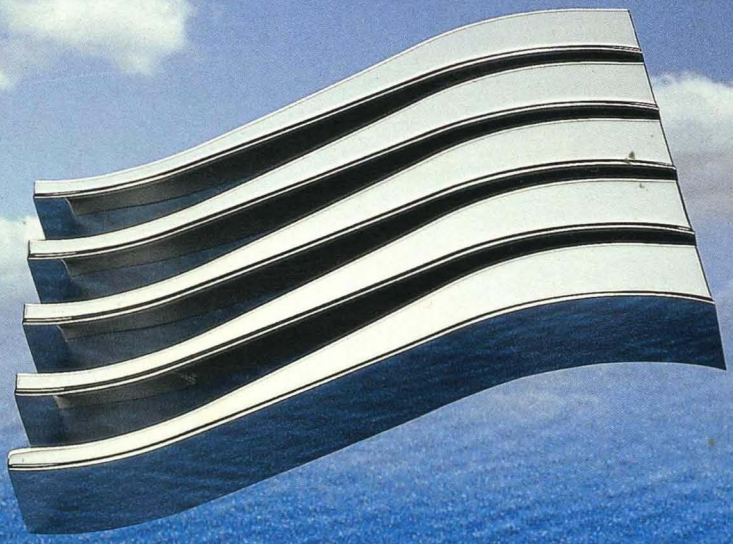
WindRiver
SYSTEMS

VxWorks®

Programmer's Guide

5.3.1



Edition 1

VxWorks®

Programmer's Guide

5.3.1

Edition 1

WindRiver
SYSTEMS

Corporate Headquarters
Wind River Systems, Inc.
1010 Atlantic Avenue
Alameda, CA 94501-1153
USA

toll free (US): 800/545-WIND
telephone: 510/748-4100
facsimile: 510/814-2010

Europe
Wind River Systems, S.A.R.L.
19, Avenue de Norvège
Immeuble B4, Bâtiment 3
Z.A. de Courtaboeuf 1
91953 Les Ulis Cédex
FRANCE

telephone: 33-1-60-92-63-00
facsimile: 33-1-60-92-63-15

Japan
Wind River Systems Japan
Pola Ebisu Bldg. 11F
3-9-19 Higashi
Shibuya-ku
Tokyo 150
JAPAN

telephone: 81-3-5467-5900
facsimile: 81-3-5467-5877

CUSTOMER SUPPORT

	Telephone	E-mail	Fax
Corporate:	800/872-4977 toll free, U.S. & Canada 510/748-4100 direct	support@wrs.com	510/814-2164
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.

VxWorks Programmer's Guide, 5.3.1
Edition 1
4 Mar 97
Part #: DOC-12067-ZD-00

Contents

Appendices

1 Overview

List of Tables

List of Figures

1.1 Introduction

This manual describes VxWorks, the high-performance real-time operating system component of the Tornado development system. This manual includes the following information:

- How to use VxWorks facilities in the development of real-time applications.
- How to use the target-resident tools included in VxWorks.
- How to use the optional components VxVMI, VxMP, and Wind Foundation Classes.

This chapter begins by providing pointers to information on how to set up and start using VxWorks as part of the Tornado development system. It then provides an overview of the role of VxWorks in the development of real-time applications, an overview of VxWorks facilities, a summary of Wind River Systems customer services, and a summary of the document conventions followed in this manual.

1.2 Getting Started with the Tornado Development System

See the following documents for information on installing and configuring the Tornado development system, including VxWorks. Information on configuration differs depending on whether your development host is UNIX or Windows; thus, the *Tornado User's Guide* is host specific.

- The *Wind River Products Installation Guide* provides information on installing all components of the Tornado Development System.

- The *Tornado User's Guide* provides information on configuring and connecting the host and target environments, building your VxWorks application, booting VxWorks, and running Tornado.

For either host, *8. Configuration* in this manual provides advanced VxWorks configuration information.

For a complete overview of Tornado documentation, see the documentation guide in the *Tornado User's Guide.*

1.3 VxWorks: A Partner in the Real-time Development Cycle

UNIX and Windows hosts are excellent systems for program development and for many interactive applications. However, they are not appropriate for real-time applications. On the other hand, traditional real-time operating systems provide poor environments for application development or for non-real-time components of an application, such as graphical user interfaces (GUIs).

Rather than trying to create a single operating system that "does it all," the Wind River philosophy is to utilize two complementary and cooperating operating systems (VxWorks and UNIX, or VxWorks and Windows) and let each do what it does best. VxWorks handles the critical real-time chores, while the host machine is used for program development and for applications that are not time-critical.

You can scale VxWorks to include exactly the feature combinations your application requires. During development, you can include additional features to speed your work (such as the networking facilities), then exclude them to save resources in the final version of your application.

You can use the cross-development host machine to edit, compile, link, and store real-time code, but then run and debug that real-time code on VxWorks. The resulting VxWorks application can run standalone—either in ROM or disk-based—with no further need for the network or the host system.

However, the host machine and VxWorks can also work together in a hybrid application, with the host machine using VxWorks systems as real-time "servers" in a networked environment. For instance, a VxWorks system controlling a robot might itself be controlled by a host machine that runs an expert system, or several VxWorks systems running factory equipment might be connected to host machines that track inventory or generate reports.

1.4 VxWorks Facilities: An Overview

This section provides a summary of VxWorks facilities; they are described in more detail in the following subsections. For details on any of these facilities, see the appropriate chapters in this manual.

- **High-Performance Real-time Kernel Facilities**

  The VxWorks kernel, *wind*, includes multitasking with preemptive priority scheduling, intertask synchronization and communications facilities, interrupt handling support, watchdog timers, and memory management.

- **POSIX Compatibility**

  VxWorks provides most interfaces specified by the 1003.1b standard (formerly the 1003.4 standard), simplifying your ports from other conforming systems.

- **I/O System**

  VxWorks provides a fast and flexible ANSI C-compatible I/O system, including UNIX standard buffered I/O and POSIX standard asynchronous I/O. VxWorks includes the following drivers:

  Network driver – for network devices (Ethernet, shared memory)
  Pipe driver – for intertask communication
  RAM "disk" driver – for memory-resident files
  SCSI driver – for SCSI hard disks, diskettes, and tape drives
  Keyboard driver – for PC x86 keyboards (x86 BSP only)
  Display driver – for PC x86 VGA displays (x86 BSP only)
  Disk driver – for IDE and floppy disk drives (x86 BSP only)
  Parallel port driver – for PC-style target hardware

- **Local File Systems**

  VxWorks provides fast file systems tailored to real-time applications. One file system is compatible with the MS-DOS® file system, another with the RT-11 file system, a third is a "raw disk" file system, and a fourth supports SCSI tape devices.

- **Network Facilities**

  VxWorks provides "transparent" access to other VxWorks and TCP/IP-networked systems, a BSD[1] Sockets-compliant programming interface, remote procedure calls (RPC), SNMP (optional), remote file access (including NFS client and server facilities and a non-NFS facility utilizing RSH, FTP, or TFTP),

1. *BSD* stands for Berkeley Software Distribution, and refers to a version of UNIX.

BOOTP, and proxy ARP. All VxWorks network facilities comply with standard Internet protocols, both loosely coupled over serial lines or standard Ethernet connections and tightly coupled over a backplane bus using shared memory.

- **Virtual Memory (Including VxVMI Option)**

  VxWorks provides both bundled and unbundled (VxVMI) virtual memory support for boards with an MMU, including the ability to make portions of memory noncacheable or read-only, as well as a set of routines for virtual-memory management.

- **Shared-Memory Objects (VxMP Option)**

  The VxMP option provides facilities for sharing semaphores, message queues, and memory regions between tasks on different processors.

- **Target-resident Tools**

  In the Tornado development system, the development tools reside on the host system; see the *Tornado User's Guide* for details. However, a target-resident shell, module loader and unloader, and symbol table can be configured into the VxWorks system if necessary.

- **Wind Foundation Classes**

  In addition to general C++ support including the Iostreams library from AT&T, the optional component Wind Foundation Classes adds the following C++ object libraries:

  - VxWorks Wrapper Class library
  - Tools.h++ library from Rogue Wave
  - Booch Components library from Rogue Wave

- **Utility Libraries**

  VxWorks provides an extensive set of utility routines, including interrupt handling, watchdog timers, message logging, memory allocation, string formatting and scanning, linear and ring buffer manipulations, linked-list manipulations, and ANSI C libraries.

- **Performance Evaluation Tools**

  VxWorks performance evaluation tools include an execution timer for timing a routine or group of routines, and utilities to show CPU utilization percentage by task.

- **Target Agent**

  The target agent allows a VxWorks application to be remotely debugged using the Tornado development tools.

- **Board Support Packages**

  Board Support Packages (BSPs) are available for a variety of boards and provide routines for hardware initialization, interrupt setup, timers, memory mapping, and so on.

- **VxWorks Simulator (VxSim)**

  The optional component VxWorks Simulator, available for UNIX environments only, simulates a VxWorks target for use as a prototyping and testing environment.

Multitasking and Intertask Communications

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows real-time applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The intertask communication facilities allow these tasks to synchronize and coordinate their activity.

The VxWorks multitasking kernel, *wind*, uses interrupt-driven, priority-based task scheduling. It features fast context switch times and low interrupt latency. Under VxWorks, any subroutine can be *spawned* as a separate task, with its own context and stack. Other basic task control facilities allow tasks to be suspended, resumed, deleted, delayed, and moved in priority. See 2.3 *Tasks*, p.30 and the reference entry for **taskLib**.

The *wind* kernel supplies semaphores as the basic task synchronization and mutual-exclusion mechanism. There are several kinds of semaphores in *wind*, specialized for different application needs: binary semaphores, counting semaphores, mutual-exclusion semaphores, and POSIX semaphores. All of these semaphore types are fast and efficient. In addition to being available to application developers, they have also been used extensively in building higher-level facilities in VxWorks.

For intertask communications, the *wind* kernel also supplies message queues, pipes, sockets, and signals. The optional component VxMP provides shared-memory objects as a communication mechanism for tasks executing on different CPUs. For information on all these facilities, see *6. Shared-Memory Objects* and

2.4 *Intertask Communications*, p.54. In addition, semaphores are described in the **semLib** and **semPxLib** reference entries; message queues are described in the **msgQLib** and **mqPxLib** reference entries; pipes are described in the **pipeDrv** reference entry and 2.4.5 *Pipes*, p.88; sockets are described in the **sockLib** reference entry and 2.4.6 *Network Intertask Communication*, p.89; and signals are described in the **sigLib** reference entry and 2.4.7 *Signals*, p.90.

POSIX Interfaces

POSIX (the Portable Operating System Interface) is a set of standards under development by representatives of the software community, working under an ISO/IEEE charter. The purpose of this effort is to support application portability at the source level across operating systems. This effort has yielded a set of interfaces (POSIX standard 1003.1b, formerly called 1003.4) for real-time operating system services. Using these interfaces makes it easier to move applications from one operating system to another.

For a list of POSIX facilities, look under **POSIX** in the keyword index in the *VxWorks Reference Manual* or in the *Tornado Online Manuals*. Nearly all POSIX 1003.1b interfaces are available in VxWorks, including POSIX interfaces for:

- asynchronous I/O
- semaphores
- message queues
- memory management
- queued signals
- scheduling
- clocks and timers

In addition, several interfaces from the traditional POSIX 1003.1 standard are also supported.

I/O System

The VxWorks I/O system provides uniform device-independent access to many kinds of devices. You can call seven basic I/O routines: ***creat( )***, ***remove( )***, ***open( )***, ***close( )***, ***read( )***, ***write( )***, and ***ioctl( )***. Higher-level I/O routines (such as ANSI C-compatible ***printf( )*** and ***scanf( )*** routines) are also provided.

VxWorks also provides a standard buffered I/O package (*stdio*) that includes ANSI C-compatible routines such as ***fopen( )***, ***fclose( )***, ***fread( )***, ***fwrite( )***, ***getc( )***, and ***putc( )***. These routines increase I/O performance in many cases.

The VxWorks I/O system also includes POSIX-compliant asynchronous I/O: a library of routines that perform input and output operations concurrently with a task's other activities.

VxWorks includes device drivers for serial communication, disks, RAM disks, SCSI tape devices, intertask communication devices (called *pipes*), and devices on a network. Application developers can easily write additional drivers, if needed. VxWorks allows dynamic installation and removal of drivers without rebooting the system.

Internally, the VxWorks I/O system allows individual drivers complete control over how the user requests are serviced. Drivers can easily implement different protocols, unique device-specific routines, and even different file systems, without interference from the I/O system itself. VxWorks also supplies several high-level packages that make it easy for drivers to implement common device protocols and file systems.

For a detailed discussion of the I/O system, see *3. I/O System*. Relevant reference entries include **ioLib** for basic I/O routines available to tasks, **fioLib** and **ansiStdio** for various format-driven I/O routines, **aioPxLib** for asynchronous I/O, and **iosLib** and **tyLib** for routines available to driver writers. Also see the reference entries for the supplied drivers.

Local File Systems

VxWorks includes several local file systems for use with block devices (disks). These devices all use a standard interface so that file systems can be freely mixed with device drivers. A local file system for SCSI tape devices is also included. The VxWorks I/O architecture makes it possible to have several different file systems on a single VxWorks system, even at the same time.

MS-DOS Compatible File System: dosFs

VxWorks provides the *dosFs* file system, which is compatible with the MS-DOS file system (for MS-DOS versions up to and including 6.2). The capabilities of dosFs offer considerable flexibility appropriate to the varying demands of real-time applications. Major features include:

- A hierarchical arrangement of files and directories, allowing efficient organization and permitting an arbitrary number of files to be created on a volume.

- The ability to specify contiguous file allocation on a per-file basis. Contiguous files offer enhanced performance, while non-contiguous files result in more efficient use of disk space.
- Compatibility with widely available storage and retrieval media. Diskettes created with dosFs and on MS-DOS personal computers can be freely interchanged and hard drives created with MS-DOS can be read by dosFs if it is correctly configured.
- Optional case-sensitive file names, with name lengths not restricted to the MS-DOS eight-character + extension convention.

Services for file-oriented device drivers using dosFs are implemented in **dosFsLib**.

RT-11 Compatible File System: rt11Fs

VxWorks provides the *rt11Fs* file system, which is compatible with that of the RT-11 operating system. This file system has been used for real-time applications because all files are contiguous. However, rt11Fs does have some drawbacks. It lacks a hierarchical file organization that is particularly useful on large disks. Also, the rigid contiguous allocation scheme may result in fragmented disk space. For these reasons, dosFs is preferable to rt11Fs.

The VxWorks implementation of the RT-11 file system includes byte-addressable random access (seeking) to all files. Each open file has a block buffer for optimized reading and writing.

Services for file-oriented device drivers using rt11Fs are implemented in **rt11FsLib**.

Raw Disk File System: rawFs

VxWorks provides *rawFs*, a simple "raw disk file system" for use with disk devices. rawFs treats the entire disk much like a single large file. The rawFs file system permits reading and writing portions of the disk, specified by byte offset, and it performs simple buffering. When only simple, low-level disk I/O is required, rawFs has the advantages of size and speed.

Services for file-oriented device drivers using rawFs are implemented in **rawFsLib**.

SCSI Sequential File System: tapeFs

VxWorks provides a file system for tape devices that do not use a standard file or directory structure on tape. The tape volume is treated much like a raw device where the entire volume is a large file. Any data organization on this large file is the responsibility of a higher-level layer.

Services for SCSI sequential device drivers using tapeFs are implemented in **tapeFsLib**.

Alternative File Systems

In VxWorks, the file system is not tied to the device or its driver. A device can be associated with any file system. Alternatively, you can supply your own file systems that use standard drivers in the same way, by following the same standard interfaces between the file system, the driver, and the VxWorks I/O system.

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. VxWorks uses the Internet protocols as implemented in BSD 4.3 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.

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 4.3 UNIX, or you can use the *zbuf socket interface* to streamline throughput. (The TCP subset of the zbuf interface is sometimes called "zero-copy TCP.")

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.

For information on sockets, see *5.2.6 Sockets*, p.251 and *5.2.7 The Zbuf Socket Interface*, p.264 and the reference entries for **sockLib** and **zbufSockLib**.

Remote Procedure Calls (RPC)

Originally designed by Sun Microsystems using the Sun ONC standard and 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. See the RPC documentation (publicly and commercially available) and the reference entry for **rpcLib**.

Simple Network Management Protocol (WindNet SNMP Option)

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 exactly 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 real-time 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.137, *5.3.4 Remote File Transfer Using TFTP*, p.291, and *3.7.5 Non-NFS Network Devices*, p.138.

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.*

Virtual Memory (Including VxVMI Option)

Virtual memory support is provided for boards with Memory Management Units (MMU). Bundled virtual memory support provides the ability to mark buffers noncacheable. This is useful for multiprocessor environments where memory is shared across processors or where DMA transfers take place. For information on

bundled virtual memory support, see *7. Virtual Memory Interface* and the reference entries for **vmBaseLib** and **cacheLib**.

Unbundled virtual memory support is available as the optional component VxVMI. VxVMI provides the ability to make text segments and the exception vector table read-only, and includes a set of routines for developers to manage their own virtual memory contexts. For information on VxVMI, see *7. Virtual Memory Interface* and the reference entry for **vmLib**.

Shared-Memory Objects (VxMP Option)

The following shared-memory objects (available with VxWorks as the optional component, VxMP) are used for communication and synchronization between tasks on different CPUs:

- Shared semaphores can be used to synchronize tasks on different CPUs as well as provide mutual exclusion to shared data structures.
- Shared message queues allow tasks on multiple processors to exchange messages.
- Shared memory management is available to allocate common data buffers for tasks on different processors.

For information on VxMP, see *6. Shared-Memory Objects* and the reference entries for **smObjLib**, **smObjShow**, **semSmLib**, **msgQSmLib**, **smMemLib**, and **smNameLib**.

Target-Resident Tools

In the Tornado development system, a full suite of development tools reside and execute on the host machine; see the *Tornado User's Guide* for details. However, a target-resident shell, symbol table, and module loader/unloader can be configured into the VxWorks system if necessary, for example, to create a dynamically configured run-time system.

For information on these target-resident tools, see *9. Target Shell* and the reference entries for **shellLib**, **usrLib**, **dbgLib**, **loadLib**, **unldLib**, and **symLib**.

C++ Development (including Wind Foundation Classes Option)

VxWorks supports C++ development. The GNU C++ compiler is shipped with Tornado. The Iostreams library provides support for formatted I/O in C++. The standard Tornado interactive development tools such as the debugger, the shell, and the incremental loader include C++ support.

In addition, you can order the Wind Foundation Classes optional component to add the following libraries:

- VxWorks Wrapper Class library
- Tools.h++ library from Rogue Wave
- Booch Components library from Rogue Wave

For more information on these libraries, see *10. C++ Development*.

Utility Libraries

VxWorks supplies many subroutines of general utility to application developers. These routines are organized as a set of subroutine libraries, which are described below. We urge you to use these libraries wherever possible. Using library utilities reduces both development time and memory requirements for the application.

Interrupt Handling Support

VxWorks supplies routines for handling hardware interrupts and software traps without having to resort to assembly language coding. Routines are provided to connect C routines to hardware interrupt vectors, and to manipulate the processor interrupt level.

For information on interrupt handling, see the **intLib** and **intArchLib** reference entries. Also see *2. Basic OS* for information about the context where interrupt-level code runs and for special restrictions that apply to interrupt service routines.

Watchdog Timers

A watchdog facility allows callers to schedule execution of their own routines after specified time delays. As soon as the specified number of ticks have elapsed, the specified "timeout" routine is called at the interrupt level of the system clock, unless the watchdog is canceled first. This mechanism is entirely different from the kernel's task delay facility. For information on watchdog timers, see *2.6 Watchdog Timers*, p.99 and the reference entry for **wdLib**.

