): 800/545-WIND		
telephone: 510/748-4100	telephone: 33-1-60-92-63-00	telephone: 81-3-5778-6001
facsimile: 510/749-2010	facsimile: 33-1-60-92-63-15	facsimile: 81-3-5778-6002

CUSTOMER SUPPORT

	Telephone	E-mail	Fax
Corporate:	800/872-4977 toll free, U.S. & Canada 510/748-4100 direct	support@wrs.com	510/749-2164
Europe:	33-1-69-07-78-78	support@wrsec.fr	33-1-69-07-08-26
Japan:	011-81-3-5467-5900	support@kk.wrs.com	011-81-3-5467-5877

If you purchased your Wind River Systems product from a distributor, please contact your distributor to determine how to reach your technical support organization.

Please provide your license number when contacting Customer Support.

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.

Chapter Overview

The first several chapters in this guide describe the network protocols supported by the standard VxWorks network stack and explain how to configure VxWorks to include a particular protocol or utility and how to configure the protocol or utility itself.

Chapter *13. Booting over the Network* explains how to boot VxWorks from the network. Included are instructions for booting over the Ethernet, the serial line (using PPP or SLIP), or the memory backplane.

Chapter *14. Upgrading 4.3 BSD Network Drivers* describes the issues associated with porting a 4.3 BSD network driver to work within the VxWorks network stack. You can choose between two upgrade paths: you can do a simple upgrade of the driver to the standard 4.4 BSD network driver interface; or, if your driver needs to support features such as multiple protocols, it should be rewritten to use the MUX interface.

Supported Protocols and Utilities

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

- SLIP, Serial Line IP

- CSLIP, Compressed Serial Line IP

- PPP, the Point-to-Point Protocol

- IP, Internet Protocol

- UDP, User Datagram Protocol

- TCP, Transmission Control Protocol

- DHCP, Dynamic Host Configuration Protocol

- BOOTP, Bootstrap Protocol

- DNS, Domain Name System

- ARP, Address Resolution Protocol, and Proxy ARP

- OSPF, Open Shortest Path First

- RIP, Routing Information Protocol

- Sockets (TCP, UPD, multicasting, routing, and Zbuf)

- RPC, Remote Procedure Calls

- RSH, Remote Shell

- FTP, File Transfer Protocol

- NFS, Network File System

- TFTP, Trivial File Transfer Protocol

- **rlogin**, Remote Login

- **telnet**, Remote Login

1.1.1 Network

One key to VxWorks's effective relationship with host development machines is its extensive networking facilities. By providing a fast, easy-to-use connection between the target and host systems, the network allows full use of the host machine as a development system, as a debugging host, and as a provider of non-real-time services in a final system.

VxWorks currently supports loosely coupled network connections over serial lines (using SLIP, CSLIP, or PPP) or Ethernet networks (IEEE 802.3). It also supports tightly coupled connections over a backplane bus using shared memory. The

standard 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.

MUX Interface

A standard BSD 4.3 network driver can be ported to VxWorks with little effort. However, VxWorks also supports an improved network driver interface. This interface, called the MUX[1], adds support for advanced features such as multicasting, polled-mode Ethernet, and zero-copy transmission. This interface also decouples the network driver and network protocol layers, allowing you add new network drivers without the need to alter the network protocol, or to add a new network protocol without the need to modify the existing MUX-based network interface drivers.

Porting a driver to the MUX interface involves more work than a simple port to BSD 4.4, but it is worth it if your driver must support multicasting, polled-mode Ethernet, and other advanced features. More information about the process of adding new drivers and protocols to the VxWorks network stack can be found in the *Network Protocol Toolkit User's Guide*.

Sockets

VxWorks provides standard BSD socket calls, which allow real-time VxWorks tasks and other processes to communicate in any combination with each other over the network. There are two sets of VxWorks socket calls: you can use sockets that are source-compatible with BSD 4.4 UNIX, or you can use the *zbuf socket interface* to streamline throughput.[2]

Any task can open one or more sockets, to which other sockets can be connected. 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 another process or VxWorks task on the same CPU or on another CPU, or with a process running under some other host operating system. Similarly, tasks using the zbuf socket interface are not aware of whether their communications partners are using standard sockets, or are also using the zbuf interface.

---

1. MUX: short for multiplexer, because it multiplexes access to physical network devices.
2. The TCP subset of the zbuf interface is sometimes called "zero-copy TCP."

For information on sockets, see *2.6 Networking APIs*, p.116, and the reference entries for **sockLib** and **zbufSockLib**.

Remote Procedure Calls (RPC)

Originally designed by Sun Microsystems using the Sun ONC standard and now available in the public domain, 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**.

Simple Network Management Protocol (WindNet SNMP)

The WindNet SNMPv1/v2c optional component allows VxWorks targets to be managed and configured remotely through SNMP (the Simple Network Management Protocol). Application developers can customize the SNMP management information base to include information specific to each application and environment.

For detailed information about WindNet SNMP, see the *WindNet SNMPv1/v2c VxWorks Component Release Supplement*.

Remote File Access: NFS, RSH, FTP, TFTP

Remote file access across the network is also available. A program running on VxWorks can use the host machine as a virtual file system. Files on any host machine can be accessed through the network as if they were local to the VxWorks system. A program running under VxWorks does not need to know where that file is, or how to access it. For example, **/dk/file** might be a file local to the VxWorks system, while **host:file** might be a file located on another machine entirely.

Conversely, VxWorks can allow host machines to use files maintained on VxWorks just as transparently – programs running on the host need not know that the files they use are maintained on the VxWorks system.

VxWorks includes the Sun Microsystems standard Network File System (NFS). VxWorks systems can run NFS clients, using files from other systems that export files over NFS, or run NFS servers, exporting files to other systems. Alternatively, VxWorks can use the following protocols to provide transparent remote file access:

- The Remote Shell protocol (RSH) can be used as a client, accessing files on UNIX host systems running an RSH server.

- The File Transfer Protocol (FTP) provides remote access to VxWorks files from other systems using FTP.

- The Trivial File Transfer Protocol (TFTP) provides read/write capability to and from a remote server.

See the reference entries for **nfsLib**, **remLib**, **ftpLib**, **ftpdLib**, **tftpLib**, and **tftpdLib**, and the following sections: *3.7.4 Network File System (NFS) Devices*, p.124, *2.10 File Access Applications*, p.153, and *3.7.5 Non-NFS Network Devices*, p.126.

Boot Parameter Access from Host

BOOTP is a basic bootstrap protocol which allows a booting target to configure itself dynamically by obtaining the required parameters from the host via the network, instead of using information encoded in the target's non-volatile RAM or ROM. The actual transfer of the boot image is performed by a file transfer program. BOOTP and TFTP are commonly used together for network booting.

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 subnetworks are assigned addresses with the same subnet number. Because they appear to reside on the same 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

Configuring the VxWorks Network Stack

2.1 Introduction

This chapter introduces the configuration and use of the standard VxWorks network stack, the details of which are given in subsequent chapters.

2.1.1 The Data Link Layer

Chapter *3. Data Link Layer Network Components* is a discussion of 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, a serial line, or even the shared-memory backplane. These drivers are the foundation of the network stack. For information on booting VxWorks using these drivers, see chapter *13. Booting over the Network*.

2.1.2 The MUX, TCP/IP, and Associated Protocols

After the chapter on the data link layer, chapter *4. TCP/IP Under VxWorks* introduces the MUX and the TCP/IP protocol suite. Under VxWorks, TCP/IP 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 standard VxWorks data link layer. For more information on the MUX, see the *Network Protocol Toolkit User's Guide*.

The discussion of IP, TCP, and UDP is primarily an overview that prepares you for a discussion of their configuration needs under VxWorks. However, this chapter

does describe ARP and Proxy ARP in some detail. 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 as one logical network. This proxy ARP scheme is an alternative to the use of explicit subnets for accessing the shared-memory network.

2.1.3 Network Configuration Protocols

The next group of protocols discussed, in chapter *5. Network Configuration Protocols*, are the network configuration protocols:

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

The networking stack can use either DHCP or BOOTP to set up and maintain its network configuration information. At boot time, both DHCP and BOOTP can provide IP addresses and related information. BOOTP assigns IP addresses permanently. The DHCP protocol extends BOOTP to allow the assignment of IP addresses on a temporary basis. Thus, the client receives an IP address on lease. When the lease expires, the client must renegotiate the lease. As a result, DHCP remains active during run-time.

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 network management station (NMS) from which you can remotely configure, monitor, and control network devices called *agents*. Thus, SNMP is a network configuration protocol, but in a very different sense of the term.

Beyond providing a few paragraphs of description, this manual does not discuss SNMP. For detailed information on using SNMP with VxWorks, see *WindNet SNMP VxWorks Optional Product Supplement.*

2.1.4 Routing Applications

Chapter *6. Dynamic Routing Protocols* discusses the routing applications:

- RIP, 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.

- OSPF, Open Shortest Path First

  Like RIP, OSPF updates the information in the routing tables. However, OSPF is more complex than RIP. This complexity enhances functionality. Thus, an OSPF router can handle inter-networks that are too large for RIP. Unfortunately, this complexity also makes OSPF much harder to configure.

2.1.5 Networking APIs

Chapter *7. Networking APIs* 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. Although this interface is WRS-specific, the interface can communicate with standard BSD sockets. Thus, the other end of the socket connection can use the standard BSD interface even if you chose to use the zbuf interface on the VxWorks side of the connection.

2.1.6 DNS, Domain Name System

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**. DNS is discussed briefly in chapter *8. DNS: Domain Name System*. For detailed information on DNS, see *RFC-1034* and *RFC-1035*.

2.1.7 SNTP, 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. SNTP is discussed briefly in chapter *9. SNTP: A Time Protocol*.

2.1.8 Remote Access Applications

Chapters *10. RPC: Remote Procedure Calls*, *11. File Access Applications*, and *12. rlogin and telnet: Host Access Applications* discuss the applications that provide remote access over the network. VxWorks supports the following:

- 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)

2.2 Advice Concerning the Use of config.h and configAll.h

There are times when VxWorks configuration must be fine-tuned beyond the capabilities of the configuration tool described in the *Tornado User's Guide: Projects*. In these cases, you should edit configuration header files manually to modify constants and macros.

The **config.h** file sets values for constants that the build uses to determine the contents of a VxWorks image. The tricky part of editing **config.h** is determining where you want to insert the **#define** or **#undef** statement. This is because all **config.h** files contain a **#include "configAll.h"** statement.

⚠ **WARNING:** You should avoid modifying the **configAll.h** file.

Within the **configAll.h** file, there are **#ifdef** blocks that depend on the value of a symbolic constant whose value you might want to modify. You should insert any change to **config.h** that modifies such constants *before* the inclusion of **configAll.h**.

2

However, **configAll.h** sometimes undefines a symbolic constant that you might want to define. In such cases, you should put your **#define** statement *after* the inclusion of **configAll.h**. As a consequence, before you modify a value in a BSP's **config.h** file, you must first check the **configAll.h** file for dependencies on the constant you want to modify.

If you modify the **config.h** file, be sure to refer to the *Tornado User's Guide: Projects* for advice on how to coordinate these changes with ones you make using the configuration tool.

3
Data Link Layer Network Components

3.1 Introduction

The data link layer consists of drivers that directly handle communication with the physical medium. It is their job to transmit and receive frames on the physical network medium. VxWorks includes three different classes of data link layer drivers: the Ethernet drivers; the Serial Line Interface Protocol (SLIP) driver; and the shared-memory network driver, which provides communication over a backplane. VxWorks also supports the creation of custom interface drivers.

3.2 Ethernet Driver Support

Ethernet is one medium among many over which the VxWorks network can operate. Ethernet is a local area network specification that is supported by numerous vendors. It is ideal for most VxWorks applications, but, with the exception of certain protocols, such as BOOTP and DHCP, nothing in either the VxWorks or host network systems is inherently tied to Ethernet.

If you are writing or porting an Ethernet driver to VxWorks, it should conform to the MUX interface for network drivers (for information on how to write a driver that works with the MUX, see the *Network Protocol Toolkit User's Guide*). This interface includes support for features such as multicasting and polled-mode Ethernet. If these features do not matter to you, a simpler port might be possible; see *14. Upgrading 4.3 BSD Network Drivers*. However, the MUX is the future of

network driver interfaces under VxWorks; eventually, all network drivers must be ported to the MUX.

3.3 Serial Line IP 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.

PPP also provides a simple form of encapsulation for IP datagrams on serial lines. However, unlike SLIP or CSLIP, PPP provides support for multiple protocols on a single serial line, dynamic negotiation of the IP addresses at each end, and much more. Of course, it comes at the cost of additional overhead with each frame, and extra frames when the link is first created.

⚠ **CAUTION:** The VxWorks implementation of PPP supports only IP. For more information on PPP, see *3.4 PPP, the Point-to-Point Protocol for Serial Line IP*, p.15.

3.3.1 Serial Line Driver 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 macro is **INCLUDE_SLIP.**

2. Specify the device to be used for the SLIP connection, the SLIP Channel Identifier. The relevant configuration macro 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 macro 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 macro 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 macro 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 macro is **CSLIP_ALLOW**.

⚠ **CAUTION:** If you want to use VxSim for Solaris with PPP as the backend, you must reconfigure VxWorks to remove BSD 4.3 compatability. (The relevant configuration macro is **BSD43_COMPATIBLE**). Otherwise, you get an exception in the WDB task when the target server tries to connect to the WDB agent.

3.4 PPP, the Point-to-Point Protocol for Serial Line IP

The VxWorks implementation of the PPP (Point-to-Point Protocol) is comprised of several different protocols that work together with the PPP network interface driver. Although PPP can, in theory, support a variety of protocols, the VxWorks implementation supports only the TCP/IP stack. The VxWorks PPP implementation is comprised of 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.

PPP is one method by which VxWorks can communicate with other operating systems over a serial line connection. PPP supports Internet Protocol (IP) layer networking software over point-to-point configurations, such as long-distance telephone lines or RS-232 serial connections between machines. If either end of a PPP connection has other network interfaces (such as Ethernet) and is able to forward packets to other machines, a PPP connection can serve as a gateway between networks.

The basic functionality provided by PPP is similar to that of the Serial Line Internet Protocol (SLIP), with the advantage that PPP is extensible and offers various configurable options.

PPP provides a standard method for transporting multi-protocol datagrams over point-to-point links. It is designed for simple links which transport packets between two peers. These links provide full-duplex, simultaneous operation and are assumed to deliver packets in the order in which they are sent. It is intended that PPP provide a common solution for easy connectivity among a variety of hosts, bridges, and routers.

3.4.1 Reference Material on PPP

The following is a list of relevant Requests for Comments (RFCs) associated with the VxWorks 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.

3.4.2 PPP Features

PPP supports the following features:

- **PPP client and server connection support** (either *active* or *passive* mode). In active mode (default), the PPP software attempts to initiate a PPP link with the peer. In passive mode, the PPP software waits for a peer to try to open a link.

- **Multiple unit support**. Up to 16 PPP interfaces can be active at any one time.

- **Asynchronous character mapping**. Users can specify control characters that should be escaped by the peer upon transmission to avoid misinterpretation by the serial driver library or by lower-level modem software.

- **Van Jacobsen (VJ) compression**. This feature reduces the regular 40-byte TCP/IP header to 3 or 8 bytes, thereby saving valuable link bandwidth.

- **Address, control, and protocol field compression**. These types of compression allow the PPP network interface driver to reduce the transmission of extraneous PPP header information, thereby saving valuable link bandwidth.

- **Link state and link statistics querying**. Internal PPP counters and protocol state information may be obtained through query routines. This enables applications to monitor and manage the PPP link.

- **IP address negotiation**. Using IP address negotiation, one peer may assign the other peer an IP address once the PPP link is established.

- **Echo request and reply**. One peer may request that the other peer respond to link-layer echoes. This allows for an automatic monitoring of the link's physical status.

- **Connect and disconnect hooks**. Use of connect and disconnect hooks allows applications to implement routines supporting modem control, dialing software, connection scripting, etc.

- **Challenge-Handshake Authentication Protocol (CHAP) and Password Authentication Protocol (PAP)**. These authentication protocols ensure that the remote peer is authorized to establish a PPP link and that the correct IP address is used.

- **Proxy ARP routing**. Use of this feature allows hosts on the proxy-server peer's connected network to access the proxy-client peer without manually adding routing entries.

3.4.3 The Point-to-Point Protocol Compared to SLIP

For many years, transferring Internet Protocol (IP) packets over serial lines was handled almost exclusively by the Serial Line Internet Protocol (SLIP). SLIP is a simple link-layer driver that is installed between IP stack code and a serial driver.

While SLIP uses less object code than PPP and processes packets more efficiently (using compressed headers in CSLIP), it can carry only IP packets and it is not extensible. Furthermore, SLIP has several different protocol implementations that do not always communicate smoothly with each other. Nevertheless, its general ease of use and large installed base has made it the *de facto* standard for using IP over point-to-point serial lines.

The Point-to-Point Protocol (PPP) was developed to address the shortcomings of SLIP. Unlike SLIP, PPP is being defined and tracked by the Internet Engineering Task Force (IETF), and the protocol specifications have been published in multiple Request For Comments (RFC) documents. Although SLIP is still an attractive choice for systems that only require basic IP-packet transfers, the advantages of PPP are prompting the rapid growth of its installed base.

PPP supports several features that make it more suitable than SLIP for certain applications:

▪ **Multi-Protocol Support**. PPP packet framing includes a protocol field in the header. This allows for the transfer of packets among different network-layer protocols over a link. At present, the only protocols supported by this PPP implementation are IP and the basic PPP protocols (LCP, IPCP, PAP, and CHAP).

▪ **Extensibility**. The protocol field in the frame header makes PPP able to accommodate new protocols (both public and proprietary). The Internet Assigned Numbers Authority (IANA) tracks the allocation of protocol field values.

▪ **Error Detection**. PPP framing also includes a Frame Check Sequence (FCS). This field automatically ensures the data integrity of every packet received by the PPP network interface driver. If an error is detected, the received packet is dropped and an input error is recorded.

▪ **Link Management**. The entire structure of PPP is based around the concept of a point-to-point *link* which is established between *peers* (the local and remote systems on either end of the serial connection). The link has several phases and states associated with its life and is managed by its own separate protocol, the Link Control Protocol (LCP). This concept of a link creates an environment that can support features like option negotiation, link-layer user authentication, link quality management, and loopback detection.

▪ **Option Negotiation**. PPP allows for the dynamic negotiation of options between peers. To some extent, this allows one end of the link to configure the peer. This is especially useful in heterogeneous environments where a PPP server may

need to assign certain properties to the peer, such as the Maximum Receive Unit (MRU).

- **Authentication**. PPP supports link-layer authentication through two widely used authentication protocols: PAP and CHAP. Both of these protocols check that the peer is authorized to establish a link with the local host by sending and/or receiving password information.

- **IP Address Negotiation**. Built into the PPP control protocol for IP is the ability to assign an IP address to a peer. This feature allows one peer to act as a PPP server and assign addresses as clients dial in. The IP address can be re-used when the PPP link is terminated.

While many applications do not require any of the features above, they may need to interact with other systems that are using PPP and not SLIP. These two protocols can *not* communicate with each other; this is perhaps the most compelling reason for using PPP.

3.4.4 An Overview of PPP

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)

Encapsulation

PPP encapsulation provides for simultaneous multiplexing of different network-layer protocols over the same link. The PPP encapsulation has been carefully designed to retain compatibility with most commonly used supporting hardware. The frame format of a standard PPP frame structure is shown in Figure 3-1.

Figure 3-1 **Format of Standard PPP Frame Structure**

Flag 01111110	Address 11111111	Control 00000011	Protocol 8/16 bits	Information **************	FCS 16/32 bits	Flag 01111110	Inter-frame or Next Address

Link Control Protocol (LCP)

In order to promote versatility and be portable to a wide variety of environments, PPP provides a Link Control Protocol (LCP). LCP is used when establishing links and negotiating a variety of configuration options. It is also used to create automatic agreement on encapsulation format options, to handle variable size limits placed on packets, to detect looped-back links and other common configuration errors, and to terminate links.

Other optional facilities provided by LCP include: authentication of the peer on the link by using authentication protocols such as PAP or CHAP, and determination when a link is functioning properly and when it is failing. After the link is established, PPP also allows an optional authentication. For more information, see RFC 1548.

Internet Protocol Control Protocol (IPCP)

The IP Control Protocol (IPCP) is the Network Control Protocol (NCP) for IP. IPCP is responsible for configuring, enabling, and disabling the IP protocol modules on both ends of the point-to-point link. It uses the same packet exchange mechanism as LCP. IPCP packets are not exchanged until PPP has established a link. IPCP is also responsible for IP address negotiation between peers. For more information, see RFC 1332.

Password Authentication Protocol (PAP)

The Password Authentication Protocol (PAP) provides a simple method by which the peer establishes its identity using a two-way handshake. This is done only upon the initial link creation. Once a link is established, an ID/password pair is sent repeatedly by the peer to the authenticator until authentication is acknowledged or the connection is terminated.

PAP is not a robust authentication protocol. Passwords are sent over the circuit "in the clear," without protection from playback or repeated trial-and-error attacks. The peer is in control of the frequency and timing of the attempts. This authentication method is most appropriately used when a plain-text password must be available to simulate a login at a remote host. For information about using PAP, see *Using PAP*, p. 33, or refer to RFC 1334.

Challenge-Handshake Authentication Protocol (CHAP)

Challenge-Handshake Authentication Protocol (CHAP) is a more robust authentication protocol offering better security than PAP. CHAP periodically verifies the identity of a peer using a three-way handshake. This is done after an initial link is established, and can be repeated later at any time.

After a link is established, the authenticator sends a "challenge" message to the peer. The peer responds with a value calculated by a one-way hash function. The authenticator checks the response against its own calculation of the expected hash value. If the values match, the authentication is acknowledged; otherwise the connection is terminated.

CHAP provides protection against playback attack by issuing ever-changing challenges at specified time intervals. The use of repeated challenges is intended to limit the time of exposure to any single attack. The authenticator is in control of the frequency and timing of the challenges.

CHAP authentication for any particular link relies on the use of a "secret" known only to the authenticator and the peer. The secret is not sent over the link; therefore the server and its peer must both have access to it. In Tornado, this is achieved using various methods explained in *Using CHAP*, p. 34. For further technical details, refer to RFC 1334.

3.4.5 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 macro is **INCLUDE_PPP**.

---

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.

⚠ **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 macro 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.33).

This PPP implementation includes many optional features (approximately 50 in all) that can be configured in to enable the PPP capabilities listed in *3.4.2 PPP Features*, p.16. 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.26.)

- At run-time, by filling in a PPP options structure. Use this method with ***pppInit( )***. (See *Initializing a PPP Link*, p.26.)

- At run-time, by setting options in a PPP options file. This method is used with either ***usrPPPInit( )*** or ***pppInit( )***, and can be used to change the selection of PPP options previously configured by one of the other two configuration methods, provided that the PPP options file can be read 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 Table 3-3.

Selecting PPP Options By Configuring VxWorks

The various configuration options offered by this PPP implementation can be initialized at compile-time by defining a number of configuration options.

3

> **NOTE:** See the *Tornado User's Guide: Projects* for information on how to set some configuration options through a graphical user interface, without directly editing **config.h**.

First, make sure the **PPP_OPTIONS_STRUCT** constant is defined in **config.h** (it is defined by default). Unless **PPP_OPTIONS_STRUCT** is defined, configuration options in **config.h** cannot be enabled.

Then, specify the default serial interface that will be used by *usrPPPInit( )* by defining the **PPP_TTY** constant. 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** constant. Table 3-1 lists the principal configuration constants used with PPP.

Table 3-1 **PPP Configuration Constants**

Constant	Purpose
INCLUDE_PPP	Include PPP. [*]
INCLUDE_PPP_CRYPT	Include DES cryptographic package.
PPP_OPTIONS_STRUCT	Enable configuration options set in **config.h**.
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 compatability off. The relevant configuration macro is **BSD43_COMPATIBLE**. Otherwise, you get an exception in the WDB task when the target server tries to connect to the WDB agent.

Table 3-2 shows the two basic formats used for configuration options in **config.h**. The full list of options available with PPP appears in column 1 of Table 3-3. By default, all of these options are disabled. To enable any **PPP_OPT_***option* setting, define its value to be 1 (these option constants are boolean values). To set any **PPP_STR_***optionstring* entry, define it by representing the desired value as a string. For example, to set **PPP_STR_MTU** to 1000, enter:

```
#define    PPP_STR_MTU    "1000"
```

Setting **PPP_OPTIONS_STRUCT**, **PPP_TTY**, and **PPP_CONNECT_DELAY** in **config.h**, as well as any configuration options, constitutes a modification to the

Table 3-2 **PPP Configuration Options in config.h**

Configuration Option	Option Included
PPP_OPT_option	Specify a PPP configuration option.
PPP_STR_optionstring	Specify a PPP configuration option string.

configuration file. 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.26) or can include it in the boot code (see *Booting over the Network*, p.185).

Selecting 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.26).

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 in column 2 of Table 3-3. 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);
    }
```

3

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 constants in **config.h** or the option structure **PPP_OPTION**.

When using *usrPPPInit( )* to initialize PPP, define the configuration constant **PPP_OPTIONS_FILE** to be the absolute path name of the options file (**NULL** by default). When using *pppInit( )*, pass in a character string that specifies the absolute path name 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 *Selecting PPP Options Using an Options Structure*, p.24. 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
```

3.4.6 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 *PPP Options*, p.27.)

The ***usrPPPInit( )*** routine is in **config/all/bootConfig.c** and **src/config/usrNetwork.c**. There are four ways it can be called:

If the boot device is set to **ppp**, ***usrPPPInit( )*** is called as follows:

– From **bootConfig.c** when booting from boot ROMs.
– From **usrNetwork.c** when booting from VxWorks boot code.

The PPP interface can also be initialized 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 options that were specified at compile-time (see *Selecting PPP Options By Configuring VxWorks*, p.22). 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 *Selecting PPP Options Using an Options Structure*, p.24). 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

---

2. The ***usrPPPInit( )*** routine can specify the unit number as a parameter. If this number is omitted, PPP defaults to 0.