Message Logging

A simple message logging facility allows applications to send error or status messages to a logging task, which then formats and outputs the messages to a system-wide logging device (such as the system console, disk, or accessible memory). The message logging facility can be used from either interrupt level or task level. For information on this facility, see *3.5.3 Message Logging*, p.122 and the reference entry for **logLib**.

Memory Allocation

VxWorks supplies a memory management facility useful for dynamically allocating, freeing, and reallocating blocks of memory from a memory pool. Blocks of arbitrary size can be allocated, and you can specify the size of the memory pool. This memory scheme is built on a much more general mechanism that allows VxWorks to manage several separate memory pools.

String Formatting and Scanning

VxWorks includes a complete set of ANSI C library string formatting and scanning subroutines that implement ***printf*( )/*scanf*( )** format-driven encoding and decoding and associated routines. See the reference entries for **fioLib** and **ansiStdio**.

Linear and Ring Buffer Manipulations

The library **bLib** contains buffer manipulation routines such as copying, filling, comparing, and so on, that have been optimized for speed. The library **rngLib** provides a set of general ring buffer routines that manage first-in-first-out (FIFO) circular buffers. Additionally, these ring buffers have the property that a single writer and a single reader can access a ring buffer "simultaneously" without being required to interlock their accesses explicitly.

Linked-List Manipulations

The library **lstLib** contains a complete set of routines for creating and manipulating doubly-linked lists.

ANSI C Libraries

VxWorks provides all C libraries specified by ANSI X3.159-1989. The ANSI C specification includes the following libraries: **assert, ctype, errno, float, limits, locale, math, setjmp, signal, stdarg, stdio, stddef, stdlib, string,** and **time**.

The header files **float.h**, **limits.h**, **errno.h**, and **stddef.h** provide ANSI-specified definitions and declarations. The more commonly used libraries are described in the following reference entries:

ansiCtype	routines for character manipulation.
ansiMath	trigonometric, exponential, and logarithmic routines.
ansiSetjmp	routines for implementing a non-local goto.
ansiStdarg	routines for traversing a variable-length argument list.
ansiStdio	routines for manipulating streams for input/output.
ansiStdlib	a variety of routines, including those for type translation, memory allocation, and random number generation.
sigLib	signal-manipulation routines.

Performance Evaluation

To understand and optimize the performance of a real-time system, it can be useful to time some of the VxWorks or application routines. VxWorks provides various timing facilities to help with this task.

The VxWorks execution timer can time any subroutine or group of subroutines. Because the system clock is too slow to provide the resolution necessary to time especially fast routines, the timer can also repeatedly execute a group of routines until the time of a single iteration is known to a reasonable accuracy. For information on the execution timer, see the **timexLib** reference entry.

VxWorks also provides the *spy* utility, which provides CPU utilization information for each task: the CPU time consumed, the time spent at interrupt level, and the amount of idle time. Time is displayed in ticks and in percentages. For information on this utility, see the **spyLib** reference entry.[2]

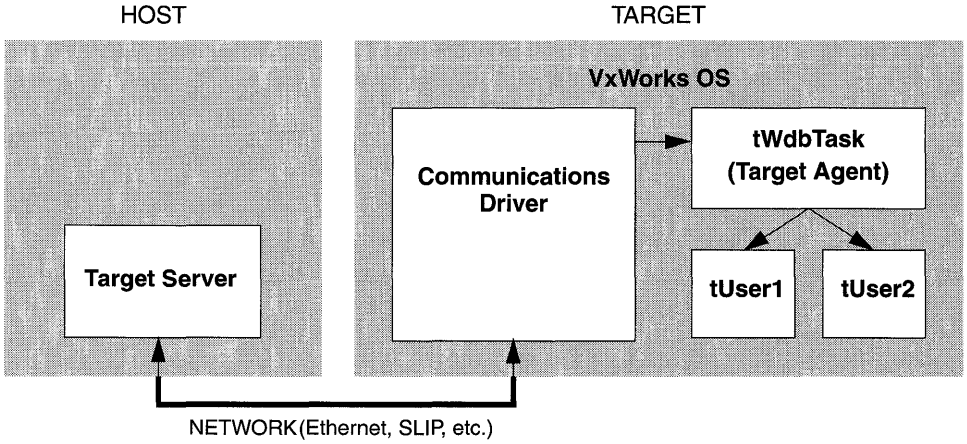
Even more powerful monitoring of the VxWorks system is available using the optional product WindView; for more information, see the *WindView User's Guide*.

2. You can also use this utility through the Tornado browser; see the *Tornado User's Guide: Browser* for details.

Target Agent

The *target agent* follows the WDB (Wind DeBug) protocol, allowing a VxWorks target to be connected to the Tornado development tools. In the target agent's default configuration, shown in Figure 1-1, the agent runs as the VxWorks task **tWdbTask**. The Tornado target server sends debugging requests to the target agent. The debugging requests often result in the target agent controlling or manipulating other tasks in the system.

Figure 1-1 **Interaction Between Target Server and Target Agent**



By default, the target server and agent communicate using the network. However, you can use alternative communication paths. For more information on the default configuration or alternative configurations of the target agent, see the *Tornado User's Guide: Getting Started.* For information on the Tornado target server, see the *Tornado User's Guide: Overview.*

Board Support Packages (BSPs)

Two target-specific libraries, **sysLib** and **sysALib**, are included with each port of VxWorks. These libraries are the heart of VxWorks portability; they provide an identical software interface to the hardware functions of all boards. They include facilities for hardware initialization, interrupt handling and generation, hardware clock and timer management, mapping of local and bus memory spaces, memory sizing, and so on.

Each BSP also includes a boot ROM or other boot mechanism. Many of these import the run-time image from the development host. For information on boot ROMs and other booting mechanisms see the *Tornado User's Guide: Getting Started.*

For information on target-specific libraries, see *8.2 The Board Support Package (BSP)*, p.427 and the target-specific reference entries for your board type.

VxWorks Simulator (VxSim Option)

VxSim, the VxWorks Simulator, is a UNIX program that simulates a VxWorks target for use as a prototyping and testing environment. This optional product is available for UNIX environments only.

VxSim is essentially a port of VxWorks to UNIX. In most regards, its capabilities are identical to a true VxWorks system running on remote target hardware. You can link in an application and rebuild the VxWorks image exactly the same way as in any other VxWorks cross-development environment. All Tornado development tools can be used with VxSim.

The difference between VxSim and a remote VxWorks target environment is that in VxSim, the image executes on the UNIX machine itself as a UNIX process. There is no emulation of instructions, because the code is in the host's own CPU architecture. VxSim includes a User Level IP (ULIP) driver, allowing it to obtain an Internet address and communicate with the host (or other nodes on the network) using the VxWorks networking tools.

Because target hardware interaction is not possible, device-driver development may not be suitable for simulation. However, the VxWorks scheduler is implemented in the VxSim UNIX process, maintaining true tasking interaction with respect to priorities and preemption. This means that any application that is written in a portable style and with minimal hardware interaction should be portable between VxSim and VxWorks.

For more information on VxSim, see the *VxSim User's Guide.*

1.5 Customer Services

A full range of support services is available from Wind River Systems to ensure that you have the opportunity to make optimal use of the extensive features of VxWorks.

This section summarizes the major services available. For more detailed information, consult the *Tornado User's Guide: Customer Service.*

Training

In the United States, Wind River Systems holds regularly scheduled classes on Tornado and VxWorks. Customers can also arrange to have Tornado classes held at their facility. The easiest way to learn about WRS training services, schedules, and prices is through the World Wide Web. Point your site's Web browser at the following URL:

```
http://www.wrs.com/trainmain.html
```

You can also receive the training schedule from an automatic e-mail server. Send e-mail with the following text in the header:

```
To: server@wrs.com
Subject: training
```

You can contact the Training Department at:

Phone:	510/748–4100
	800/545–WIND
Fax:	510/814–2010
E-mail:	training@wrs.com

Outside of the United States, call your local distributor or nearest Wind River Systems office for training information. See the back cover of this manual for a list of Wind River Systems offices.

Customer Support

Direct contact with a staff of software engineers experienced in VxWorks is available through the Wind River Systems Customer Support program. For information on how to contact WRS Customer Support, see the copyright page at the front of this manual.

1.6 Documentation Conventions

Typographical Conventions

VxWorks documentation uses the conventions shown in Table 1-1 to differentiate various elements. Parentheses are always included to indicate a subroutine name, as in ***printf*( )**.

Table 1-1 **Font Usage for Special Terms**

Term	Example
files, pathnames	**/etc/hosts**
libraries, drivers	**memLib, nfsDrv**
host tools	**more, chkdsk**
subroutines	***semTake*( )**
boot commands	**p**
code display	`main ();`
keyboard input	`make CPU=MC68040 ...`
display output	`value = 0`
user-supplied parameters	*name*
constants	**INCLUDE_NFS**
C keywords, **cpp** directives	**#define**
named key on keyboard	**RETURN**
control characters	**CTRL+C**
lower-case acronyms	*fd*

Cross-References

Cross-references in this guide to a *reference entry* for a tool or module refer to an entry in the *VxWorks Reference Manual* (for target libraries or subroutines) or to the reference appendix in the *Tornado User's Guide* (for host tools). These references are also provided in the *Tornado Online Manuals*. For more information about how to access online documentation, see the *Tornado User's Guide: Documentation Guide*.

Other references from one book to another are always at the chapter level, and take the form *Book Title: Chapter Name*.

Pathnames

The top-level Tornado directory structure includes three major directories (see the *Tornado User's Guide: Directories and Files*). Because all VxWorks files reside in the **target** directory, this manual uses relative pathnames starting below that directory. For example, if you install Tornado in **/usr/wind**, the full pathname for the file shown as **config/all/configAll.h** is **/usr/wind/target/config/all/configAll.h**.

NOTE: In this manual, forward slashes are used as pathname delimiters for both UNIX and Windows filenames.

2 Basic OS

List of Tables

List of Figures

List of Examples

2.1 Introduction

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows a real-time application to be constructed as a set of independent tasks, each with its own thread of execution and set of system resources. The intertask communication facilities allow these tasks to synchronize and communicate in order to coordinate their activity. In VxWorks, the intertask communication facilities range from fast semaphores to message queues and pipes to network-transparent sockets.

Another key facility in real-time systems is hardware interrupt handling, because interrupts are the usual mechanism to inform a system of external events. To get the fastest possible response to interrupts, *interrupt service routines (ISRs)* in VxWorks run in a special context of their own, outside of any task's context.

This chapter discusses the multitasking kernel, tasking facilities, intertask communication, and interrupt handling facilities, which are at the heart of the VxWorks run-time environment.

2.2 Wind Features and POSIX Features

The POSIX standard for real-time extensions (1003.1b) specifies a set of interfaces to kernel facilities. To improve application portability, the VxWorks kernel, *wind*, includes both POSIX interfaces and interfaces designed specifically for VxWorks.

This manual (especially in this chapter) uses the qualifier "Wind" to identify facilities designed expressly for use with the VxWorks *wind* kernel. For example, you can find a discussion of Wind semaphores contrasted to POSIX semaphores in *Comparison of POSIX and Wind Semaphores*, p.68.

2.3 Tasks

It is often essential to organize applications into independent, though cooperating, programs. Each of these independent programs, while executing, is called a *task*. In VxWorks, tasks have immediate, shared access to most system resources, while also maintaining enough separate context to maintain individual threads of control.

2.3.1 Multitasking

Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events. The VxWorks real-time kernel, *wind*, provides the basic multitasking environment. Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. Each apparently independent program is called a *task*. Each task has its own *context*, which is the CPU environment and system resources that the task sees each time it is scheduled to run by the kernel. On a context switch, a task's context is saved in the task control block (TCB). A task's context includes:

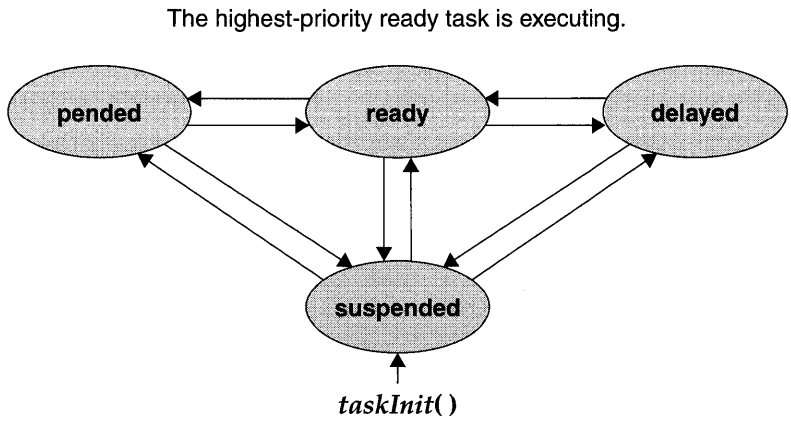
- a thread of execution, that is, the task's program counter
- the CPU registers and (optionally) floating-point registers
- a stack for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a timeslice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

In VxWorks, one important resource that is *not* part of a task's context is memory address space: all code executes in a single common address space. Giving each task its own memory space requires virtual-to-physical memory mapping, which is available only with the optional product VxVMI; for more information, see *7. Virtual Memory Interface*.

2.3.2 Task State Transition

The kernel maintains the current state of each task in the system. A task changes from one state to another as the result of kernel function calls made by the

Figure 2-1 **Task State Transitions**



ready	→	**pended**	*semTake( ) / msgQReceive( )*
ready	→	**delayed**	*taskDelay( )*
ready	→	**suspended**	*taskSuspend( )*
pended	→	**ready**	*semGive( ) / msgQSend( )*
pended	→	**suspended**	*taskSuspend( )*
delayed	→	**ready**	expired delay
delayed	→	**suspended**	*taskSuspend( )*
suspended	→	**ready**	*taskResume( ) / taskActivate( )*
suspended	→	**pended**	*taskResume( )*
suspended	→	**delayed**	*taskResume( )*

Table 2-1 **Task State Transitions**

State Symbol	Description
READY	The state of a task that is not waiting for any resource other than the CPU.
PEND	The state of a task that is blocked due to the unavailability of some resource.
DELAY	The state of a task that is asleep for some duration.
SUSPEND	The state of a task that is unavailable for execution. This state is used primarily for debugging. Suspension does not inhibit state transition, only task execution. Thus *pended-suspended* tasks can still unblock and *delayed-suspended* tasks can still awaken.
DELAY + S	The state of a task that is both delayed and suspended.
PEND + S	The state of a task that is both pended and suspended.
PEND + T	The state of a task that is pended with a timeout value.
PEND + S + T	The state of a task that is both pended with a timeout value and suspended.
state **+ I**	The state of task specified by *state*, plus an inherited priority.

application. When created, tasks enter the *suspended* state. Activation is necessary for a created task to enter the *ready* state. The activation phase is extremely fast, enabling applications to pre-create tasks and activate them in a timely manner. An alternative is the *spawning* primitive, which allows a task to be created and activated with a single function. Tasks can be deleted from any state.

The *wind* kernel states are shown in the state transition diagram in Figure 2-1, and a summary of the corresponding *state symbols* you will see when working with Tornado development tools is shown in Table 2-1.

2.3.3 Wind Task Scheduling

Multitasking requires a scheduling algorithm to allocate the CPU to ready tasks. Priority-based preemptive scheduling is the default algorithm in *wind*, but you can select round-robin scheduling for your applications as well. The routines listed in Table 2-2 control task scheduling.

Table 2-2 **Task Scheduler Control Routines**

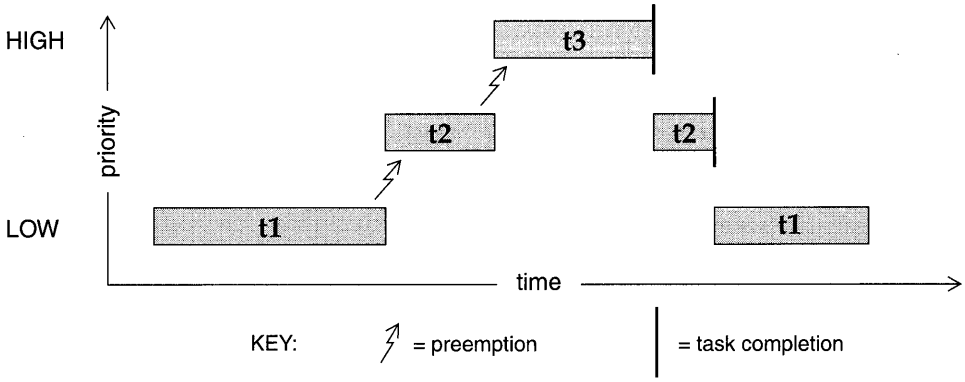
Call	Description
kernelTimeSlice( )	Control round-robin scheduling.
taskPrioritySet( )	Change the priority of a task.
taskLock( )	Disable task rescheduling.
taskUnlock( )	Enable task rescheduling.

Preemptive Priority Scheduling

With a preemptive priority-based scheduler, each task has a priority and the kernel ensures that the CPU is allocated to the highest priority task that is ready to run. This scheduling method is *preemptive* in that if a task that has higher priority than the current task becomes ready to run, the kernel immediately saves the current task's context and switches to the context of the higher priority task. In Figure 2-2, task **t1** is preempted by higher-priority task **t2**, which in turn is preempted by **t3**. When **t3** completes, **t2** continues executing. When **t2** completes execution, **t1** continues executing.

The *wind* kernel has 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest. Tasks are assigned a priority when created;

Figure 2-2 **Priority Preemption**



however, while executing, a task can change its priority using ***taskPrioritySet( )***. The ability to change task priorities dynamically allows applications to track precedence changes in the real world.

Round-Robin Scheduling

Preemptive priority scheduling can be augmented with round-robin scheduling. A round-robin scheduling algorithm attempts to share the CPU fairly among all ready tasks of the *same priority*. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single task can usurp the processor by never blocking, thus never giving other equal-priority tasks a chance to run.

Round-robin scheduling achieves fair allocation of the CPU to tasks of the same priority by an approach known as *time slicing*. Each task of a group of tasks executes for a defined interval, or *time slice*; then another task executes for an equal interval, in rotation. The allocation is fair in that no task of a priority group gets a second slice of time before the other tasks of a group are given a slice.

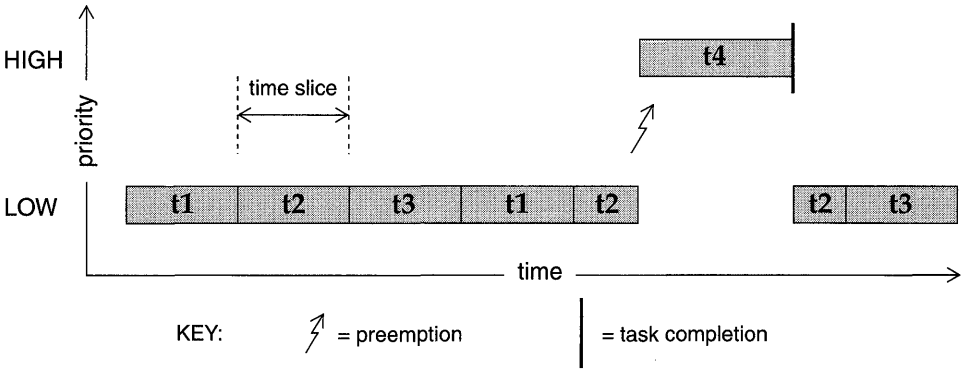
Round-robin scheduling can be enabled with the routine ***kernelTimeSlice( )***, which takes a parameter for a time slice, or interval. This interval is the amount of time each task is allowed to run before relinquishing the processor to another equal-priority task.

More precisely, a run-time counter is kept for each task and incremented on every clock tick. When the specified time-slice interval is completed, the counter is cleared and the task is placed at the tail of the queue of tasks at its priority. New

tasks joining a priority group are placed at the tail of the group with a run-time counter initialized to zero.