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:

- When a terminate request packet is received 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 */
```

PPP Options

Table 3-3 lists all the configuration options supported by PPP. Each option is shown in its three forms, corresponding to the configuration methods explained in the following sections:

Column 1: *Selecting PPP Options By Configuring VxWorks*, p.22

Column 2: *Selecting PPP Options Using an Options Structure*, p.24

Column 3: *Setting PPP Options Using an Options File*, p.25.

A brief description of each option follows the three formats. Configuration options specified in the options file **PPP_OPTIONS_FILE** take precedence over any previously set in **config.h** or set by passing the structure **PPP_OPTIONS** to *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 macro 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 3-3   **PPP Configuration Options**

Options			Description
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.
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.

Table 3-3   **PPP Configuration Options** *(Continued)*

Options			Description
Set in config.h	**Set using options structure**	**Set using options file**	
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.
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.

Table 3-3   **PPP Configuration Options** *(Continued)*

Options			**Description**
Set in config.h	**Set using options structure**	**Set using options file**	
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.
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.

Table 3-3   **PPP Configuration Options** *(Continued)*

Options			Description
Set in config.h	**Set using options structure**	**Set using options file**	
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.
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.

PPP Authentication

PPP provides security through two authentication protocols: PAP (see *Password Authentication Protocol (PAP)*, p.20) and CHAP (see *Challenge-Handshake Authentication Protocol (CHAP)*, p.21). This section introduces the use of PPP link-layer authentication (introduced in *Link Control Protocol (LCP)*, p.20), and describes the format of the secrets files.

In VxWorks, 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. PPP in VxWorks 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 which 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 3-4 is an example of a secrets file. It could be either a PAP or CHAP secrets file, since their formats are identical.

Table 3-4 **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**	

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.

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 in Table 3-3.)

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 macro 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 macro 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 macro 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 macro 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.

The code in Example 3-1 demonstrates how to hook the example routines, *connectRoutine***( )** and *disconnectRoutine***( )**, into the PPP connection establishment mechanism and termination mechanism, respectively.

Example 3-1   **Using Connect and Disconnect Hooks**

```
#include <vxWorks.h>
#include <pppLib.h>

/* type declarations */

void        attachRoutine (void);
STATIC int  connectRoutine(int unit, int fd);
STATIC int  disconnectRoutine(int unit, int fd);

void attachRoutine (void)
    {
    /* add connect hook to unit 0 */
    pppHookAdd (0, connectRoutine, PPP_CONNECT_HOOK);

    /* add disconnect hook to unit 0 */
    pppHookAdd (0 , disconnectRoutine, PPP_DISCONNECT_HOOK);
    }

STATIC int connectRoutine
    (
    int     unit,
    int     fd
    )

    {
    BOOL    connectOk  = FALSE;

    /* user specfic connection code */
        {
        ..........
        connectOk = TRUE;
        }
    if (connectOk)
        return (OK);
    else
        return (ERROR);
    }
```

3

```
STATIC int disconnectRoutine
    (
    int     unit,
    int     fd
    )
    {
    BOOL disconnectOk = FALSE;
    /* user specific code */
        {
        ..........
        disconnectOk = TRUE;
        }
    if (disconnectOk)
        return (OK);
    else
        return (ERROR);
    }
```

3.4.7 PPP with Tornado

PPP can be used in two ways in the Tornado environment. The PPP link can serve as an additional network interface apart from the existing default network interface, or it can be the default network interface on the target, causing PPP to serve as a network back end for the target server on the host.

PPP Link as an Additional Network Interface

1. To use this option, rebuild the VxWorks image with PPP included. For more information on how to include PPP, see *3.4.5 PPP Configuration*, p.21.

2. Boot the image from the regular Tornado boot ROM.

3. Start the Tornado target server and launch Tornado.

4. Start the Tornado shell, and invoke *usrPPPInit***( )** from the shell. You can also use *pppInit***( )** from an application to configure the PPP link. For more information on these routines, see *Initializing a PPP Link*, p.26.

PPP Link as a Network Back End for the Target Server

1. Configure VxWorks with PPP capability. The relevant configuration macro is **INCLUDE_PPP**. Make new boot ROMs for the target with this configuration. For more information, see *3.4.5 PPP Configuration*, p.21.

2. Rebuild a new VxWorks image for the target.

3. Configure and start the **pppd** daemon on the host. For example, on a Sun host using SunOS, the following command can be run to start the daemon:

    ```
    % pppd passive /dev/ttyb 38400
    ```

4. Change the boot configuration parameters to use the PPP link. For example:

    ```
    [VxWorks Boot}: c
    boot device          : ppp,38400
    processor number     : 0
    host name            : host
    file name            : /usr/wind/target/config/mv177/vxWorks
    inet on ethernet (e) : 90.0.0.165:ffffff00
    host inet (h)        : 90.0.0.5
    gateway inet (g)     : 90.0.0.5
    user (u)             : thardy
    flags (f)            : 0x4
    target name (tn)     : luna
    ```

5. After booting you should see messages similar to the following:

    ```
    Attaching network interface ppp0...
    ppp0: ppp 2.1.2 started by
    ppp0: Connect: ppp0 <--> /tyCo/1
    ppp0: local  IP address 90.0.0.165
    ppp0: remote IP address 90.0.0.5
    done.
    Attaching network interface lo0... done.
    Loading... 361620 + 70448 + 34350
    Starting at 0x1000...

    Attaching network interface ppp0...
    ppp0: ppp 2.1.2 started by
    ppp0: Connect: ppp0 <--> /tyCo/1
    ppp0: local  IP address 90.0.0.165
    ppp0: remote IP address 90.0.0.5
    done.
    Attaching network interface lo0... done.
    NFS client support not included.

                    VxWorks
    Copyright 1984-1995  Wind River Systems, Inc.
             CPU: Motorola MVME177
         VxWorks: 5.3
      BSP version: 1.1/0
    Creation date: Jan 26 1996
             WDB: Ready.
    ```

You are now ready to start the target server and run Tornado. For more information on starting Tornado refer to the *Tornado User's Guide*.

3

The PPP connection is a network back end established on a serial link. When using the PPP link to communicate with the target, all the Tornado tools work in the same way as any other network back end. (See the *Tornado User's Guide*.)

⚠ **CAUTION:** System-level debugging is not available when using the PPP link. To perform system-level debugging, use the regular serial back end described in the *Tornado User's Guide*.

3.4.8 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. (These can always be added 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. If, however, difficulties using PPP with VxWorks persist, contact the Wind River Systems technical support organization.

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 macro 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 PPP for VxWorks. The following steps may help you troubleshoot basic authentication problems.

1.   Turn on the debug option for PPP. The relevant configuration macro is **PPP_OPT_DEBUG**. You can also use the alternative options in Table 3-3. 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.

3.5 Shared-Memory Network on the Backplane

The VxWorks network can also be used for communication among multiple processors on a common *backplane*. In this case, data is passed through shared memory. This is implemented in the form of a standard network driver so that all the higher levels of network components are fully functional over this shared-memory "network." Thus, all the high-level network facilities provided over an Ethernet are also available over the shared-memory network.

3

A multiprocessor backplane bus contains a separate Internet network. Each shared-memory network has its own network/subnet number. As usual, each processor (host) on the shared-memory network has a unique Internet address.

⚠ **CAUTION:** This is different if you are using proxy ARP. See *4.7 ARP and Proxy ARP for Transparent Subnets*, p.84 for additional information.

In the example shown in Figure 3-2, two CPUs are on a backplane. The shared-memory network's Internet address is 161.27.0.0. Each CPU on the shared-memory network has a unique Internet address, 161.27.0.1 for **vx1** and 161.27.0.2 for **vx2**.

Figure 3-2   **Shared-Memory Network**
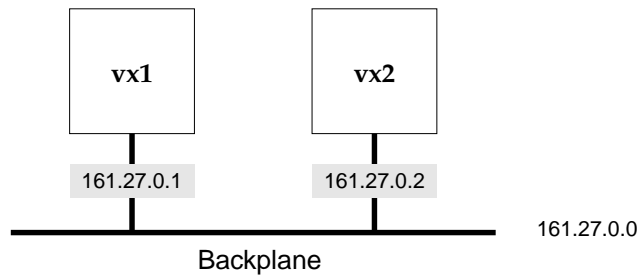
The routing capabilities of the VxWorks IP layer allow processors on a shared-memory network to reach systems on other networks over a *gateway* processor on the shared-memory network. The gateway processor has connections to both the shared-memory network and an external network. These connections allow higher-level protocols to transmit data between any processor on the shared-memory network and any other host or target system on the external network.

The low-level data transfer mechanism of the shared-memory network driver is also available directly. This allows alternative protocols to be run over the shared-memory network in addition to the standard ones.

The following features allow the VxWorks shared-memory network driver to send network packets from one processor on the backplane to another:

▪ Packets are transferred across the backplane through a pool of *shared memory* that can be accessed by all processors on the backplane.

▪ Access to the shared-memory pool is interlocked by use of a test-and-set instruction.

- Processors can poll the shared-memory data structures for input packets receive notification of packet arrival through interrupts.

The shared-memory network is configured by various configuration constants and by parameters specified to the VxWorks boot ROMs. The following sections give the details of the backplane network operation and configuration.

3.5.1 The Backplane Shared-Memory Pool

The basis of the VxWorks shared-memory network is the *shared-memory pool.* This is a contiguous block of memory that must be accessible to all processors on the backplane. Typically this memory is either part of one of the processors' on-board, dual-ported memory, or a separate memory board.

Backplane Processor Numbers

The processors on the backplane are each assigned a unique *backplane processor number* starting with 0. The assignment of numbers is arbitrary, except for processor 0, which by convention is the shared-memory network master, described in the next section.

The processor numbers are established by the parameters supplied to the boot ROMs when the system is booted. These parameters can be burned into ROM, set in the processor's NVRAM (if available), or entered manually.

The Shared-Memory Network Master: Processor 0

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

- Initializing the shared-memory pool and the *shared-memory anchor.*
- Maintaining the *shared-memory heartbeat*.
- Functioning (usually) as the gateway to the external (Ethernet) network.
- Allocating the shared-memory pool from its dual-ported memory (in some configurations).

No processor can use the shared-memory network until the 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 pool is initialized, the processors, including the master, are all peers.

The configuration module **target/src/config/usrNetwork.c** sets the processor number of the master to 0. The master usually boots from the external (Ethernet) network directly. The master has two Internet addresses in the system: its Internet address on the Ethernet, and its address on the shared-memory network. See the reference entry for **usrConfig**.

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

The Shared-Memory Anchor

The location of the shared-memory pool depends on the system configuration. In many situations, you want to allocate the shared memory at run-time rather than fixing its location at the time the system is built.

Of course, all processors on the shared-memory network must be able to access the shared-memory pool, even if its location is not assigned at compile time. The shared-memory anchor serves as a common point of reference for all processors. The anchor is a small data structure assigned at a fixed location at compile time. This location is usually in low memory of the dual-ported memory of one of the processors. Sometimes the anchor structure is stored at some fixed address on the separate memory board.

The anchor contains a pointer to the actual shared-memory pool. The master sets this pointer during initialization. The value of the pointer to the shared-memory pool is actually an offset from the anchor itself. Thus, the anchor and pool must be in the same address space so that the offset is valid for all processors.

The backplane anchor address is established by configuration constants or by boot parameters. For the shared-memory network master, the anchor address is assigned in the master's configuration at the time the system image is built. The shared memory anchor address, *as seen by the master*, is also set during configuration. The relevant configuration macro is **SM_ANCHOR_ADRS**.

For the other processors on the shared-memory network, a default anchor 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 network. For this reason, the anchor address can also be specified in the boot parameters if the shared-memory network is the

boot device. To do this, enter the address (separated by an equal sign, "=") after
the shared-memory network boot device specifier **sm**. For example, the following
line sets the anchor address to 0x800000:

```
boot device:  sm=0x800000
```

In this case, this is the address of the anchor *as seen by the processor being booted*.

The Shared-Memory Heartbeat

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

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

Figure 3-3   **Shared-Memory Heartbeat**
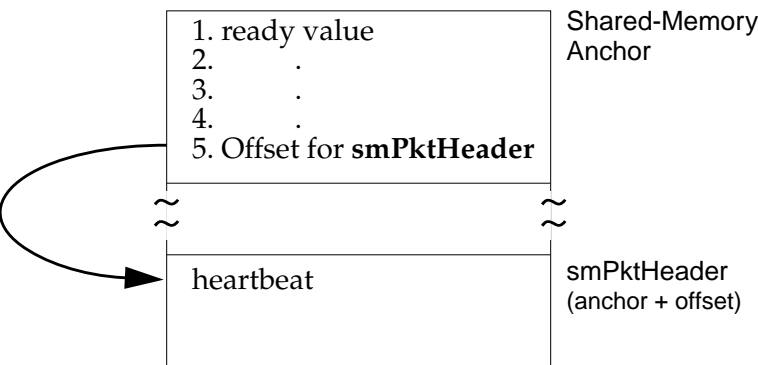
Thus, if the anchor were located at 0x800000:

```
[VxWorks Boot]: d 0x800000
800000: 8765 4321 0000 0001 0000 0000 0000 002c *.eC!............,*
800010: 0000 0170 0000 0000 0000 0000 0000 0000 *...p............*
800020: 0000 0000 0000 0000 0000 0000 0000 0000 *................*
```

The offset to the shared-memory pool is 0x170. To view the start of the shared-memory pool, display 0x800170:

```
[VxWorks Boot]: d 0x800170
800170: 0000 0050 0000 0000 0000 0bfc 0000 0350 *...P...........P*
```

In this example, the value of the shared-memory heartbeat is 0x50. Examine this location again to determine whether the network is alive. If the value has changed, the network is alive.

Shared-Memory Location

As mentioned previously, shared memory is assigned a fixed location at compile time or it is allocated dynamically at run-time. The location is determined by the value of the shared memory size set during configuration (configuration constant: **SM_MEM_ADRS**). This constant can be specified as follows:

- **NONE** (-1) means that the shared-memory pool is to be dynamically allocated from the master's on-board dual-ported memory.

- An absolute address that is *different* from the anchor address. The shared memory anchor address (configuration constant: **SM_ANCHOR_ADRS**) indicates that the shared-memory pool starts at that fixed address.

- For convenience, an absolute address that is the *same* as the anchor address means the shared-memory pool starts immediately after the anchor data structure; the size of that structure need not be known in advance.

Shared Memory Size

The size of the shared-memory pool is set during configuration. The relevant configuration macro is **SM_MEM_SIZE**.

The size required for the shared-memory pool 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 pool is divided into 2KB packets. Thus, the maximum number of packets available on the backplane network is (*poolsize* – 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. Having too small a pool slows down communications.

On-Board and Off-Board Options

The configuration of VxWorks includes a conditional compilation constant that makes it easy to select a pair of typical configurations, for instance between an *off-board* shared-memory pool and an *on-board* shared memory pool. The relevant configuration macro is **SM_OFF_BOARD**.

A typical *off-board* configuration establishes the backplane anchor and memory pool at an absolute address of 0x800000 on a separate memory board with a pool size of 512KB.

The *on-board* configuration establishes the shared-memory anchor at a low address in the master processor's dual-ported memory. The shared-memory pool size is set to 64KB allocated from the master's own memory at run time.

→ **NOTE:** These configurations are provided as examples. Change them to suit your needs.

Because the shared-memory pool is accessed by all processors on the backplane, that memory must be configured as non-cacheable. On some systems, this requires that you change the **sysPhysMemDesc[ ]** table in **sysLib.c**. Specifically, any board whose MMU is enabled (the default) must disable caching for off-board memory. Fortunately, if the VME address space used for the shared-memory pool already has a virtual-to-physical mapping in the table, the memory is already marked non-cacheable. Otherwise, you must add the appropriate mapping (with caching disabled).

For the MC680*x*0 family of processors, virtual addresses must equal physical addresses. For the 68030, if the MMU is off, caching must be turned off globally; see the reference entry for **cacheLib**. Note that the default for all BSPs is to have their VME bus access set to non-cacheable in **sysPhysMemDesc[ ]**. See *VxWorks Programmer's Guide: Virtual Memory Interface*.

Test-and-Set to Shared Memory

Unless some form of mutual exclusion is provided, multiple processors can simultaneously access certain critical data structures of the shared-memory pool and cause fatal errors. The VxWorks shared-memory network uses an indivisible test-and-set instruction to obtain exclusive use of a shared-memory data structure. This translates into a *read-modify-write* (RMW) cycle on the backplane bus.

It is important that the selected shared memory supports the RMW cycle on the bus and guarantee the indivisibility of such cycles. This is especially problematic

if the memory is dual-ported, as the memory must then also 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.c**. The shared-memory network driver calls this routine to set up mutual exclusion on shared-memory data structures.

⚠️ **CAUTION:** Configure the shared memory test-and-set type for VxWorks (configuration constant: **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**).

3.5.2 Interprocessor Interrupts

Each processor on the backplane has a single *input queue* for packets received from other processors. There are three methods processors use to determine when to examine their input queues: polling, bus interrupts, and 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. Interrupt-driven communication is much more efficient than polling.

However, most backplane buses have a limited number of interrupt lines available on the backplane (for example, VMEbus has seven). Although a processor can use one of these interrupt lines as its input interrupt, each processor must have its own interrupt line. In addition, not all processor boards are capable of generating bus interrupts. Nor can you always use bus interrupts.

As an alternative interrupt mechanism, 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. Each board can be set, with hardware jumpers or software registers, to use a different address for its mailbox interrupt.

To generate a mailbox interrupt, a processor writes to that location. There is effectively no limit to the number of processors that can use mailbox interrupts,

because each interrupt requires only a single address on the bus. 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. This information is used by the shared-memory network drivers of the other processors when sending packets.

The interrupt type and parameters for each processor are specified during configuration. The relevant configuration macro is **SM_INT_TYPE** (also **SM_INT_ARG***n*). The possible values are defined in the header file **smNetLib.h**. Table 3-5 summarizes the available interrupt types and parameters.

Table 3-5 **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

3.5.3 Sequential Addressing

Sequential addressing is a method of assigning IP addresses to processors on the network based on their processor number. Addresses are assigned in ascending order, with the master having the lowest address, as shown in Figure 3-4.

Using sequential addressing, a target on the shared-memory 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 starting IP address.

Sequential addressing provides a more uniform environment for the shared-memory network. Because a target can determine both its own Internet address

Figure 3-4 **Sequential Addressing**


```
┌────────┐      ┌────────┐      ┌────────┐
│ CPU 0  │      │ CPU 1  │      │ CPU 2  │
└────────┘      └────────┘      └────────┘
150.12.17.1     150.12.17.2     150.12.17.3
                                                   sm0
        (Shared-Memory Network)
```
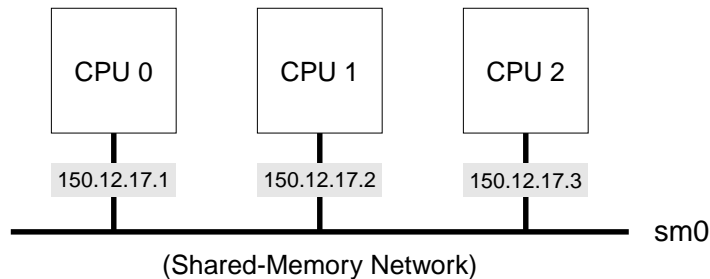
and the Internet addresses of all other targets on the shared-memory network, hardware-to-IP translation (ARP) is unnecessary over the VxWorks shared-memory network, and is therefore eliminated.

When setting up a shared-memory network with sequential addressing, choose a block of IP addresses and assign the lowest address in this block to the master.

When the shared-memory network driver is initialized by the master with *smNetInit*( ), the starting IP address is passed as a parameter and stored in the shared-memory pool.

Each target sets its interface address with *ifAddrSet*( ). This routine checks that the assigned address matches the expected address for its location on the backplane, based on the processor number from the boot parameters. If any other address is specified, the operation fails. To determine the starting address for an active shared-memory network, use *smNetShow*( ).

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

```
-> smNetShow
value = 0 = 0x0
```

The following output displays on the standard output device:

```
Anchor Local Addr: 0x800000, SOFT TAS
Sequential addressing enabled. Master address: 150.12.17.1
heartbeat = 453, header at 0x800170, free pkts = 235.
cpu    int type      arg1      arg2      arg3      queued pkts
-----  -----------  ---------  ---------  ---------  --------------
 0     mbox-1        0x2d      0x803f      0x10              0
 1     mbox-1        0x2d      0x813f      0x10              0
input packets = 366  output packets = 376
input errors = 0     output errors = 1
collisions = 0
```

With sequential addressing, when booting a slave, the backplane IP address and gateway IP boot parameters are no longer 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)         : darger
flags (f)        : 0x0

[VxWorks Boot] : @
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)         : darger
flags (f)        : 0x0

Backplane anchor at 0x800000... Attaching network interface sm0...
done.
Backplane inet address: 150.12.17.2
Subnet Mask: 0xffffff00
Gateway inet address: 150.12.17.1
Attaching network interface lo0... done.
Loading... 364512 + 27976 + 20128
Starting at 0x1000...
```

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

3.5.4 Shared-Memory Network Configuration

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

- All shared-memory network host names and addresses are present in **/etc/hosts**.

- All shared-memory 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 Ethernet as the gateway to the shared-memory network. The gateway entry is not needed if you are using proxy ARP. For more information, see *4.7 ARP and Proxy ARP for Transparent Subnets*, p. 84.

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

To illustrate the previous discussion, this section presents an example of a simple shared-memory 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 pool, and an Ethernet controller board. The additional memory board is not essential, but provides a configuration that is easier to describe.

Figure 3-5 illustrates the overall configuration. The Ethernet network is assigned network number 150, and the shared-memory network is assigned 161. The host **h1** is assigned the Internet address 150.12.0.1.

Figure 3-5   **Example Shared-Memory Network**
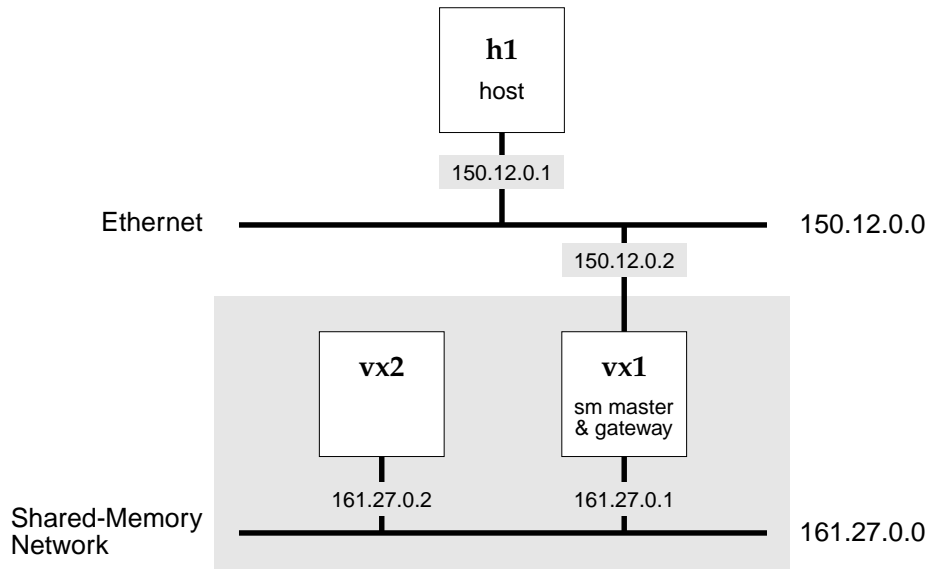
The master is **vx1**, and functions as the gateway between the Ethernet and shared-memory networks. It therefore has two Internet addresses: 150.12.0.2 on the Ethernet network and 161.27.0.1 on the shared-memory network.

The other backplane processor is **vx2**; it is assigned the shared-memory 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 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 3-6 shows the example address assignments.

Table 3-6   **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, because the host system never accesses that system with that address—but it is useful to include it in the file to ensure 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 target systems' names:

```
vx1
vx2
```

To inform the UNIX system of the existence of the Ethernet-to-shared-memory 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
```

3

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
```

The target system's configurations include the parameters shown in Table 3-7. The backplane master, **vx1**, uses the following boot parameters:

```
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
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**, has the following boot parameters:[3]

```
boot device            : sm=0x800000
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 network configured for the first time can be tricky. If you have trouble, here are a few troubleshooting procedures you can use. Take one step at a time.

---

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

Table 3-7 **Configuration Constants**

Constant	Value	Comment
shared memory anchor address (**SM_ANCHOR_ADRS**)	0x800000	Address of anchor as seen by **vx1**.
shared memory address (**SM_MEM_ADRS**)	0x800000	Address of shared-memory pool as seen by **vx1**. Zero indicates that local memory should be allocated.
shared memory size (**SM_MEM_SIZE**)	0x80000	Size of shared-memory pool, in bytes.
shared memory interrupt type (**SM_INT_TYPE**)	**SM_INT_MAILBOX_1**	Interrupt targets with 1-byte write mailbox.
shared memory interrupt type - argument 1 (**SM_INT_ARG1**)	**VME_AM_SUP_SHORT_IO**	Mailbox in short I/O space.
shared memory interrupt type - argument 2 (**SM_INT_ARG2**)	`(0xc000|(sysProcNum * 2))`	Mailbox at: 0xc000 for **vx1** 0xc002 for **vx2**
shared memory interrupt type - argument 3 (**SM_INT_ARG3**)	0	Write 0 value to mailbox.
shared memory packet size (**SM_PKTS_SIZE**)	**DEFAULT_PKTS_SIZE**	
max # of cpus for shared network (**SM_CPUS_MAX**)	**DEFAULT_CPUS_MAX**	

1. Boot a single processor in the backplane without any additional memory or processor cards. Omit the **inet on backplane** parameter to prevent the processor from trying to initialize the shared-memory network.

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 network. The following message appears during the reboot:

```
Backplane anchor at anchor-addrs...Attaching network interface
sm0...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
```

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

5. 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.

6. 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.

7. 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).

8. Use the **d** command on the second processor to look for the shared-memory anchor. The anchor begins with the ready value of 0x8765 (see Figure 3-3). You can also look for the shared-memory heartbeat; see *The Shared-Memory Heartbeat*, p. 44.

9. After you have found the anchor from the second processor, enter the boot parameter for the boot device with that address as the anchor address:

```
boot device: sm=0x800000
```

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

10. 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 network. If not, then you have probably specified the anchor address incorrectly on the second processor. If the second processor is attached, then the problem is more likely to be with the gateway or with the host system configuration.

11. 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*.

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

3.6 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/NPT interface, the VxWorks network can use your custom interface as readily as it uses the Ethernet interface. The only exception being the BOOTP and DHCP protocols, which currently assume Ethernet.

4

TCP/IP Under VxWorks

4.1 Introduction

Typically, most VxWorks systems use the TCP/IP protocol suite. Although you can port other networking protocols to VxWorks, this document generally assumes you are using TCP/IP and the numerous networking components, utilities, and services built on TCP/IP. Included in the Internet Protocol suite are the transport layer protocols TCP and UDP. These protocols, along with OSPF (a separately purchasable option), are layered on top of the IP (network) layer. In turn, the IP layer rests on top of the MUX.