If a task is preempted by a higher priority task during its interval, its run-time count is saved and then restored when the task is again eligible for execution. Figure 2-3 shows round-robin scheduling for three tasks of the same priority: **t1**, **t2**, and **t3**. Task **t2** is preempted by a higher priority task **t4** but resumes at the count where it left off when **t4** is finished.

Figure 2-3 **Round-Robin Scheduling**



Preemption Locks

The *wind* scheduler can be explicitly disabled and enabled on a per-task basis with the routines ***taskLock( )*** and ***taskUnlock( )***. When a task disables the scheduler by calling ***taskLock( )***, no priority-based preemption can take place while that task is running.

However, if the task explicitly blocks or suspends, the scheduler selects the next highest-priority eligible task to execute. When the preemption-locked task unblocks and begins running again, preemption is again disabled.

Note that preemption locks prevent task context switching but do not lock out interrupt handling.

Preemption locks can be used to achieve mutual exclusion; however, keep the duration of preemption locking to a minimum. For more information, see *2.4.2 Mutual Exclusion*, p.55.

2.3.4 Tasking Control

The following sections give an overview of the basic VxWorks tasking routines, which are found in the VxWorks library **taskLib**. These routines provide the means for task creation, control, and information. See the reference entry for **taskLib** for further discussion. For interactive use, you can control VxWorks tasks from the host-resident shell; see the *Tornado User's Guide: Shell*.

Task Creation and Activation

The routines listed in Table 2-3 are used to create tasks.

Table 2-3 **Task Creation Routines**

Call	Description
taskSpawn( )	Spawn (create and activate) a new task.
taskInit( )	Initialize a new task.
taskActivate( )	Activate an initialized task.

The arguments to ***taskSpawn*( )** are the new task's name (an ASCII string), priority, an "options" word, stack size, main routine address, and 10 arguments to be passed to the main routine as startup parameters:

```
id = taskSpawn ( name, priority, options, stacksize, main, arg1, ...arg10 );
```

The ***taskSpawn*( )** routine creates the new task context, which includes allocating the stack and setting up the task environment to call the main routine (an ordinary subroutine) with the specified arguments. The new task begins execution at the entry to the specified routine.

The ***taskSpawn*( )** routine embodies the lower-level steps of allocation, initialization, and activation. The initialization and activation functions are provided by the routines ***taskInit*( )** and ***taskActivate*( )**; however, we recommend you use these routines only when you need greater control over allocation or activation.

Task Names and IDs

When a task is spawned, you can specify an ASCII string of any length to be the task name. VxWorks returns a task ID, which is a 4-byte handle to the task's data

structures. Most VxWorks task routines take a task ID as the argument specifying a task. VxWorks uses a convention that a task ID of 0 (zero) always implies the calling task.

A task name should not conflict with any existing task name. Furthermore, to use the Tornado development tools to their best advantage, task names should not conflict with globally visible routine or variable names. To avoid name conflicts, VxWorks uses a convention of prefixing all task names started from the target with the letter *t* and task names started from the host with the letter *u*.

You may not want to name some or all of your application's tasks. If a NULL pointer is supplied for the *name* argument of ***taskSpawn( )***, then VxWorks assigns a unique name. The name is of the form **t***N*, where *N* is a decimal integer that increases by one for each unnamed task that is spawned.

NOTE: In the shell, task names are resolved to their corresponding task IDs to simplify interaction with existing tasks; see the *Tornado User's Guide: Shell*.

The **taskLib** routines listed in Table 2-4 manage task IDs and names.

Table 2-4 **Task Name and ID Routines**

Call	Description
taskName( )	Get the task name associated with a task ID.
taskNameToId( )	Look up the task ID associated with a task name.
taskIdSelf( )	Get the calling task's ID.
taskIdVerify( )	Verify the existence of a specified task.

Task Options

When a task is spawned, an option parameter is specified by performing a logical OR operation on the desired options, listed in the following table. Note that **VX_FP_TASK** must be specified if the task performs any floating-point operations.

To create a task that includes floating-point operations, use:

```
tid = taskSpawn ("tMyTask", 90, VX_FP_TASK, 20000, myFunc, 2387, 0, 0,
                 0, 0, 0, 0, 0, 0, 0, 0);
```

Task options can also be examined and altered after a task is spawned by means of the routines listed in Table 2-6. Currently, only the **VX_UNBREAKABLE** option can be altered.

Table 2-5 **Task Options**

Name	Hex Value	Description
VX_FP_TASK	0x8	Execute with the floating-point coprocessor.
VX_NO_STACK_FILL	0x100	Do not fill stack with 0xee.
VX_PRIVATE_ENV	0x80	Execute task with a private environment.
VX_UNBREAKABLE	0x2	Disable breakpoints for the task.

Table 2-6 **Task Option Routines**

Call	Description
taskOptionsGet( )	Examine task options.
taskOptionsSet( )	Set task options.

Task Information

The routines listed in Table 2-7 get information about a task by taking a snapshot of a task's context when called. The state of a task is dynamic, and the information may not be current unless the task is known to be dormant (that is, suspended).

Table 2-7 **Task Information Routines**

Call	Description
taskIdListGet( )	Fill an array with the IDs of all active tasks.
taskInfoGet( )	Get information about a task.
taskPriorityGet( )	Examine the priority of a task.
taskRegsGet( )	Examine a task's registers.
taskRegsSet( )	Set a task's registers.
taskIsSuspended( )	Check if a task is suspended.
taskIsReady( )	Check if a task is ready to run.
taskTcb( )	Get a pointer to task's control block.

Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in Table 2-8 to delete tasks and protect tasks from unexpected deletion.

WARNING: Make sure that tasks are not deleted at inappropriate times: a task must release all shared resources it holds before an application deletes the task.

Table 2-8 **Task-Deletion Routines**

Call	Description
exit( )	Terminate the calling task and free memory (task stacks and task control blocks only).*
taskDelete( )	Terminate a specified task and free memory (task stacks and task control blocks only).*
taskSafe( )	Protect the calling task from deletion.
taskUnsafe( )	Undo a *taskSafe*( ) (make the calling task available for deletion).

* Memory that is allocated by the task during its execution is *not* freed when the task is terminated.

Tasks implicitly call *exit*( ) if the entry routine specified during task creation returns. Alternatively, a task can explicitly call *exit*( ) at any point to kill itself. A task can kill another task by calling *taskDelete*( ).

When a task is deleted, no other task is notified of this deletion. The routines *taskSafe*( ) and *taskUnsafe*( ) address problems that stem from unexpected deletion of tasks. The routine *taskSafe*( ) protects a task from deletion by other tasks. This protection is often needed when a task executes in a critical region or engages a critical resource.

For example, a task might take a semaphore for exclusive access to some data structure. While executing inside the critical region, the task might be deleted by another task. Because the task is unable to complete the critical region, the data structure might be left in a corrupt or inconsistent state. Furthermore, because the semaphore can never be released by the task, the critical resource is now unavailable for use by any other task and is essentially frozen.

Using *taskSafe*( ) to protect the task that took the semaphore prevents such an outcome. Any task that tries to delete a task protected with *taskSafe*( ) is blocked. When finished with its critical resource, the protected task can make itself available

for deletion by calling ***taskUnsafe( )***, which readies any deleting task. To support nested deletion-safe regions, a count is kept of the number of times ***taskSafe( )*** and ***taskUnsafe( )*** are called. Deletion is allowed only when the count is zero, that is, there are as many "unsafes" as "safes." Protection operates only on the calling task. A task cannot make another task safe or unsafe from deletion.

The following code fragment shows how to use ***taskSafe( )*** and ***taskUnsafe( )*** to protect a critical region of code:

```
taskSafe ();
semTake (semId, WAIT_FOREVER);  /* Block until semaphore available */
.
.   critical region
.
semGive (semId);                    /* Release semaphore */
taskUnsafe ();
```

Deletion safety is often coupled closely with mutual exclusion, as in this example. For convenience and efficiency, a special kind of semaphore, the *mutual-exclusion semaphore*, offers an option for deletion safety. For more information, see *Mutual-Exclusion Semaphores*, p.62.

Task Control

The routines listed in Table 2-9 provide direct control over a task's execution.

Table 2-9 **Task Control Routines**

Call	Description
taskSuspend( )	Suspend a task.
taskResume( )	Resume a task.
taskRestart( )	Restart a task.
taskDelay( )	Delay a task; delay units are ticks.
nanosleep( )	Delay a task; delay units are nanoseconds.

VxWorks debugging facilities require routines for suspending and resuming a task. They are used to freeze a task's state for examination.

Tasks may require restarting during execution in response to some catastrophic error. The restart mechanism, ***taskRestart( )***, recreates a task with the original creation arguments. The Tornado shell also uses this mechanism to restart itself in response to a task-abort request; for information, see the *Tornado User's Guide: Shell*.

Delay operations provide a simple mechanism for a task to sleep for a fixed duration. Task delays are often used for polling applications. For example, to delay a task for half a second without making assumptions about the clock rate, call:

```
taskDelay (sysClkRateGet ( ) / 2);
```

The routine ***sysClkRateGet( )*** returns the speed of the system clock in ticks per second. Instead of ***taskDelay( )***, you can use the POSIX routine ***nanosleep( )*** to specify a delay directly in time units. Only the units are different; the resolution of both delay routines is the same, and depends on the system clock. For details, see *2.7 POSIX Clocks and Timers*, p.100.

As a side effect, ***taskDelay( )*** moves the calling task to the end of the ready queue for tasks of the same priority. In particular, you can yield the CPU to any other tasks of the same priority by "delaying" for zero clock ticks:

```
taskDelay (NO_WAIT); /* allow other tasks of same priority to run */
```

A "delay" of zero duration is only possible with ***taskDelay( )***; ***nanosleep( )*** considers it an error.

2.3.5 Tasking Extensions

To allow additional task-related facilities to be added to the system without modifying the kernel, *wind* provides *task create, switch,* and *delete hooks*, which allow additional routines to be invoked whenever a task is created, a task context switch occurs, or a task is deleted. There are spare fields in the task control block (TCB) available for application extension of a task's context. These hook routines are listed in Table 2-10; for more information, see the reference entry for **taskHookLib**.

Table 2-10 **Task Create, Switch, and Delete Hooks**

Call	Description
taskCreateHookAdd( )	Add a routine to be called at every task create.
taskCreateHookDelete( )	Delete a previously added task create routine.
taskSwitchHookAdd( )	Add a routine to be called at every task switch.
taskSwitchHookDelete( )	Delete a previously added task switch routine.
taskDeleteHookAdd( )	Add a routine to be called at every task delete.
taskDeleteHookDelete( )	Delete a previously added task delete routine.

User-installed switch hooks are called within the kernel context. Thus, switch hooks do not have access to all VxWorks facilities. Table 2-11 summarizes the routines that can be called from a task switch hook; in general, any routine that does not involve the kernel can be called.

Table 2-11 **Routines that Can Be Called by Task Switch Hooks**

Library	Routines
bLib	All routines
fppArchLib	*fppSave( ), fppRestore( )*
intLib	*intContext( ), intCount( ), intVecSet( ), intVecGet( ), intLock( ), intUnlock( )*
lstLib	All routines except *lstFree( )*
mathALib	All are callable if *fppSave( )/fppRestore( )* are used
rngLib	All routines except *rngCreate( )* and *roundlet( )*
taskLib	*taskIdVerify( ), taskIdDefault( ), taskIsReady( ), taskIsSuspended( ), taskTcb( )*
vxLib	*vxTas( )*

2.3.6 POSIX Scheduling Interface

The POSIX 1003.1b scheduling routines, provided by **schedPxLib**, are shown in Table 2-12. These routines let you use a portable interface to get and set task priority, get the scheduling policy, get the maximum and minimum priority for tasks, and if round-robin scheduling is in effect, get the length of a time slice. To understand how to use the routines in this alternative interface, be aware of the minor differences between the POSIX and Wind methods of scheduling.

Differences Between POSIX and Wind Scheduling

POSIX and Wind scheduling routines differ in the following ways:

- POSIX scheduling is based on *processes*, while Wind scheduling is based on *tasks*. Tasks and processes differ in several ways. Most notably, tasks can address memory directly while processes cannot; and processes inherit only some specific attributes from their parent process, while tasks operate in exactly the same environment as the parent task.

Tasks and processes are alike in that they can be scheduled independently.

- VxWorks documentation uses the term *preemptive priority* scheduling, while the POSIX standard uses the term *FIFO*. This difference is purely one of nomenclature: both describe the same priority-based policy.
- The POSIX scheduling algorithms are applied on a process-by-process basis. The Wind methodology, on the other hand, applies scheduling algorithms on a system-wide basis—either all tasks use a round-robin scheme, or all use a preemptive priority scheme.
- The POSIX priority numbering scheme is the inverse of the Wind scheme. In POSIX, the higher the number, the higher the priority; in the Wind scheme, the *lower* the number, the higher the priority, where 0 is the highest priority. Accordingly, the priority numbers used with the POSIX scheduling library (**schedPxLib**) do not match those used and reported by all other components of VxWorks. You can override this default by setting the global variable **posixPriorityNumbering** to FALSE. If you do this, the Wind numbering scheme (smaller number = higher priority) is used by **schedPxLib**, and its priority numbers match those used by the other components of VxWorks.

The POSIX scheduling routines are included when **INCLUDE_POSIX_SCHED** is defined in **configAll.h**; see *8. Configuration* for information on configuring VxWorks.

Table 2-12 **POSIX Scheduling Calls**

Call	Description
sched_setparam( )	Set a task's priority.
sched_getparam( )	Get the scheduling parameters for a specified task.
sched_setscheduler( )	Set scheduling policy and parameters for a task.
sched_yield( )	Relinquish the CPU.
sched_getscheduler( )	Get the current scheduling policy.
sched_get_priority_max( )	Get the maximum priority.
sched_get_priority_min( )	Get the minimum priority.
sched_rr_get_interval( )	If round-robin scheduling, get the time slice length.

Getting and Setting POSIX Task Priorities

The routines *sched_setparam( )* and *sched_getparam( )* set and get a task's priority, respectively. Both routines take a task ID and a **sched_param** structure (defined in **h/sched.h**). A task ID of 0 sets or gets the priority for the calling task. The **sched_priority** member of the **sched_param** structure specifies the new task priority when *sched_setparam( )* is called. The routine *sched_getparam( )* fills in the **sched_priority** with the specified task's current priority.

Example 2-1 **Getting and Setting POSIX Task Priorities**

```
/* This example sets the calling task's priority to 150, then verifies
 * that priority. To run from the shell, spawn as a task:
 *   -> sp priorityTest
 */

/* includes */
#include "vxWorks.h"
#include "sched.h"

/* defines */
#define PX_NEW_PRIORITY 150

STATUS priorityTest (void)
    {
    struct sched_param myParam;

    /* initialize param structure to desired priority */
    myParam.sched_priority = PX_NEW_PRIORITY;
    if (sched_setparam (0, &myParam) == ERROR)
        {
        printf ("error setting priority\n");
        return (ERROR);
        }

    /* demonstrate getting a task priority as a sanity check; ensure it
     * is the same value that we just set.
     */

    if (sched_getparam (0, &myParam) == ERROR)
        {
        printf ("error getting priority\n");
        return (ERROR);
        }

    if (myParam.sched_priority != PX_NEW_PRIORITY)
        {
        printf ("error - priorities do not match\n");
        return (ERROR);
        }
    else
        printf ("task priority = %d\n", myParam.sched_priority);

    return (OK);
    }
```

The routine *sched_setscheduler( )* is designed to set both scheduling policy and priority for a single POSIX process (which corresponds in most other cases to a single Wind task). In the VxWorks kernel, *sched_setscheduler( )* controls only task priority, because the kernel does not allow tasks to have scheduling policies that differ from one another. If its policy specification matches the current system-wide scheduling policy, *sched_setscheduler( )* sets only the priority, thus acting like *sched_setparam( )*. If its policy specification does not match the current one, *sched_setscheduler( )* returns an error.

The only way to change the scheduling policy is to change it for all tasks; there is no POSIX routine for this purpose. To set a system-wide scheduling policy, use the Wind function *kernelTimeSlice( )* described in *Round-Robin Scheduling*, p.33.

Getting and Displaying the Current Scheduling Policy

The POSIX routine *sched_getscheduler( )* returns the current scheduling policy. There are two valid scheduling policies in VxWorks: preemptive priority scheduling (in POSIX terms, **SCHED_FIFO**) and round-robin scheduling by priority (**SCHED_RR**).

Example 2-2 **Getting POSIX Scheduling Policy**

```
/* This example gets the scheduling policy and displays it. */

/* includes */

#include "vxWorks.h"
#include "sched.h"

STATUS schedulerTest (void)
    {
    int policy;

    if ((policy = sched_getscheduler (0)) == ERROR)
        {
        printf ("getting scheduler failed\n");
        return (ERROR);
        }

    /* sched_getscheduler returns either SCHED_FIFO or SCHED_RR */

    if (policy == SCHED_FIFO)
        printf ("current scheduling policy is FIFO\n");
    else
        printf ("current scheduling policy is round robin\n");

    return (OK);
    }
```

Getting Scheduling Parameters: Priority Limits and Time Slice

The routines *sched_get_priority_max( )* and *sched_get_priority_min( )* return the maximum and minimum possible POSIX priority values, respectively.

If round-robin scheduling is enabled, you can use *sched_rr_get_interval( )* to determine the length of the current time-slice interval. This routine takes as an argument a pointer to a **timespec** structure (defined in **time.h**), and writes the number of seconds and nanoseconds per time slice to the appropriate elements of that structure.

Example 2-3 **Getting the POSIX Round-Robin Time Slice**

```
/* The following example checks that round-robin scheduling is enabled,
 * gets the length of the time slice, and then displays the time slice.
 */

/* includes */

#include "vxWorks.h"
#include "sched.h"

STATUS rrgetintervalTest (void)
    {
    struct timespec slice;

    /* turn on round robin */

    kernelTimeSlice (30);

    if (sched_rr_get_interval (0, &slice) == ERROR)
        {
        printf ("get-interval test failed\n");
        return (ERROR);
        }

    printf ("time slice is %l seconds and %l nanoseconds\n",
            slice.tv_sec, slice.tv_nsec);
    return (OK);
    }
```

2.3.7 *Task Error Status:* errno

By convention, C library functions set a single global integer variable **errno** to an appropriate error number whenever the function encounters an error. This convention is specified as part of the ANSI C standard.

Layered Definitions of errno

In VxWorks, **errno** is simultaneously defined in two different ways. There is, as in ANSI C, an underlying global variable called **errno**, which you can display by name using Tornado development tools; see the *Tornado User's Guide*. However, **errno** is also defined as a macro in **errno.h**; this is the definition visible to all of VxWorks except for one function. The macro is defined as a call to a function ***__errno( )*** that returns the address of the global variable, **errno** (as you might guess, this is the single function that does not itself use the macro definition for **errno**). This subterfuge yields a useful feature: because ***__errno( )*** is a function, you can place breakpoints on it while debugging, to determine where a particular error occurs. Nevertheless, because the result of the macro **errno** is the address of the global variable **errno**, C programs can set the value of **errno** in the standard way:

```
errno = someErrorNumber;
```

As with any other **errno** implementation, take care not to have a local variable of the same name.

A Separate errno *Value for Each Task*

In VxWorks, the underlying global **errno** is a single predefined global variable that can be referenced directly by application code that is linked with VxWorks (either statically on the host or dynamically at load time). However, for **errno** to be useful in the multitasking environment of VxWorks, each task must see its own version of **errno**. Therefore **errno** is saved and restored by the kernel as part of each task's context every time a context switch occurs. Similarly, *interrupt service routines (ISRs)* see their own versions of **errno**.

This is accomplished by saving and restoring **errno** on the interrupt stack as part of the interrupt enter and exit code provided automatically by the kernel (see *2.5.1 Connecting Application Code to Interrupts*, p.94). Thus, regardless of the VxWorks context, an error code can be stored or consulted with direct manipulation of the global variable **errno**.