Also included in this section is a detailed description of ARP and proxy ARP.

4.1.1 MUX, an Interface between the Data Link and Network Layers

VxWorks provides the MUX interface to support independence between the network protocol layer and the data link layer. To use a driver in the data link layer, the network protocol calls the appropriate MUX routine. Likewise, when a driver in the data link layer needs to access the network layer (whether IP or another protocol), it calls the appropriate MUX routine. Neither protocol nor driver deal with each other directly. Thus, neither needs specific knowledge of the other, which makes it easier to plug in a new protocol over existing drivers. For more information on the MUX interface, see the *Network Protocol Toolkit User's Guide*.

Attaching to the MUX

To attach the TCP/IP stack to the MUX for a particular interface, use the *ipAttach*( ) routine. When an interface is shut down, *ipDetach*( ) will release the TCP/IP stack components for that interface.

4.2 IP, Internet Protocol

A network protocol handles network communications at the level just above the MUX interface to the drivers that provide raw Ethernet and backplane transmission mechanisms. In VxWorks, the most commonly used network protocol is the Internet Protocol (IP)—the network protocol of the Internet protocol suite often referred to as *TCP/IP*.

With IP, each *host* (computer) in the network has a unique 4-byte Internet address (described in *4.2.1 Internet Addresses*, p.58). IP accepts packets addressed to a particular host and tries to deliver them. If multiple networks are connected by routers, IP forwards a packet from router to router until the packet reaches a network where it can be delivered directly. IP also breaks up and reassembles packets to fit the packet size of the physical network. However, IP makes no guarantees that packets are delivered to the destination correctly. Although it is possible to access IP directly, most applications use one of the higher-level protocols such as UDP or TCP.

The VxWorks network also fully supports the associated Internet Control Message Protocol (ICMP) and the Ethernet Address Resolution Protocol (ARP), as required by the relevant RFCs.

4.2.1 Internet Addresses

Each Internet host has a unique Internet address and an associated address mask. An Internet address is four bytes long and contains information that identifies a network as well as a specific host on that network. The number of bits used for network identification versus host identification can differ according to the class of the address.

Depending on the address class, the networking software uses different masks to separate the bits that carry the network address from those that carry the host

4

address. The address mask is set to a default value according to class if subnets are not used. For more information, see *4.3.3 Subnet Configuration*, p.72 and *4.7.8 Proxy ARP and Its Consequences for Subnet Configuration*, p.93.

The following list describes the Internet addresses used to accommodate different network configurations.

Class A: These addresses support a small number of networks, each with a large number of hosts.

Class B: These addresses support a moderate number of networks, each with a moderate number of hosts.

Class C: These addresses support a large number of networks, each with a small number of hosts.

Class D: These addresses support IP multicasting.

These classes are distinguished by the high-order bits of an Internet address as shown in Figure 4-1.

Figure 4-1 **Internet Address Classes**

CLASS	ADDRESS	EXAMPLE
A	**0** network: 7 bits / host: 24 bits	**90.1.2.3**
B	**1 0** network: 14 bits / host: 16 bits	**128.0.1.2**
C	**1 1 0** network: 21 bits / host: 8 bits	**192.0.0.1**
D	**1 1 1 0** multicast group ID: 28 bits	**224.0.0.1**

By convention, Internet addresses are usually represented in dotted-decimal notation, which lists the 32-bit number as a string of four 8-bit values separated by dots. Internally, the Internet address is often kept as a simple 32-bit value (of type **struct in_addr**[1]). For example, the Internet address **0x5A010203** is 90.1.2.3 in standard dotted-decimal notation. Each Internet address class has a unique address range determined by the high-order bits and the default address mask

---

1. Other declarations are possible, but **struct in_addr** is more forward compatible and less subject to a change in the size of the address.

(used for separating the bits used for the network portion of the address) as shown in Table 4-1.

VxWorks includes utilities for manipulating Internet addresses. For example, there are routines for converting between dot notation and integer notation, routines for extracting network and host portions of an address, and routines for creating a new address from a network and host number.

See the reference entry for **inetLib**.

Table 4-1   **Internet Address Ranges**

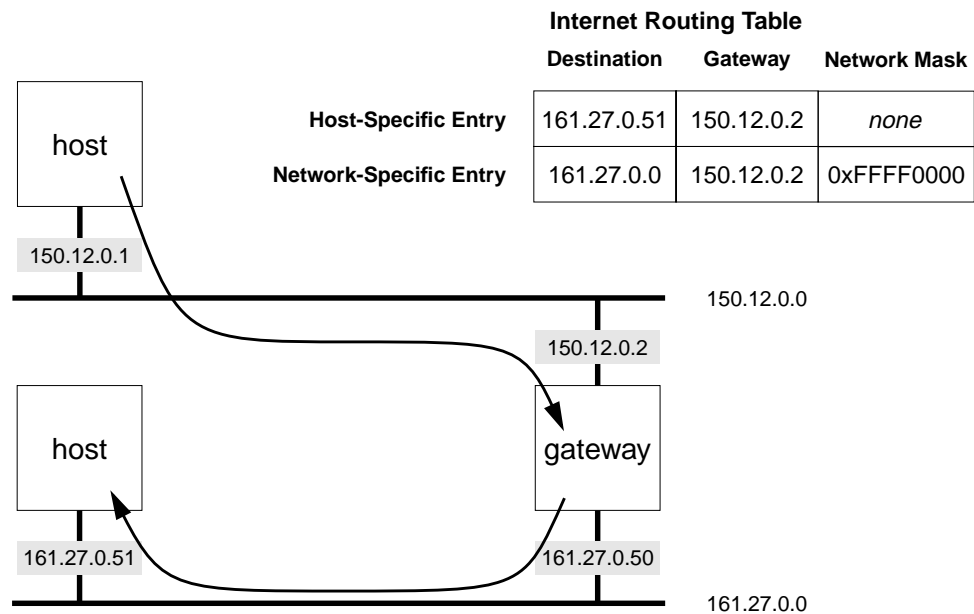| Class	High Order Bits	Default Address Mask	Address Range
A	0	`0xff000000`	0.0.0.0 - 126.255.255.255
Reserved			127.0.0.0 - 127.255.255.255
B	10	`0xffff0000`	128.0.0.0 - 191.255.255.255
C	110	`0xffffff00`	192.0.0.0 - 223.255.255.255
D	1110	None	224.0.0.0 to 239.255.255.255

4.2.2 Packet Routing

The IP protocol handles packet routing. Each route entry in the routing table is a mapping between a destination address and the network interface through which the packet is transmitted.

The route entries in a routing table are of two types: host-specific and network-specific. Host-specific route entries contain the host destination address and the address of the gateway to use for packets destined for this host. Network-specific route entries contain a network destination address and the Internet address of the gateway to use for packets destined for this network.

The VxWorks networking software establishes a route before transmitting each packet. Given a destination address, VxWorks searches the routing table to find a matching route entry and thus the network interface through which it transmits the packet. A route entry is considered a match if the logical **AND** of the network mask and the given search key (destination address) equals the destination address stored in the route entry. For example, if a route entry has 147.11.44.00 as a destination and **0xFFFFFF00** as a network mask, a search key of 147.11.44.155 matches. However, a search key of 147.11.43.155 does not match.

Figure 4-2 **Internet Routing**

Internet Routing Table

	Destination	Gateway	Network Mask
Host-Specific Entry	161.27.0.51	150.12.0.2	*none*
Network-Specific Entry	161.27.0.0	150.12.0.2	0xFFFF0000

For a host route, a mask of **0xFFFFFFFF** is assumed. If a default route is added with the destination address as "0.0.0.0", an implicit netmask of **0x00000000** is assumed. The destination address for a default route entry is always "0.0.0.0". An implicit network mask of **0x00000000** is assumed for this route entry.

When searching[2] the routing table for a destination address, the search algorithm first tries to match the complete host address. If the host address match is not found, the search tries to match the network address of the provided destination address. If a network address match is not found, the search returns the route entry of the default address (if any).

If the search of the routing table finds a matching entry which has a clone flag set, a new route entry is created (cloned) from the found entry. For example:

If: The routing table on a VxWorks target contains a route to an Ethernet network interface whose IP address is 147.11.44.155.

And: The subnet mask is **0xFFFFFF00**.

———————————————

2. For more information about the routing table structure and the algorithms used to search the routing table, see *TCP/IP Illustrated, Volume 2*, by Gary R. Wright and W. Richard Stevens.

And: The **RTF_CLONING** (0x100) flag is set.

Then: A search for 147.11.44.156 matches the 147.11.44.155 entry and creates a new route entry, a copy (clone) of the route entry for 147.11.44.155.

This new route entry for 147.11.44.156 is used by the link level address resolution protocol (ARP) to supply the corresponding Ethernet hardware address to the gateway address field. The route flags **RTF_HOST** (0x004) and **RTF_LLINFO** (0x400) are set to specify that it is host route and the gateway address is a link level address rather than a regular Internet address.

For information on configuring and adding routes to the routing table, see *Adding a Route on VxWorks*, p.68.

4.2.3 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 4-2 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 4-2   **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++;
```

4.3 VxWorks Manual Network Configuration Utilities

VxWorks includes a variety of utilities you can use to assign Internet addresses to network interfaces, hosts, and broadcasting. VxWorks also includes utilities you can use to explicitly add a gateway or configure a subnet. Not included in this section is information on the automatic network configuration protocols, DHCP and BOOTP. For more information on these utilities and their configuration needs, see *Network Configuration Protocols*, p.101.

4.3.1 Assigning Internet Addresses

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

Associating Internet Addresses with Network Interfaces

A system's physical connection to a network is called a *network interface.* Each network interface must be assigned a unique Internet (*inet*) address. A system can be connected to several networks and thus have several network interfaces.

On a UNIX system, the Internet address of a network interface is specified using the **ifconfig** command. For example, to associate the Internet address 150.12.0.1 with the interface ln0, enter:

```
% ifconfig ln0 150.12.0.1
```

This is usually done in the UNIX startup file **/etc/rc.boot**. For more information, see the UNIX reference entry for **ifconfig**.

In VxWorks, the Internet address of a network interface is specified, and a new route to that interface is constructed, by calling *ifAddrSet( )*. For example, to associate the Internet address 150.12.0.1 with the interface ln0, enter:

```
ifAddrSet ("ln0", "150.12.0.1");
```

➜ **NOTE:** The subnet mask used in determining the network portion of the address used in *ifAddrSet*( ) will be that set by *ifMaskSet*( ), or the default class mask if *ifMaskSet*( ) has not been called. It is standard to call *ifMaskSet*( ) prior to any calls to *ifAddrSet*( ).

For more information, see the Tornado reference entries for **ifLib** and *ifAddrSet*( ).

The VxWorks network startup routine, *usrNetInit*( ) in **usrNetwork.c**, automatically sets the address of the interface used to boot VxWorks to the Internet address specified in the VxWorks boot parameters.

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 system on the network. For more information on **/etc/hosts**, see your UNIX system reference entry **hosts**(5).

In VxWorks, call *hostAdd*( ) to associate system names with Internet addresses. Make one call to *hostAdd*( ) for each system the VxWorks target communicates with, as follows:

```
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 your VxWorks target. For more information, see *DNS: Domain Name System*, p.161.

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 can be accessed with the names **host**, **myHost**, and **widget**:

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

The 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:[3]

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

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

Assigning 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). This is shown in the demo program **target/src/demo/dg/dgTest.c**.

Unfortunately, there is some ambiguity about which address is to be interpreted as 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 newer systems, including VxWorks, *accept* either address on incoming packets as being a broadcast packet. But when an application *sends* a broadcast packet, it must use the correct broadcast address for its system.

_____

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

VxWorks normally uses a host part of all ones as the broadcast address. Thus a datagram sent to Internet address 150.255.255.255 (**0x5AFFFFFF**) 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.137.

4.3.2 Adding Gateways to a Network

One of the primary functions of IP is to transport packets from one host to another. Communication between two hosts on the same physical network requires little effort on the part of IP. In this case, IP can deliver the packet directly to the destination. However, if the destination of a packet is not local, IP cannot deliver the packet directly. In this case, IP hands the packet off to a gateway.

A gateway 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 the network on the other side of the gateway, the gateway can deliver the packet directly. Otherwise, the gateway passes the packet to still another gateway. This process, called routing, continues until the packet is delivered or expires.

To support routing, most systems contain a table that identifies a default gateway as well as gateways associated with specific IP addresses. To maintain this table and determine network connectivity, a VxWorks system uses the Routing Information Protocol (RIP) either version 1 or 2. The RIP implementation provided with VxWorks is based on the BSD 4.4 **routed** program.

To setup the initial routing information available on a VxWorks target, use the functions of the **routeLib** library. Using the **routeLib** routines, you can establish the default gateway you want the target to use. These routines also let you associate a destination IP address with a specific gateway. For a listing of the routing configuration functions, see the reference entry for **routeLib**.

NOTE: An OSPF-based router is available for VxWorks as a separately purchasable option. You can use this router instead of RIP to set up and maintain routing information.

Adding a Route 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 Route 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 Route on VxWorks

VxWorks provides a set of functions that you can use to edit the routing table. However, before you edit the table, it is generally a good idea to look at what is already there.

- ### Inspecting the Routing Table

  To inspect the contents of the routing table, use *routeShow( )*. If a VxWorks target boots through an Ethernet network interface, a typical *routeShow( )* call would display the following:[4]

  ```
  -> routeShow()

  ROUTE NET TABLE
  destination     gateway          flags  Refcnt  Use      Interface
  --------------------------------------------------------------------
  147.11.44.0     147.11.44.165    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 147.11.44.0 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**.

- ### Editing the Routing Table

  VxWorks includes a number of functions that you can use to edit the routing table. These functions are as follows:[5]

  routeAdd( )
  > Adds a route to the routing table.

---

4. This assumes that the VxWorks image is configured with network show routines. The relevant configuration macro is **INCLUDE_NET_SHOW**.
5. These routines manage static routing entries. These entries are not updated, modified, or deleted by the dynamic routing protocols (RIP or OSPF). They remain active until you explicitly delete them.

routeNetAdd( )

> This function is the same as *routeAdd( )* except that the destination address is assumed to be a network. This is useful for adding a route to a sub-network that is not on the same physical network as the local network.

routeDelete( )

> Deletes a route from the routing table.

mRouteAdd( )

> Adds routes differentiated by masks and service quality.

mRouteDelete( )

> Deletes routes differentiated by masks and service quality.

→ **NOTE:** The functions *routeAdd( )* and *routeDelete( )* implicitly derive the netmask from the given destination and gateway address. Please refer to the manual pages for the function calls mentioned above.

To add gateways to the VxWorks network routing tables, use *routeAdd( )* or *mRouteAdd( )*. For example, to use *routeAdd( )*:

```
/* routeAdd ("destinationAddr", "gatewayAddr") */
/* To send to network 161.27.0.0 use 150.12.0.2 */
routeAdd ("161.27.0.0", "150.12.0.2");
```

Both addresses can be specified either by dotted decimal notation or by the host names defined by the routine *hostAdd( )*. If the destination address is a subnet, you can use *routeNetAdd( )* instead.

For example, consider two VxWorks machines **vx2** and **vx3** (shown in Figure 4-3), both interfaced to network 161. Suppose that **vx3** is a gateway between networks 150 and 161 and that its Internet address on network 161 is 161.27.0.3.

The following calls can then be made on **vx2** to establish **vx3** as a gateway to network 150:

```
-> routeAdd ("150.12.0.0", "vx3");
```

or:

```
-> routeAdd ("150.12.0.0", "161.27.0.3");
```

To confirm that a route is in the routing table, use the *routeShow( )* routine.[6] Other routing routines are available in the library **routeLib**.

---

6. This function is not built into the Tornado shell. The relevant configuration macro is **INCLUDE_NET_SHOW**.

Figure 4-3 **Routing Example**

Network 161

161.27.0.2

vx2

161.27.0.3

vx3

150.12.0.1

Network 150

150.12.0.2

h1
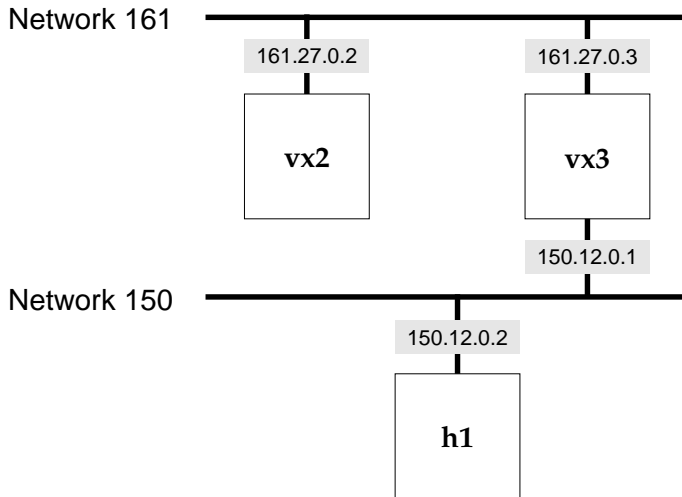
The VxWorks network startup routine, *usrNetInit*( ) in **usrNetwork.c**,
automatically adds the gateway specified in the boot parameters (if any) to the
routing tables. In this case, the address specified in the gateway field (**g =**) is
added as the gateway to the network of the boot host.

To add the default route entry to the routing table:

```
routeAdd ("0.0.0.0", "gatewayAddrs");
```

If a default route is added to the routing table, any packet for which a host route or
a network route cannot be established, is forwarded to the address provided in the
default route entry.

To delete an entry from the routing table, use *routeDelete*( ).

```
/* Delete route to node 161.27.0.51 using gateway 150.12.0.2 */
routeDelete ("161.27.0.51", "150.12.0.2");
```

▪ **Using** *mRouteAdd*( )

To use *mRouteAdd*( ) you need to specify a little more information than for a
simple *routeAdd*( ) call. However, this extra information does provide support for
additional routing services that take into account the type of service or mask
associated with a particular route. The general format of a *mRouteAdd*( ) call is as
follows:

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

Thus, to specify that the route to the 90.0.0.0 network use the 91.0.0.3 router:

```
mRouteAdd ("90.0.0.0", "91.0.0.3", 0xffffff00, 0, 0);
```

To delete a route that was added using *mRouteAdd( )*, call *mRouteDelete( )*. The general format of a call to *mRouteDelete( )* is as follows:

```
mRouteDelete ("destination", netmask, tos)
```

Thus, to delete the route just added above:

```
mRouteDelete("90.0.0.0", 0xffffff00, 0);
```

The netmask and type of service must match those of the route you want to delete. Otherwise the route is not deleted.

Using *mRouteAdd( )*, you can specify multiple routes to a single destination. These routes differ only in factors such as the netmask or the type of service. For example:

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

Now packets destined for "90.0.0.0" can get there using either of the two different gateways. The distinguishing factor is the netmask, although you could have used the *type-of-service* or *flags* values to distinguish the routes. For more information, see the *mRouteAdd( )* reference entry.

▪ **Setting the Type Of Service**

The *type-of-service* parameter to *mRouteAdd( )* takes any of the following values:

IPTOS_LOWDELAY[7]
IPTOS_THROUGHPUT
IPTOS_RELIABILITY
IPTOS_MINCOST

The routing engine uses these values to pick among multiple routes when a user application requests a certain type of service for their socket. Applications choose their type of service as input to a *setsockopt( )* call. See the *setsockopt( )* reference entry for more information.

---

7. The **IPTOS** constants are defined in **netinet/ip.h**.

■ **Setting the Routing Priority**

Within VxWorks it is now possible to have routes chosen on a priority scheme. All routes that are installed in the system have a *routing protocol type* associated with them. These types are as follows:

M2_ipRouteProto_other[8]
M2_ipRouteProto_local
M2_ipRouteProto_netmgmt
M2_ipRouteProto_icmp
M2_ipRouteProto_egp
M2_ipRouteProto_ggp
M2_ipRouteProto_hello
M2_ipRouteProto_rip
M2_ipRouteProto_is_is
M2_ipRouteProto_es_is
M2_ipRouteProto_ciscoIgrp
M2_ipRouteProto_bbnSpfIgp
M2_ipRouteProto_ospf
M2_ipRouteProto_bgp

All routes added by *mRouteAdd( )* take a protocol type of **M2_ipRouteProto_other**.

To set the routing priority, use *routeProtoPrioritySet( )*:

```
void routeProtoPrioritySet
    (
    int    proto,              /* protocol no, from m2Lib.h */
    int    prio                /* priority, >= 0 , <= 200 */
    )
```

Using this routine, you can give a certain class of routes precedence over routes of other classes. For example, you could use *routeProtoPrioritySet( )* to give OSPF-installed routes precedence over routes installed by RIP or *mRouteAdd( )*. See the *routeProtoPrioritySet( )* reference entry for more information.

4.3.3 Subnet Configuration

An Internet address consists of a network address portion and a host address portion. There are different classes of Internet addresses in which different parts of the 32-bit address are assigned to each portion. This provides a great deal of

---

8. The **M2_ipRouteProto** constants are defined in **h/m2Lib.h**.

flexibility in network addressing. Even so, in some environments network addresses are a scarce resource.

A single network address can be subdivided into multiple sub-networks using a technique called *subnet addressing*. This technique involves extending the network portion of the addresses used on a particular set of physical networks. The interpretation of the Internet address is altered to include more bits in the network portion and fewer in host portion. For example, if a network uses a type B address (131.1.0.0), the third byte can be used for the subnet and the fourth byte for the host address, as shown in Figure 4-4. Internal to the subnet, the Internet address is interpreted as 131.1.7 for the network portion and 81 for the host portion.

Figure 4-4   **Subnetting**
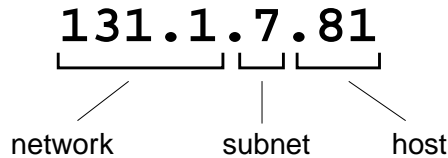
The specification of which bits are to be interpreted as the network address is called the *net mask.* A net mask is a 32-bit value with 1's in all bit positions to be interpreted as the network portion. In the example in Figure 4-4, the netmask is **0xFFFFFF00**. In VxWorks, use *ifMaskSet*( ) to specify the subnet mask for a particular network interface (see also the reference entry for *ifMaskSet*( )).

To specify a net mask, you must correctly access the host from which you are booting. This can be done by appending **:***mask* to the Internet address specifications for the Ethernet and/or backplane interfaces in the boot parameters, where *mask* is the desired net mask in hexadecimal. For example, when entering boot parameters interactively, it might look as follows:

```
inet on ethernet (e)      : 131.1.7.81:ffffff00
inet on backplane (b)     : 131.1.81.1:ffffff00
```

When specifying the boot parameters in a boot string, the same Internet address specification looks as follows:

```
e=131.1.7.81:ffffff00 b=131.1.81.1:ffffff00
```

4.4 UDP, User Datagram Protocol

The User Datagram Protocol (UDP), part of the TCP/IP suite, provides a simple *datagram*-based process-to-process 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, it is this relatively low-overhead process-to-process delivery mechanism that makes UDP so useful to many other protocols and utilities, such as BOOTP, DHCP, DNS, RIP, SNMP, and NFS.

4.5 TCP, Transmission Control Protocol

The Transmission Control Protocol (TCP), part of the TCP/IP suite, 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.

4.6 Configuring the Network Stack

Changing the default network stack configuration involves setting values for various **#define** statements. To configure the amount of memory that the network

stack uses for the network memory pool, modify the **clDescTbl** table defined in **target/src/config/usrNetwork.c**.

4.6.1 Network Protocol Scalability

By default, the build creates a VxWorks image that includes the code implementing the TCP, UDP, ICMP, and IGMP protocols. If you want to exclude one of these protocols, reconfigure VxWorks. The relevant configuration macro is found in the table below:

INCLUDE_TCP	Includes the TCP protocol.
INCLUDE_UDP	Includes the UDP protocol.
INCLUDE_ICMP	Includes the ICMP protocol.
INCLUDE_IGMP	Includes the Internet Group Management Protocol (IGMP).

4.6.2 Setting #defines for the IP, TCP, UDP, and ICMP Protocols

This section describes the configuration for the network layer protocols. Table 4-3 describes all configuration options. For some options, the default value is specified using symbolic constants. These symbolic constants are defined in **netLib.h**. To override any default values assigned to these constants, reconfigure VxWorks with the appropriate values set.

Table 4-3   **Network Configuration Options**

Configuration Constant	Default Value and Description
TCP Default Flags (**TCP_FLAGS_DFLT**)	Default Value: **TCP_DO_RFC1323** Includes RFC1323 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 this option cannot be negotiated by the peer, it should drop the option. If the host does not understand this option, it terminates the connection. For such hosts, you must turn off this option.
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.

Table 4-3 **Network Configuration Options**

Configuration Constant	Default Value and Description
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. To establish a maximum segment size, TCP typically uses the maximum transmission unit of the network interface on which the connection is established.
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 in excess of 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.
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. TCP drops the connection after sending out the maximum number of keep alive probes.
UDP Configuration Flags (**UDP_FLAGS_DFLT**)	Default Value: **UDP_DO_CKSUM_SND	UDP_DO_CKSUM_RCV** Tells UDP to calculate a UDP header and 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 connection.
UDP Receive Buffer Size (**UDP_RCV_SIZE_DFLT**)	Default Value: 41600 Sets the default receive buffer size of a UDP connection.

Table 4-3   **Network Configuration Options**

Configuration Constant	Default Value and Description
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 default value enables forwarding of packets and enables sending ICMP redirect messages (if it is necessary to redirect packets through a different router). The RFC requires that you send and receive checksums. To prevent sending a checksum, clear the **IP_DO_CHECKSUM_SND** bit. Likewise, clear **IP_DO_CHECKSUM_RCV** to prevent a checksum receive.
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 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.

Table 4-3   **Network Configuration Options**

Configuration Constant	Default Value and Description
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 has its own time to live, which, by default, is 30 seconds. This means that a fragment is deleted from the queue if it cannot be assembled in 30 seconds. If the network is extremely busy, the IP fragment queue can accumulate a lot of fragments that are waiting to be reassembled. This clutter can cause the queue to grow very large and thus take up a lot of system memory. To alleviate this problem, you can reduce the value of this configuration constant.

4.6.3 Network Memory Pool Configuration

VxWorks allocates and initializes memory for the network stack only once, at network initialization time. Out of this pre-allocated memory, the network stack uses **netBufLib** routines to set up a memory pool. From this memory pool, the network stack uses **netBufLib** routines to get the memory needed for data transfer.

The **netBufLib** routines deal with data in terms of **mBlk** structures, **clBlk** structures, and clusters. The **mBlk** and **clBlk** structures provide information necessary to manage the data stored in clusters. The clusters, which come in different sizes, contain the data described by the **mBlk** and **clBlk** structures. When VxWorks sets up the network stack memory pool, it needs to know the number of **mBlk**s, **clBlk**s, as well as the number of clusters per cluster size. The default counts are specified by symbolic constants defined in **h/netBufLib.h**. These constants are described in Table 4-4.

⚠   **CAUTION:** Change these constants only after you fully understand what they do. Setting inappropriate values can make the TCP/IP stack inoperable.

Table 4-4   **Configuration Constants for Network Memory Pools**

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

Default Memory Pool Configuration for the Network Stack

By default, the VxWorks network stack creates six pools (all within the main network memory pool) for clusters ranging in size from 64 bytes to 2048 bytes. However, valid cluster sizes can range from 64 bytes increasing by powers of two to 64K (65535). If your network stack needs clusters of a valid but non-default size, you can edit the **clDescTbl** table defined in **target/src/config/usrNetwork.c**. The following is an example and **clDescTbl[ ]** table:

```
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}
        };
```

To add a cluster pool for clusters of 4096 bytes each, edit **clDescTbl[ ]** as follows: [9]

```
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},
        {4096,              NUM_4096,       NULL,           0}
        };
```

⚠ **CAUTION:** Adjust the counts for **mBlk**s, **clBlk**s, clusters, and cluster pools only after collecting data on system behavior and deciding whether or how you would like to change that behavior. Carefully planned changes can significantly improve performance, but reckless changes can significantly reduce performance.

The values shown above are reasonable defaults, but the network requirements for your system could be radically different. For example, your use of the network stack could require more clusters of a particular size. Making such a determination is a matter of experimentation and analysis. However, as background information, you need to understand how data divides up into **mBlk**s, **clBlk**s, and clusters.

---

9. For this particular **clDescTbl[ ]** table only, you can specify **memArea** values as NULL and **memSize** values as 0. When the network initialization code actually allocates the necessary memory, it resets these values appropriately. For all other **clDescTbl[ ]** tables, you must provide these values explicitly before calling *netPoolInit( )*. For more information, see the reference entry for *netPoolInit( )*.

The **mBlk** structure is the primary vehicle through which you access data in a memory pool established by *netPoolInit( )*. Because the **mBlk** structure merely references the data, this lets network layers communicate data without actually having to copy the data. In addition, data can be chained using **mBlk**s. Thus, you can pass an arbitrarily large amount of data by passing the **mBlk** at the head of an **mBlk** chain. Consider Figure 4-5. [10]

Figure 4-5   **Presentation of Two Packets to the TCP Layer**
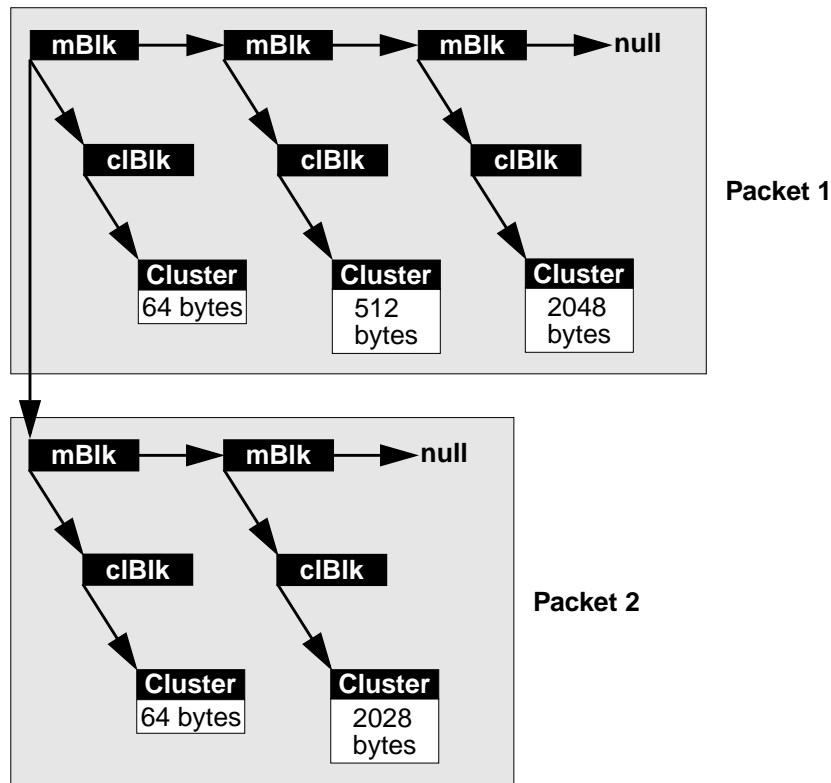
As shown in Figure 4-5, an **mBlk** references data only indirectly – through a **clBlk** structure. This indirection makes it easier for multiple **mBlk**s to share the same cluster. See Figure 4-6.

_____

10. To support chaining across multiple packets, the **mBlk** structure contains two members that support of chaining. One member points to the next **mBlk** in the current packet. The other member points to the head **mBlk** in the next packet (if any).

Figure 4-6   **Two mBlks Can Share the Same Cluster**
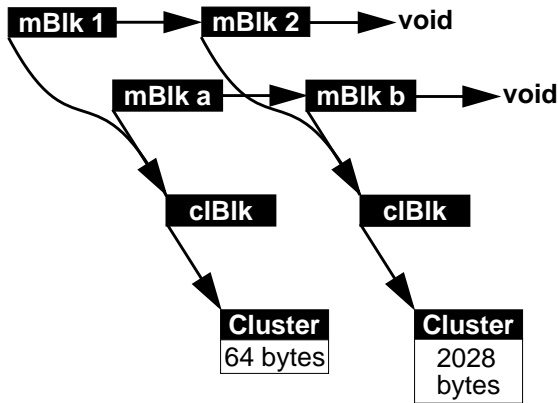
Please note that using a **clBlk** structure instead of a pointer to provide a level of indirection is not an extravagance. The **clBlk** structure tracks how many **mBlk**s share its underlying cluster. This is critical when it comes time to free an **mBlk**/**clBlk**/cluster construct. If you use *netMblkClFree( )* to free the construct, the **mBlk** is freed back to the pool and the reference count in the **clBlk** is decremented. If the reference count drops to zero, the **clBlk** and cluster are also freed back to the memory pool.

Configuring Memory Pools Out of Private Memory

The **clDescTbl[ ]** shown in the previous section did not make use of the **memArea** and **memSize** members. For the default network buffer pool, the memory allocation calls are handled internally and the values of **memArea** and **memSize** are set for you. However, if necessary, you can supply these values and thus explicitly determine the size and location of the memory pools. For more information on how to use the **memArea** and **memSize** members, see the reference entry for *netPoolInit( )*.

4.6.4 Testing Network Connections

You can use the *ping( )* utility from VxWorks 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 is useful for testing routing tables and host tables, or determining whether another machine is responding to network requests.

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 numeric 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, and 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.

4.7 ARP and Proxy ARP for Transparent Subnets

ARP (Address Resolution Protocol) provides dynamic mapping from an IP
address to the corresponding hardware or MAC address. Using ARP, VxWorks
implements a proxy ARP scheme over the shared-memory network that can make
distinct networks appear as one logical network (that is, the networks share the
same address space). This proxy ARP scheme is an alternative to the use of explicit
subnets for accessing the shared-memory network. See *3.5 Shared-Memory Network
on the Backplane*, p.40. [11]

Previously, the shared-memory network (backplane) had to be partitioned as a
separate subnet, and routes to that subnet had to be added to each host that
required access to the shared-memory network. Each shared-memory network
took up an individual subnet number; therefore, if a large number of shared-
memory networks were present on a network, precious subnet numbers were
rapidly consumed. However, with proxy ARP, the shared-memory network
appears to use the same subnet/network as the Ethernet. Therefore, subnet
numbers are not assigned.

If the shared-memory network is attached to a large network with many networks
and subnets, network configuration becomes difficult. Proxy ARP simplifies
network configuration because there is only one network to deal with and
additional configuration on the host is unnecessary.

---

11. Proxy ARP is described in Request For Comments (RFC) 925 "Multi LAN Address Resolu-
    tion," and an implementation is discussed in RFC 1027 "Using ARP to Implement Trans-
    parent 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.

4.7.1 ARP Introduction

ARP is used to resolve a host's IP address into a hardware address. This is done by broadcasting an ARP request on the physical medium (typically Ethernet). The destination host sees the request and recognizes the destination IP address as its own. It then sends a reply with its hardware address.

In the example in Figure 4-7, host **h1** wants to communicate with host **h4**. It needs **h4**'s hardware address, so it broadcasts an ARP request. Host **h4** sees the ARP request and replies with its hardware address. **h1** records **h4**'s IP-to-hardware mapping and proceeds to communicate with it.

Figure 4-7   **ARP Example**
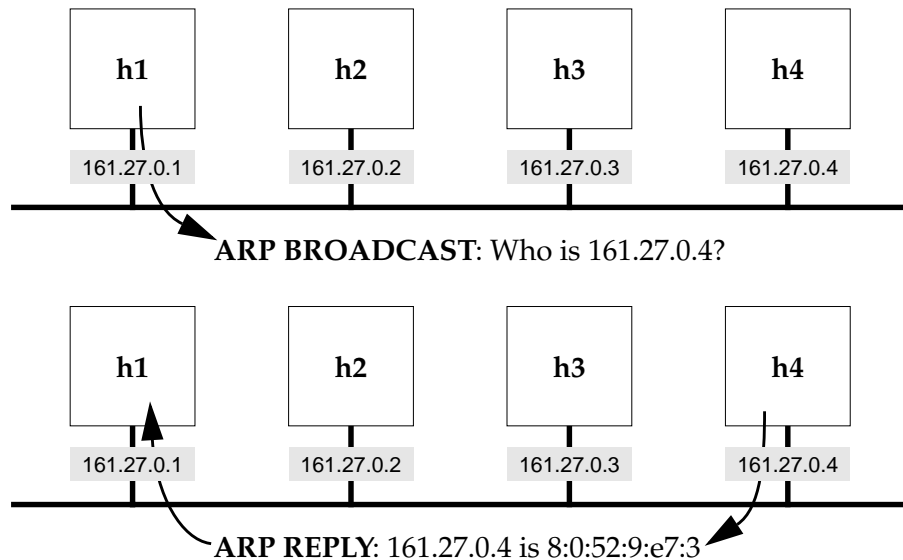
For a host to communicate with another host on a different subnet or network (as indicated by the IP addresses and the subnet mask), it must use a gateway. In Figure 4-8, **vx3** acts as a gateway between Network A and Network B. Each host must have a routing entry for the gateway in its routing table. The routing table for **vx1** to communicate with Network B includes entries like the following:

```
node            destination         gateway
vx1             150.12.2.0          150.12.1.1 (network)
```

The routing table for **h1** to communicate with Network A includes entries like the following:

```
node            destination          gateway
h1              150.12.1.0             150.12.2.1 (network)
```

A sender cannot send an ARP request for a host on another subnet or network. Instead, if it doesn't know the hardware address for the gateway listed in its routing table, it sends an ARP request for the gateway's hardware address.

Figure 4-8   **Subnets and ARP**
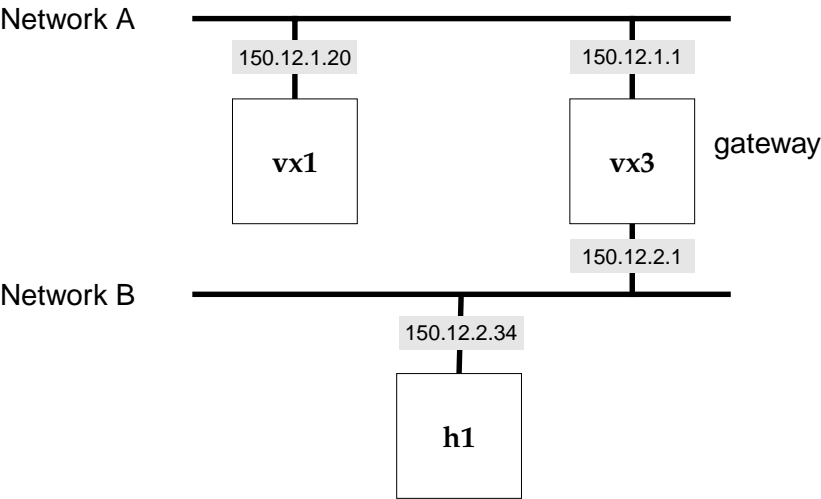
4.7.2 *Proxy ARP Overview*

With proxy ARP, nodes on different physical networks are assigned addresses with the same subnet number. Because they appear to reside on the same network, they can communicate directly and use ARP to resolve hardware addresses. The gateway node provides this network transparency by watching for and answering ARP requests that cross network boundaries. The node providing this transparency is the *proxy server*.

The example configuration shown in Figure 4-8 changes slightly when proxy ARP is used. As shown in Figure 4-9, the nodes **vx1** and **h1** now look as if they are on the same subnet. Nodes **h1** and **vx1** are fooled by **vx3** into thinking they can send

directly to each other, when they are actually sending to **vx3**. The gateway node, **vx3**, ensures that the packets get to the correct destination.

Figure 4-9   **Proxy ARP Example**
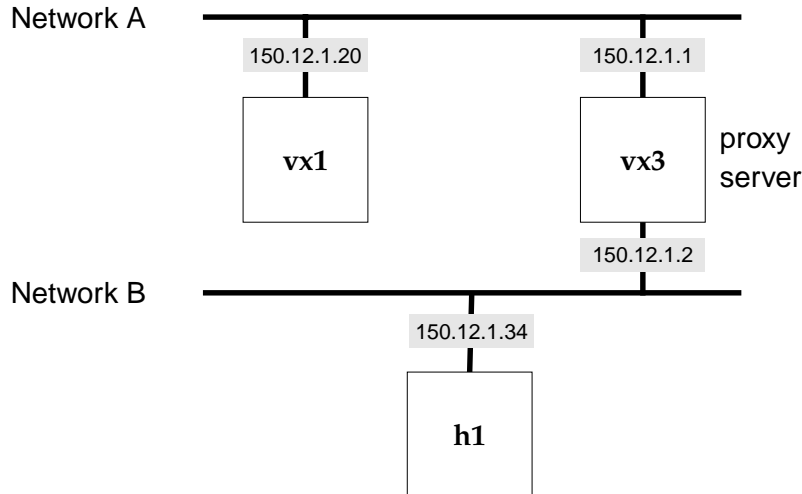
4.7.3 Routing Issues and the Proxy Server

The proxy server provides network transparency by listening to and answering ARP messages, and by manipulating its routing tables. Suppose the proxy server had two interfaces: shared-memory network and Ethernet. Nodes residing on different interfaces can have the same network address if network-specific routes with an explicit mask of **0xFFFFFFFF** were used on one interface (shared-memory network) and network routing was done on the other (Ethernet).

The proxy clients in the shared-memory network are added with a subnet mask of **0xFFFFFFFF** so that arp route entries to the proxy clients can be cloned from the route added by the **proxyLib**. In the proxy server, the backplane IP address should also have an explicit mask of **0xFFFFFFFF**. For example:

```
Inet on the backcplane (b): 150.12.1.199:FFFFFFFF
```

In the example in Figure 4-10, **vx1** and **h1** have the same network address, 150.12.1.0. The proxy server, **vx3**, has a routing table like the following example:

```
Destination                                                    Gateway
150.12.1.6 (network route with netmask 0xffffffff).......) 150.12.1.8
150.12.1.7 (network route with netmask 0xffffffff........) 150.12.1.8
150.12.1.0 (network)                                        150.12.1.60
```

Figure 4-10 **Proxy Server Example**
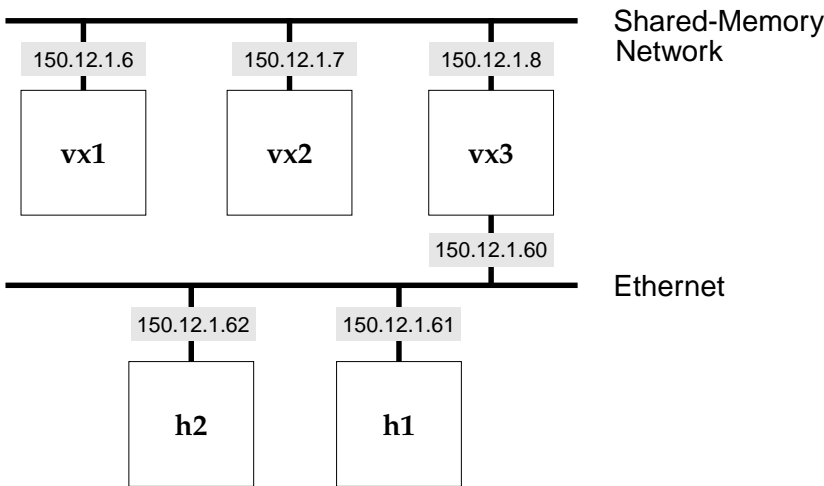
The network on which the proxy server performs network-specific routing (or for which it is acting as a proxy) with the mask **0xFFFFFFFF** is referred to as the *proxy network*. The proxy server has a network-specific route with mask **0xFFFFFFFF** to each node on the proxy network. The network interface on which the proxy server performs network routing is called the *main network*. In the example in Figure 4-10, the shared-memory network is the proxy network and the Ethernet is the main network. The routing table of **vx3** has network-specific routes with mask **0xFFFFFFFF** for both **vx1** and **vx2**. To send to nodes **h1** and **h2**, it uses the network route (150.12.1.0). There can be multiple proxy networks per main network. However, there can only be one main network per network/subnet number.

Although network-specific routes with netmask **0xFFFFFFFF** can be used on all interfaces for complete generality, a VxWorks shared-memory network usually is configured so that one side of the proxy server contains the majority of nodes (the Ethernet side). Therefore, in this case it is reasonable to use this network as the main network.

4.7.4 Proxy ARP Protocol

The following subsections describe how the proxy ARP protocol responds to ARP requests for proxy clients, non-proxy clients, and how it responds to replies from the main network.

ARP Requests for Proxy Clients

If the proxy server receives an ARP request from the main network for a node on a proxy network (*proxy client*), the proxy server generates an ARP reply with its own hardware address as the source hardware address. If the node that generated the request is also on the proxy network, the destination proxy client answers for itself. In the example in Figure 4-10, if **vx1** broadcasts an ARP request for 150.12.1.7, **vx2** replies to the request, not the proxy server **vx3**. However, if **h1** broadcasts an ARP request for 150.12.1.7, the proxy server (**vx3**) replies with its own hardware address.

ARP Requests from Proxy Clients for Non-proxy Clients

If an ARP request comes from a proxy network and the destination address is not a proxy client, the proxy server tries to resolve the request. If the hardware address of the destination is known, the server generates and sends an ARP reply to the source proxy client. If the hardware address is unknown, the server forwards the ARP request to the proxy network's corresponding main network, replacing the source hardware address in the ARP message with its own outgoing interface hardware address. For example, in Figure 4-10, **vx1** sends an ARP request for 150.12.1.62. If **vx3** knows the hardware address, it sends an ARP reply to **vx1**. Otherwise it forwards the request to the Ethernet.

ARP Replies from the Main Network

If the proxy server gets an ARP reply, the server checks to see if the destination is a proxy client. If it is, and the server previously forwarded this request, then the server forwards the ARP reply back to the proxy client (replacing the source hardware address in the ARP reply message with its own). In the previous example, if **h2** replies to the request for the Ethernet address of 150.12.1.62, the proxy server (**vx3**) records the address for itself and then forwards the reply to **vx1** (with **vx3**'s own hardware address substituted for **h2**'s).

4.7.5 Broadcast Datagrams

All nodes on a logical network are expected to receive an IP broadcast for that network (for example, 150.12.1.255). Thus, broadcasts must be passed through the proxy server so that nodes on both the proxy network and the main network receive them. Because most broadcast traffic is extraneous, it is desirable to minimize the number of forwarded shared-memory network broadcasts, thus keeping shared-memory network traffic to a minimum.

To minimize and control shared-memory network broadcast traffic, the proxy server can be configured to forward broadcasts only to a specified set of destination UDP ports. Ports are enabled using the routine *proxyPortFwdOn( )*, and are disabled with *proxyPortFwdOff( )*. Only the BOOTP/DHCP server port (67) is enabled by default.

If a broadcast datagram originates from 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 4-11, if a datagram comes from **sm1**, it gets forwarded to **ln0** and **sm0**.

Figure 4-11   **Broadcast Datagram Forwarding**

proxy network 0 (sm0)
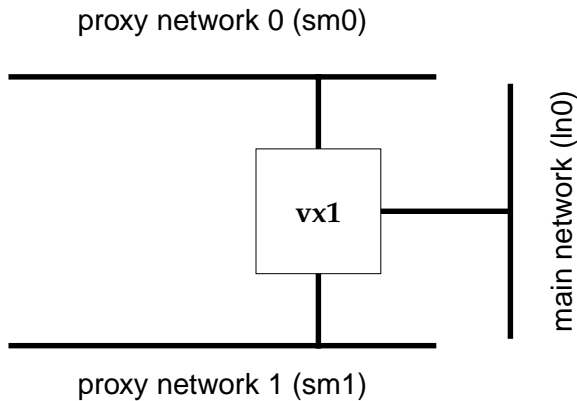
proxy network 1 (sm1)

If the datagram originates from a main network (and the port is enabled), the server forwards the broadcasts to all the main network's proxy networks. For example, in Figure 4-11, a datagram 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.

Although forwarding broadcasts between interfaces is potentially dangerous (due to broadcast storms and forwarding loops), the restrictions put on the configuration make these situations unlikely. Even so, forwarding broadcasts between proxy and main interfaces is not recommended. Therefore, forward broadcasts only on necessary ports.

4.7.6 Special Configuration Needs for Multi-Homed Proxy Clients

Using multi-homed proxy clients requires that you make changes to the routing and broadcast configuration of your VxWorks system. These changes are described in the following subsections.

Routing Configuration Considerations 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 4-12, **vx1** is the proxy server and **vx2** is a proxy client with an interface on the main network. **vx2** must be configured to have network-specific routes with mask **0xFFFFFFFF** to each of the other proxy clients (**vx4** and **vx5**), and a network-specific route to the main network. Otherwise any traffic from **vx2** to **vx4** (or **vx5**) unnecessarily travels over the main network through the proxy server (**vx1**).

The following is an example of **vx2**'s routing table. The routing table is manipulated using *routeAdd( )* and *routeDelete( )*. 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 Considerations 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). Broadcasts can be disabled on an interface using *ifFlagChange( )*. (See the reference entry.)

Figure 4-12 **Routing Example**
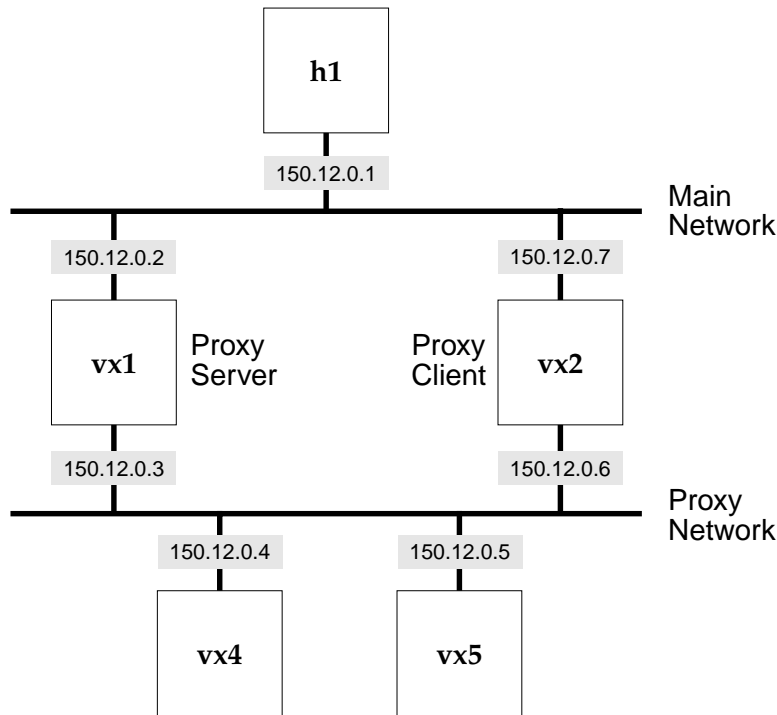
4.7.7 Single-Tier Configuration for Shared-Memory Networks under Proxy ARP

Proxy ARP works only for a single tier of shared-memory networks. That is, only interfaces directly attached to the proxy server are supported. Example configurations that work are shown in Figure 4-13 and Figure 4-15.However, the configuration shown in Figure 4-14 does not work because ARP requests are not forwarded over proxy networks, and there can be only one proxy server per shared-memory network. This single-tier restriction means that problems such as network circles, broadcast storms, and continually forwarded ARP requests are avoided.

To work, the configuration in Figure 4-14 requires a combination of proxy ARP and IP (or standard subnet) routing. The modified configuration is shown in Figure 4-16, where Proxy Network 1 has become an IP routing network with a

Figure 4-13   **Single-Tier Example Using Proxy ARP with Two Branches**
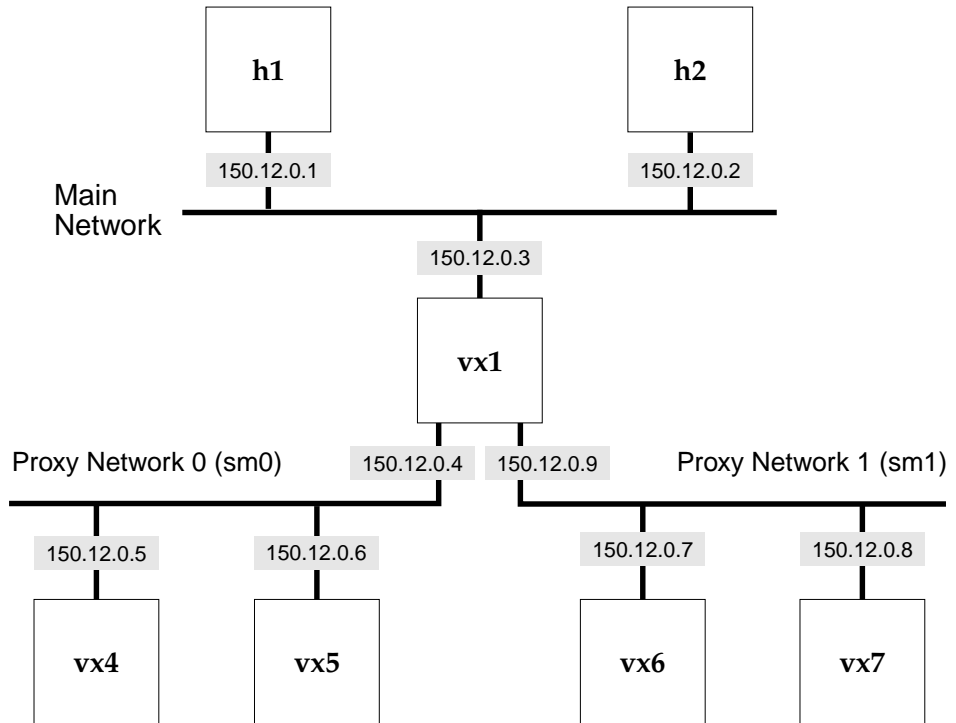
different network address. For **vx6** to send to **h2** in the modified configuration, it
requires the following entry in its routing table:

```
Destination              Gateway
150.12.0.0 (network)       161.27.0.1
```

For **h2** to send to **vx6**, it requires the following entry in its routing table:

```
Destination              Gateway
161.27.0.0 (network)       150.12.0.6
```

4.7.8 Proxy ARP and Its Consequences for Subnet Configuration

If the main network on which the proxy server is connected is subnetted, then all
the interfaces (both proxy and main) must reside on the same subnet as the main

Figure 4-14 **Multi-Tier Configuration that *CANNOT* Be Used with Proxy ARP**
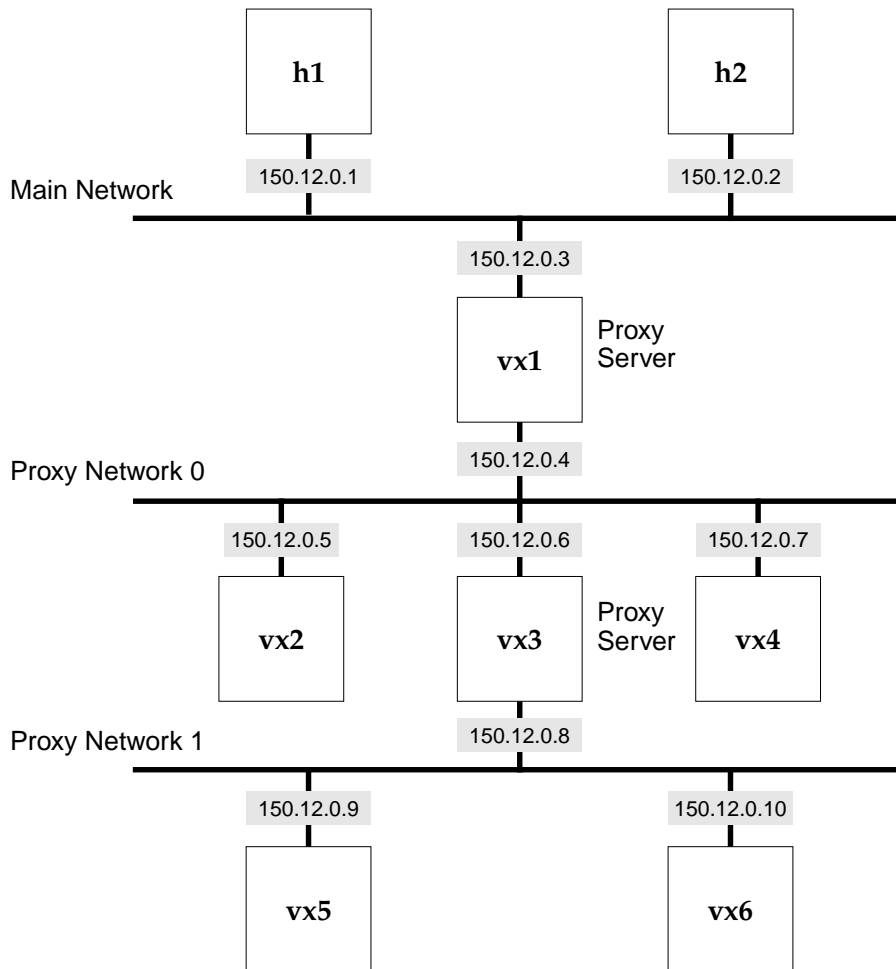
network. That is, the main network interface and the proxy network interface on the proxy server and all the proxy clients must have the same subnet mask.

To enable proxy ARP for the shared-memory network, reconfigure VxWorks and rebuild it with the proxy server. The relevant configuration macro is

Figure 4-15   **Another Single-Tier Example Using Proxy ARP**
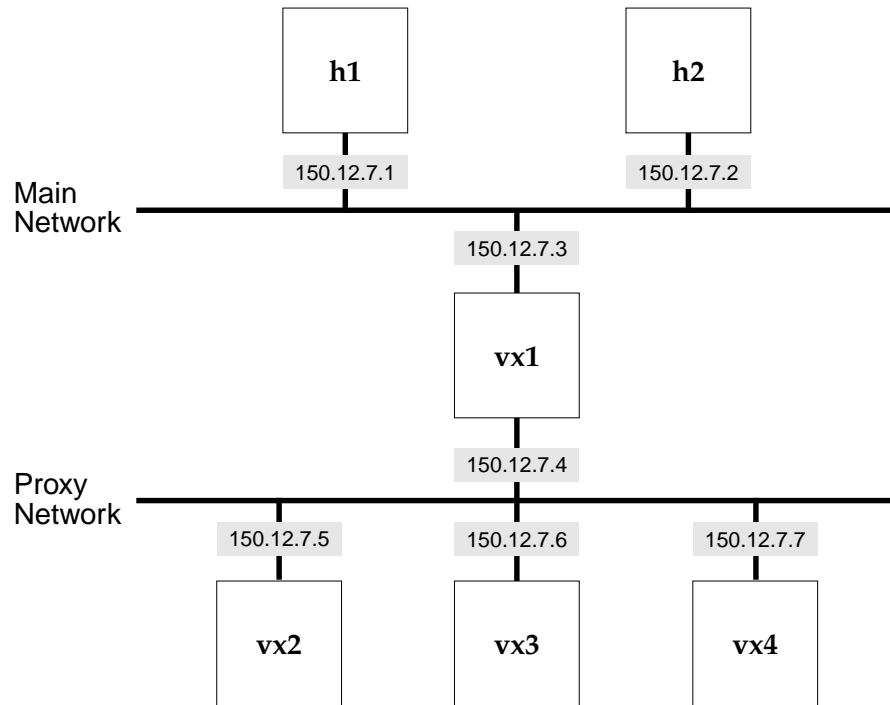
INCLUDE_PROXY_SERVER. If the target is processor zero (the shared-memory network master), the proxy server is enabled using the boot parameter **inet on ethernet (e)** as the main network, and the boot parameter **inet on backplane (b)** as the proxy network. From the example in Figure 4-15, **vx1**'s corresponding boot parameters are as follows:

```
inet on ethernet (e)    : 150.12.7.3:ffffff00
inet on backplane (b)   : 150.12.7.4
```

Proxy ARP Server Configuration

The proxy server for the shared-memory network must be the master board. As previously mentioned, the server must be configured for proxy servers. The relevant configuration macro is **INCLUDE_PROXY_SERVER**. If sequential addressing is not used, then the master backplane inet address must be specified as well as the slaves' backplane and gateway inet addresses. This configuration gives you greater control over the addresses that are assigned to the target boards.

Sequential and Default Addressing

If such control is not required, it is possible to have the proxy server assign the inet addresses to the proxy clients. When VxWorks is configured for sequential addressing, the proxy server assigns incremental inet addresses to the slave boards based on the proxy server's backplane inet address. The relevant configuration macro is **INCLUDE_SM_SEQ_ADDR**. For example, if the proxy server has a backplane inet address of 150.12.0.4, the inet address assigned to the first slave is 150.12.0.5, to the second slave 150.12.0.6, and so on. (See Figure 4-16.)

Using sequential addressing frees you from having to specify a backplane or a gateway inet address for each proxy client. All the addresses are assigned by the proxy server at boot time.

It is also possible to have the proxy server's backplane address configured by default. This allows for greater flexibility in the assignment of backplane inet addresses. You are only required to assign the inet address for the proxy server's interface to the main network. The backplane address is assigned automatically by adding 1 (one) to the network interface address.

To assign the proxy server's backplane address by default, you must use a configuration with default addressing as well as sequential addressing. The relevant configuration macro is **INCLUDE_PROXY_DEFAULT_ADDR**. This frees you from having to specify the backplane inet address of the proxy server and the proxy clients, and the gateway address of the proxy clients.

For example, with VxWorks so configured: if the proxy server is given the inet network address of 150.12.0.3, its backplane address is 150.12.0.4. The first proxy client is assigned the inet address 150.12.0.5, the second 150.12.0.6, and so on.

Note that with 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

Figure 4-16   **Multi-Tier Example Using Proxy ARP and IP Routing**
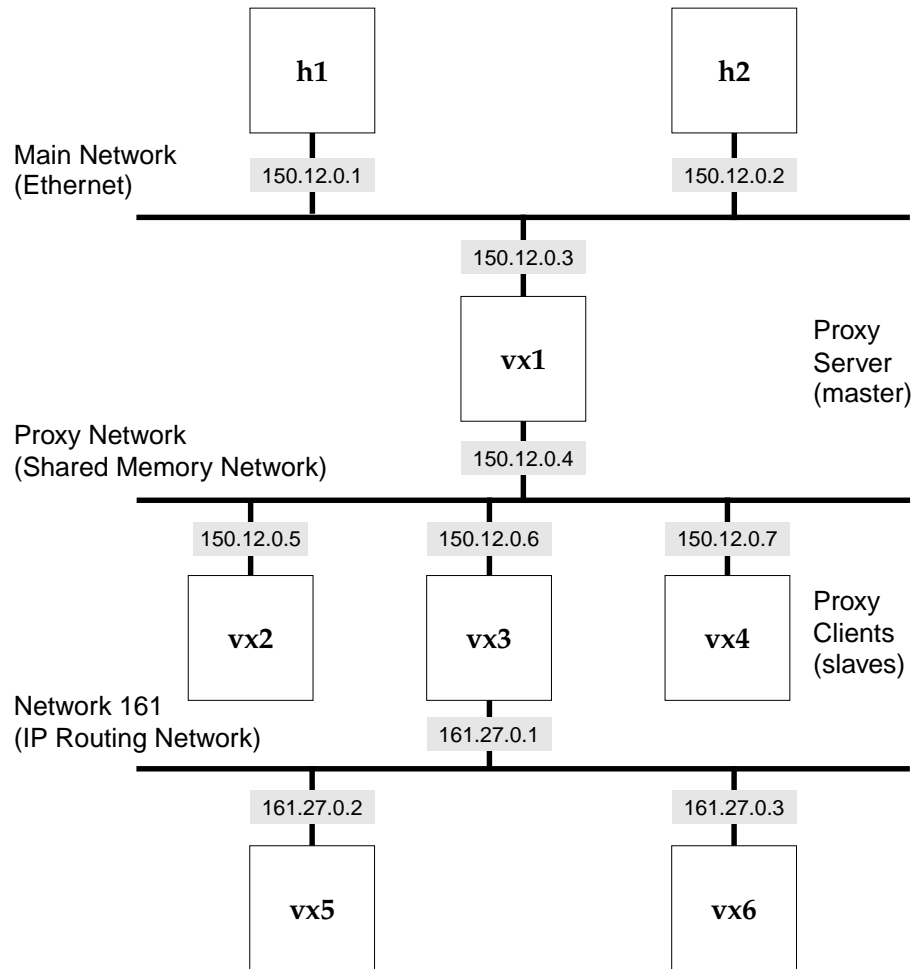
of specifying the gateway. For additional information on booting with proxy ARP, see *13.5 Booting from the Shared-Memory Network*, p.201.

VxWorks Images for 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 configuring VxWorks, see the *Tornado User's Guide: Projects*.

- **Proxy ARP and Shared Memory Definition in the VxWorks Configuration:**

(1) PING client (configuration constant: **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 only for default addressing

(5) Default address for bp (**INCLUDE_PROXY_DEFAULT_ADDR**) – required only for default addressing

- **Master Definition in config.h:**

```
#define PROXY_ARP_MASTER
#define SM_OFF_BOARD=FALSE
```

- **Slave definition in config.h:**

```
#define PROXY_ARP_SLAVE
#define SM_OFF_BOARD=TRUE
```

Setting Up Boot Parameters and Booting

See *3.5 Shared-Memory Network on the Backplane*, p. 40 for information on booting shared memory networks. After booting **vx1** (the master), 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. For general information on boot parameters, see the *Tornado User's Guide: Getting Started*.

Creating Network Connections

From vx1 (the master): Use *routeAdd*( ) to tell the master (the proxy server) about the IP routing network by running the following:

```
-> routeAdd ("161.27.0.0", "150.12.0.6")
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 4-16:

```
-> userNetIfAttach ("ln", "161.27.0.1")
Attaching network interface ln0...done.
value = 0 = 0x0
-> userNetIfConfig ("ln", "161.27.0.1", "t0-1", 0xffffff00)
value = 0 = 0x
```

NOTE: Substitute the appropriate network boot device for *"ln"*. The correct boot device is the first one given by *ifShow*( ).

Diagnosing Shared Memory Booting Problems

See *Troubleshooting*, p.53 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.

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.

5
Network Configuration Protocols

5.1 Introduction

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

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

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. Unfortunately, predicting the duration of such sessions is not usually possible at the time the leases are assigned.

Fortunately, a DHCP client has the ability to recontact its server 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, a separately purchasable optional networking product that is compatible with VxWorks. For detailed usage information on SNMP, see the *WindNet SNMP VxWorks Optional Product Supplement*.

DHCP and BOOTP are Supported for Ethernet Devices Only

Both the DHCP and BOOTP clients use broadcasts to discover an appropriate server. As a result, both protocols require network devices capable of link-layer broadcasts. In addition, the current VxWorks implementations of DHCP and BOOTP assume Ethernet. Thus, under VxWorks, DHCP and BOOTP support only Ethernet devices and the shared-memory network driver.

5.2 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 serial reassignment of network addresses. The DHCP implementation provided with VxWorks conforms to the Internet standard RFC 1541.

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 DHCP client also maintains any leases it has retrieved. Likewise, the DHCP server can process both BOOTP and DHCP messages. Both the client and server implementations support all options described in RFC 1533. The DHCP relay agent provides forwarding of DHCP and BOOTP messages across subnet boundaries. [1]

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 corresponding parameters are not specified by the server.

_____

1. In addition to the supported target-resident server, VxWorks also includes source for an unsupported UNIX-compatible DHCP server. See **target/unsupported/dhcp-1.3beta**.

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.

5.2.1 Configuring VxWorks to Include the DHCP Components

To control which DHCP feature VxWorks includes, you must reconfigure VxWorks. The relevant configuration macro is found in the list below:

DHCP server
>   Includes the DHCP server code in the VxWorks image. (Configuration flag: **INCLUDE_DHCPS**)

DCHPv4 runtime client
>   Includes the DHCP client code in the VxWorks image. You need this code if you want the target to boot using DHCP. (Configuration flag: **INCLUDE_DHCPC**)

DHCP relay agent
>   Includes the DHCP relay agent in the VxWorks image. Include the DHCP relay agent if the VxWorks target relays information from a DHCP server on a different subnet. (Configuration flag: **INCLUDE_DHCPR**)

After defining any of the above constants, rebuild VxWorks.

5.2.2 Configuring the DHCP Client

The following configuration macros are defined by default for the DHCP client:

DHCP Client Target Port
>   Port monitored by DHCP servers. Default: 67. (Configuration constant: **DHCPC_SPORT**)

DHCP Client Host Port
>   Port monitored by DHCP clients. Default: 68. (Configuration constant:
>   **DHCPC_CPORT**)

DHCP Client Maximum Leases
>   Maximum number of simultaneous leases. Default: 4. (Configuration
>   constant: **DHCPC_MAX_LEASES**)

DHCP Client Timeout Value
>   Seconds to wait for multiple offers. Default: 5. (Configuration constant:
>   **DHCPC_OFFER_TIMEOUT**)

DHCP Client Default Lease
>   Desired lease length in seconds. Default: 3600. (Configuration constant:
>   **DCHPC_DEFAULT_LEASE**)

DHCP Client Minimum Lease
>   Minimum allowable lease length (seconds). Default: 30. (Configuration
>   constant: **DHCPC_MIN_LEASE**)

You can configure VxWorks to set these parameters to any desired value. However, 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).

For more information on using a DHCP client to retrieve network configuration parameters at boot-time, see *13. Booting over the Network*.

→ **NOTE:** In addition to setting values for the defines mentioned above, most real-world uses of DHCP require that you provide an event hook routine to handle lease events. For more information, see the *dhcpcEventHookAdd( )* reference entry.

5.2.3 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 must 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 macros. For more information on configuring VxWorks, see the *Tornado User's Guide: Projects*. The relevant configuration macros are those in the following list:

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.108. (Configuration constant: **DHCPS_LEASE_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.108. (Configuration constant: **DHCPS_ADDRESS_HOOK**)

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. (Configuration constant: **DHCPS_DEFAULT_LEASE**)

DHCPS_MAX_LEASE
> Default: 3600. This constant specifies the maximum lease length in
> seconds. This value applies if no explicit value is set in the address pool.

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.
> (Configuration constant: **DHCP_MAX_HOPS**)

DHCP Server/Relay Agent Host Port
> Default: 67. This value specifies the port monitored by DHCP servers.
> (Configuration constant: **DHCP_SPORT**)

DHCP Server/Relay Agent Target Port
> Default: 68. This value specifies the port monitored by DHCP clients.
> (Configuration constant: **DHCPS_CPORT**)

To determine its initial configuration data, the supported DHCP server uses the
dhcpsLeaseTbl structure defined in **usrNetwork.c**. This structure 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:dfll=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 an 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 which 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 (discussed below).

The sample entries shown above demonstrate the possible server-issued lease types:

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 which 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.

The lease type is used by the server 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, configurations which are available only to DHCP clients are preferred.

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 expansion of the address pool without
rebuilding the VxWorks 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, configure VxWorks with the DHCP Server Lease
Storage Routine set to a string containing the routine name. The relevant
configuration macro is **DHCPS_LEASE_HOOK**.

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 using *dhcpsLeaseEntryAdd( )*. To preserve these entries, configure VxWorks with the DHCP Server Address Storage Routine set to the name of the storage routine. The relevant configuration macro is **DHCPS_ADDRESS_HOOK**. If this configuration is not done, active leases using alternate entries are not renewed when the server is restarted.

The **cmd** 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 return or replace any previously stored data.

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

---

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

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.

Configuring the Unsupported DHCP Server

The unsupported DHCP server is 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. As a convenience, WRS provides the code for the unsupported
sever in **target/unsupported/dhcp-1.3beta**. 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.

5.2.4 Configuring the Supported DHCP Relay Agent

The relay agent uses some of the same configuration constants as the DHCP server:

DHCP Server/Relay Agent Network Radius
Default: 4. Hops before discard, up to 16. (Configuration constant:
DHCP_MAX_HOPS)

DHCP Server/Relay Agent Host Port
Default: 67. Port monitored by DHCP servers. (Configuration constant:
DHCPS_SPORT)

DHCP Server/Relay Agent Target Port
> Default: 68. Port monitored by DHCP clients. (Configuration constant:
> **DHCPS_CPORT**)

If DHCP relay is configured into VxWorks (The relevant configuration macro is
INCLUDE_DHCPR), the build generates a VxWorks image that includes the DHCP
relay agent. The relay agent reads the data structure contained in **usrNetwork.c** to
obtain the IP addresses of target DHCP servers or other relay agents. That data
structure, **dhcpsRelayTbl**, has the following format:

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

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 value specified
in the VxWorks configuration is exceeded, the message is silently discarded. This
value is only increased when the message is forwarded by a DHCP agent. 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 VxWorks image.

5.2.5 DHCP Within an Application

The target-resident DHCP client can retrieve multiple sets of configuration
parameters. 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. */

dhcpcOptionSet (pLeaseCookie, _DHCP_LEASE_TIME_TAG, 1800, 0, NULL);

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
(such as a lease duration of 30 minutes) the code makes a call to *dhcpcOptionSet*( ).
Because the DHCP protocol requires that all requested parameters be specified
before a lease is established, the *dhcpcOptionSet*( ) call 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 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 requested
parameter was provided by the server. 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;
...
result = dhcpcOptionGet (pLeaseCookie, _DHCP_DFLT_WWW_SERVER_TAG, 4,
                         &webServer);
```

———————————————————————

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

```
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.

5.3 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 file name specifying a bootable image. To retrieve the boot file, the target can use a file transfer program, such as TFTP, FTP, or RSH.[4]

→ **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 VxWorks target can retrieve the boot parameters stored on a host system. This lets you set up VxWorks 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

───────────────────────────

4. 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."

(**bootptab**). The format of this database is server specific. An example **bootptab** file is described in *About the BOOTP Database*, p.114.

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 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.

5.3.1 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.

About 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 VxWorks Target

To register a VxWorks 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 file name explicitly. The home directory (**hd**) and the boot file (**bf**) are taken from **global.dummy**.

When performing the boot file name 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 file name calls for it (for example, if it is relative), the server prefixes the home directory to the file name. 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 file names 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**.

5.4 SNMP, Simple Network Management Protocol

WindNet SNMP is an optional component that provides VxWorks with SNMP (Simple Network Management Protocol) capabilities. It is a "bilingual" product, supporting both SNMP version 1 and version 2c. SNMP enables network devices, called *agents*, to be monitored, controlled, and configured remotely from a network management station.

WindNet SNMP allows a target to be managed and configured remotely by an SNMP manager. The Management Information Base (MIB) specifies the network management variables stored in an agent. You can customize the agent by extending its MIB. The newly-added MIB can include information specific to your application and environment. As shipped, WindNet SNMP supports the standard Management Information Base-II (MIB-II) definitions.

WindNet SNMP is extensible. In addition to the base functionality, you can make extensions to the SNMP agent's MIB to include information specific to your application and environment.

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

6
Dynamic Routing Protocols

6.1 Introduction

VxWorks retrieves routing information by searching for it in a routing table. To set up and manage this table manually, use the routines *routeAdd( )* or *mRouteAdd( )*. However, because the network environment is constantly in flux, the information in the routing table grows obsolete as machines join or leave the network. To update the routing table dynamically, VxWorks supports two protocols:

- RIP (Routing Information Protocol)
- OSPF (Open Shortest Path First)

RIP, the older and simpler protocol, comes bundled with VxWorks, and is intended for small to medium-sized networks. OSPF is a separately purchasable option for use with VxWorks. OSPF is superior to RIP in many ways. For example, OSPF is a link-state protocol, not a distance-vector protocol, like RIP. The messages from a *distance-vector protocol* contain a vector of distances (a hop count). Each router uses these distance-vectors to update its routing tables.

A router running a *link-state protocol*, such as OSPF, is more active about getting the information it needs. An OSPF router actively tests the status of its links to its neighbors, and then shares this information with other OSPF routers. One result of this more active approach is faster network stabilization after a change, such as the loss of a router or a link. Unfortunately, this enhanced stabilization comes at the price of increased complexity. As a result, it requires considerably more thought to configure OSPF correctly.

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. While this might seem like a large number, there are already many existing corporate networks that exceed this limit.[1]

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 backwards 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 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.

───────────────────────────

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.

6.2.1 VxWorks Includes Supplemental Debugging Routines for RIP

Provided with the RIP server are 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 a lot 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 timeout 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 propagate to other routers.
RTS_INTERNAL	Route is internal, do not propagate.
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).

Routing information is pushed down into VxWorks's routing table periodically, but there can be periods when the two are out of sync. This periodic updating (as opposed to continuous updating) avoids route thrashing, where transient routes are pushed into the system but then need to be removed immediately.

_____

2. The timeout is the length of time for which the route remains current. If a route is not updated after 3 minutes, it is flushed from the routing table.

6.2.2 Configuring RIP

To include the RIP server, reconfigure the VxWorks image. The relevant configuration macro is **INCLUDE_RIP**. The RIP server starts up when **usrNetwork.c** calls *ripLibInit( )*. This routine takes four parameters. You set the value of these parameters by editing the configuration and adjusting the following configuration items:

RIP Supplier Flag

> Set to 1 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. (Configuration macro: **RIP_SUPPLIER**)

RIP Gateway Flag

> Set to 1 tells the server that it is a router to the greater Internet. If this is not the case, set this constant to 0 (the default). (Configuration macro: **RIP_GATEWAY**)

⚠ **WARNING:** Do not set **RIP_GATEWAY** to 1 unless this really is the general gateway. Setting this to 1 makes the RIP server send a default route (0.0.0.0) out with every routing update. This tells all the other listening servers that this server is the default route for its subnet. This causes all packets to go to this router if they do not have a route that matches an existing entry in their routing table.

RIP Multicast Flag

> Set to 1 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. (Configuration macro: **RIP_MULTICAST**)

RIP Version Number

> Set to 1 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 to 1, you put the server in full version 2 mode. (Configuration macro: **RIP_VERSION**)

BSD 4.3 Compatible Sockets

> Undefine this constant if you want to use RIP with VxWorks. By default, this constant is already defined. (Configuration macro: **BSD43_COMPATIBLE**). It is also automatically defined if VxWorks is

configured to use sockets. The relevant configuration macro is
INCLUDE_BSD_SOCKET.

⚠️ **CAUTION:** The RIP server does not support separate routing domains. Only
routing domain 0, the default, is supported.

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.

6.3 OSPF, Open Shortest Path First

OSPF is an optional and separately purchasable product available from Wind
River Systems. The implementation of OSPF supported under VxWorks is OSPF
version 2, as defined in RFC-1583. OSPF is an "open" protocol because it was
defined in an open way by the Internet Engineering Task Force (IETF). It is a
shortest-path-first protocol because its routing protocol is of the Shortest Path First
family.

In addition to implementing routing management, the library associated with this
implementation of OSPF provides interfaces that you can use to configure the
OSPF MIBs (as defined in RFC-1253). You can invoke these services directly, or you
can invoke these services indirectly by using the relevant method routines of an
SNMP agent. Because OSPF is a complicated protocol to set up and maintain, it is
best to use an SNMP agent to handle configuration.

Another consequence of the complexity of OSPF is that documenting it is beyond
the scope of this manual. This manual assumes that you already have a good
understanding of OSPF and require only the specifics of this implementation. If
this is not the case, you should study RFC-1253 and the OSPF reference entries. You
should also consult one of the many published texts on OSPF and routing
protocols.[3]

---

3. For example, *Routing in the Internet* by Christian Huitema.

6.3.1 Including OSPF in VxWorks

To include OSPF, reconfigure VxWorks. The relevant configuration macro is **INCLUDE_OSPF**.