Error Return Convention

Almost all VxWorks functions follow a convention that indicates simple success or failure of their operation by the actual return value of the function. Many functions return only the status values **OK** (0) or **ERROR** (-1). Some functions that normally return a nonnegative number (for example, ***open( )*** returns a file descriptor) also

return **ERROR** to indicate an error. Functions that return a pointer usually return **NULL** (0) to indicate an error. In most cases, a function returning such an error indication also sets **errno** to the specific error code.

The global variable **errno** is never cleared by VxWorks routines. Thus, its value always indicates the last error status set. When a VxWorks subroutine gets an error indication from a call to another routine, it usually returns its own error indication without modifying **errno**. Thus, the value of **errno** that is set in the lower-level routine remains available as the indication of error type.

For example, the VxWorks routine ***intConnect( )***, which connects a user routine to a hardware interrupt, allocates memory by calling ***malloc( )*** and builds the interrupt driver in this allocated memory. If ***malloc( )*** fails because insufficient memory remains in the pool, it sets **errno** to a code indicating an insufficient-memory error was encountered in the memory allocation library, **memLib**. The ***malloc( )*** routine then returns **NULL** to indicate the failure. The ***intConnect( )*** routine, receiving the **NULL** from ***malloc( )***, then returns its own error indication of **ERROR**. However, it does not alter **errno**, leaving it at the "insufficient memory" code set by ***malloc( )***. For example:

```
if ((pNew = malloc (CHUNK_SIZE)) == NULL)
        return (ERROR);
```

We recommend that you use this mechanism in your own subroutines, setting and examining **errno** as a debugging technique. A string constant associated with **errno** can be displayed using ***printErrno( )*** if the **errno** value has a corresponding string entered in the error-status symbol table, **statSymTbl**. See the reference entry **errnoLib** for details on error-status values and building **statSymTbl**.

Assignment of Error Status Values

VxWorks **errno** values encode the module that issues an error, in the most significant two bytes, and use the least significant two bytes for individual error numbers. All VxWorks module numbers are in the range 1–500; **errno** values with a "module" number of zero are used for source compatibility.

All other **errno** values (that is, positive values greater than or equal to **501<<16**, and all negative values) are available for application use.

See the reference entry on **errnoLib** for more information about defining and decoding **errno** values with this convention.

2.3.8 Task Exception Handling

Errors in program code or data can cause hardware exception conditions such as illegal instructions, bus or address errors, divide by zero, and so forth. The VxWorks exception handling package takes care of all such exceptions. The default exception handler suspends the task that caused the exception, and saves the state of the task at the point of the exception. The kernel and other tasks continue uninterrupted. A description of the exception is transmitted to the Tornado development tools, which can be used to examine the suspended task; see the *Tornado User's Guide: Shell* for details.

Tasks can also attach their own handlers for certain hardware exceptions through the *signal* facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. Signals are also used for signaling software exceptions as well as hardware exceptions. They are described in more detail in 2.4.7 *Signals*, p.90 and in the reference entry for **sigLib**.

2.3.9 Shared Code and Reentrancy

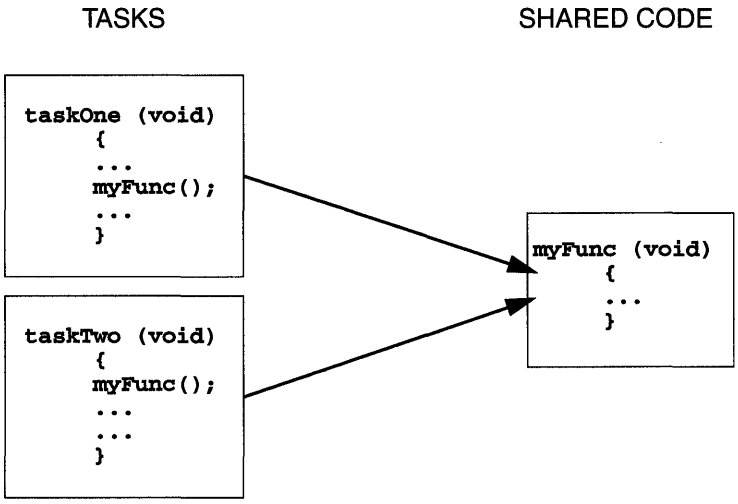
In VxWorks, it is common for a single copy of a subroutine or subroutine library to be invoked by many different tasks. For example, many tasks may call ***printf*( )**, but there is only a single copy of the subroutine in the system. A single copy of code executed by multiple tasks is called *shared code*. VxWorks dynamic linking facilities make this particularly easy. Shared code also makes the system more efficient and easier to maintain; see Figure 2-4.

Shared code must be *reentrant*. A subroutine is reentrant if a single copy of the routine can be called from several task contexts simultaneously without conflict. Such conflict typically occurs when a subroutine modifies global or static variables, because there is only a single copy of the data and code. A routine's references to such variables can overlap and interfere in invocations from different task contexts.

Most routines in VxWorks are reentrant. However, all routines which have a corresponding *name_r*( ) routine should be assumed non-reentrant. For example, because ***ldiv*( )** has a corresponding routine ***ldiv_r*( )**, you can assume that ***ldiv*( )** is not reentrant.

VxWorks I/O and driver routines are reentrant, but require careful application design. For buffered I/O, we recommend using file-pointer buffers on a per-task basis. At the driver level, it is possible to load buffers with streams from different tasks, due to the global file descriptor table in VxWorks. This may or may not be desirable, depending on the nature of the application. For example, a packet driver

Figure 2-4 **Shared Code**



can mix streams from different tasks because the packet header identifies the destination of each packet.

The majority of VxWorks routines use the following reentrancy techniques:

- dynamic stack variables
- global and static variables guarded by semaphores
- task variables

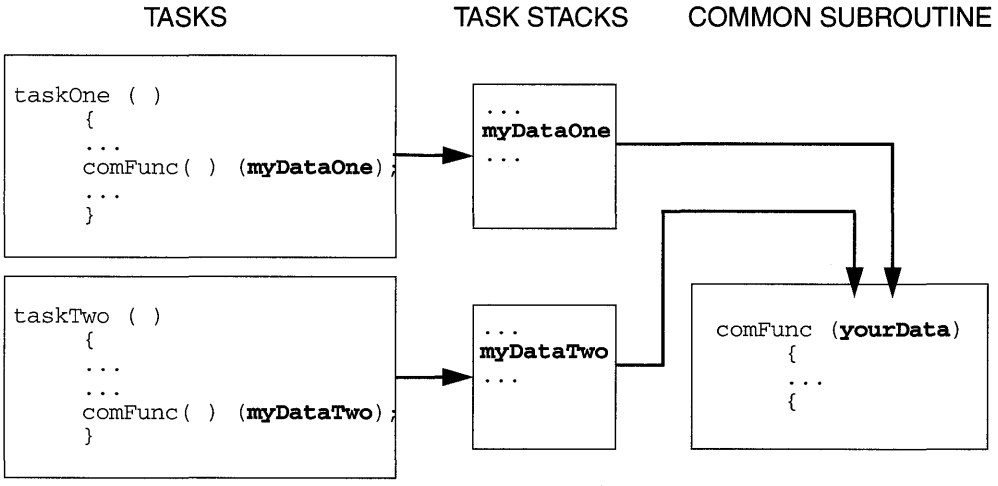
We recommend applying these same techniques when writing application code that can be called from several task contexts simultaneously.

Dynamic Stack Variables

Many subroutines are *pure* code, having no data of their own except dynamic stack variables. They work exclusively on data provided by the caller as parameters. The linked-list library, **lstLib**, is a good example of this. Its routines operate on lists and nodes provided by the caller in each subroutine call.

Subroutines of this kind are inherently reentrant. Multiple tasks can use such routines simultaneously without interfering with each other, because each task does indeed have its own stack. See Figure 2-5.

Figure 2-5 **Stack Variables and Shared Code**



Guarded Global and Static Variables

Some libraries encapsulate access to common data. One example is the memory allocation library, **memLib**, which manages pools of memory to be used by many tasks. This library declares and uses its own static data variables to keep track of pool allocation.

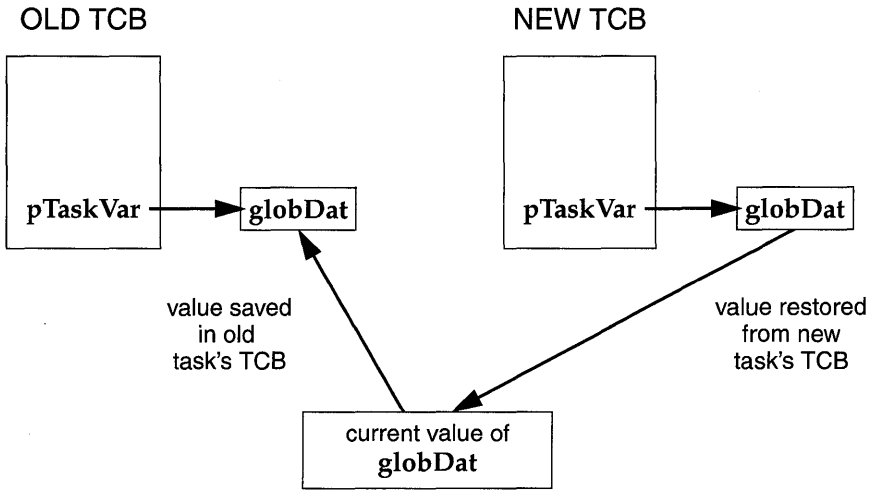
This kind of library requires some caution because the routines are not inherently reentrant. Multiple tasks simultaneously invoking the routines in the library might interfere with access to common variables. Such libraries must be made explicitly reentrant by providing a *mutual-exclusion* mechanism to prohibit tasks from simultaneously executing critical sections of code. The usual mutual-exclusion mechanism is the semaphore facility provided by **semLib** and described in *2.4.3 Semaphores*, p.57.

Task Variables

Some routines that can be called by multiple tasks simultaneously may require global or static variables with a distinct value for each calling task. For example, several tasks may reference a private buffer of memory and yet refer to it with the same global variable.

To accommodate this, VxWorks provides a facility called *task variables* that allows 4-byte variables to be added to a task's context, so that the value of such a variable is switched every time a task switch occurs to or from its owner task. Typically, several tasks declare the same variable (4-byte memory location) as a task variable. Each of those tasks can then treat that single memory location as its own private variable; see Figure 2-6. This facility is provided by the routines ***taskVarAdd( )***, ***taskVarDelete( )***, ***taskVarSet( )***, and ***taskVarGet( )***, which are described in the reference entry for **taskVarLib**.

Figure 2-6 **Task Variables and Context Switches**



Use this mechanism sparingly. Each task variable adds a few microseconds to the context switching time for its task, because the value of the variable must be saved and restored as part of the task's context. Consider collecting all of a module's task variables into a single dynamically allocated structure, and then making all accesses to that structure indirectly through a single pointer. This pointer can then be the task variable for all tasks using that module.

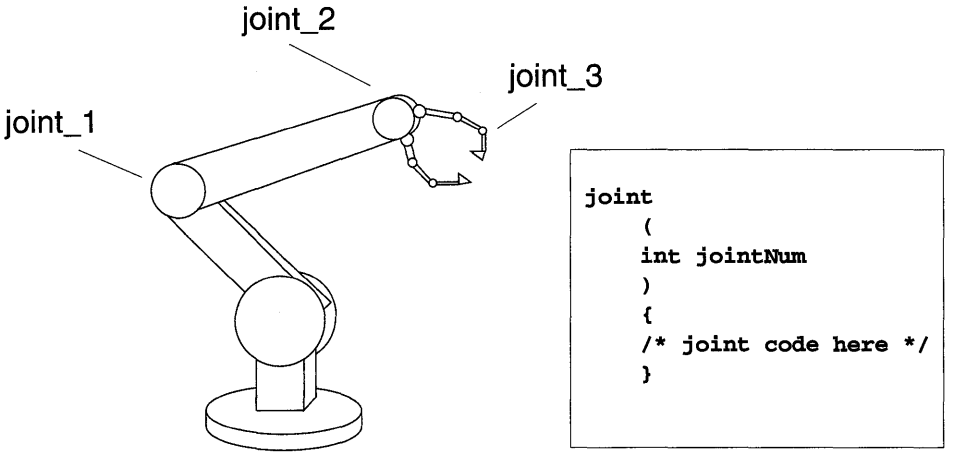
Multiple Tasks with the Same Main Routine

With VxWorks, it is possible to spawn several tasks with the same main routine. Each spawn creates a new task with its own stack and context. Each spawn can also pass the main routine different parameters to the new task. In this case, the same rules of reentrancy described in *Task Variables*, p.50 apply to the entire task.

This is useful when the same function needs to be performed concurrently with different sets of parameters. For example, a routine that monitors a particular kind of equipment might be spawned several times to monitor several different pieces of that equipment. The arguments to the main routine could indicate which particular piece of equipment the task is to monitor.

In Figure 2-7, multiple joints of the mechanical arm use the same code. The tasks manipulating the joints invoke ***joint*( )**. The joint number (**jointNum**) is used to indicate which joint on the arm to manipulate.

Figure 2-7 **Multiple Tasks Utilizing Same Code**



2.3.10 VxWorks System Tasks

VxWorks includes several system tasks, described in the following sections.

The Root Task: tUsrRoot

The root task, **tUsrRoot**, is the first task executed by the kernel. The entry point of the root task is ***usrRoot*( )** in **config/all/usrConfig.c** and initializes most VxWorks facilities. It spawns such tasks as the logging task, the exception task, the network task, and the **tRlogind** daemon. Normally, the root task terminates and is deleted after all initialization has occurred. You are free to add any necessary initialization to the root task. For more information, see *8.3 Configuring VxWorks*, p.430.

The Logging Task: tLogTask

The log task, **tLogTask**, is used by VxWorks modules to log system messages without having to perform I/O in the current task context. For more information, see *3.5.3 Message Logging*, p.122 and the reference entry for **logLib**.

The Exception Task: tExcTask

The exception task, **tExcTask**, supports the VxWorks exception handling package by performing functions that cannot occur at interrupt level. It must have the highest priority in the system. Do not suspend, delete, or change the priority of this task. For more information, see the reference entry for **excLib**.

The Network Task: tNetTask

The **tNetTask** daemon handles the task-level functions required by the VxWorks network.

The Target Agent Task: tWdbTask

The target agent task, **tWdbTask**, is created if the target agent is set to run in task mode; see *8.4.1 Scaling Down VxWorks*, p.447. It services requests from the Tornado target server; for information on this server, see the *Tornado User's Guide: Overview.*

Tasks for Optional Components

The following VxWorks system tasks are created if their associated configuration constants are defined; for more information, see *8.3 Configuring VxWorks*, p.430.

tShell If you have included the target shell in the VxWorks configuration, it is spawned as this task. Any routine or task that is invoked from the target shell, rather than spawned, runs in the **tShell** context. For more information, see *9. Target Shell.*

tRlogind If you have included the target shell and the **rlogin** facility in the VxWorks configuration, this daemon allows remote users to log in to VxWorks. It accepts a remote login request from another VxWorks or host system and spawns **tRlogInTask** and **tRlogOutTask**. These tasks exist as long as the remote user is logged on. During the remote

session, the shell's (and any other task's) input and output are redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**. For more information, see *3.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)*, p.131 and the reference entry for **ptyDrv**.

tTelnetd If you have included the target shell and the **telnet** facility in the VxWorks configuration, this daemon allows remote users to log in to VxWorks with **telnet**. It accepts a remote login request from another VxWorks or host system and spawns the input task **tTelnetInTask** and output task **tTelnetOutTask**. These tasks exist as long as the remote user is logged on. During the remote session, the shell's (and any other task's) input and output are redirected to the remote user. A *tty*-like interface is provided to the remote user through the use of the VxWorks pseudo-terminal driver, **ptyDrv**. See *3.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)*, p.131 and the reference entry for **ptyDrv** for further explanation.

tPortmapd If you have included the RPC facility in the VxWorks configuration, this daemon is an RPC server that acts as a central registrar for RPC servers running on the same machine. RPC clients query the **tPortmapd** daemon to find out how to contact the various servers.

tRdbTask If you have included the RDB facility in the VxWorks configuration, this daemon services requests made by remote source-level debuggers. The RDB modules fill a role roughly analogous to that of the target agent, except that the RDB connection relies on VxWorks facilities, such as the target-resident symbol table and the target-resident dynamic linker. For more information on remote debugging, see the *Tornado User's Guide: Debugger.*

2.4 Intertask Communications

The complement to the multitasking routines described in the *2.3 Tasks*, p.30 is the intertask communication facilities. These facilities permit independent tasks to coordinate their actions.

VxWorks supplies a rich set of intertask communication mechanisms, including:

- *Shared memory,* for simple sharing of data.

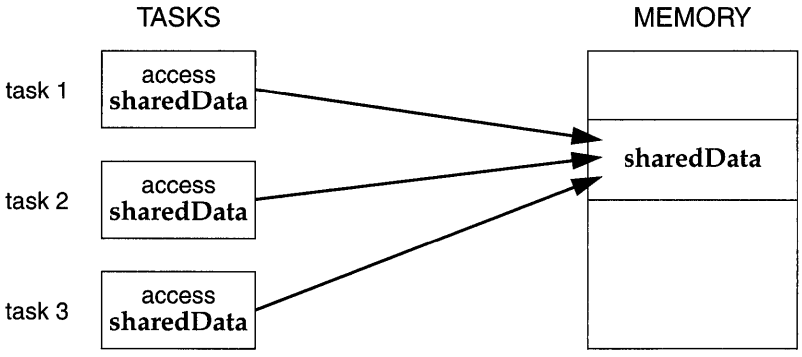
- *Semaphores*, for basic mutual exclusion and synchronization.
- *Message queues* and *pipes*, for intertask message passing within a CPU.
- *Sockets* and *remote procedure calls*, for network-transparent intertask communication.
- *Signals*, for exception handling.

The optional product, VxMP, provides intertask communication over the backplane for tasks running on different CPUs. This includes shared semaphores, shared message queues, shared memory, and the shared name database.

2.4.1 Shared Data Structures

The most obvious way for tasks to communicate is by accessing shared data structures. Because all tasks in VxWorks exist in a single linear address space, sharing data structures between tasks is trivial; see Figure 2-8. Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced directly by code running in different contexts.

Figure 2-8 **Shared Data Structures**



2.4.2 Mutual Exclusion

While a shared address space simplifies exchange of data, interlocking access to memory is crucial to avoid contention. Many methods exist for obtaining exclusive access to resources, and vary only in the scope of the exclusion. Such methods include disabling interrupts, disabling preemption, and resource locking with semaphores.

Interrupt Locks and Latency

The most powerful method available for mutual exclusion is the disabling of interrupts. Such a lock guarantees exclusive access to the CPU:

```
funcA ()
    {
    int lock = intLock();
    .
    .   critical region that cannot be interrupted
    .
    intUnlock (lock);
    }
```

While this solves problems involving mutual exclusion with ISRs, it is inappropriate as a general-purpose mutual-exclusion method for most real-time systems, because it prevents the system from responding to external events for the duration of these locks. Interrupt latency is unacceptable whenever an immediate response to an external event is required. However, interrupt locking can sometimes be necessary where mutual exclusion involves ISRs. In any situation, keep the duration of interrupt lockouts short.

Preemptive Locks and Latency

Disabling preemption offers a somewhat less restrictive form of mutual exclusion. While no other task is allowed to preempt the current executing task, ISRs are able to execute:

```
funcA ()
    {
    taskLock ();
    .
    .   critical region that cannot be interrupted
    .
    taskUnlock ();
    }
```

However, this method can lead to unacceptable real-time response. Tasks of higher priority are unable to execute until the locking task leaves the critical region, even though the higher-priority task is not itself involved with the critical region. While this kind of mutual exclusion is simple, if you use it, make sure to keep the duration short. A better mechanism is provided by semaphores, discussed in *2.4.3 Semaphores*, p.57.

2.4.3 Semaphores

VxWorks semaphores are highly optimized and provide the fastest intertask communication mechanism in VxWorks. Semaphores are the primary means for addressing the requirements of both mutual exclusion and task synchronization:

- For *mutual exclusion*, semaphores interlock access to shared resources. They provide mutual exclusion with finer granularity than either interrupt disabling or preemptive locks, discussed in *2.4.2 Mutual Exclusion*, p.55.
- For *synchronization*, semaphores coordinate a task's execution with external events.

There are three types of Wind semaphores, optimized to address different classes of problems:

binary	The fastest, most general-purpose semaphore. Optimized for synchronization or mutual exclusion.
mutual exclusion	A special binary semaphore optimized for problems inherent in mutual exclusion: priority inheritance, deletion safety, and recursion.
counting	Like the binary semaphore, but keeps track of the number of times a semaphore is given. Optimized for guarding multiple instances of a resource.

VxWorks provides not only the Wind semaphores, designed expressly for VxWorks, but also POSIX semaphores, designed for portability. An alternate semaphore library provides the POSIX-compatible semaphore interface; see *POSIX Semaphores*, p.67.

The semaphores described here are for use on a single CPU. The optional product VxMP provides semaphores that can be used across processors; see *6. Shared-Memory Objects*.

Semaphore Control

Instead of defining a full set of semaphore control routines for each type of semaphore, the Wind semaphores provide a single uniform interface for semaphore control. Only the creation routines are specific to the semaphore type. Table 2-13 lists the semaphore control routines.

The ***semBCreate( )***, ***semMCreate( )***, and ***semCCreate( )*** routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by

Table 2-13 **Semaphore Control Routines**

Call	Description
semBCreate( )	Allocate and initialize a binary semaphore.
semMCreate( )	Allocate and initialize a mutual-exclusion semaphore.
semCCreate( )	Allocate and initialize a counting semaphore.
semDelete( )	Terminate and free a semaphore.
semTake( )	Take a semaphore.
semGive( )	Give a semaphore.
semFlush( )	Unblock all tasks that are waiting for a semaphore.

the other semaphore-control routines. When a semaphore is created, the queue type is specified. Tasks pending on a semaphore can be queued in priority order (**SEM_Q_PRIORITY**) or in first-in first-out order (**SEM_Q_FIFO**).

WARNING: The ***semDelete( )*** call terminates a semaphore and deallocates any associated memory. Take care when deleting semaphores, particularly those used for mutual exclusion, to avoid deleting a semaphore that another task still requires. Do not delete a semaphore unless the same task first succeeds in taking it.

Binary Semaphores

The general-purpose binary semaphore is capable of addressing the requirements of both forms of task coordination: mutual exclusion and synchronization. The binary semaphore has the least overhead associated with it, making it particularly applicable to high-performance requirements. The mutual-exclusion semaphore described in *Mutual-Exclusion Semaphores*, p.62 is also a binary semaphore, but it has been optimized to address problems inherent to mutual exclusion. Alternatively, the binary semaphore can be used for mutual exclusion if the advanced features of the mutual-exclusion semaphore are deemed unnecessary.

A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with ***semTake( )***, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 2-9. If the semaphore is available (full), the semaphore becomes unavailable (empty) and the task continues executing immediately. If the semaphore is unavailable (empty), the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

Figure 2-9 **Taking a Semaphore**

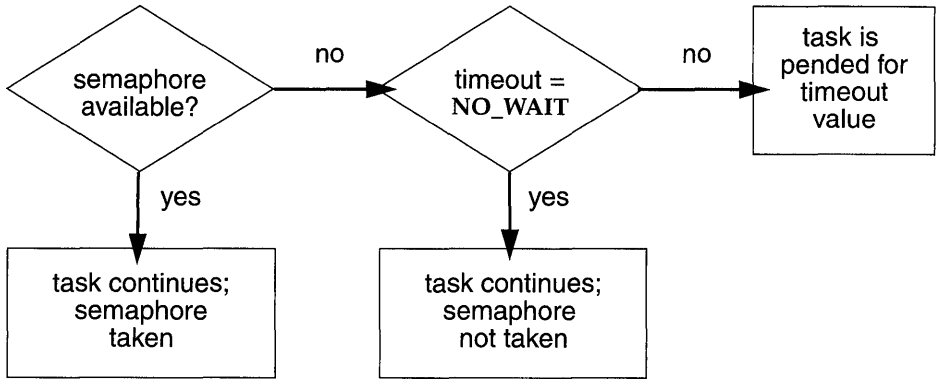
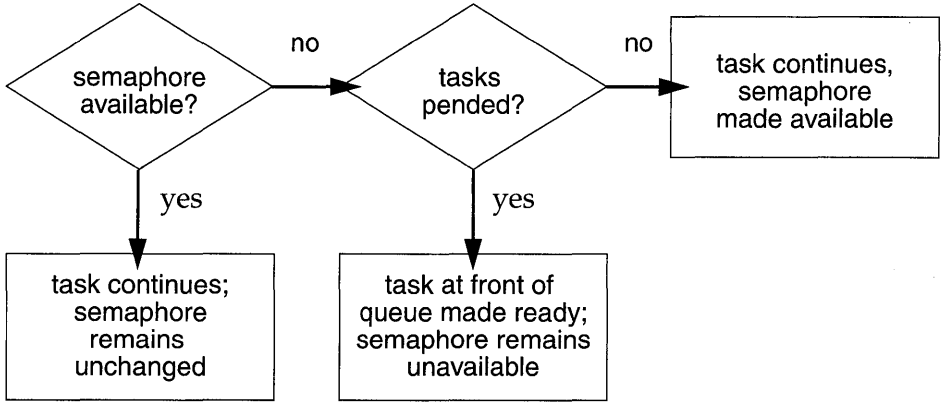


Figure 2-10 **Giving a Semaphore**



When a task gives a binary semaphore, using ***semGive( )***, the outcome also depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 2-10. If the semaphore is already available (full), giving the semaphore has no effect at all. If the semaphore is unavailable (empty) and no task is waiting to take it, then the semaphore becomes available (full). If the semaphore is unavailable (empty) and one or more tasks are pending on its availability, then the first task in the queue of blocked tasks is unblocked, and the semaphore is left unavailable (empty).

Mutual Exclusion

Binary semaphores interlock access to a shared resource efficiently. Unlike disabling interrupts or preemptive locks, binary semaphores limit the scope of the mutual exclusion to only the associated resource. In this technique, a semaphore is created to guard the resource. Initially the semaphore is available (full).

```
/* includes */
#include "vxWorks.h"
#include "semLib.h"

SEM_ID semMutex;

/* Create a binary semaphore that is initially full. Tasks *
 * blocked on semaphore wait in priority order.            */

semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

When a task wants to access the resource, it must first take that semaphore. As long as the task keeps the semaphore, all other tasks seeking access to the resource are blocked from execution. When the task is finished with the resource, it gives back the semaphore, allowing another task to use the resource.

Thus all accesses to a resource requiring mutual exclusion are bracketed with *semTake*( ) and *semGive*( ) pairs:

```
semTake (semMutex, WAIT_FOREVER);
.
.  critical region, only accessible by a single task at a time
.
semGive (semMutex);
```

Synchronization

When used for task synchronization, a semaphore can represent a condition or event that a task is waiting for. Initially the semaphore is unavailable (empty). A task or ISR signals the occurrence of the event by giving the semaphore (see *2.5 Interrupt Service Code*, p.93 for a complete discussion of ISRs). Another task waits for the semaphore by calling *semTake*( ). The waiting task blocks until the event occurs and the semaphore is given.

Note the difference in sequence between semaphores used for mutual exclusion and those used for synchronization. For mutual exclusion, the semaphore is initially full, and each task first takes, then gives back the semaphore. For synchronization, the semaphore is initially empty, and one task waits to take the semaphore given by another task.

In Example 2-4, the *init*( ) routine creates the binary semaphore, attaches an ISR to an event, and spawns a task to process the event. The routine *task1*( ) runs until it

calls ***semTake( )***. It remains blocked at that point until an event causes the ISR to call ***semGive( )***. When the ISR completes, ***task1( )*** executes to process the event. There is an advantage of handling event processing within the context of a dedicated task: less processing takes place at interrupt level, thereby reducing interrupt latency. This model of event processing is recommended for real-time applications.

Example 2-4 **Using Semaphores for Task Synchronization**

```
/* This example shows the use of semaphores for task synchronization. */

/* includes */
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* replace arch with architecture type */

SEM_ID syncSem;         /* ID of sync semaphore */

init (
    int someIntNum
    )
    {
    /* connect interrupt service routine */
    intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);

    /* create semaphore */
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    /* spawn task used for synchronization. */
    taskSpawn ("sample", 100, 0, 20000, task1, 0,0,0,0,0,0,0,0,0,0);
    }

task1 (void)
    {
    ...
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    printf ("task 1 got the semaphore\n");
    ... /* process event */
    }

eventInterruptSvcRout (void)
    {
    ...
    semGive (syncSem);    /* let task 1 process event */
    ...
    }
```

Broadcast synchronization allows all processes that are blocked on the same semaphore to be unblocked atomically. Correct application behavior often requires a set of tasks to process an event before any task of the set has the opportunity to process further events. The routine ***semFlush( )*** addresses this class of synchronization problem by unblocking all tasks pended on a semaphore.

Mutual-Exclusion Semaphores

The mutual-exclusion semaphore is a specialized binary semaphore designed to address issues inherent in mutual exclusion, including priority inversion, deletion safety, and recursive access to resources.

The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

- It can be used only for mutual exclusion.
- It can be given only by the task that took it.
- It cannot be given from an ISR.
- The ***semFlush( )*** operation is illegal.

Priority Inversion

Priority inversion arises when a higher-priority task is forced to wait an indefinite period of time for a lower-priority task to complete. Consider the scenario in Figure 2-11: **t1**, **t2**, and **t3** are tasks of high, medium, and low priority, respectively. **t3** has acquired some resource by taking its associated binary guard semaphore. When **t1** preempts **t3** and contends for the resource by taking the same semaphore, it becomes blocked. If we could be assured that **t1** would be blocked no longer than the time it normally takes **t3** to finish with the resource, there would be no problem because the resource cannot be preempted. However, the low-priority task is vulnerable to preemption by medium-priority tasks (like **t2**), which could inhibit **t3** from relinquishing the resource. This condition could persist, blocking **t1** for an indefinite period of time.

The mutual-exclusion semaphore has the option **SEM_INVERSION_SAFE**, which enables a *priority-inheritance* algorithm. The priority-inheritance protocol assures that a task that owns a resource executes at the priority of the highest-priority task blocked on that resource. Once the task priority has been elevated, it remains at the higher level until all mutual-exclusion semaphores that the task owns are released; then the task returns to its normal, or standard, priority. Hence, the "inheriting" task is protected from preemption by any intermediate-priority tasks. This option must be used in conjunction with a priority queue (**SEM_Q_PRIORITY**).

In Figure 2-12, priority inheritance solves the problem of priority inversion by elevating the priority of **t3** to the priority of **t1** during the time **t1** is blocked on the semaphore. This protects **t3**, and indirectly **t1**, from preemption by **t2**.

The following example creates a mutual-exclusion semaphore that uses the priority inheritance algorithm:

```
semId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

Figure 2-11 **Priority Inversion**

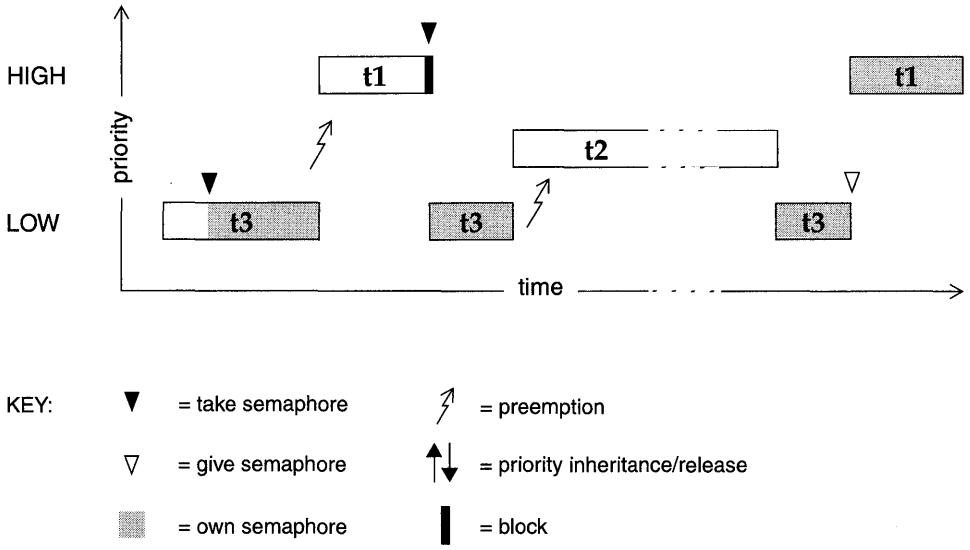
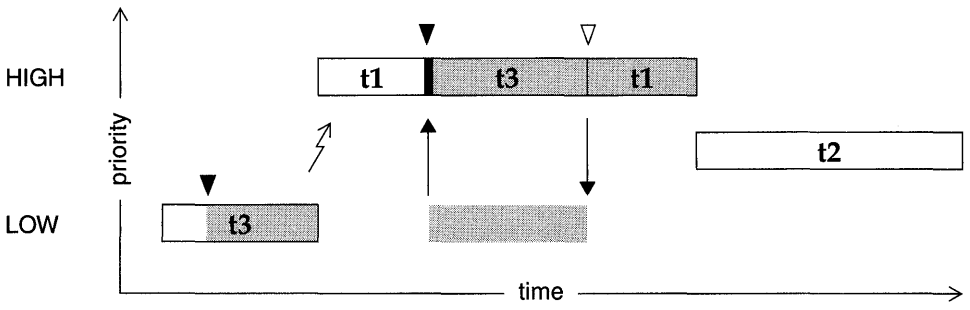


Figure 2-12 **Priority Inheritance**



Deletion Safety

Another problem of mutual exclusion involves task deletion. Within a critical region guarded by semaphores, it is often desirable to protect the executing task from unexpected deletion. Deleting a task executing in a critical region can be catastrophic. The resource might be left in a corrupted state and the semaphore guarding the resource left unavailable, effectively preventing all access to the resource.

The primitives ***taskSafe( )*** and ***taskUnsafe( )*** provide one solution to task deletion. However, the mutual-exclusion semaphore offers the option **SEM_DELETE_SAFE**, which enables an implicit ***taskSafe( )*** with each ***semTake( )***, and a ***taskUnsafe( )*** with each ***semGive( )***. In this way, a task can be protected from deletion while it has the semaphore. This option is more efficient than the primitives ***taskSafe( )*** and ***taskUnsafe( )***, as the resulting code requires fewer entrances to the kernel.

```
semId = semMCreate (SEM_Q_FIFO | SEM_DELETE_SAFE);
```

Recursive Resource Access

Mutual-exclusion semaphores can be taken *recursively*. This means that the semaphore can be taken more than once by the task that owns it before finally being released. Recursion is useful for a set of routines that must call each other but that also require mutually exclusive access to a resource. This is possible because the system keeps track of which task currently owns the mutual-exclusion semaphore.

Before being released, a mutual-exclusion semaphore taken recursively must be *given* the same number of times it is *taken*. This is tracked by a count that increments with each ***semTake( )*** and decrements with each ***semGive( )***.

Example 2-5 **Recursive Use of a Mutual-Exclusion Semaphore**

```
/* Function A requires access to a resource which it acquires by taking
 * mySem; function A may also need to call function B, which also
 * requires mySem:
 */

/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;

/* Create a mutual-exclusion semaphore. */

init ()
    {
    mySem = semMCreate (SEM_Q_PRIORITY);
    }
```

```
funcA ()
    {
    semTake (mySem, WAIT_FOREVER);
    printf ("funcA: Got mutual-exclusion semaphore\n");
    ...
    funcB ();
    ...
    semGive (mySem);
    printf ("funcA: Released mutual-exclusion semaphore\n");
    }
funcB ()
    {
    semTake (mySem, WAIT_FOREVER);
    printf ("funcB: Got mutual-exclusion semaphore\n");
    ...
    semGive (mySem);
    printf ("funcB: Releases mutual-exclusion semaphore\n");
    }
```

Counting Semaphores

Counting semaphores are another means to implement task synchronization and mutual exclusion. The counting semaphore works like the binary semaphore except that it keeps track of the number of times a semaphore is given. Every time a semaphore is given, the count is incremented; every time a semaphore is taken, the count is decremented. When the count reaches zero, a task that tries to take the semaphore is blocked. As with the binary semaphore, if a semaphore is given and a task is blocked, it becomes unblocked. However, unlike the binary semaphore, if a semaphore is given and no tasks are blocked, then the count is incremented. This means that a semaphore that is given twice can be taken twice without blocking. Table 2-14 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Table 2-14 **Counting Semaphore Example**

Semaphore Call	Count after Call	Resulting Behavior
semCCreate( )	3	Semaphore initialized with initial count of 3.
semTake( )	2	Semaphore taken.
semTake( )	1	Semaphore taken.
semTake( )	0	Semaphore taken.
semTake( )	0	Task blocks waiting for semaphore to be available.
semGive( )	0	Task waiting is given semaphore.
semGive( )	1	No task waiting for semaphore; count incremented.

Counting semaphores are useful for guarding multiple copies of resources. For example, the use of five tape drives might be coordinated using a counting semaphore with an initial count of 5, or a ring buffer with 256 entries might be implemented using a counting semaphore with an initial count of 256. The initial count is specified as an argument to the ***semCCreate( )*** routine.

Special Semaphore Options

The uniform Wind semaphore interface includes two special options. These options are not available for the POSIX-compatible semaphores described in *POSIX Semaphores*, p.67.

Timeouts

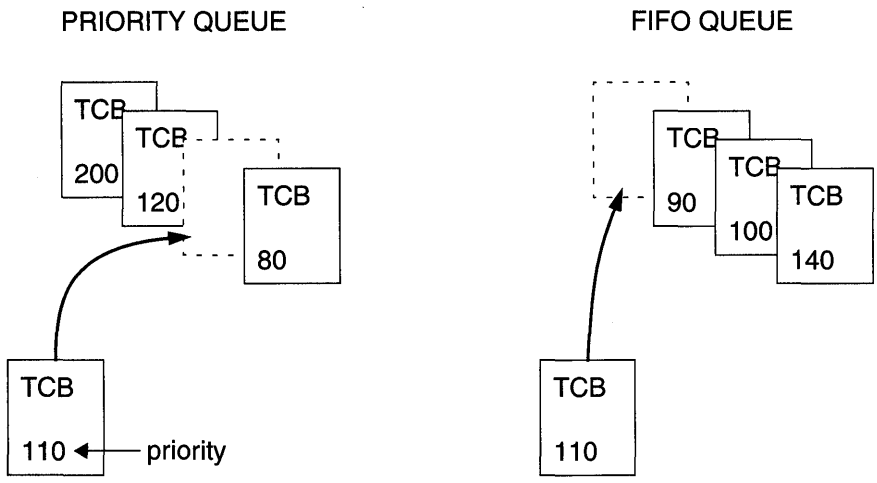
Wind semaphores include the ability to time out from the pended state. This is controlled by a parameter to ***semTake( )*** that specifies the amount of time in ticks that the task is willing to wait in the pended state. If the task succeeds in taking the semaphore within the allotted time, ***semTake( )*** returns **OK**. The **errno** set when a ***semTake( )*** returns **ERROR** due to timing out before successfully taking the semaphore depends upon the timeout value passed. A ***semTake( )*** with **NO_WAIT** (0), which means *do not wait at all*, sets **errno** to **S_objLib_OBJ_UNAVAILABLE**. A ***semTake( )*** with a positive timeout value returns **S_objLib_OBJ_TIMEOUT**. A timeout value of **WAIT_FOREVER** (-1) means *wait indefinitely.*

Queues

Wind semaphores include the ability to select the queuing mechanism employed for tasks blocked on a semaphore. They can be queued based on either of two criteria: first-in first-out (FIFO) order, or priority order; see Figure 2-13.