To start OSPF, call *ospfInit***( )** after VxWorks has completed booting. This routine is defined as follows:

```
STATUS ospfInit
    (
    int priority,    /* Priority of tasks. */
    int options,
    int stackSize,   /* Task stack size. */
    int routerId     /* The ID for this router. */
    )
```

After OSPF is up and running, you must configure the OSPF MIB. To do this, use the various *m2Ospf* routines. The parameters to these routines are specified in the OSPF MIB as defined in RFC1253. The RFC provides explanations for all of these routines and parameters. In addition, the VxWorks implementation of OSPF supports additional configuration functions:

*ospfAddExtRoute***( )**
> Imports an external route into the OSPF domain.

*ospfDelExtRoute***( )**
> Deletes a route imported using *ospfAddExtRoute***( )**.

*ospfAddNbmaDest***( )**
> Adds a destination on a NBMA (Non Broadcast Multi Access) link.

These routines are not MIB routines. They are convenience interfaces provided for adding and removing external routes. For more information, see reference entries for these routines.

Example 6-1   **Sample OSPF Configuration**

This section provides an example router setup as well as the code necessary to make the example work. In the example system, a router is attached to 2 subnets 160.10.10.00 and 160.10.11.00 with 0xffffff00 as the subnetmask. The interface addresses are 160.10.10.5 and 160.10.11.5. The diagram in Figure 6-1 shows this setup.

To set this up programatically, execute the following code:

```
void ospfSetup ()
    {
    /* This is a generic setup for all interfaces in the system. */
    M2_OSPF_AREA_ENTRY  area;
```

Figure 6-1   **Example Router Setup**

```
                                    160.10.11.00
        160.10.11.5

        Interface B
          Router
        Interface A

        160.10.10.5
                                    160.10.10.00
```
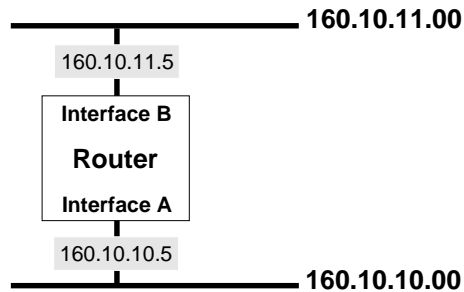
```
M2_OSPF_IF_ENTRY    intf;
area.ospfAreaId = 0x2;   /* using area id 2
area.ospfAuthType = 0;   /* no authentication
if (m2OspfAreaEntrySet (M2_OSPF_AREA_ID |
    M2_OSPF_AUTH_TYPE, area) != OK)
    {
        return (ERROR);
    };

/* First we set up Interface A */

/* set the interface address */
intf.ospfIfIpAddress = 0xa00a0a05; /* 160.10.10.5 */

/* address less interface is false */
intf.ospfAddressLessIf = 0;

/* interface area id set to 2 */
intf.ospfIfAreaId =  2;

/* interface type */
intf.ospfIfType =  1;

/* router priority */
intf.ospfIfRtrPriority = 5;

/* various  time   intervals */
intf.ospfIfTransitDelay   =   1;
intf.ospfIfRetransInterval  =  3;
intf.ospfIfHelloInterval = 10;
intf.ospfIfRtrDeadInterval = 40;
intf.ospfIfPollInterval = 30;

/* enable ospf on interface */
intf.ospfIfAdminStat = 1;

/* set the parameters for this interface */
if (m2OspfIfEntrySet(
    M2_OSPF_ADDRESS_LESS_IF |
    M2_OSPF_IF_AREA_ID |
```

```
    M2_OSPF_IF_TYPE |
    M2_OSPF_IF_RTR_PRIORITY |
    M2_OSPF_IF_RETRANS_INTERVAL |
    M2_OSPF_IF_HELLO_INTERVAL |
    M2_OSPF_IF_RTR_DEAD_INTERVAL |
    M2_OSPF_IF_POLL_INTERVAL |
    M2_OSPF_IF_ADMIN_STAT,
    & intf) != OK)
    {
    return (ERROR);
    }

/* similar sequence for Interface B */
intf.ospfIfIpAddress = 0xa00a0b05; /* 160.10.11.5 */
intf.ospfAddressLessIf  = 0;
intf.ospfIfAreaId  =     2;
intf.ospfIfType   =    1;
intf.ospfIfRtrPriority =  0;
intf.ospfIfTransitDelay  = 1;
intf.ospfIfRetransInterval  =  3;
intf.ospfIfHelloInterval = 10;
intf.ospfIfRtrDeadInterval = 40;
intf.ospfIfPollInterval = 30;
intf.ospfIfAdminStat = 1;

if (m2OspfIfEntrySet ( M2_OSPF_ADDRESS_LESS_IF |
    M2_OSPF_IF_AREA_ID |
    M2_OSPF_IF_TYPE |
    M2_OSPF_IF_RTR_PRIORITY |
    M2_OSPF_IF_RETRANS_INTERVAL |
    M2_OSPF_IF_HELLO_INTERVAL |
    M2_OSPF_IF_RTR_DEAD_INTERVAL |
    M2_OSPF_IF_POLL_INTERVAL |
    M2_OSPF_IF_ADMIN_STAT, & intf) != OK)
    {
    return (ERROR);
    }
}
```

After this code has executed, the system uses OSPF to route between the two
interfaces, A and B. The system continues to use OSPF until either the system is
shut off or further calls are made into the system using the *m2Ospf* interfaces.

7
Networking APIs

7.1 Introduction

This section 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 exactly 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.

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 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.

There are a number of complex network programming issues that are beyond the scope of this guide. For additional information, consult a socket-programming book, such as one of the following:

- *Internetworking with TCP/IP Volume III* by Douglas Comer and David Stevens

- *UNIX Network Programming* by Richard Stevens

- *The Design and Implementation of the 4.3 BSD UNIX Operating System* by Leffler, McKusick, Karels and Quarterman

- *TCP/IP Illustrated, Vol. 1*, by Richard Stevens

- *TCP/IP Illustrated, Vol. 2*, by Gary Wright and Richard Stevens

7.2.1 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 phone to another. After the connection is established, you can write and read data (talk and listen).

Table 7-1 shows the steps in establishing socket communications with TCP, and the analogy of each step with telephone communications.

Table 7-1   **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 phone numbers.
listen( )		Allow others to connect to socket.	Allow others to call.
	connect( )	Request connection to another socket.	Dial another phone's number.
accept( )		Complete connection between sockets.	Answer phone 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 7-1   **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.sa_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 useage 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.sa_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 (clientAddr.sin_port),
            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 */
            }
```

7.2.2 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 7-2 **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 */
    };
```

133

```
/* udpClient.c - UDP client example */

/*
DESCRIPTION
This file contains the client-side of the VxWorks UDP example code.
The example code demonstrates the useage 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);
    }

/* bind not required - port number is dynamic */

/* build server socket address */

sockAddrSize = sizeof (struct sockaddr_in);
bzero ((char *) &serverAddr, sockAddrSize);
serverAddr.sa_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: myRey = 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 useage 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.sa_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), clientRequest.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 *routeAdd( )*.

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 7-2).

Table 7-2 **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 7-3 lists some of the well-known multicast groups.

The following code samples define two routines, *mcastSend( )* and *mcastRcv( )*. These routines demonstrate how to use UDP sockets to handle sending and receiving multicast traffic.

Table 7-3   **Well-Known Multicast Groups**

Group	VxWorks 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.
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).

*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 is coming, 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 7-3   **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 sucessful 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.sa_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.sa_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 */
            }

        if (sockDesc != ERROR)
            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.sa_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:
      {
      if (sockDesc != ERROR)
          close (sockDesc);
      if ((status != OK) && (recvBuf != NULL))
          {
          free (recvBuf);
          recvBuf = NULL;
          }
      }
```

```
        return (recvBuf);
        }
```

7.3 Zbuf Sockets

VxWorks includes an alternative set of socket calls based on a data abstraction called a *zbuf*, which permits you to share data buffers (or portions of data buffers) between separate software modules. The *zbuf socket interface* allows applications to 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 this new 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 (set during configuration, using the macros **TCP_SND_SIZE_DFLT** or **UDP_SND_SIZE_DFLT**) must exceed that of any zbufs sent over the socket

To link (and initialize) the zbuf socket interface, reconfigure VxWorks. The relevant configuration macro is **INCLUDE_ZBUF_SOCK**.

7.3.1 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. *7.3.2 Manipulating the Zbuf Data Structure*, p.145 describes the routines to create and manage zbufs, and *7.3.3 Zbuf Socket Calls*, p.154 introduces the remaining zbuf-specific socket routines. See also the reference entries for **zbufLib** and **zbufSockLib**.

7.3.2 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, with offsets relative to the first segment. This is the most efficient addressing scheme to refer to bytes a, b, or c in the figure.

Figure 7-1 **Zbuf Addressing Relative to First Segment (NULL)**
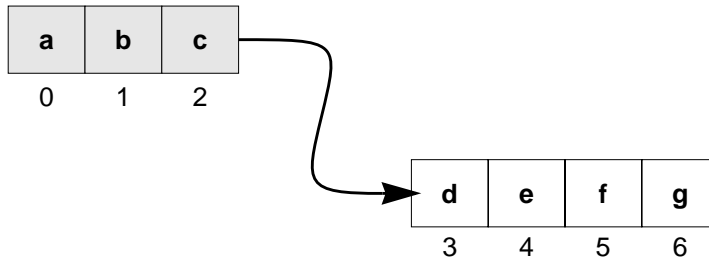
Figure 7-2 shows the same zbuf, but labelled with offsets relative to the second segment. This is the most efficient addressing scheme to refer to bytes d, e, f, or g in the figure.

Figure 7-2 **Zbuf Addressing Relative to Second Segment**
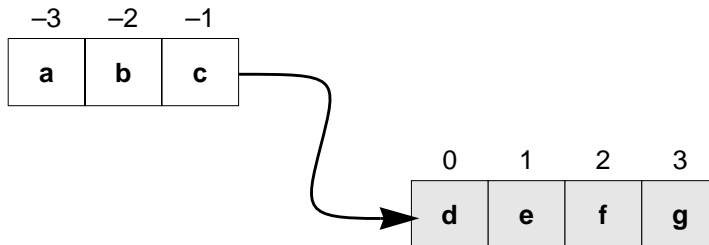
Two special shortcuts give the fastest access to either the beginning or the end of a zbuf. The constant **ZBUF_END** refers to the position after all existing bytes in the zbuf. Similarly, **ZBUF_BEGIN** refers to the position before all existing bytes. These constants are the only offsets with meanings 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 a portion of 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***( )** specifying 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.
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.

7.3.3 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 socket libraries (called automatically if the configuration has zbuf sockets enabled. The relevant configuration macro is **INCLUDE_SOCK_ZBUF**).
zbufSockSend( )	Send zbuf data to a TCP socket.
zbufSockSendto( )	Send a zbuf message to 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 7-1 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 7-1.

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 */
        else if (zbufInsert (zRecvTotal, NULL, ZBUF_END, zRecv) == NULL)
            {
            zbufDelete (zRecv);
            zbufDelete (zRecvTotal);
            return (NULL);
            }

        first = FALSE;                  /* can append now... */
        }

    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.

7

8

DNS: Domain Name System

8.1 Introduction

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. As a result, the network has a dynamic decentralized 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).

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.

8.2 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. WRS (Wind River Systems) is a commercial organization, thus it fits under the commercial domain. The NIC has given WRS the authority to manage the name space under "wrs.com". WRS uses this space to name all the hosts in its network.

8.3 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 in 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. As a result, under VxWorks, the functions *resolvGetHostByName*( ) and *resolvGetHostByAddr*( ) require two extra parameters (for a detailed description of the interface, see the reference entries for these routines).

Under VxWorks, the 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 the resolver library's response 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.

8.3.1 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( )* have 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 image.

8.3.2 Resolver Configuration

The resolver library is not included by default in the VxWorks image. Thus, to include the resolver in your VxWorks image, you must modify **config.h** as follows:

1. Reconfigure VxWorks with the DNS resolver on. The relevant configuration macro is **INCLUDE_DNS_RESOLVER**.

2. Establish the IP address of the Domain Name Server. Change the default value for the constant:

    ```
    #define RESOLVER_DOMAIN_SERVER "ip_address"
    ```

    The IP address of the server needs to be in dotted decimal notation (for example, 90.0.0.3).

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 *routeAdd( )* to add the route to the routing table. However, if you have included a routing protocol such as RIP or OSPF in your VxWorks image, these protocols add the route for you.

─────────────────────────

1. For initialization, call *resolvParamsGet( )* and *resolvParamsSet( )*. See the reference entries for these routines.
2. The boot configuration table is maintained by **hostLib**.
3. Both *hostGetByName( )* and *hostGetByAddr( )* are **hostLib** functions.

4. Define the domain to which the resolver belongs by changing the default Resolver Domain in the VxWorks configuration (defined by the configuration constant **RESOLVER_DOMAIN**).

    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

SNTP: A Time Protocol

9.1 Introduction

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.

9.2 Using the SNTP Client

To include the SNTP client, reconfigure your VxWorks image. The relevant configuration macro 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.

9.3 Using the SNTP Server

To include the SNTP server, reconfigure your VxWorks image. The relevant configuration macro is **INCLUDE_SNTPS**. VxWorks automatically calls *sntpsInit( )* during system startup. Depending on the value of the SNTP Server Mode Selection (set by the configuration constant **SNTPS_MODE**), the server executes in one of two modes, **SNTP_PASSIVE** or **SNTP_ACTIVE**.

If the SNTP Server Mode Selection is set to **SNTP_PASSIVE**, the server waits for requests from clients and sends replies containing an NTP timestamp. If the SNTP Server Mode Selection is set to **SNTP_ACTIVE**, the server periodically transmits NTP timestamp information at fixed intervals.

When executing in active mode, the SNTP server uses two other configuration settings, the SNTP Server Destination Address (configuration constant **SNTPS_DSTADDR**) and the SNTP Server Update Interval (configuration constant **SNTPS_INTERVAL**) to determine the target IP address and broadcast 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 constant **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 constant **SNTPS_TIME_HOOK**) set to the appropriate routine name. You can also call *sntpsClockSet( )*. See the reference entry fro *sntpsClockSet( )* for more information.

10

RPC: Remote Procedure Calls

10.1 Introduction

Remote Procedure Call (RPC) implements a client-server model of task interaction. In this model, client tasks request services of server tasks, and then wait for their reply. RPC formalizes this model and provides a standard protocol for passing requests and returning replies. Thus, a VxWorks or host system client task can request services from VxWorks or the host servers in any combination.

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. Also, 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 implementation of RPC was originally designed by Sun Microsystems and is in the public domain. For more information, see the public domain RPC documentation (supplied in source form in the directories **target/unsupported/rpc4.0/doc** and **target/unsupported/rpc4.0/man**), and the reference entry for **rpcLib**.

11

File Access Applications

11.1 Introduction

Using RSH, FTP, or NFS, applications running under VxWorks can access files on any host development system (over the network) exactly as if they were local to the VxWorks system. For example, **/dk0/file** might be a file local to the VxWorks system, while **/host/file** might be a file located on another machine entirely. To VxWorks applications, the files operate in exactly the same way; only the name is different. Transparent file access is available through any of three different protocols:[1]

- **Remote Shell (RSH)** is serviced by the remote shell daemon **rshd** on the host system. See the reference entry for **remLib**.

- **Internet File Transfer Protocol (FTP)** client and server functions are provided with a library of routines in **ftpLib** to transfer files between FTP servers on the network and invoke other FTP functions. See the reference entries for **ftpLib** and **ftpdLib**.

- **Network File System (NFS)** client protocol is implemented in the I/O driver **nfsDrv** to access files on any NFS server on the network. This I/O driver was tested with many different implementations of NFS file servers on various operating systems. The NFS server protocol is implemented (for dosFs file systems) in two libraries, **mountLib** and **nfsdLib**.

_____

1. If you are developing on a Windows host, check your Windows and networking software documentation for information on which of these protocols is available and how to use them.

An alternative remote file transfer protocol is the Trivial File Transfer Protocol (TFTP). The VxWorks implementation provides both client and server functions, and is used only to retrieve a VxWorks image at boot time. See the reference entries for **tftpLib** and **tftpdLib**.

11.2 RSH and FTP

The VxWorks I/O driver **netDrv** implements remote file access using either of the protocols, RSH or FTP. The **netDrv** driver uses these protocols to read the entire remote file into local memory when the file is opened, and to write the file back when it is closed (if it was modified).

The VxWorks I/O driver **nfsDrv** implements remote file access using NFS. This protocol transfers only the data actually read or written to the file and thus is considerably more efficient, both in terms of memory utilization and throughput. However, it is somewhat more cumbersome to set up initially than the other protocols. The following sections describe the implementation and configuration of these protocols.

A separate VxWorks I/O device is created for every host that services remote file accesses. When a file on one of these devices is accessed, **netDrv** uses either RSH or FTP to transfer the file to or from VxWorks:

- Using RSH, **netDrv** remotely executes the **cat** command to copy the entire requested file to and from the target. The RSH protocol is serviced by the remote shell daemon **rshd**. See the reference entry for **remLib**.

- Using FTP, **netDrv** uses the **RETR** and **STOR** commands to retrieve and store the entire requested file. The **netDrv** driver uses a library of routines, in **ftpLib**, that implements the client side for the Internet File Transfer Protocol. VxWorks tasks can transfer files to and from FTP servers on the network and invoke other FTP functions. See the reference entry for **ftpLib**.

VxWorks can also function as an FTP server. The FTP daemon running on a VxWorks server handles calls from host system and VxWorks clients, and can also boot another VxWorks system. To boot from the VxWorks server with a local disk, specify the Internet address of the VxWorks server in the **host inet** field of the boot parameters, supply a password in the **ftp password** field, and specify the shared-memory network as the boot device.

In the following example (also see Figure 11-1), a slave on the shared-memory network boots from the master CPU's local SCSI disk. (For more information on shared-memory networks, see *3.5 Shared-Memory Network on the Backplane*, p. 40.) Note that VxWorks requires a non-empty **ftp password** field. This is true even if VxWorks is configured with FTP server security turned off (the default). The relevant configuration macro is **INCLUDE_FTPD_SECURITY**. However, if FTP security checks are turned on, the **ftp password** field must contain a valid password for the specified user.

The following boot parameters are for the slave processor (**vxSlave**):

```
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
```

The FTP server daemon is initialized on the VxWorks server based on the configuration. The relevant configuration macro is **INCLUDE_FTP_SERVER**. See also the reference entry for **ftpdLib**.

Figure 11-1 **FTP Boot Example**

Shared-Memory Network
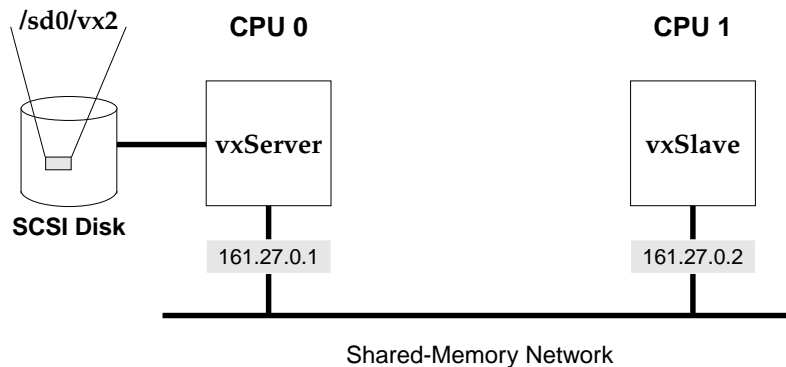
11.2.1 *Allowing Remote File Access with RSH*

An RSH request includes the name of the requesting user. The request is treated like a remote login by that user.

For Windows hosts, the availability and functionality of this facility is determined by your version of Windows and the networking software you are using. See that documentation for details.

For UNIX hosts, such remote logins are restricted by means of the host file **.rhosts** in the user's home directory, and more globally with the host file **/etc/hosts.equiv**. The **.rhosts** file contains a list of system names (as defined in **/etc/hosts**) that have access to that user's login. Therefore, make sure that the user's home directory has a **.rhosts** file listing the VxWorks systems, each on a separate line, that are allowed to access files remotely using the user's name.

The **/etc/hosts.equiv** file provides a less selective mechanism. Systems listed in this file are allowed login access to any user defined on the local system (except the super-user **root**). Thus, adding VxWorks system names to **/etc/hosts.equiv** allows those VxWorks systems to access files using any user name on the UNIX system.

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.

11.2.2 Creating VxWorks Network Devices that use RSH or FTP

The routine *netDevCreate( )* is used to create a VxWorks I/O device for a particular remote host system:

```
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( )*. It is traditional to use as the device name 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 can be accessed by appending the host path name to the device name. For example, the file name **mars:/usr/darger/myfile** refers to the file **/usr/darger/myfile** on the **mars** system. This file can be read and/or written exactly like a local file. For example, the following Tornado shell command opens that file for I/O access:

```
-> fd = open ("mars:/usr/darger/myfile", 2)
```

The VxWorks network startup routine, *usrNetInit*( ) in **usrNetwork.c**, automatically creates a network device for the host name specified in the VxWorks boot parameters. If no FTP password was specified in the boot parameters, the network device is specified with the RSH protocol. If a password was specified, FTP is used.

11.2.3 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"
```

The VxWorks network startup routine, *usrNetInit*( ) in **usrNetwork.c**, initially sets the user name and password to those specified in the boot parameters.

11.2.4 File Permissions

For a VxWorks system to have access to a particular file on a host, permissions on the host system must be set up so that the user name that VxWorks is using has permission to read that file (and write it, if necessary). This means that it must have permission to access all directories in the path, as well as the file itself.

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.

11.3 NFS

The I/O driver **nfsDrv**, which provides NFS client support, uses the client routines in the library **nfsLib** to access files on an NFS file server.

VxWorks also allows you to run an NFS server to export files to other systems. The server task **mountd** allows other systems on the network to mount VxWorks file systems (dosFs only); then the server task **nfsd** allows them to read and write to those files. The VxWorks NFS server facilities are implemented in the following libraries:

mountLib

Mount Protocol library. Provides functions to manage exporting file systems.

nfsdLib

NFS Server library. Provides functions to manage requests from remote NFS clients.

The routines in the VxWorks NFS libraries are implemented using RPC. For more information, see the reference entries for these libraries and *RPC: Remote Procedure Calls*, p.167.

11.3.1 VxWorks Target as Client

To access files on UNIX, NFS clients *mount* file systems from NFS servers. On a UNIX NFS server, the file **/etc/exports** specifies which of the server's file systems can be mounted by NFS clients. For example, if **/etc/exports** contains the following line:

```
/usr
```

then the file system **/usr** can be mounted by NFS clients such as VxWorks. If a file system is not listed in this file, it cannot be mounted by other machines. Other

optional fields in **/etc/exports** allow the exported file system to be restricted to certain machines or users.

⚠ **CAUTION:** On Windows, most networking packages that support NFS also supply a mechanism for exporting files so that they are visible on the network. See your Windows and networking software documentation for information on this facility.

Creating VxWorks Network Devices that Use NFS

Access to a remote NFS file system is established by mounting that file system locally and creating an I/O device for it using the routine *nfsMount*( ):

```
nfsMount ("host", "hostFileSys", "localName")
```

Its arguments are:

host
>The host name of the NFS server where the file system resides.

hostFileSys
>The name of the desired host file system or subdirectory.

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"
```

The host name **mars** must already be in VxWorks's list of hosts (added with the routine *hostAdd*( )). VxWorks then creates a local I/O device **/vwusr** that refers to the mounted file system. A reference on VxWorks 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 VxWorks is configured with "NFS mount all" on, VxWorks mounts all host-exported NFS file systems. The relevant configuration macro is **INCLUDE_NFS_MOUNT_ALL**. Otherwise, the network startup routine, *usrNetInit*( ) in **usrNetwork.c**, tries to mount the file system from which VxWorks was booted—as long as NFS is included in the VxWorks configuration and the VxWorks boot file begins with a slash (**/**). For example, if NFS is included and you boot **/usr/wind/target/config**/*bspname*/**vxWorks**, then VxWorks attempts to mount **/usr** from the boot host with NFS.

Setting the User ID for Remote File Access with NFS

When making an NFS request to a host system, the NFS server expects more information than the user's name. NFS is built on top of Remote Procedure Call (RPC) and uses a type of RPC authentication known as **AUTH_UNIX**. This mechanism requires the user ID and a list of group IDs to which the user belongs. These parameters can be set on VxWorks using *nfsAuthUnixSet( )*. For example, to set the user ID to 1000 and the group ID to 200 for the machine **mars**, use:

```
-> nfsAuthUnixSet "mars", 1000, 200, 0
```

The routine *nfsAuthUnixPrompt( )* 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**.

A default user ID and group ID is specified during configuration by setting the user identifier for NFS access (the configuration constant **NFS_USER_ID**, set by default to 2001) and the group identifier for NFS access (the configuration constant **NFS_GROUP_ID**, set by default to 100) respectively. The NFS authentication parameters are set to these values at system startup. If NFS file access is unsuccessful, make sure that the configuration is correct.

11.3.2 VxWorks Target as Server

To export a dosFs file system with NFS, carry out the following steps:

- Initialize a dosFs file system, with the option that makes it NFS-exportable.
- Register the file system for export, with a call to *nfsExport( )*.

To use the file system from another machine after you export it, you must also:

- Mount the remote VxWorks file system using local host facilities.

To include NFS Server support, reconfigure VxWorks. The relevant configuration macro is **INCLUDE_NFS_SERVER**. If you wish, you can run a VxWorks system with only NFS Server support (no client support).

Initializing an NFS-Exportable File System

To export a dosFs file system with NFS, you must initialize that file system with the **DOS_OPT_EXPORT** option (see *VxWorks Programmer' Guide: Volume Configuration*).

With this option, the dosFs initialization code creates some small additional in-memory data structures; these structures make the file system exportable.

The following steps initialize a DOS file system called **/export** 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 with the usual dosFs facilities, but also specify the option **DOS_OPT_EXPORT**. If your NFS client is PC-based, it may also require the **DOS_OPT_LOWERCASE** option. For example, if the device already has a valid dosFs file system on it (see *VxWorks Programmer's Guide: Using an Already Initialized Disk*), initialize it as follows:

   ```
   dosFsDevInitOptionsSet (DOS_OPT_EXPORT);
   dosFsDevInit ("/export", 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 dosFs file system from a VxWorks platform called **vxTarget**, and how to mount it from a typical UNIX system.

1. After the file system (**/export** in this example) is initialized, the following function call specifies it as a file system to be exported with NFS:

    ```
    nfsExport ("/export", 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 place-holder 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 normally need root access to mount file systems.