Priority ordering better preserves the intended priority structure of the system at the expense of some overhead in ***semTake( )*** in sorting the tasks by priority. A FIFO queue requires no priority sorting overhead and leads to constant-time performance. The selection of queue type is specified during semaphore creation with ***semBCreate( )***, ***semMCreate( )***, or ***semCCreate( )***. Semaphores using the priority inheritance option (**SEM_INVERSION_SAFE**) must select priority-order queuing.

Figure 2-13 **Task Queue Types**



POSIX Semaphores

POSIX defines both *named* and *unnamed* semaphores, which have the same properties, but use slightly different interfaces. The POSIX semaphore library provides routines for creating, opening, and destroying both named and unnamed semaphores. The POSIX semaphore routines provided by **semPxLib** are shown in Table 2-15.

With named semaphores, you assign a symbolic name[1] when opening the semaphore; the other named-semaphore routines accept this name as an argument.

The POSIX terms *wait* (or *lock*) and *post* (or *unlock*) correspond to the VxWorks terms *take* and *give*, respectively.

The initialization routine ***semPxLibInit( )*** is called by default when **INCLUDE_POSIX_SEM** is defined in **configAll.h**. The routines ***sem_open( )***, ***sem_unlink( )***, and ***sem_close( )*** are for opening and closing/destroying named

1. Some host operating systems, such as UNIX, require symbolic names for objects that are to be shared among processes. This is because processes do not normally share memory in such operating systems. In VxWorks, there is no requirement for named semaphores, because all objects are located within a single address space, and reference to shared objects by memory location is standard practice.

semaphores only; ***sem_init( )*** and ***sem_destroy( )*** are for initializing and destroying unnamed semaphores only. The routines for locking, unlocking, and getting the value of semaphores are used for both named and unnamed semaphores.

Table 2-15 **POSIX Semaphore Routines**

Call	Description
semPxLibInit( )	Initialize the POSIX semaphore library (non-POSIX).
sem_init( )	Initialize an unnamed semaphore.
sem_destroy( )	Destroy an unnamed semaphore.
sem_open( )	Initialize/open a named semaphore.
sem_close( )	Close a named semaphore.
sem_unlink( )	Remove a named semaphore.
sem_wait( )	Lock a semaphore.
sem_trywait( )	Lock a semaphore only if it is not already locked.
sem_post( )	Unlock a semaphore.
sem_getvalue( )	Get the value of a semaphore.

WARNING: The ***sem_destroy( )*** call terminates an unnamed semaphore and deallocates any associated memory; the combination of ***sem_close( )*** and ***sem_unlink( )*** has the same effect for named semaphores. Take care when deleting semaphores, particularly mutual exclusion semaphores, to avoid deleting a semaphore still required by another task. Do not delete a semaphore unless the deleting task first succeeds in locking that semaphore. (Likewise, for named semaphores, close semaphores only from the same task that opens them.)

Comparison of POSIX and Wind Semaphores

POSIX semaphores are *counting* semaphores; that is, they keep track of the number of times they are given.

The Wind semaphore mechanism is similar to that specified by POSIX, except that Wind semaphores offer additional features: priority inheritance, task-deletion safety, the ability for a single task to take a semaphore multiple times, ownership of mutual-exclusion semaphores, semaphore timeouts, and the choice of queuing mechanism. When these features are important, Wind semaphores are preferable.

Using Unnamed Semaphores

In using unnamed semaphores, normally one task allocates memory for the semaphore and initializes it. A semaphore is represented with the data structure **sem_t**, defined in **semaphore.h**. The semaphore initialization routine, ***sem_init( )***, allows you to specify the initial value.

Once the semaphore is initialized, any task can use the semaphore by locking it with ***sem_wait( )*** (blocking) or ***sem_trywait( )*** (non-blocking), and unlocking it with ***sem_post( )***.

As noted earlier, semaphores can be used for both synchronization and mutual exclusion. When a semaphore is used for synchronization, it is typically initialized to zero (locked). The task waiting to be synchronized blocks on a ***sem_wait( )***. The task doing the synchronizing unlocks the semaphore using ***sem_post( )***. If the task blocked on the semaphore is the only one waiting for that semaphore, the task unblocks and becomes ready to run. If other tasks are blocked on the semaphore, the task with the highest priority is unblocked.

When a semaphore is used for mutual exclusion, it is typically initialized to a value greater than zero (meaning that the resource is available). Therefore, the first task to lock the semaphore does so without blocking; subsequent tasks block (if the semaphore value was initialized to 1).

Example 2-6 **POSIX Unnamed Semaphores**

```
/* This example uses unnamed semaphores to synchronize an action between
 * the calling task and a task that it spawns (tSyncTask). To run from
 * the shell, spawn as a task:
 *   -> sp unnameSem
 */

/* includes */

#include "vxWorks.h"
#include "semaphore.h"

/* forward declarations */

void syncTask (sem_t * pSem);

void unnameSem (void)
    {
    sem_t * pSem;

    /* reserve memory for semaphore */

    pSem = (sem_t *) malloc (sizeof (sem_t));
```

```
    /* initialize semaphore to unavailable */

    if (sem_init (pSem, 0, 0) == -1)
        {
        printf ("unnameSem: sem_init failed\n");
        return;
        }

    /* create sync task */

    printf ("unnameSem: spawning task...\n");
    taskSpawn ("tSyncTask", 90, 0, 2000, syncTask, pSem);

    /* do something useful to synchronize with syncTask */

    /* unlock sem */

    printf ("unnameSem: posting semaphore - synchronizing action\n");
    if (sem_post (pSem) == -1)
        {
        printf ("unnameSem: posting semaphore failed\n");
        return;
        }

    /* all done - destroy semaphore */

    if (sem_destroy (pSem) == -1)
      {
      printf ("unnameSem: sem_destroy failed\n");
      return;
      }
    }

void syncTask
    (
    sem_t * pSem
    )
    {
    /* wait for synchronization from unnameSem */

    if (sem_wait (pSem) == -1)
        {
        printf ("syncTask: sem_wait failed \n");
        return;
        }
    else
        printf ("syncTask:sem locked; doing sync'ed action...\n");

    /* do something useful here */
    }
```

Using Named Semaphores

The ***sem_open( )*** routine either opens a named semaphore that already exists, or, as an option, creates a new semaphore. You can specify which of these possibilities you want by combining the following flag values:

O_CREAT Create the semaphore if it does not already exist (if it exists, either fail or open the semaphore, depending on whether **O_EXCL** is also specified).

O_EXCL Open the semaphore only if newly created; fail if the semaphore exists already.

The possible effects of a call to ***sem_open( )***, depending on which flags are set and on whether the semaphore accessed already exists, are shown in Table 2-16. There is no entry for **O_EXCL** alone, because using that flag alone is not meaningful.

Table 2-16 **Possible Outcomes of Calling *sem_open( )***

Flag Settings	Semaphore Exists	Semaphore Does Not Exist
None	Semaphore is opened	Routine fails
O_CREAT	Semaphore is opened	Semaphore is created
O_CREAT and **O_EXCL**	Routine fails	Semaphore is created

A POSIX named semaphore, once initialized, remains usable until explicitly destroyed. Tasks can explicitly mark a semaphore for destruction at any time, but the semaphore remains in the system until no task has the semaphore open.

If **INCLUDE_SHOW_ROUTINES** is defined in the VxWorks configuration (for details, see *8. Configuration*), you can use ***show( )*** from the Tornado shell to display information about a POSIX semaphore:[2]

```
-> show semId
value = 0 = 0x0
```

The output is sent to the standard output device, and provides information about the POSIX semaphore **mySem** with two tasks blocked waiting for it:

```
Semaphore name          :mySem
sem_open() count         :3
Semaphore value         :0
No. of blocked tasks       :2
```

2. This is not a POSIX routine, nor is it designed for use from programs; use it from the Tornado shell (see the *Tornado User's Guide: Shell* for details).

For a group of collaborating tasks to use a named semaphore, one of the tasks first creates and initializes the semaphore (by calling ***sem_open( )*** with the **O_CREAT** flag). Any task that needs to use the semaphore thereafter opens it by calling ***sem_open( )*** with the same name (but without setting **O_CREAT**). Any task that has opened the semaphore can use it by locking it with ***sem_wait( )*** (blocking) or ***sem_trywait( )*** (non-blocking) and unlocking it with ***sem_post( )***.

To remove a semaphore, all tasks using it must first close it with ***sem_close( )***, and one of the tasks must also unlink it. Unlinking a semaphore with ***sem_unlink( )*** removes the semaphore name from the name table. After the name is removed from the name table, tasks that currently have the semaphore open can still use it, but no new tasks can open this semaphore. The next time a task tries to open the semaphore without the **O_CREAT** flag, the operation fails. The semaphore vanishes when the last task closes it.

Example 2-7 **POSIX Named Semaphores**

```
/* In this example, nameSem() creates a task for synchronization. The
 * new task, tSyncSemTask, blocks on the semaphore created in nameSem().
 * Once the synchronization takes place, both tasks close the semaphore,
 * and nameSem() unlinks it. To run this task from the shell, spawn
 * nameSem as a task:
 *   -> sp nameSem, "myTest"
 */

/* includes */
#include "vxWorks.h"
#include "semaphore.h"
#include "fcntl.h"

/* forward declaration */
int syncSemTask (char * name);

int nameSem
    (
    char * name
    )
    {
    sem_t * semId;

    /* create a named semaphore, initialize to 0*/
    printf ("nameSem: creating semaphore\n");
    if ((semId = sem_open (name, O_CREAT, 0, 0)) == (sem_t *) -1)
        {
        printf ("nameSem: sem_open failed\n");
        return;
        }

    printf ("nameSem: spawning sync task\n");

    taskSpawn ("tSyncSemTask", 90, 0, 2000, syncSemTask, name);
```

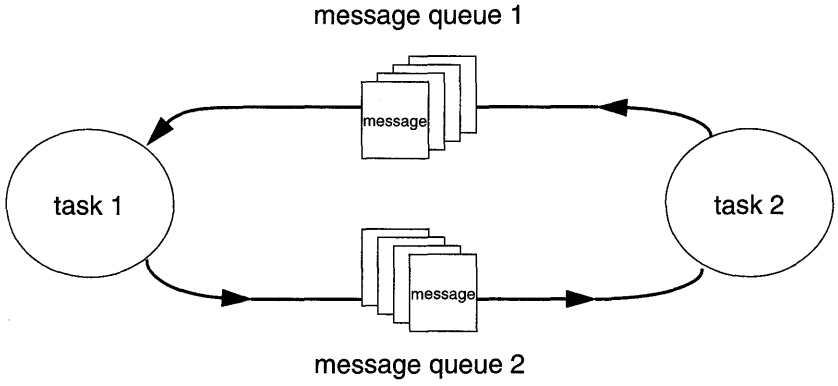
```
    /* do something useful to synchronize with syncSemTask */

    /* give semaphore */
    printf ("nameSem: posting semaphore - synchronizing action\n");
    if (sem_post (semId) == -1)
        {
        printf ("nameSem: sem_post failed\n");
        return;
        }

    /* all done */
    if (sem_close (semId) == -1)
        {
        printf ("nameSem: sem_close failed\n");
        return;
        }

    if (sem_unlink (name) == -1)
        {
        printf ("nameSem: sem_unlink failed\n");
        return;
        }

    printf ("nameSem: closed and unlinked semaphore\n");
    }

int syncSemTask
    (
    char * name
    )
    {
    sem_t * semId;

    /* open semaphore */
    printf ("syncSemTask: opening semaphore\n");
    if ((semId = sem_open (name, 0)) == (sem_t *) -1)
        {
        printf ("syncSemTask: sem_open failed\n");
        return;
        }

    /* block waiting for synchronization from nameSem */
    printf ("syncSemTask: attempting to take semaphore...\n");
    if (sem_wait (semId) == -1)
        {
        printf ("syncSemTask: taking sem failed\n");
        return;
        }

    printf ("syncSemTask: has semaphore, doing sync'ed action ...\n");

    /* do something useful here */

    if (sem_close (semId) == -1)
        {
        printf ("syncSemTask: sem_close failed\n");
        return;
        }
    }
```

2.4.4 Message Queues

Modern real-time applications are constructed as a set of independent but cooperating tasks. While semaphores provide a high-speed mechanism for the synchronization and interlocking of tasks, often a higher-level mechanism is necessary to allow cooperating tasks to communicate with each other. In VxWorks, the primary intertask communication mechanism within a single CPU is *message queues*. The optional product, VxMP, provides global message queues that can be used across processors; for more information, see *6. Shared-Memory Objects*.

Message queues allow a variable number of messages, each of variable length, to be queued. Any task or ISR can send messages to a message queue. Any task can receive messages from a message queue. Multiple tasks can send to and receive from the same message queue. Full-duplex communication between two tasks generally requires two message queues, one for each direction; see Figure 2-14.

Figure 2-14 **Full Duplex Communication Using Message Queues**



There are two message-queue subroutine libraries in VxWorks. The first of these, **msgQLib**, provides Wind message queues, designed expressly for VxWorks; the second, **mqPxLib**, is compatible with the POSIX standard (1003.1b) for real-time extensions. See *Comparison of POSIX and Wind Message Queues*, p.86 for a discussion of the differences between the two message-queue designs.

Wind Message Queues

Wind message queues are created and deleted with the routines shown in Table 2-17. This library provides messages that are queued in FIFO order, with a single exception: there are two priority levels, and messages marked as high priority are attached to the head of the queue.

Table 2-17 **Wind Message Queue Control**

Call	Description
msgQCreate( )	Allocate and initialize a message queue.
msgQDelete( )	Terminate and free a message queue.
msgQSend( )	Send a message to a message queue.
msgQReceive( )	Receive a message from a message queue.

A message queue is created with ***msgQCreate( )***. Its parameters specify the maximum number of messages that can be queued in the message queue and the maximum length in bytes of each message. Enough buffer space is preallocated for the specified number and length of messages.

A task or ISR sends a message to a message queue with ***msgQSend( )***. If no tasks are waiting for messages on that queue, the message is added to the queue's buffer of messages. If any tasks are already waiting for a message from that message queue, the message is immediately delivered to the first waiting task.

A task receives a message from a message queue with ***msgQReceive( )***. If messages are already available in the message queue's buffer, the first message is immediately dequeued and returned to the caller. If no messages are available, then the calling task blocks and is added to a queue of tasks waiting for messages. This queue of waiting tasks can be ordered either by task priority or FIFO, as specified in an option parameter when the queue is created.

Timeouts

Both ***msgQSend( )*** and ***msgQReceive( )*** take timeout parameters. When sending a message, the timeout specifies how many ticks to wait for buffer space to become available, if no space is available to queue the message. When receiving a message, the timeout specifies how many ticks to wait for a message to become available, if no message is immediately available. As with semaphores, the value of the timeout parameter can have the special values of **NO_WAIT** (0), meaning always return immediately, or **WAIT_FOREVER** (-1), meaning never time out the routine.

Urgent Messages

The *msgQSend*( ) function allows specification of the priority of the message as either normal (**MSG_PRI_NORMAL**) or urgent (**MSG_PRI_URGENT**). Normal priority messages are added to the tail of the list of queued messages, while urgent priority messages are added to the head of the list.

Example 2-8 **Wind Message Queues**

```
/* In this example, task t1 creates the message queue and sends a message
 * to task t2. Task t2 receives the message from the queue and simply
 * displays the message.
 */

/* includes */
#include "vxWorks.h"
#include "msgQLib.h"

/* defines */
#define MAX_MSGS (10)
#define MAX_MSG_LEN (100)

MSG_Q_ID myMsgQId;

task2 (void)
    {
    char msgBuf[MAX_MSG_LEN];

    /* get message from queue; if necessary wait until msg is available */
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* display message */
    printf ("Message from task 1:\n%s\n", msgBuf);
    }

#define MESSAGE "Greetings from Task 1"
task1 (void)
    {
    /* create message queue */
    if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY))
        == NULL)
        return (ERROR);

    /* send a normal priority message, blocking if queue is full */
    if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,
                  MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
    }
```

POSIX Message Queues

The POSIX message queue routines, provided by **mqPxLib**, are shown in Table 2-18. These routines are similar to Wind message queues, except that POSIX message queues provide named queues and messages with a range of priorities.

Table 2-18 **POSIX Message Queue Routines**

Call	Description
mqPxLibInit( )	Initialize the POSIX message queue library (non-POSIX).
mq_open( )	Open a message queue.
mq_close( )	Close a message queue.
mq_unlink( )	Remove a message queue.
mq_send( )	Send a message to a queue.
mq_receive( )	Get a message from a queue.
mq_notify( )	Signal a task that a message is waiting on a queue.
mq_setattr( )	Set a queue attribute.
mq_getattr( )	Get a queue attribute.

The initialization routine ***mqPxLibInit*( )** makes the POSIX message queue routines available; the system initialization code must call it before any tasks use POSIX message queues. As shipped, ***usrInit*( )** calls ***mqPxLibInit*( )** when **INCLUDE_POSIX_MQ** is defined in **configAll.h.**

Before a set of tasks can communicate through a POSIX message queue, one of the tasks must create the message queue by calling ***mq_open*( )** with the **O_CREAT** flag set. Once a message queue is created, other tasks can open that queue by name to send and receive messages on it. Only the first task opens the queue with the **O_CREAT** flag; subsequent tasks can open the queue for receiving only (**O_RDONLY**), sending only (**O_WRONLY**), or both sending and receiving (**O_RDWR**).

To put messages on a queue, use ***mq_send*( )**. If a task attempts to put a message on the queue when the queue is full, the task blocks until some other task reads a message from the queue, making space available. To avoid blocking on ***mq_send*( )**, set **O_NONBLOCK** when you open the message queue. In that case, when the

queue is full, *mq_send( )* returns -1 and sets **errno** to **EAGAIN** instead of pending, allowing you to try again or take other action as appropriate.

One of the arguments to *mq_send( )* specifies a message priority. Priorities range from 0 (lowest priority) to 31 (highest priority).

When a task receives a message using *mq_receive( )*, the task receives the highest-priority message currently on the queue. Among multiple messages with the same priority, the first message placed on the queue is the first received (FIFO order). If the queue is empty, the task blocks until a message is placed on the queue. To avoid pending on *mq_receive( )*, open the message queue with **O_NONBLOCK**; in that case, when a task attempts to read from an empty queue, *mq_receive( )* returns -1 and sets **errno** to **EAGAIN**.

To close a message queue, call *mq_close( )*. Closing the queue does not destroy it, but only asserts that your task is no longer using the queue. To request that the queue be destroyed, call *mq_unlink( )*. *Unlinking* a message queue does not destroy the queue immediately, but it does prevent any further tasks from opening that queue, by removing the queue name from the name table. Tasks that currently have the queue open can continue to use it. When the last task closes an unlinked queue, the queue is destroyed.