For example, on a typical UNIX system, the following command (executed with root privilege) mounts the **/export** file system from the VxWorks system **vxTarget**, using the name **/mnt** for it on UNIX:

```
# /etc/mount vxTarget:/export /mnt
```

Properties of NFS-Exported File Systems

Several global variables allow you to specify dosFs facilities related to NFS support. Because these facilities use global variables, you can export previously existing dosFs file systems without altering the existing configuration stored with the file system data on disk.

However, because these are global variables, you must take care to avoid race conditions if more than one task initializes dosFs file systems. If your application initializes file systems for NFS on the fly, you may need mutual exclusion surrounding these global variable settings and the corresponding file system initialization.

You can specify a single user ID, group ID, and mode (permissions) for all files within a dosFs file system. To specify these values, define the following global variables before initializing a dosFs file system with either *dosFsDevInit*( ) or *dosFsMkfs*( ):

dosFsUserId
> Numeric user ID. Default: 65534.

dosFsGroupId
> Numeric group ID. Default: 65534.

dosFsFileMode
> Numeric file access mode (that is, permissions with UNIX encoding).
> Default: 511 (octal, 777).

These settings remain in effect for the file system until you reboot.

> ⚠ **WARNING: dosFsFileMode** controls only how the file access mode is reported to
> NFS clients; it does not override local access restrictions on the DOS file system. In
> particular, if any file in an exported file system has **DOS_ATTR_RDONLY** set in its
> file-attribute byte, no modifications to that file are permitted regardless of what
> **dosFsFileMode** says.

You can also set the current date and time for the DOS file system using
*dosFsDateSet***( )** and *dosFsTimeSet***( )**. For a discussion of these routines and other
standard dosFs facilities, see *VxWorks Programmer's Guide: MS-DOS-Compatible File
System: dosFs*.

11

Limitations of the VxWorks NFS Server

The VxWorks NFS Server can export only dosFs file systems, which leads to the
following DOS limitations:

– File names in dosFs normally share the DOS limit of 8 characters with a three-
  character extension. An optional dosFs feature allows (at the expense of DOS
  compatibility) file names up to forty characters long. To enable this extension,
  create the file system with the **DOS_OPT_LONGNAMES** option (defined in
  dosFsLib.h).

– DOS file systems do not provide for permissions, user IDs, and group IDs on
  individual files. You can provide a single user ID, a single group ID, and a
  single set of permissions for all files on an entire DOS file system by defining
  the global variables **dosFsUserId**, **dosFsGroupId**, and **dosFsFileMode**,
  described in the reference entry for **dosFsLib**.

– Because the DOS file system does not provide file permissions, VxWorks 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.

11.4 TFTP

The Trivial File Transfer Protocol (TFTP) is implemented on top of the Internet User Datagram Protocol (UDP). 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.

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.

11.4.1 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 rooting all TFTP requests into the directory specified (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.

11.4.2 VxWorks TFTP Server

The TFTP server daemon is initialized by default when VxWorks is appropriately configured. The relevant configuration macro is **INCLUDE_TFTP_SERVER**. See the reference entry for **tftpdLib**.

11.4.3 VxWorks TFTP Client

Include the VxWorks TFTP client side by reconfiguring VxWorks. The relevant configuration macro 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**.

11

12

rlogin and telnet: Host Access Applications

12.1 Introduction

VxWorks supports the host access applications **telnet** and **rlogin**. VxWorks also includes **remLib**, a library for the execution of commands on a remote shell.

12.2 rlogin

You can log in to a host system from a VxWorks terminal using *rlogin( )*. For more information on the VxWorks side of this communication, see the reference entry for **rLogLib**.

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 *11.2.1 Allowing Remote File Access with RSH*, p.171.

12.3 telnet

Like **rlogin**, **telnet** is another remote login utility. However, **telnet** does not require any previous setup of the "rhosts" file. For more information on how to use telnet with a VxWorks target, see the reference entry for **telnetLib**.

12.4 remLib

The VxWorks remote command execution facilities allow applications running under VxWorks to invoke commands on a remote system and have the results returned on *standard output* and *standard error* over socket connections. This is accomplished using the *remote shell* protocol, which on UNIX systems is serviced by the remote shell daemon **rshd**. See the reference entry for **remLib**.

13
Booting over the Network

13.1 Introduction

To boot VxWorks over the network, a VxWorks target needs to know certain configuration parameters that describe itself, the network, and its relationship to the network. The goals of the boot program are as follows:

1. Gather configuration information (such as networking parameters).

2. Format the configuration information as an ASCII string, a boot line.

3. Store the boot line in a known memory location.

4. Retrieve and load a run-time VxWorks image.

5. Pass control to the run-time VxWorks image.

When the run-time VxWorks image needs the configuration information gathered by the boot program, it reads the boot line that the boot program stored at the known memory location.

The following sections describe the boot parameters and how to set them. This section also discusses the protocols, network utilities, and network devices available to the boot program. These various media, protocols, and utilities combine to produce a great variety of different boot systems (Ethernet with DHCP with TFTP, serial line with BOOTP with RSH, shared memory backplane with BOOTP with FTP, and so on.) This section describes booting using three representative systems, one example for each physical medium.

13.2 About the Boot Program

It is possible to write your own boot program from scratch, provided that the program leaves a correctly formed boot line at the known memory location and then retrieves, loads, and runs the VxWorks image. While that might sound simple, in practice, it requires a considerable amount of work. This is because the boot program typically needs access to a variety of networking utilities in order to gather all the information needed for the boot line, as well as a file transfer utility to retrieve the boot image.

For example, in many environments, the boot program must include a DHCP client to negotiate for a lease on an IP address. In addition, in order to get the VxWorks image, the boot program typically needs access to a file transfer utility, such as FTP. Thus, such a program must include a network device driver, a DHCP client, a networking stack, an FTP client, and more.

To create a boot program without coding everything from scratch, you can use the appropriate BSP and Tornado to control the configuration of **bootConfig.c**, the VxWorks boot program. The resulting boot program knows how to format a boot line and store it in the known memory location (as well as NVRAM, if available).[1] However, a boot program that uses the network to retrieve a run-time image needs its own boot parameters, such as the name of its network device.

13.2.1 How the Boot Program Gets Its Boot Parameters

The default VxWorks boot program has a built-in default boot line.[2] However, that line might be incomplete, and certain values might not be valid. Before the boot program tries to use its default boot line, the boot program looks in NVRAM, if available. The boot program also accepts a boot line from user input.

From these sources, a *non*-networked boot program must be able to find appropriate values for *all* necessary boot parameters. However, if the boot program has network access, the initial boot line needs to define only those parameters required to initialize and use its networking utilities. The boot program can then use these networking utilities to retrieve the missing boot parameters and the run-time VxWorks image from a remote source.

─────────────────────────────

1. This boot line address is configurable. The relevant configuration macro is **BOOT_LINE_ADRS**. If this behavior is inappropriate for your target device, you can copy **bootConfig.c**, modify the copy, and then use the resulting code as your boot program.
2. The default boot line is configurable. The relevant configuration macro is **DEFAULT_BOOT_LINE**.

Figure 13-1   **Sources of Boot Parameters**
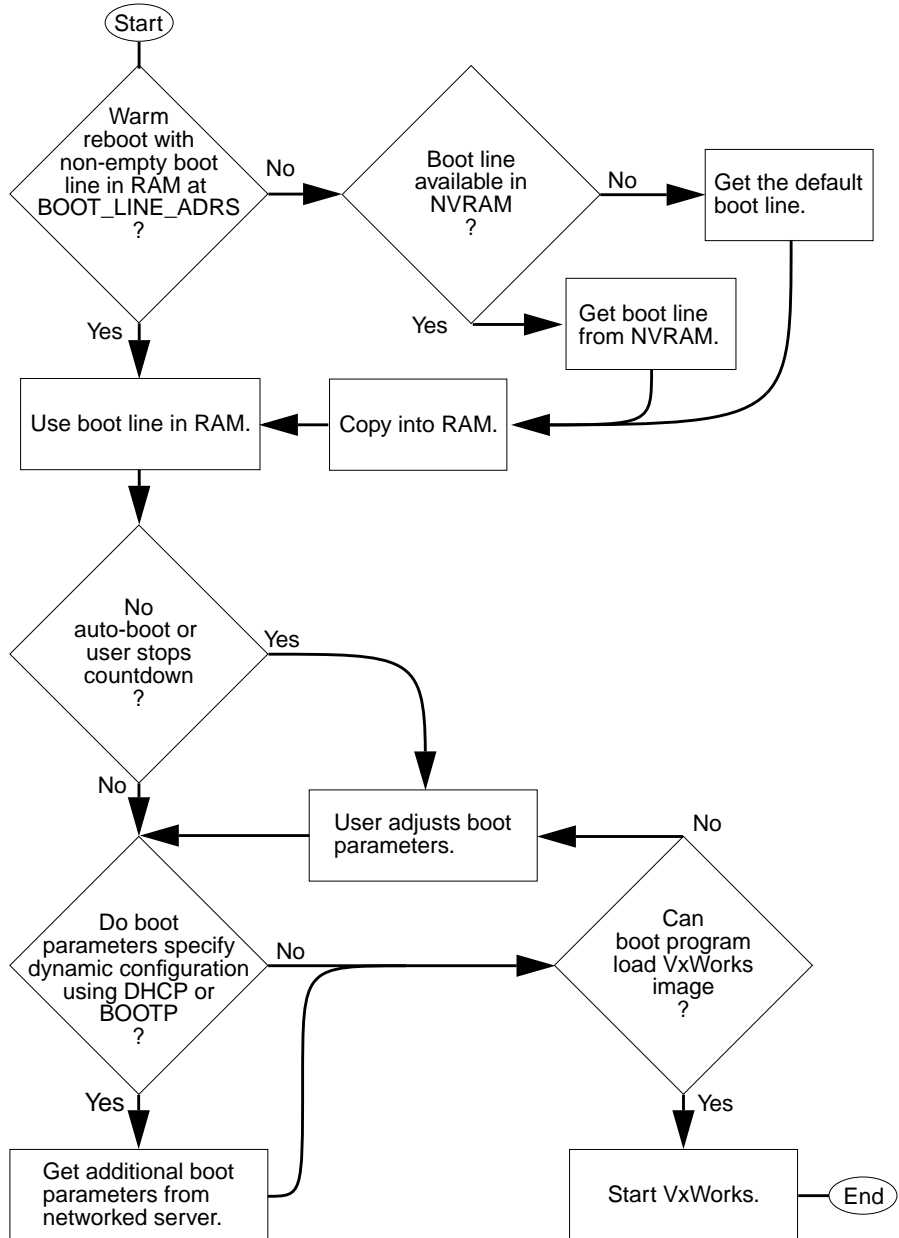
13.2.2 The General Format of a Boot Line

When a VxWorks target boots, it uses the boot line to fill in a **BOOT_PARAMS**
structure. The boot program and the VxWorks run-time image use this structure to
track boot parameters. The **BOOT_PARAMS** structure is defined as follows:

```
typedef struct                          /* BOOT_PARAMS */
    {
    char bootDev [BOOT_DEV_LEN];         /* boot device code */
    char hostName [BOOT_HOST_LEN];       /* name of host */
    char targetName [BOOT_HOST_LEN];     /* name of target */
    char ead [BOOT_ADDR_LEN];            /* ethernet internet addr */
    char bad [BOOT_ADDR_LEN];            /* backplane internet addr */
    char had [BOOT_ADDR_LEN];            /* host internet addr */
    char gad [BOOT_ADDR_LEN];            /* gateway internet addr */
    char bootFile [BOOT_FILE_LEN];       /* name of boot file */
    char startupScript [BOOT_FILE_LEN];  /* name of startup script file */
    char usr [BOOT_USR_LEN];             /* user name */
    char passwd [BOOT_PASSWORD_LEN];     /* password */
    char other [BOOT_OTHER_LEN];         /* available for applications */
    int  procNum;                        /* processor number */
    int  unitNum;                        /* network device unit number */
    int  flags;                          /* configuration flags */
    } BOOT_PARAMS;
```

This structure is shown here because its member names provide a convenient set
of labels for discussing boot parameters. For example, using the member names
shown above, this document can represent the general format of a boot line is as
follows:

bootDev(*unitNum*, *procNum*) *hostName*:*bootFile* **e=**ead **b=**bad **h=**had **g=**gad **u=**userName
pw=passWord **f=**flags **tn=**targetName **s=**startupScript **o=**other

The labeled parameters e, b, h, and so on, are not order sensitive. You can leave
them blank. For example, "pw= " specifies an empty password parameter. If the
labeled parameter is optional or supplied later by DHCP or BOOTP, you can omit
it from the boot line entirely.

As an example of a typical boot line, consider the following:

```
ln(0, 0) bear:/usr/wpwr/target/config/mz7122/vxworks e=90.0.0.2
b=91.0.0.2 h=100.0.0.4 g=90.0.0.3 u=papa pw=biggrump f=0x80 tn=goldilox
s=bear:/usr/papa/startup o=
```

To get a listing of the boot parameters, type **p** at the boot prompt (if a parameter
currently has no setting, the **p** command does not list it). The labels used in the **p**
listing differ somewhat from the names of the structure members.

13.2.3 *Boot Parameters Needed for DHCP, BOOTP, and Network Device 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 13-1. 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.[3]

Table 13-1   **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 to boot from. For example, **ln** specifies the Lance driver. Which device you specify determines the physical medium over which the boot program attempts a networked boot. Currently, VxWorks supports drivers operating over three physical media: Ethernet, a serial line, and the memory backplane. For information on the configuration needs of these drivers, see *2.2 Data Link Layer Network Components*, p.11. 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 to the MUX interface, see *Using the MUX Interface*, p.191.
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 CPU 0.

----

3. 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 13-1   **Boot Parameters Needed for DHCP, BOOTP, and Network Device Initialization**

Parameter Name from BOOT_PARAMS	Parameter Name from p Command Listing (followed by description)
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).

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 with a downloaded symbol table. The relevant configuration macro 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.

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.

13.2.4 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**.[4] See Table 13-2.

Table 13-2   **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 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 *Subnet Configuration*, p.67). 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 *Subnet Configuration*, p.67).
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.
bootFile	**file name** The full path name of the file containing the VxWorks run-time image.

---

4. If you accidentally include both a DHCP and BOOTP client in a boot program, the program uses the DHCP client.

13.2.5 Boot Parameters Needed to Set Up Remote File Access and Get the VxWorks Image

To get a file containing the VxWorks run-time image, the boot program needs to know the name of the file containing the run-time image. The boot program also needs to know the identity of the machine that hosts the file. These are provided in the boot parameters: **had** and **fileName**. Typically, the boot program gets these parameters from a DHCP or BOOTP message.[5]

To retrieve the image, the boot program uses any of three protocols: TFTP, FTP, or RSH. If the 0x80 bit in the **flags** parameter is set, the boot program uses TFTP. Otherwise, the boot program uses FTP or RSH to retrieve the image. However, in order to use RSH, the boot program needs a user ID, **usr**. If the boot program must use FTP, it needs both a user ID and a password, **passwd**. In addition, if the VxWorks run-time image is resident on a SCSI drive, you can use the **other** parameter to specify the network interface to which the VxWorks image attaches after it boots.

To support remote file access for startup code in the VxWorks image, you also need to supply a host name, **hostName**. This parameter is not critical to the boot program. You can leave it empty, if you want. However, entering a meaningful **hostName** can make messages from the boot program a little easier to read. For detailed descriptions of boot parameters for remote file access, see Table 13-3.

Table 13-3 **Boot Parameters for Remote File Access**

Parameter Name from BOOT_PARAMS	Parameter Name from p Command Listing (followed by description)
usr	**user (u)** The user name to use with FTP or RSH.
passwd	**ftp password (pw)** The password to use with FTP. If the password is empty, the boot program uses RSH instead of FTP. Supplying a password also has configuration consequences for remote file access and remote login. To access remote files, the target creates a device named according to the value of the **hostName** parameter. This device is an instance of the **netDrv** utility. This utility provides remote file access using either RSH or FTP to retrieve the remote file. If you enter a password here, **netDrv** uses FTP. For more information on **netDrv**, see *2.10.1 RSH and FTP*, p.154.

---

5. If the target reaches the host through a gateway, the target also needs the **gad** value, which is included in the same DHCP or BOOTP message that provided the **had** value.

Table 13-3   **Boot Parameters for Remote File Access**

Parameter Name from BOOT_PARAMS	Parameter Name from **p** Command Listing (followed by description)
other	**other (o)** If you are booting from the network, this parameter has no predetermined use. It is optional, and you can leave it empty. However, when booting from a disk (a subject outside the scope of this chapter), **bootDev** refers to the disk not the network device. If the boot program finds that you have entered a non-empty value for other, the boot program assumes that this value is the name of a network device that you want to attach.

13.2.6 Optional Boot Parameters

Table 13-4 lists the optional boot parameters. Omitting these parameters from the boot line does not prevent the target from booting.

Table 13-4   **Optional Boot Parameters**

Parameter Name from BOOT_PARAMS	Parameter Name from **p** Command Listing (followed by description)
hostName	**host name** Can contain the name of the host that supplies the boot file. The startup code in the VxWorks run-time image uses this name for the **netDrv** device that it creates to handle remote file access, but this name is entirely optional. Leaving it empty breaks nothing, although using a meaningful name here does make the messages from the boot program a little easier to follow. For example, this parameter is used to name the current working directory (if any), which is the **netDrv** device "*hostName*:". The target also adds this name (if any) to its host table.
targetName	**target name (tn)** Contains the name of the target. VxWorks adds this name to its host and route tables.
startupScript	**startup script (s)** Names the script, if any, to execute in a target shell upon completion of boot.

13.3 Setting the VxWorks Boot Parameters

To set boot parameter values, you can enter the parameter values manually at boot time, or you can set various **#define**s to create a default boot line. You also have the option of using either a DHCP or BOOTP client to retrieve certain parameters (such as IP address, boot file name, and so on) from a remote server.

13.3.1 Supplying Boot Parameters Using **#define** *Values*

The default boot line is specified during configuration. The relevant configuration macro is **DEFAULT_BOOT_LINE**. A valid setting looks like:

```
"ei(0,0)host:/vw/config/mv166/vxWorks h=90.0.0.3 e=90.0.0.50 u=target"
```

When control passes to the run-time VxWorks image, VxWorks parses the boot line at the known memory location and loads the values into a **BOOT_PARAMS** structure and checks for missing boot parameters. At this point, if you booted using TFTP, the only parameters that could be missing are: **hostName**, **targetName**, **usr**, **passwd**, **startupScript**, and **other**. Otherwise, the boot would have failed.[6]

However, these parameters can be useful to the run-time image even if they were not essential to the boot program. Therefore, it is possible to include values for these parameters in the run-time VxWorks image. To do this, reconfigure VxWorks. For more information on configuring VxWorks, see the *Tornado User's Guide: Projects*. The relevant configuration macros are the following:

HOST_NAME_DEFAULT
Supplies a **hostName** value, the name of the system that supplied the boot file, **bootFile**. This name is added to the target's host table to provide a convenient label for the host machine at the IP address, **had**. The **hostName** is added to the host and route tables. At startup, the run-time VxWorks image creates a **netDrv** device named "*hostName*:".

TARGET_NAME_DEFAULT
Supplies a **targetName** value, the name of the target. This name is added to the target's host table to provide a convenient label for the target's IP address, **had**.

_____

6. If you booted using RSH, you had to have provided the boot program with a value for **usr**. If you booted using FTP, you had to have provided values for **usr** and **passwd**.

HOST_USER_DEFAULT

> Supplies a **usr** value, the login name that this target's **netDrv** device should use.

HOST_PASSWORD_DEFAULT

> Supplies a **passwd** value, the password (if any), to use. If there is a non-empty **passwd**, this has configuration consequences for **netDrv** as well as for remote login from the VxWorks target to a remote system. A password here tells **netDrv** device, "*hostName*:",[7] to use FTP for remote file access. Otherwise, it uses RSH.

SCRIPT_DEFAULT

> Supplies a value for **startupScript**, the startup script for the VxWorks target-based shell (if any). You can use this script to do things such as redirect output.

OTHER_DEFAULT

> Supplies an **other** value. If the booted VxWorks image must access a local SCSI disk, you can use the **other** parameter to specify which network interface to attach.

13.3.2 Supplying Boot Parameters Manually

If you supply a boot line manually, its values take precedence over parameters specified by any other source (except those few parameters retrieved by DHCP or BOOTP). To check the values of all currently set boot parameters, type **p** at the VxWorks prompt. This command lists all currently set boot parameters and their values. For example:

```
[VxWorks Boot]: p
boot device          : ln
processor number     : 0
flags (f)            : 0xc0
unit number          : 0
```

Before you can boot the target from the network, you must enter appropriate values for **bootDev** (boot device), **procNum** (processor number), **flags**, and **unitNum** (unit number). The target can store these values in NVRAM (if available) and use them in subsequent boot sessions.

To change the boot device, processor number, flags value, or unit number, enter a **c** at the boot prompt. This runs a script that prompts you for individual boot

---

7. The presence or lack of a **passwd** affects only the "*hostName*:" device. Other instances of **netDrv** are individually configurable.

parameters. To bypass the script and enter the whole boot line all at once, enter a
$ followed by the boot line.

If you intend to use a DHCP or BOOTP message to supply additional parameters,
make sure that you specify a **flags** value that sets the 0x40 bit (the dynamic
configuration bit). Although this setting is the same for both DHCP and BOOTP,
the VxWorks startup code automatically uses DHCP if available. If DHCP is
unavailable, the startup code uses BOOTP. Typically, the boot image contains
either a DHCP or BOOTP client and not both. However, if, for some reason, you
have included both clients in the boot image, the BOOTP client is ignored and the
boot image is larger than it need be.

To make a DHCP or BOOTP client available at boot time, you must construct boot
ROMs that contain the client code.

13.3.3 Supplying Boot Parameters from a DHCP or BOOTP Server

If the boot program includes a DHCP or BOOTP client, and you have set the 0x40
bit in the **flags** boot parameter, the boot program uses the DHCP or BOOTP client
to retrieve values for **ead** (from which the boot program derives **bad**), **had**, **gad**,
and **bootFile**.

NOTE: This section provides a general discussion of the issues that arise when
using a DHCP or BOOTP server to provide boot parameters. For more on
configuring the DHCP or BOOTP servers, see *Configuring VxWorks to Include the
DHCP Components*, p.96, or *BOOTP Configuration*, p.106.

IP Lease Length

When booting with DHCP, the value of the minimum lease length is critical.
Because the network address assigned by DHCP is only valid for a finite period of
time, the DHCP client must spawn a monitor task to renew the lease as necessary.
However, this cannot occur until after the VxWorks boot file has been downloaded.

Thus, the minimum lease setting must be large enough to allow this download to
complete. Otherwise, the server which supplied the IP address may reassign it
after the lease expires, and the VxWorks image will inadvertently use an invalid IP
address. The default value is acceptable for an Ethernet link, but might need to be
increased for slower connections, such as serial links.

Booting a VxWorks Target with DHCP from a Windows NT Server

The Windows NT implementation of the DHCP server is geared towards providing configuration parameters for other Windows NT workstations. As a result, it does not provide all the information necessary to boot a VxWorks target successfully.

In particular, the Windows NT implementation of the DHCP server does not provide a boot file name. Thus, to use DHCP to boot a VxWorks target from a Windows NT Server, you must enter the boot file name manually at the boot prompt. In addition, the Windows NT implementation of DHCP server does not provide for the case in which the DHCP server that provides the configuration parameters is resident on one machine while the boot file is resident on another. Thus, you have to enter the host IP address manually.

Normally, the Tornado target server retrieves the run-time VxWorks image from the target. Unfortunately, because the NT DHCP server does not provide the target with the name of this file, the target cannot provide the file name to the target server. To get around this lack in the NT implementation of DHPC, you must supply Tornado with the name of the VxWorks run-time image. To do this, go the Create Target Server window (accessible from the Tornado Launcher window by selecting Target: Create), then use the Core field to specify a VxWorks image file. This file need not be the actual image used by the target. It can be a locally accessible copy of that image.

Getting the Target Ethernet Address

When you configure the BOOTP server, you need the target's hardware address to use as a key into the BOOTP database. You get this address from the target device. If the device is running VxWorks, you can use the *ifShow*( ) command. In the following example, the target's Ethernet address is 00:00:4b:0b:b3:a8.[8]

```
-> ifShow "ln0"
value = 0 = 0x0
```

The output is sent to the standard output device, and looks like the following:

```
ln (unit number 0):
Flags: (0x63) UP BROADCAST ARP RUNNING
Internet address: 150.12.1.240
Broadcast address: 150.12.1.255
Netmask 0xffff0000 Subnetmask 0xffffff00
Ethernet address is 00:00:4b:0b:b3:a8
Metric is 0
```

---

8. The *ifShow*( ) function is not built in to the Tornado shell but must be activated by turning on network debugging. The relevant configuration macro is **INCLUDE_TCP_SHOW**.

```
Maximum Transfer Unit size is 1500
5 packets received; 6 packets sent
0 input errors; 0 output errors
6 collisions
```

If the device has not yet booted, you can use the **n** command to retrieve the information from the boot ROMs. For example:

```
[VxWorks Boot]: n ln
Attaching network interface enp0... done
Address for device "ln" == 02:cf:1f:e0:20:24
```

13.4 Booting from the Ethernet

The following procedure describes how to boot from a UNIX host over the Ethernet for a VxWorks target that uses DHCP or BOOTP to retrieve boot parameters and TFTP to retrieve the file containing the run-time image.

1. Copy the VxWorks image to the boot directory on the boot host:[9]

   ```
   % cp vxWorks.st /usr/wind/target/vxBoot/vxWorks.vx245
   ```

2. Make sure that the permissions on the boot file make it accessible to all:

   ```
   % chmod 644 vxWorks.vx245
   % ls -l
   total 609
   -rw-r--r-- 1 target 519880 Jul 6 19:36 vxWorks.vx245
   ```

3. On the target, set the flag value to 0xc0. This enables automatic configuration (using DHCP or BOOTP, 0x40) and file retrieval using TFTP (0x80).

   To check the current value, enter **p** at the boot prompt:

   ```
   [VxWorks Boot]: p
   ```

   The target responds:

   ```
   boot device        : ln
   processor number   : 0
   flags (f)          : 0xc0
   unit number        : 0
   ```

───────────────────────────────

9. The file suffix, **.vx245**, is of no special significance. It just distinguishes this image from other VxWorks images that might reside in the directory.

To change a value, enter **c** at the boot prompt.

4. Boot the target. To do this, enter an **@** at the prompt:

```
[VxWorks Boot]: @
```

If the target uses DHCP, it responds:

```
boot device        : ln
unit number        : 0
processor number   : 0
user (u)           : stephenm
flags (f)          : 0x40

Attaching network interface ln0... done.
Getting boot parameters via network interface ln0.

DHCP Server:147.11.46.24
    Boot file: /usr/wind/target/vxBoot/vxWorks.vx245
    Boot host: 147.11.46.24
    Boot device Addr (ln0): 147.11.46.174
    Subnet mask: 0xffffff00
    Subnet gateway: 147.11.46.24
Attaching network interface lo0... done.
Loading... 374624 + 57008 + 20036
Starting at 0x1000...

Host Name: bootHost
Target Name: vxTarget
User: target
Attaching network interface ln0... done.
Attaching network interface lo0... done.
```

If the target uses BOOTP, it responds:

```
boot device        : ln
processor number   : 0
flags (f)          : 0xc0
Attaching network interface ln0... done.
Getting boot parameters via network interface ln0.
Bootp Server:150.12.1.159
Boot file: /usr/wind/target/vxBoot/vxWorks.vx245
Boot host: 150.12.1.159
Boot device Addr (ln0): 150.12.1.245
Subnet mask: 0xffffff00
Attaching network interface lo0... done.
Loading... 374624 + 57008 + 20036
Starting at 0x1000...

Host Name: bootHost
Target Name: vxTarget
User: target
Attaching network interface ln0... done.
Attaching network interface lo0... done.
```

13

Getting a Symbol Table File

VxWorks can be configured to omit the symbol table, by turning off the "downloadable symbol table" during configuration. The relevant configuration macro is **INCLUDE_NET_SYM_TBL**.[10] Instead, the run-time VxWorks image (not the boot program) downloads the symbol table file from the same remote directory that contained the VxWorks image. To retrieve this file, VxWorks uses the **netDrv** I/O driver.

When you copy the VxWorks image to the host boot directory:

```
% cp vxWorks /usr/wind/target/vxBoot/vxWorks.vx245
```

You must also copy the symbol file to the same directory:

```
% cp vxWorks.sym /usr/wind/target/vxBoot/vxWorks.vx245.sym
```

Note that the name of the symbol file is the bootfile name with a **.sym** suffix.

NOTE: Because **netDrv** uses either RSH or FTP to access the remote files, the boot parameters must specify a value for the **usr** boot parameter. If you want to use FTP, you must also specify a **passwd**. Otherwise, the target uses RSH. For more information on **netDrv**, see *2.10.1 RSH and FTP*, p.154.

13.4.1 Troubleshooting

If possible, put the BOOTP server in debugging mode.

No BOOTP Reply or Problems with TFTP

If there is no BOOTP reply:

- Make sure a BOOTP server is running on the host.
- Verify that the target address is correct.
- Be sure the boot file for the target exists and is accessible.

If the TFTP server is started with the **-s** option, it roots its requests in the specified directory. This can cause a conflict with BOOTP.

For example, suppose the boot file is specified in **bootptab** as **/tftpboot/vxBoot/vxWorks.vx245**. After getting the request, the BOOTP server checks for the existence of this file, and then sends a reply. In response to the

---

10. You need this symbol table to be resident on the target only if you want to use target-resident (pre-Tornado) tools. Host-based tools do not require a target-resident symbol table.

BOOTP reply, the target sends a TFTP request to get the file
/tftpboot/vxBoot/vxWorks.vx245.

However, if the TFTP server was started with the **-s /tftpboot** option, the request
fails because the server looks for the file in **/tftpboot/tftpboot/vxBoot** rather than
in **/tftpboot/vxBoot**. If this is a problem, link **/tftpboot/tftpboot** to **/tftpboot**. To
do this, use the following commands:

```
% cd /tftpboot
% ln -s . tftpboot
```

Multiple BOOTP Servers

If there are multiple BOOTP servers on the network, the target uses the parameters
specified in the first reply message it receives. In the previous example, the server
from which the reply message came is specified in an output line like the
following:

```
Bootp Server:150.12.1.159
```

Subnet Mismatch between DHCP Server and Client

The DHCP server distributes IP addresses only to clients that reside on the same
subnet as the server. If you want the client to use a server on a different subnet, you
must setup a DHCP relay agent on the target's subnet.

13.5 Booting from the Shared-Memory Network

Targets on the shared-memory network can boot using BOOTP only if proxy ARP
is enabled (see *Proxy ARP Overview*, p.80). A target on the shared-memory
network keys its entry in the BOOTP database by its IP address. A shared-memory
network target's entry in the BOOTP database looks something like:

```
vx232:ip=150.12.1.232:tc=global.dummy
```

A shared-memory network's master entry in the BOOTP database looks like:

```
vx230:ht=ethernet:ha=0000530e0018:ip=150.12.1.230:tc=global.dummy
```

The following example is a master processor using a combination of BOOTP, TFTP,
proxy ARP, sequential addressing, and proxy default addressing for booting:[11]

```
    [VxWorks Boot]: @
    boot device      : ln
    processor number : 0
    flags (f)        : 0xc0

    Attaching network interface ln0... done.
    Getting boot parameters via network interface ln0.
    Bootp Server:150.12.1.159
[1]     Boot file: /usr/wind/target/vxBoot/vxWorks.vx230
[1]     Boot host: 150.12.1.159
[1]     Boot device Addr (ln0): 150.12.1.230
[1]     Subnet mask: 0xffffff00
    Attaching network interface lo0... done.
    Loading... 370356 + 28040 + 20196
    Starting at 0x1000...

[2] Host Name: bootHost
[2] Target Name: vxTarget
[2] User: target
    Attaching network interface ln0... done.
    Initializing backplane net with anchor at 0x800000... done.
    Backplane anchor at 0x800000... Attaching network interface sm0...
done.

[3] Backplane address: 150.12.1.231
    Creating proxy network: 150.12.1.231
    Attaching network interface lo0... done.
```

The parameters from the preceding output came from the following sources:

[1] These lines all display information retrieved from the BOOTP database.

[2] These lines display information you set during configuration (defaults).

[3] These lines appear because you have configured VxWorks with "proxy arp
    server," "auto address setup," and "default address for bp" (the configuration
    macros **INCLUDE_PROXY_SERVER**, **INCLUDE_SM_SEQ_ADDR**, and
    INCLUDE_PROXY_DEFAULT_ADDR). (Note that the address is one more than
    that of parameter **inet on ethernet**, in this case 150.12.1.230.)

The following example shows booting a slave processor using a combination of
BOOTP, TFTP, and sequential addressing:

```
    [VxWorks Boot]: @
    boot device      : sm=0x800000
    processor number : 1
    flags (f)        : 0x1c0

    Backplane anchor at 0x800000... Attaching network interface sm0...
done.
```

---

11. The use of sequential addressing can make it difficult to use DHCP.

```
[1]  Backplane inet address: 150.12.1.232
     registering proxy client: 150.12.1.232.done.
     Getting boot parameters via network interface sm0.
     Bootp Server:150.12.1.159
[2]  Boot file: /usr/wind/target/vxBoot/vxWorks.vx232
[2]  Boot host: 150.12.1.159
[2]  Subnet mask: 0xffffff00
     Attaching network interface lo0... done.
     Loading... 370356 + 28040 + 20196
     Starting at 0x1000...
[3]  Host Name: bootHost
[3]  Target Name: vxTarget
[3]  User: target
     Backplane anchor at 0x800000... Attaching network interface sm0...
done.
     Attaching network interface lo0... done.
```

The parameters from the preceding output came from the following sources:

[1] These lines appear because you have configured VxWorks with "proxy arp client" and "auto address setup" (the configuration constants **INCLUDE_PROXY_CLIENT** and **INCLUDE_SM_SEQ_ADDR**)[12]. (Note that the address is equal to the master CPU's backplane address plus the client's processor number.)

[2] These lines all display information retrieved from the BOOTP database.

[3] These lines display information you set during configuration(defaults).

13.6 Booting from the Serial Line

VxWorks can communicate with the host operating system over serial connections as well as over networks and backplanes. Over a serial line connection, you can boot VxWorks using either SLIP or PPP.

13.6.1 Booting VxWorks Using SLIP

If you have configured VxWorks to use SLIP, you can use the Serial Line Internet Protocol to boot VxWorks. The relevant configuration macro is **INCLUDE_SLIP**. SLIP supports IP layer software with point-to-point configurations such as RS-232

---

12. The use of sequential IP addressing makes this procedure incompatible with DHCP.

serial connections between machines or long-distance telephone lines. If either end of a SLIP connection has other network interfaces (such as Ethernet), it can forward packets to other machines.

NOTE: Both target and host must agree on the MTU size. On a VxWorks system, the default MTU size is 576.

Optionally, you can use compressed TCP/IP headers over SLIP. This variant of the protocol is known as CSLIP (compressed SLIP). Only the TCP/IP headers are compressed, not the data itself; this implies that CSLIP improves the responsiveness of interactive communications (such as remote shells), where the ratio of header size to data is large, but makes little difference for large data transfers (such as downloading object code). Because compression applies only to TCP/IP headers, not to other forms of IP, CSLIP has no impact on applications that use UDP rather than TCP (for example, CSLIP has no effect on NFS).[13]

When booting using SLIP (or its CSLIP variant), specify the boot device as follows:

```
boot device: sl    or    sl=device
```

Using the form **sl=***device* allows you to specify the SLIP *tty*, overriding the configuration constant **SLIP_TTY**.

The following is a boot example for the configuration shown in Figure 13-2:

```
boot device          : sl=/tyCo/1
processor number     : 0
host name            : phobos
file name            : /usr/wind/target/config/ads302/vxWorks
inet on ethernet (e) : 150.12.1.2
host inet (h)        : 150.12.1.1
user (u)             : jane
target name (tn)     : vxJane
```

When the boot device is SLIP, the SLIP interface is configured by *usrSlipInit( )* in **target/src/config/usrNetwork.c**. This sets up the SLIP *tty*, and configures the point-to-point connection using the target and host IP addresses specified in the boot parameters. If a gateway address is specified, the SLIP driver adds a routing entry from the gateway address to the host address. If a gateway address is not

---

13. If your host operating system does not include SLIP or CSLIP facilities, consider using a publicly available implementation. One popular implementation for SunOS 4.1.*x*, the Van Jacobson CSLIP 2.7 release, is provided in **target/unsupported/cslip-2.7**. This code is publicly available, and is not supported by Wind River Systems. It is included only as a convenience.

Figure 13-2   **SLIP Configuration Example**
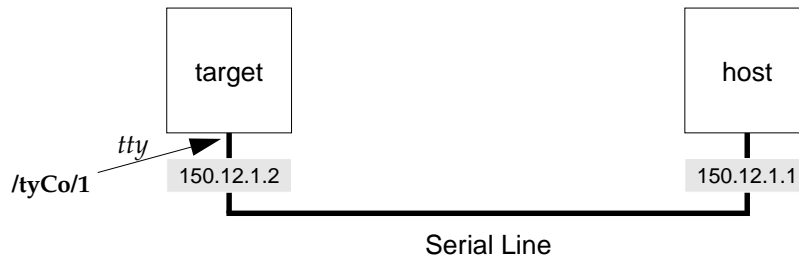
specified, the SLIP driver assumes that the point-to-point peer address on the other end of the serial line is the gateway and enters the appropriate routing entry.

If you do not have a second serial port, then you must use the console port as the SLIP port. To do this:

1.   Set the console serial port (configuration constant **CONSOLE_TTY**) to **NONE** and define the SLIP channel identifier (configuration constant **SLIP_TTY**):

```
#define    CONSOLE_TTY    NONE
#define    SLIP_TTY       0       /* use port number 0 for slip */
```

2.   Remake VxWorks and burn new boot ROMs before booting.

To access a UNIX file system, the Internet addresses specified in the target boot parameters must be consistent with those specified when the host connection is created.

13.6.2 Booting VxWorks Using PPP

To boot VxWorks using PPP, first configure PPP into the system (see *PPP Configuration*, p.18) and remake the VxWorks and boot ROM images. After a new boot ROM image has been built, burned into ROM, and installed in the target board, bootstrap the target board to the VxWorks boot ROM prompt.

When booting using PPP, specify the boot device with one of the following options:

boot device: ppp
    If using **boot device: ppp**, the serial channel is set to **PPP_TTY** in the VxWorks configuration and the baud rate is set to the default baud rate of the channel.

    When the boot device is **ppp**, the PPP interface is initialized by *usrPPPInit( )*. This configures the point-to-point connection using the serial device, target,

and host IP addresses specified in the boot parameters. And it configures in the options defined at compile-time in the configuration (see the Tornado User's Guide: Projects for more information on configuring VxWorks). If a gateway address is specified, the PPP driver adds a routing entry from the gateway address to the host address. If a gateway address is not specified, the PPP software assumes that the point-to-point peer address is on the other end of the serial line and enters the appropriate routing entry.

ppp=*device*
Specifying **ppp=***device* allows you to choose the PPP *tty* (serial channel), overriding the **PPP_TTY** constant.

ppp=*device,baudrate*
Specifying **ppp=***device,baudrate* allows you to choose the PPP *tty* (serial channel) and the baud rate of the channel.

ppp,*baudrate*
The default baud rate used by the PPP *tty* (serial channel) can be configured into the system by defining the constant **PPP_BAUDRATE** (in **config.h**) as the required baud rate, and remaking VxWorks and the boot ROM images.

However, the baud rate supplied as a part of the boot device overrides any default settings. The following is a boot example for the configuration shown in Figure 13-3:

```
boot device          : ppp=/tyCo/2,38400
processor number     : 0
host name            : mars
file name            : /usr/vw/config/mv167/vxWorks
inet on ethernet (e) : 90.0.0.10
host inet (h)        : 90.0.0.1
user (u)             : jane
target name (tn)     : vxJane
```

If you want to boot VxWorks over a PPP link but do not have a console device, the following additional modifications must be made:

1. Set the console serial port (configuration constant **CONSOLE_TTY**) to **NONE** and define the *tty* port number using the constant **PPP_TTY** in **config.h**:

```
#define   PPP_TTY      0        /* use port number 0 for PPP */
```

2. Specify the default boot line (configuration constant **DEFAULT_BOOT_LINE**) before making your boot ROMs. Changing any of the default PPP settings requires new boot ROMs. An example of a default boot line is:

Figure 13-3   **PPP Configuration Example**
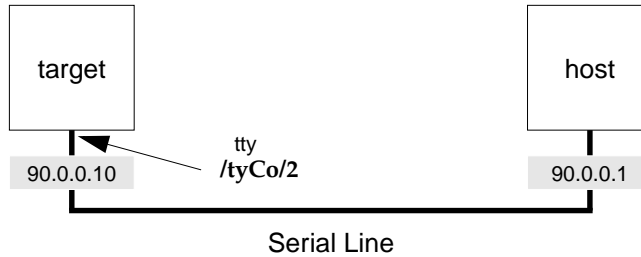

```
"ppp(0,0)mars:/usr/vw/config/mv167/vxWorks h=90.0.0.1 e=90.0.0.10 u=jane"
```

3.   If your system has nonvolatile RAM (NVRAM), edit **sysLib.c** and change
     sysNvRamGet( ) to return **ERROR**. This forces the use of the default boot line,
     instead of the value stored in NVRAM.

4.   Initialize PPP on the remote peer.

5.   Boot VxWorks with the new boot ROMs.

14
Upgrading 4.3 BSD Network Drivers

14.1 Introduction

This chapter describes two upgrade paths for 4.3 BSD network drivers. One path simply ports the 4.3 BSD network driver to the BSD 4.4 model. The other path upgrades the 4.3 BSD network driver to an NPT driver (described in the *Network Protocol Toolkit User's Guide*).

Porting a network driver to the 4.4 BSD model should require only minimal changes to the code. In fact, porting some drivers has taken less than a day's work. However, an older driver that does not already use a transmission startup routine can take longer to port.

Porting a network driver to an NPT driver requires more extensive changes. However, it is worth the effort if the driver must handle the following:

- multicasting
- polled-mode Ethernet (necessary for WDB system-mode debugging over a network, a mode that is several orders of magnitude faster than the serial link)
- zero-copy transmission
- support for network protocols other than IP

14.2 Structure of a 4.3 BSD Network Driver

The network drivers currently shipped with VxWorks are based on those available in BSD UNIX version 4.3. These drivers define only one global (user-callable) routine, the driver's *attach* routine. Typically, the name of this routine contains the word, *attach*, prefixed with two letters from the device name. For example, the AMD Lance driver's attach routine is called **lnattach( )**. The **xxattach( )** routine hooks in five function pointers that are mapped into an **ifnet** structure. These functions, listed in Table 14-1, are all called from various places in the IP protocol stack, which has intimate knowledge of the driver.

Table 14-1 **Network Interface Procedure Handles**

Function	Function Pointer	Driver-Specific Routine
initialization	**if_init**	*xxInit( )*
output	**if_output**	*xxOutput( )*
control	**if_ioctl**	*xxIoctl( )*
reset	**if_reset**	*xxReset( )*
watchdog	**if_watchdog** (optional)	*xxWatchdog( )*

Packet reception begins when the driver's interrupt routine is invoked. The interrupt routine does the least work necessary to get the packet off the local hardware, schedules an input handler to run by calling **netJobAdd( )**, and then returns. The **tNetTask** calls the function that was added to its work queue. In the case of packet reception, this the driver's **xxReceive( )** function.

The **xxReceive( )** function eventually sends the packet up to a protocol by calling **do_protocol_with_type( )**. This routine is a switch statement that figures out which protocol to hand the packet off to. This calling sequence is shown in Figure 14-1.

Figure 14-2 shows the call graph for packet transmission. After a protocol has picked an interface on which to send a packet, it calls the **xxOutput( )** routine for that interface. The output routine calls the generic **ether_output( )** function, passing it a pointer to addressing information (usually an **arpcom** structure) as well as the data to be sent. After the data is properly packed, it is placed on the output queue (using the **IF_ENQUEUE** macro), and the driver's start routine is called. The **xxTxStartup( )** routine dequeues as many packets as it can and transmits them on the physical medium.

Figure 14-1   **Packet Reception Call Graph**
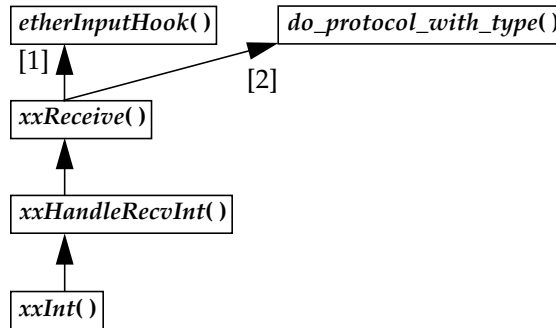
[1] The *xxReceive( )* first shows the packet to *etherInputHook( )*.

[2] If *etherInputHook( )* does not take delivery of the packet, *xxReceive( )* hands the packet to *do_protocol_with_type( )*.

Figure 14-2   **Packet Transmission Call Graph**
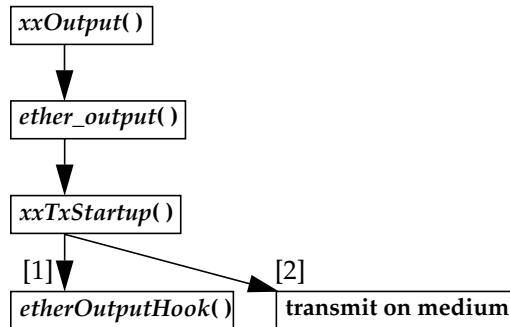
[1] The *xxTxStartup( )* first shows the packet to *etherOutputHook( )*.

[2] If *etherOutputHook( )* does not take delivery of the packet, *xxTxStartup( )* transmits the packet on the medium.

14.2.1 Etherhook Routines Provides Access to Raw Packets

You can use the *etherInputHook( )* and **etherOutputHook( )** routines to bypass the TCP/IP stack and thus get access to raw packets. On packet reception, if an *etherInputHook( )* function is installed, it receives the packet just after the driver has completed reception but before the packet goes to the protocol. If

*etherInputHook***( )** decides to prevent others from seeing the packet,
*etherInputHook***( )** returns a non-zero value and the driver considers the packet to
be delivered. If the *etherInputHook***( )** returns 0, the driver hands the packet to the
TCP/IP stack.

On packet transmission, an installed *etherOutputHook***( )** receives a packet just
before it would have been transmitted. If *etherOutputHook***( )** decides to prevent
the packet from passing on, *etherOutputHook***( )** returns a non-zero value and the
driver considers the packet to be transmitted. If the *etherOutputHook***( )** returns 0,
the driver transmits the packet.

It is only possible to install one *etherInputHook***( )** and one *etherOutputHook***( )**
function per driver. This limits the number of alternate protocols to one, unless
these *ether*Hook* routines then act as a multiplexor for more protocols.

For more information on etherhooks, see the *Tornado BSP Developer's Kit for
VxWorks, Tornado 1.0.1, User's Guide: G.4 Network Interface Hook Routines.*

⚠ **CAUTION:** Future versions of VxWorks will not support etherhooks.

14.3 Upgrading to 4.4 BSD

To upgrade a driver from 4.3 BSD to 4.4 BSD you must change how the driver uses
*ether_attach***( )**. This routine is almost always called from the driver's own
*xxattach***( )** routine and is responsible for placing the driver's entry points, listed in
Table 14-1, into the **ifnet** structure that the TCP/IP protocol to track drivers.
Consider the call to *ether_attach***( )** shown below:

```
ether_attach(
    (IFNET *) & pDrvCtrl->idr,
    unit,
    "xx",
    (FUNCPTR) NULL,
    (FUNCPTR) xxIoctl,
    (FUNCPTR) xxOutput,
    (FUNCPTR) xxReset
    );
```

As arguments, this routine expects an Interface Data Record (**idr**), a unit number,
and a quoted string that is the name of the device, in this case, "xx". The next four
arguments are the function pointers to relevant driver routines.

The first function pointer references this driver's *init*( ) routine, which this driver does not need or have. The second function pointer references the driver's *ioctl*( ) interface, which allows the upper layer to manipulate the device state. The third function pointer references the routine that outputs packets on the physical medium. The last function pointer references a routine that can reset the device if the TCP/IP stack decides that this needs to be done.

In 4.4 BSD, there is a generic output routine called *ether_output*( ) that all Ethernet device drivers can use. Thus, to convert the above *ether_attach*( ) call to a 4.4-style call, you would call *ether_attach*( ) as follows:

```
ether_attach(
    (IFNET *) & pDrvCtrl->idr,
    unit,
    "xx",
    (FUNCPTR) NULL,
    (FUNCPTR) xxIoctl,
    (FUNCPTR) ether_output, /* generic ether_output */
    (FUNCPTR) xxReset
    );
pDrvCtrl->idr.ac_if.if_start = (FUNCPTR)xxTxStartup;
```

This time, there is an extra line following the call to *ether_attach*( ). This line of code adds a transmit startup routine to the Interface Data Record. The transmit startup routine is called by the TCP/IP stack after the generic *ether_output*( ) routine is called. This extra line of code assumes that the driver already has a transmit startup routine. If a driver lacks a separate transmit startup routine, you must write one. See the template in *14.3.4 Creating a Transmit Startup Routine*, p. 214.

14.3.1 Removing the xxOutput Routine

If a 4.3 BSD driver has an *xxOutput*( ) routine, it probably looks something like the following:

```
static int xxOutput
    (
    IDR *     pIDR,
    MBUF *     pMbuf,
    SOCK * pDestAddr
    )
    {
    return (ether_output ((IFNET *)pIDR, pMbuf, pDestAddr,
        (FUNCPTR) xxTxStartup, pIDR));
    }
```

Internally, this routine calls the *ether_output*( ) routine, which expects a pointer to the startup routine as one of its arguments. However, in the 4.4 BSD model, all that

work that is now handled in the TCP/IP stack. Thus, in a 4.4 BSD driver, this code is unnecessary and should be removed.

14.3.2 Changing the Transmit Startup Routine

Under 4.3 BSD, the function prototype for a transmit startup routine is as follows:

```
static void xxTxStartup (int unit);
```

Under 4.4 BSD, the prototype has changed to the following:

```
static void xxTxStartup (struct ifnet * pDrvCtrl);
```

The 4.4 BSD version expects a pointer to a driver control structure. This change eases the burden on the startup routine. Instead of having to find its own driver control structure, it receives a pointer to a driver control structure as input.

If the driver uses *netJobAdd*( ) to schedule the transmit startup routine for task-level execution, edit the *netJobAdd*( ) call to pass in a **DRV_CTRL** structure pointer instead of a unit number.

14.3.3 Changes in Receiving Packets

Under 4.3 BSD, the driver calls *do_protocol_with_type*( ). For example:

```
do_protocol_with_type (etherType, pMbuf, &pDrvCtrl->idr, len);
```

This call expects an *etherType* (which the driver had to discover previously), a pointer to an mbuf containing the packet data, the Interface Data Record, and the length of the data.

Under 4.4 BSD, replace the call above with a call to *do_protocol*( ). For example:

```
do_protocol (pEh, pMbuf, &pDrvCtrl->idr, len);
```

The first parameter is a pointer to the very beginning of the packet (including the link level header). All the other parameters remain the same. The driver no longer needs to figure out the *etherType* for the protocol.

14.3.4 Creating a Transmit Startup Routine

Some 4.3 BSD drivers did not have a transmit startup routine. For such a driver, you must create one. The template is as follows:

```
void templateStartup
    (
    DRV_CTRL *pDrvCtrl
    )
    {
    MBUF * pMbuf;
    int length;
    TFD * pTfd;

    /* Loop until there are no more packets ready to send or we
     * have insufficient resources left to send another one. */
    while (pDrvCtrl->idr.ac_if.if_snd.ifq_head)
        {
        /* Deque a packet from the send queue. */
        IF_DEQUEUE (&pDrvCtrl->idr.ac_if.if_snd, pMbuf);

        /* Device specific code to get transmit resources, such as a
         * transmit descriptor, goes here. */

        if (Insufficient Resources)
            {
            m_freem (pMbuf);/* Make sure to free the packet. */
            return;
            }

        /* pData below is really the place in your descriptor,
         * transmit descriptor, or equivalent, where the data is
         * to be placed. */

        copy_from_mbufs (pData, pMbuf, length);
        if ((etherOutputHookRtn != NULL) &&
            (* etherOutputHookRtn)
            (&pDrvCtrl->idr, (ETH_HDR *)pTfd->enetHdr, length))
            continue;

        /* Do hardware manipulation to set appropriate bits
         * and other stuff to get the packet to actually go.
         */

        /*
         * Update the counter that determines the number of
         * packets that have been output.
         */

        pDrvCtrl->idr.ac_if.if_opackets++;

        } /* End of while loop. */
    } /* End of transmit routine. */
```

Index

Symbols

A

B

IX

IX

IX

IX