Example 2-9 **POSIX Message Queues**

```
/* In this example, the mqExInit() routine spawns two tasks that
 * communicate using the message queue.
 */

/* mqEx.h - message example header */

/* defines */
#define MQ_NAME "exampleMessageQueue"

/* forward declarations */
void receiveTask (void);
void sendTask (void);

/* testMQ.c - example using POSIX message queues */

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
#include "mqEx.h"

/* defines */
#define HI_PRIO      31
#define MSG_SIZE     16
```

```
int mqExInit (void)
    {
    /* create two tasks */
    if (taskSpawn ("tRcvTask", 95, 0, 4000, receiveTask, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ("taskSpawn of tRcvTask failed\n");
        return (ERROR);
        }

    if (taskSpawn ("tSndTask", 100, 0, 4000, sendTask, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ("taskSpawn of tSendTask failed\n");
        return (ERROR);
        }
    }

void receiveTask (void)
    {
    mqd_t    mqPXId;          /* msg queue descriptor */
    char     msg[MSG_SIZE];   /* msg buffer */
    int      prio;            /* priority of message */

    /* open message queue using default attributes */
    if ((mqPXId = mq_open (MQ_NAME, O_RDWR | O_CREAT, 0, NULL))
        == (mqd_t) -1)
        {
        printf ("receiveTask: mq_open failed\n");
        return;
        }

    /* try reading from queue */
    if (mq_receive (mqPXId, msg, MSG_SIZE, &prio) == -1)
        {
        printf ("receiveTask: mq_receive failed\n");
        return;
        }
    else
        {
        printf ("receiveTask: Msg of priority %d received:\n\t\t%s\n",
                prio, msg);
        }
    }

/* sendTask.c - mq sending example */

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "mqEx.h"

/* defines */
#define MSG   "greetings"
#define HI_PRIO 30
```

```
void sendTask (void)
    {
    mqd_t    mqPXId;          /* msg queue descriptor */

    /* open msg queue; should already exist with default attributes */
    if ((mqPXId = mq_open (MQ_NAME, O_RDWR, 0, NULL)) == (mqd_t) -1)
        {
        printf ("sendTask: mq_open failed\n");
        return;
        }

    /* try writing to queue */
    if (mq_send (mqPXId, MSG, sizeof (MSG), HI_PRIO) == -1)
        {
        printf ("sendTask: mq_send failed\n");
        return;
        }
    else
        printf ("sendTask: mq_send succeeded\n");
    }
```

Notifying a Task that a Message is Waiting

A task can use the *mq_notify( )* routine to request notification when a message for it arrives at an empty queue. The advantage of this is that a task can avoid blocking or polling to wait for a message.

The *mq_notify( )* call specifies a signal to be sent to the task when a message is placed on an empty queue. This mechanism uses the POSIX data-carrying extension to signaling, which allows you, for example, to carry a queue identifier with the signal (see *POSIX Queued Signals*, p.92).

The *mq_notify( )* mechanism is designed to alert the task only for new messages that are actually available. If the message queue already contains messages, no notification is sent when more messages arrive. If there is another task that is blocked on the queue with *mq_receive( )*, that other task unblocks, and no notification is sent to the task registered with *mq_notify( )*.

Notification is exclusive to a single task: each queue can register only one task for notification at a time. Once a queue has a task to notify, no attempts to register with *mq_notify( )* can succeed until the notification request is satisfied or cancelled.

Once a queue sends notification to a task, the notification request is satisfied, and the queue has no further special relationship with that particular task; that is, the queue sends a notification signal only once per *mq_notify( )* request. To arrange for one particular task to continue receiving notification signals, the best approach is to call *mq_notify( )* from the same signal handler that receives the notification signals. This reinstalls the notification request as soon as possible.

To cancel a notification request, specify NULL instead of a notification signal. Only the currently registered task can cancel its notification request.

Example 2-10 **Notifying a Task that a Message Queue is Waiting**

```
/* In this example, a task uses mq_notify() to discover when a message
 * is waiting for it on a previously empty queue.
 */

/* includes */
#include "vxWorks.h"
#include "signal.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"

/* defines */
#define QNAM      "PxQ1"
#define MSG_SIZE    64        /* limit on message sizes */

/* forward declarations */
static void exNotificationHandle (int, siginfo_t *, void *);
static void exMqRead (mqd_t);

/*******************************************************************************
*
* exMqNotify - example of how to use mq_notify()
*
* This routine illustrates the use of mq_notify() to request notification
* via signal of new messages in a queue. To simplify the example, a
* single task both sends and receives a message.
*/

int exMqNotify
    (
    char * pMess            /* text for message to self */
    )
    {
    struct mq_attr   attr;                  /* queue attribute structure */
    struct sigevent   sigNotify;            /* to attach notification */
    struct sigaction  mySigAction;          /* to attach signal handler */
    mqd_t        exMqId;                        /* id of message queue */

    /* Minor sanity check; avoid exceeding msg buffer */
    if (MSG_SIZE <= strlen (pMess))
        {
        printf ("exMqNotify: message too long\n");
        return (-1);
        }

    /* Install signal handler for the notify signal - fill in a
     * sigaction structure and pass it to sigaction(). Because the
     * handler needs the siginfo structure as an argument, the
     * SA_SIGINFO flag is set in sa_flags.
     */
```

```
mySigAction.sa_sigaction = exNotificationHandle;
mySigAction.sa_flags   = SA_SIGINFO;
sigemptyset (&mySigAction.sa_mask);

if (sigaction (SIGUSR1, &mySigAction, NULL) == -1)
    {
    printf ("sigaction failed\n");
    return (-1);
    }

/* Create a message queue - fill in a mq_attr structure with the
 * size and no. of messages required, and pass it to mq_open().
 */
attr.mq_flags  = O_NONBLOCK;     /* make nonblocking */
attr.mq_maxmsg = 2;
attr.mq_msgsize = MSG_SIZE;

if ( (exMqId = mq_open (QNAM, O_CREAT | O_RDWR, 0, &attr)) ==
     (mqd_t) - 1 )
    {
    printf ("mq_open failed\n");
    return (-1);
    }

/* Set up notification: fill in a sigevent structure and pass it
 * to mq_notify(). The queue ID is passed as an argument to the
 * signal handler.
 */
sigNotify.sigev_signo      = SIGUSR1;
sigNotify.sigev_notify     = SIGEV_SIGNAL;
sigNotify.sigev_value.sival_int = (int) exMqId;

if (mq_notify (exMqId, &sigNotify) == -1)
    {
    printf ("mq_notify failed\n");
    return (-1);
    }

/* We just created the message queue, but it may not be empty;
 * a higher-priority task may have placed a message there while
 * we were requesting notification. mq_notify() does nothing if
 * messages are already in the queue; therefore we try to
 * retrieve any messages already in the queue.
 */
exMqRead (exMqId);

/* Now we know the queue is empty, so we will receive a signal
 * the next time a message arrives.
 *
 * We send a message, which causes the notify handler to be
 * invoked. It is a little silly to have the task that gets the
 * notification be the one that puts the messages on the queue,
 * but we do it here to simplify the example.
 *
 * A real application would do other work instead at this point.
 */
```

```
    if (mq_send (exMqId, pMess, 1 + strlen (pMess), 0) == -1)
        {
        printf ("mq_send failed\n");
        return (-1);
        }

    /* Cleanup */
    if (mq_close (exMqId) == -1)
        {
        printf ("mq_close failed\n");
        return (-1);
        }

    /* More cleanup */
    if (mq_unlink (QNAM) == -1)
        {
        printf ("mq_unlink failed\n");
        return (-1);
        }

    return (0);
    }

/*******************************************************************************
*
* exNotificationHandle - handler to read in messages
*
* This routine is a signal handler; it reads in messages from a message
* queue.
*/

static void exNotificationHandle
    (
    int     sig,          /* signal number */
    siginfo_t * pInfo,        /* signal information */
    void *   pSigContext     /* unused (required by posix) */
    )
    {
    struct sigevent   sigNotify;
    mqd_t         exMqId;

    /* Get the Id of the message queue out of the siginfo structure.
     */
    exMqId = (mqd_t) pInfo->si_value.sival_int;

    /* Request notification again; it resets each time a notification
     * signal goes out.
     */
    sigNotify.sigev_signo = pInfo->si_signo;
    sigNotify.sigev_value = pInfo->si_value;
    sigNotify.sigev_notify = SIGEV_SIGNAL;

    if (mq_notify (exMqId, &sigNotify) == -1)
        {
        printf ("mq_notify failed\n");
```

```
        return;
        }

    /* Read in the messages
     */
    exMqRead (exMqId);
    }

/*******************************************************************************
*
* exMqRead - read in messages
*
* This small utility routine receives and displays all messages
* currently in a POSIX message queue; assumes queue has O_NONBLOCK.
*/

static void exMqRead
    (
    mqd_t         exMqId
    )
    {
    char          msg[MSG_SIZE];
    int           prio;

    /* Read in the messages - uses a loop to read in the messages
     * because a notification is sent ONLY when a message is sent on
     * an EMPTY message queue. There could be multiple msgs if, for
     * example, a higher-priority task was sending them. Because the
     * message queue was opened with the O_NONBLOCK flag, eventually
     * this loop exits with errno set to EAGAIN (meaning we did an
     * mq_receive() on an empty message queue).
     */
    while (mq_receive (exMqId, msg, MSG_SIZE, &prio) != -1)
        {
        printf ("exMqRead: received message: %s\n",msg);
        }

    if (errno != EAGAIN)
        {
        printf ("mq_receive: errno = %d\n", errno);
        }
    }
```

Message Queue Attributes

A POSIX message queue has the following attributes:

- an optional **O_NONBLOCK** flag
- the maximum number of messages in the message queue
- the maximum message size
- the number of messages currently on the queue

Tasks can set or clear the **O_NONBLOCK** flag (but not the other attributes) using *mq_setattr( )*, and get the values of all the attributes using *mq_getattr( )*.

Example 2-11 **Setting and Getting Message Queue Attributes**

```
/* This example sets the O_NONBLOCK flag, and examines message queue
 * attributes.
 */

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"

/* defines */
#define MSG_SIZE    16

int attrEx
    (
    char * name
    )
    {
    mqd_t           mqPXId;     /* mq descriptor */
    struct mq_attr  attr;       /* queue attribute structure */
    struct mq_attr  oldAttr;    /* old queue attributes */
    char            buffer[MSG_SIZE];
    int             prio;

    /* create read write queue that is blocking */
    attr.mq_flags = 0;
    attr.mq_maxmsg = 1;
    attr.mq_msgsize = 16;
    if ((mqPXId = mq_open (name, O_CREAT | O_RDWR , 0, &attr))
         == (mqd_t) -1)
        return (ERROR);
    else
        printf ("mq_open with non-block succeeded\n");

    /* change attributes on queue - turn on non-blocking */
    attr.mq_flags = O_NONBLOCK;
    if (mq_setattr (mqPXId, &attr, &oldAttr) == -1)
        return (ERROR);
    else
        {
        /* paranoia check - oldAttr should not include non-blocking.
         */
        if (oldAttr.mq_flags & O_NONBLOCK)
            return (ERROR);
        else
            printf ("mq_setattr turning on non-blocking succeeded\n");
        }

    /* try receiving - there are no messages but this shouldn't block */
    if (mq_receive (mqPXId, buffer, MSG_SIZE, &prio) == -1)
        {
        if (errno != EAGAIN)
            return (ERROR);
        else
            printf ("mq_receive with non-blocking didn't block on empty queue\n");
        }
```

```
else
    return (ERROR);

/* use mq_getattr to verify success */
if (mq_getattr (mqPXId, &oldAttr) == -1)
    return (ERROR);
else
    {
    /* test that we got the values we think we should */
    if (!(oldAttr.mq_flags & O_NONBLOCK) || (oldAttr.mq_curmsgs != 0))
        return (ERROR);
    else
        printf ("queue attributes are:\n\tblocking is %s\n\t
                message size is: %d\n\t
                max messages in queue: %d\n\t
                no. of current msgs in queue: %d\n",
                oldAttr.mq_flags & O_NONBLOCK ? "on" : "off",
                oldAttr.mq_msgsize, oldAttr.mq_maxmsg,
                oldAttr.mq_curmsgs);
    }

/* clean up - close and unlink mq */

if (mq_unlink (name) == -1)
    return (ERROR);

if (mq_close (mqPXId) == -1)
    return (ERROR);

return (OK);

}
```

Comparison of POSIX and Wind Message Queues

The two forms of message queues solve many of the same problems, but there are some significant differences. Table 2-19 summarizes the main differences between the two forms of message queues.

Table 2-19 **Message Queue Feature Comparison**

Feature	Wind Message Queues	POSIX Message Queues
Message Priority Levels	1	32
Blocked Task Queues	FIFO or priority-based	Priority-based
Receive with Timeout	Optional	Not available
Task Notification	Not available	Optional (one task)
Close/Unlink Semantics	No	Yes

Another feature of POSIX message queues is, of course, portability: if you are migrating to VxWorks from another 1003.1b-compliant system, using POSIX message queues enables you to leave that part of the code unchanged, reducing the porting effort.

Displaying Message Queue Attributes

The VxWorks ***show( )*** command produces a display of the key message queue attributes, for either kind of message queue[3]. For example, if **mqPXId** is a POSIX message queue:

```
-> show mqPXId
value = 0 = 0x0
```

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

```
Message queue name        : MyQueue
No. of messages in queue    : 1
Maximum no. of messages     : 16
Maximum message size      : 16
```

Compare this to the output when **myMsgQId** is a Wind message queue:[4]

```
-> show myMsgQId
Message Queue Id  : 0x3adaf0
Task Queuing    : FIFO
Message Byte Len  : 4
Messages Max    : 30
Messages Queued   : 14
Receivers Blocked  : 0
Send timeouts    : 0
Receive timeouts  : 0
```

Servers and Clients with Message Queues

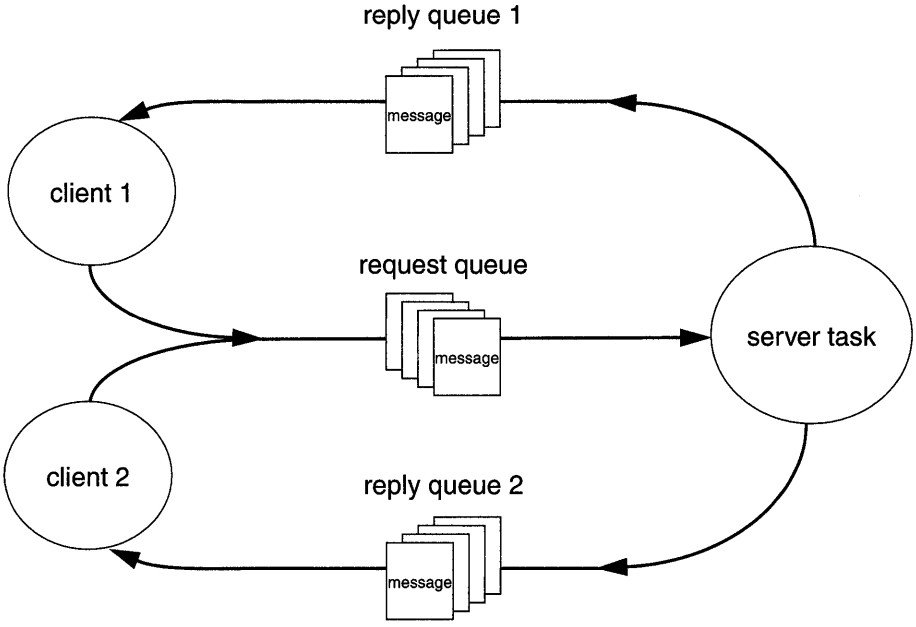
Real-time systems are often structured using a *client-server* model of tasks. In this model, server tasks accept requests from client tasks to perform some service, and usually return a reply. The requests and replies are usually made in the form of intertask messages. In VxWorks, message queues or pipes (see *2.4.5 Pipes*, p.88) are a natural way to implement this.

---

3. However, to get information on POSIX message queues, **INCLUDE_SHOW_ROUTINES** must be defined in the VxWorks configuration; for information, see *8. Configuration.*
4. The built-in ***show( )*** routine handles Wind message queues; see the *Tornado User's Guide: Shell* for information on built-in routines. You can also use the Tornado browser to get information on Wind message queues; see the *Tornado User's Guide: Browser* for details.

For example, client-server communications might be implemented as shown in Figure 2-15. Each server task creates a message queue to receive request messages from clients. Each client task creates a message queue to receive reply messages from servers. Each request message includes a field containing the **msgQId** of the client's reply message queue. A server task's "main loop" consists of reading request messages from its request message queue, performing the request, and sending a reply to the client's reply message queue.

Figure 2-15 **Client-Server Communications Using Message Queues**



The same architecture can be achieved with pipes instead of message queues, or by other means that are tailored to the needs of the particular application.

2.4.5 Pipes

Pipes provide an alternative interface to the message queue facility that goes through the VxWorks I/O system. Pipes are virtual I/O devices managed by the driver **pipeDrv**. The routine ***pipeDevCreate( )*** creates a pipe device and the underlying message queue associated with that pipe. The call specifies the name

of the created pipe, the maximum number of messages that can be queued to it, and the maximum length of each message:

```
status = pipeDevCreate ("/pipe/name", max_msgs, max_length);
```

The created pipe is a normally named I/O device. Tasks can use the standard I/O routines to open, read, and write pipes, and invoke *ioctl* routines. As they do with other I/O devices, tasks block when they read from an empty pipe until data is available, and block when they write to a full pipe until there is space available. Like message queues, ISRs can write to a pipe, but cannot read from a pipe.

As I/O devices, pipes provide one important feature that message queues cannot—the ability to be used with ***select( )***. This routine allows a task to wait for data to be available on any of a set of I/O devices. The ***select( )*** routine also works with other asynchronous I/O devices including network sockets and serial devices. Thus, by using ***select( )***, a task can wait for data on a combination of several pipes, sockets, and serial devices; see *3.3.8 Pending on Multiple File Descriptors: The Select Facility*, p.117.

Pipes allow you to implement a client-server model of intertask communications; see *Servers and Clients with Message Queues*, p.87.

2.4.6 Network Intertask Communication

Sockets

In VxWorks, the basis of intertask communications across the network is *sockets*. A socket is an endpoint for communications between tasks; data is sent from one socket to another. When you create a socket, you specify the Internet communications protocol that is to transmit the data. VxWorks supports the Internet protocols TCP and UDP. VxWorks socket facilities are source compatible with BSD 4.3 UNIX.

TCP provides reliable, guaranteed, two-way transmission of data with *stream sockets*. In a stream-socket communication, two sockets are "connected," allowing a reliable byte-stream to flow between them in each direction as in a circuit. For this reason TCP is often referred to as a *virtual circuit* protocol.

UDP provides a simpler but less robust form of communication. In UDP communications, data is sent between sockets in separate, unconnected, individually addressed packets called *datagrams*. A process creates a datagram socket and binds it to a particular port. There is no notion of a UDP "connection."

Any UDP socket, on any host in the network, can send messages to any other UDP socket by specifying its Internet address and port number.

One of the biggest advantages of socket communications is that it is "homogeneous." Socket communications among processes are exactly the same regardless of the location of the processes in the network, or the operating system under which they are running. 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 look identical to the application, except, of course, for their speed.

For more information, see *5.2.6 Sockets*, p.251 and the reference entry for **sockLib**.

Remote Procedure Calls (RPC)

Remote Procedure Calls (RPC) is a facility that allows a process on one machine to call a procedure that is executed by another process on either the same machine or a remote machine. Internally, RPC uses sockets as the underlying communication mechanism. Thus with RPC, VxWorks tasks and host system processes can invoke routines that execute on other VxWorks or host machines, in any combination.

As discussed in the previous sections on message queues and pipes, many real-time systems are structured with a client-server model of tasks. 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. Also, RPC includes tools to help generate the client interface routines and the server skeleton.

For more information on RPC, see *5.2.8 Remote Procedure Calls*, p.278.

2.4.7 Signals

VxWorks supports a software signal facility. Signals asynchronously alter the control flow of a task. Any task or ISR can raise a signal for a particular task. The task being signaled immediately suspends its current thread of execution and the task-specified signal handler routine is executed the next time the task is scheduled to run. Note that the signal handler gets invoked even if the task is blocked. Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism.

The *wind* kernel supports two types of signal interface: UNIX BSD-style signals and POSIX-compatible signals. The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b. For the sake of simplicity, we recommend that you use only one interface type in a given application, rather than mixing routines from different interfaces.

For more information on signals, see the reference entry for **sigLib**.

Basic Signal Routines

Table 2-20 shows the basic signal routines. To make these facilities available, the signal library initialization routine ***sigInit( )*** must be called, normally from ***usrInit( )*** in **usrConfig.c**, before interrupts are enabled.

Table 2-20 **Basic Signal Calls (BSD and POSIX 1003.1b)**

POSIX 1003.1b Compatible Call	UNIX BSD Compatible Call	Description
signal( )	*signal( )*	Specify the handler associated with a signal.
kill( )	*kill( )*	Send a signal to a task.
raise( )	N/A	Send a signal to yourself.
sigaction( )	*sigvec( )*	Examine or set the signal handler for a signal.
sigsuspend( )	*pause( )*	Suspend a task until a signal is delivered.
sigpending( )	N/A	Retrieve a set of pending signals blocked from delivery.
sigemptyset( ) *sigfillset( )* *sigaddset( )* *sigdelset( )* *sigismember( )*	*sigmask( )*	Manipulate a signal mask.
sigprocmask( )	*sigsetmask( )*	Set the mask of blocked signals.
sigprocmask( )	*sigblock( )*	Add to a set of blocked signals.

The colorful name ***kill( )*** harks back to the origin of these interfaces in UNIX BSD. Although the interfaces vary, the functionality of BSD-style signals and basic POSIX signals is similar.

In many ways, signals are analogous to hardware interrupts. The basic signal facility provides a set of 31 distinct signals. A *signal handler* binds to a particular signal with ***sigvec( )*** or ***sigaction( )*** in much the same way that an ISR is connected to an interrupt vector with ***intConnect( )***. A signal can be asserted by calling ***kill( )***. This is analogous to the occurrence of an interrupt. The routines ***sigsetmask( )*** and ***sigblock( )*** or ***sigprocmask( )*** let signals be selectively inhibited.

Certain signals are associated with hardware exceptions. For example, bus errors, illegal instructions, and floating-point exceptions raise specific signals.

POSIX Queued Signals

The ***sigqueue( )*** routine provides an alternative to ***kill( )*** for sending signals to a task. The important differences between the two are:

- ***sigqueue( )*** includes an application-specified value that is sent as part of the signal. You can use this value to supply whatever context your signal handler finds useful. This value is of type **sigval** (defined in **signal.h**); the signal handler finds it in the **si_value** field of one of its arguments, a structure **siginfo_t**. An extension to the POSIX ***sigaction( )*** routine allows you to register signal handlers that accept this additional argument.
- ***sigqueue( )*** enables the queueing of multiple signals for any task. The ***kill( )*** routine, by contrast, delivers only a single signal, even if multiple signals arrive before the handler runs.

VxWorks includes eight signals reserved for application use, numbered consecutively from **RTSIGMIN**. The presence of these eight reserved signals is required by POSIX 1003.1b, but the specific signal values are not; for portability, specify these signals as offsets from **RTSIGMIN** (for example, write **RTSIGMIN+2** to refer to the third reserved signal number). All signals delivered with ***sigqueue( )*** are queued by numeric order, with lower-numbered signals queuing ahead of higher-numbered signals.

POSIX 1003.1b also introduced an alternative means of receiving signals. The routine ***sigwaitinfo( )*** differs from ***sigsuspend( )*** or ***pause( )*** in that it allows your application to respond to a signal without going through the mechanism of a registered signal handler: when a signal is available, ***sigwaitinfo( )*** returns the value of that signal as a result, and does not invoke a signal handler even if one is registered. The routine ***sigtimedwait( )*** is similar, except that it can time out.

For detailed information on signals, see the reference entry for **sigLib**.

Table 2-21 **POSIX 1003.1b Queued Signal Calls**

Call	Description
sigqueue( )	Send a queued signal.
sigwaitinfo( )	Wait for a signal.
sigtimedwait( )	Wait for a signal with a timeout.

Signal Configuration

The basic signal facility is included in VxWorks by default with INCLUDE_SIGNALS (defined in **configAll.h**).

Before your application can use POSIX queued signals, they must be initialized separately with ***sigqueueInit*( )**. Like the basic signals initialization function ***sigInit*( )**, this function is normally called from ***usrInit*( )** in **usrConfig.c**, after ***sysInit*( )** runs.

To initialize the queued signal functionality, also define INCLUDE_POSIX_SIGNALS in **configAll.h**: with that definition, ***sigqueueInit*( )** is called automatically.

The constant NUM_SIGNAL_QUEUES in **configAll.h** specifies the number of signals that can be simultaneously queued for a specific task. The routine ***sigqueueInit*( )** allocates that number of buffers for use by ***sigqueue*( )**, which requires a buffer for each currently queued signal. A call to ***sigqueue*( )** fails if no buffer is available.

2.5 Interrupt Service Code

Hardware interrupt handling is of key significance in real-time systems, because it is usually through interrupts that the system is informed of external events. For the fastest possible response to interrupts, interrupt service routines (ISRs) in VxWorks run in a special context outside of any task's context. Thus, interrupt handling involves no task context switch. The interrupt routines, listed in Table 2-22, are provided in **intLib** and **intArchLib**.

Table 2-22 **Interrupt Routines**

Call	Description
intConnect( )	Connect a C routine to an interrupt vector.
intContext( )	Return TRUE if called from interrupt level.
intCount( )	Get the current interrupt nesting depth.
intLevelSet( )	Set the processor interrupt mask level.
intLock( )	Disable interrupts.
intUnlock( )	Re-enable interrupts.
intVecBaseSet( )	Set the vector base address.
intVecBaseGet( )	Get the vector base address.
intVecSet( )	Set an exception vector.
intVecGet( )	Get an exception vector.

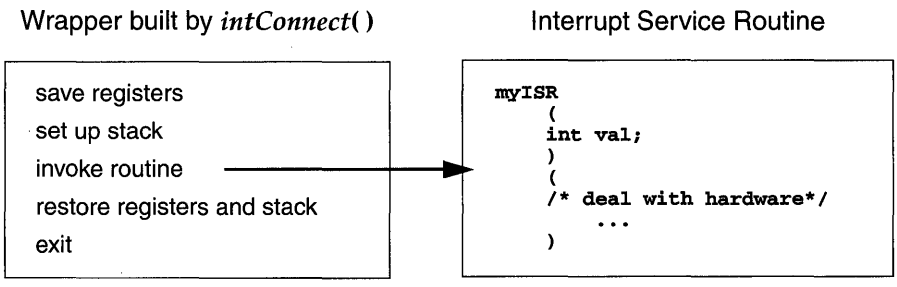
For boards with an MMU, the optional product VxVMI provides write protection for the interrupt vector table; see *7. Virtual Memory Interface.*

2.5.1 Connecting Application Code to Interrupts

You can use system hardware interrupts other than those used by VxWorks. VxWorks provides the routine ***intConnect( )***, which allows C functions to be connected to any interrupt. The arguments to this routine are the byte offset of the interrupt vector to connect to, the address of the C function to be connected, and an argument to pass to the function. When an interrupt occurs with a vector established in this way, the connected C function is called at interrupt level with the specified argument. When the interrupt handling is finished, the connected function returns. A routine connected to an interrupt in this way is called an *interrupt service routine* (ISR).

Interrupts cannot actually vector directly to C functions. Instead, ***intConnect( )*** builds a small amount of code that saves the necessary registers, sets up a stack entry (either on a special interrupt stack, or on the current task's stack) with the argument to be passed, and calls the connected function. On return from the function it restores the registers and stack, and exits the interrupt; see Figure 2-16.

Figure 2-16 **Routine Built by *intConnect*( )**




```
intConnect (INUM_TO_IVEC (someIntNum), myISR, someVal);
```

For target boards with VME backplanes, the BSP provides two standard routines for controlling VME bus interrupts, ***sysIntEnable*( )** and ***sysIntDisable*( )**.

2.5.2 Interrupt Stack

Whenever the architecture allows it, all ISRs use the same *interrupt stack*. This stack is allocated and initialized by the system at start-up according to specified configuration parameters. It must be large enough to handle the worst possible combination of nested interrupts.

Some architectures, however, do not permit using a separate interrupt stack. On such architectures, ISRs use the stack of the interrupted task. If you have such an architecture, you must create tasks with enough stack space to handle the worst possible combination of nested interrupts *and* the worst possible combination of ordinary nested calls. See the reference entry for your BSP to determine whether your architecture supports a separate interrupt stack.

Use the ***checkStack*( )** facility during development to see how close your tasks and ISRs have come to exhausting the available stack space.

2.5.3 Special Limitations of ISRs

Many VxWorks facilities are available to ISRs, but there are some important limitations. These limitations stem from the fact that an ISR does not run in a regular task context: it has no task control block, for example, and all ISRs share a single stack.

Table 2-23 **Routines that Can Be Called by Interrupt Service Routines**

Library	Routines
bLib	All routines
errnoLib	*errnoGet( )*, *errnoSet( )*
fppArchLib	*fppSave( )*, *fppRestore( )*
intLib	*intContext( )*, *intCount( )*, *intVecSet( )*, *intVecGet( )*
intArchLib	*intLock( )*, *intUnlock( )*
logLib	*logMsg( )*
lstLib	All routines except *lstFree( )*
mathALib	All routines, if *fppSave( )*/*fppRestore( )* are used
msgQLib	*msgQSend( )*
pipeDrv	*write( )*
rngLib	All routines except *rngCreate( )* and *rngDelete( )*
selectLib	*selWakeup( )*, *selWakeupAll( )*
semLib	*semGive( )* except mutual-exclusion semaphores, *semFlush( )*
sigLib	*kill( )*
taskLib	*taskSuspend( )*, *taskResume( )*, *taskPrioritySet( )*, *taskPriorityGet( )*, *taskIdVerify( )*, *taskIdDefault( )*, *taskIsReady( )*, *taskIsSuspended( )*, *taskTcb( )*
tickLib	*tickAnnounce( )*, *tickSet( )*, *tickGet( )*
tyLib	*tyIRd( )*, *tyITx( )*
vxLib	*vxTas( )*, *vxMemProbe( )*
wdLib	*wdStart( )*, *wdCancel( )*

For this reason, the basic restriction on ISRs is that they must not invoke routines that might cause the caller to block. For example, they must not try to take a semaphore, because if the semaphore is unavailable, the kernel tries to switch the caller to the pended state. However, ISRs can give semaphores, releasing any tasks waiting on them.

Because the memory facilities ***malloc( )*** and ***free( )*** take a semaphore, they cannot be called by ISRs, and neither can routines that make calls to ***malloc( )*** and ***free( )***. For example, ISRs cannot call any creation or deletion routines.

ISRs also must not perform I/O through VxWorks drivers. Although there are no inherent restrictions in the I/O system, most device drivers require a task context because they might block the caller to wait for the device. An important exception is the VxWorks pipe driver, which is designed to permit writes by ISRs.

VxWorks supplies a logging facility, in which a logging task prints text messages to the system console. This mechanism was specifically designed so that ISRs could use it, and is the most common way to print messages from ISRs. For more information, see the reference entry for **logLib**.

An ISR also must not call routines that use a floating-point coprocessor. In VxWorks, the interrupt driver code created by ***intConnect( )*** does not save and restore floating-point registers; thus, ISRs must not include floating-point instructions. If an ISR requires floating-point instructions, it must explicitly save and restore the registers of the floating-point coprocessor using routines in **fppArchLib**.

All VxWorks utility libraries, such as the linked-list and ring-buffer libraries, can be used by ISRs. As discussed earlier (*2.3.7 Task Error Status: errno*, p.45), the global variable **errno** is saved and restored as a part of the interrupt enter and exit code generated by the ***intConnect( )*** facility. Thus **errno** can be referenced and modified by ISRs as in any other code. Table 2-23 lists routines that can be called from ISRs.

2.5.4 Exceptions at Interrupt Level

When a task causes a hardware exception such as illegal instruction or bus error, the task is suspended and the rest of the system continues uninterrupted. However, when an ISR causes such an exception, there is no safe recourse for the system to handle the exception. The ISR has no context that can be suspended. Instead, VxWorks stores the description of the exception in a special location in low memory and executes a system restart.

The VxWorks boot ROMs test for the presence of the exception description in low memory and if it is detected, display it on the system console. The **e** command in the boot ROMs re-displays the exception description; see the *Tornado User's Guide: Getting Started*.

One example of such an exception is the message:

```
workQPanic: Kernel work queue overflow.
```

This exception usually occurs when kernel calls are made from interrupt level at a very high rate. It generally indicates a problem with clearing the interrupt signal or a similar driver problem.

2.5.5 Reserving High Interrupt Levels

The VxWorks interrupt support described earlier in this section is acceptable for most applications. However, on occasion, low-level control is required for events such as critical motion control or system failure response. In such cases it is desirable to reserve the highest interrupt levels to ensure zero-latency response to these events. To achieve zero-latency response, VxWorks provides the routine *intLockLevelSet( )*, which sets the system-wide interrupt-lockout level to the specified level. If you do not specify a level, the default is the highest level supported by the processor architecture.

⚠ **NOTE:** Some hardware prevents masking certain interrupt levels; check the hardware manufacturer's documentation. For example, on MC680*x*0 chips, interrupt level 7 is non-maskable. Because level 7 is also the highest interrupt level on this architecture, VxWorks uses 7 as the default lockout level—but this is in fact equivalent to a lockout level of 6, since the hardware prevents locking out level 7.

2.5.6 Additional Restrictions for ISRs at High Interrupt Levels

ISRs connected to interrupt levels that are not locked out (either an interrupt level higher than that set by *intLockLevelSet( )*, or an interrupt level defined in hardware as non-maskable) have special restrictions:

- The ISR can be connected only with *intVecSet( )*.
- The ISR cannot use any VxWorks operating system facilities that depend on interrupt locks for correct operation.

2.5.7 Interrupt-to-Task Communication

While it is important that VxWorks support direct connection of ISRs that run at interrupt level, interrupt events usually propagate to task-level code. Many VxWorks facilities are not available to interrupt-level code, including I/O to any device other than pipes. The following techniques can be used to communicate from ISRs to task-level code:

- **Shared Memory and Ring Buffers.** ISRs can share variables, buffers, and ring buffers with task-level code.
- **Semaphores.** ISRs can give semaphores (except for mutual-exclusion semaphores and VxMP shared semaphores) that tasks can take and wait for.

- **Message Queues.** ISRs can send messages to message queues for tasks to receive (except for shared message queues using VxMP). If the queue is full, the message is discarded.
- **Pipes.** ISRs can write messages to pipes that tasks can read. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message written is discarded because the ISR cannot block. ISRs must not invoke any I/O routine on pipes other than *write( )*.
- **Signals.** ISRs can "signal" tasks, causing asynchronous scheduling of their signal handlers.

2.6 Watchdog Timers

VxWorks includes a watchdog-timer mechanism that allows any C function to be connected to a specified time delay. Watchdog timers are maintained as part of the system clock ISR. Normally, functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. However, if the kernel is unable to execute the function immediately for any reason (such as a previous interrupt or kernel state), the function is placed on the **tExcTask** work queue. Functions on the **tExcTask** work queue execute at the priority level of the **tExcTask** (usually 0). Restrictions on ISRs apply to routines connected to watchdog timers. The functions in Table 2-24 are provided by the **wdLib** library.

Table 2-24 **Watchdog Timer Calls**

Call	Description
wdCreate( )	Allocate and initialize a watchdog timer.
wdDelete( )	Terminate and deallocate a watchdog timer.
wdStart( )	Start a watchdog timer.
wdCancel( )	Cancel a currently counting watchdog timer.

A watchdog timer is first created by calling *wdCreate( )*. Then the timer can be started by calling *wdStart( )*, which takes as arguments the number of ticks to delay, the C function to call, and an argument to be passed to that function. After the specified number of ticks have elapsed, the function is called with the specified

argument. The watchdog timer can be canceled any time before the delay has elapsed by calling *wdCancel*( ).

Example 2-12 **Watchdog Timers**

```
/* This example creates a watchdog timer and sets it to go off in
 * 3 seconds.
 */

/* includes */
#include "vxWorks.h"
#include "logLib.h"
#include "wdLib.h"

/* defines */
#define  SECONDS (3)

WDOG_ID myWatchDogId;
task (void)
    {
    /* Create watchdog */

    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);

    /* Set timer to go off in SECONDS - printing a message to stdout */

    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                 "Watchdog timer just expired\n") == ERROR)
        return (ERROR);

    /* ... */
    }
```

2.7 POSIX Clocks and Timers

A *clock* is a software construct (**struct timespec**, defined in **time.h**) that keeps time in seconds and nanoseconds. The software clock is updated by system-clock ticks. VxWorks provides a POSIX 1003.1b standard clock and timer interface.

The POSIX standard provides for identifying multiple virtual clocks, but only one clock is required—the system-wide real-time clock, identified in the clock and timer routines as **CLOCK_REALTIME** (also defined in **time.h**). VxWorks provides routines to access the system-wide real-time clock; see the reference entry for **clockLib**. (No virtual clocks are supported in VxWorks.)

The POSIX timer facility provides routines for tasks to signal themselves at some time in the future. Routines are provided to create, set, and delete a timer; see the reference entry for **timerLib**. When a timer goes off, the default signal (**SIGALRM**) is sent to the task. Use ***sigaction*( )** to install a signal handler that executes when the timer expires (see *2.4.7 Signals*, p.90).

Example 2-13 **POSIX Timers**

```
/* This example creates a new timer and stores it in timerid. */

/* includes */
#include "vxWorks.h"
#include "time.h"

int createTimer (void)
    {
    timer_t timerid;

    /* create timer */

    if (timer_create (CLOCK_REALTIME, NULL, &timerid) == ERROR)
        {
        printf ("create FAILED\n");
        return (ERROR);
        }

    return (OK);
    }
```

An additional POSIX function, ***nanosleep*( )**, allows specification of sleep or delay time in units of seconds and nanoseconds, as opposed to the ticks used by the Wind ***taskDelay*( )** function. Only the units are different, however, not the precision: both delay routines have the same precision, determined by the system clock rate.

2.8 POSIX Memory-Locking Interface

Many operating systems perform memory *paging* and *swapping*. These techniques allow the use of more virtual memory than there is physical memory on a system, by copying blocks of memory out to disk and back. These techniques impose severe and unpredictable delays in execution time; they are therefore undesirable in real-time systems.

Because the *wind* kernel is designed specifically for real-time applications, it never performs paging or swapping. However, the POSIX 1003.1b standard for real-time extensions also covers operating systems that perform paging or swapping. On such systems, applications that attempt real-time performance can use the POSIX *page-locking* facilities to declare that certain blocks of memory must not be paged or swapped.

To help maximize portability, VxWorks includes the POSIX page-locking routines. Executing these routines makes no difference in VxWorks, because all memory is, in effect, always locked. They are included only to make it easier to port programs between other POSIX-conforming systems and VxWorks.

The POSIX page-locking routines are in **mmanPxLib** (the name reflects the fact that these routines are part of the POSIX "memory-management" routines). Because in VxWorks all pages are always kept in memory, the routines listed in Table 2-25 always return a value of **OK** (0), and have no further effect.

The **mmanPxLib** library is included automatically when the configuration constant **INCLUDE_POSIX_MEM** is defined in **configAll.h**.

Table 2-25 **POSIX Memory Management Calls**

Call	Purpose on Systems with Paging or Swapping
mlockall( )	Lock into memory all pages used by a task.
munlockall( )	Unlock all pages used by a task.
mlock( )	Lock a specified page.
munlock( )	Unlock a specified page.

3 I/O System

List of Tables

List of Figures

List of Examples

3.1 Introduction

The VxWorks I/O system is designed to present a simple, uniform, device-independent interface to any kind of device, including:

- character-oriented devices such as terminals or communications lines
- random-access block devices such as disks
- virtual devices such as intertask *pipes* and *sockets*
- monitor and control devices such as digital/analog I/O devices
- network devices that give access to remote devices

The VxWorks I/O system provides standard C libraries for both basic and buffered I/O. The basic I/O libraries are UNIX-compatible; the buffered I/O libraries are ANSI C-compatible. Internally, the VxWorks I/O system has a unique design that makes it faster and more flexible than most other I/O systems. These are important attributes in a real-time system.

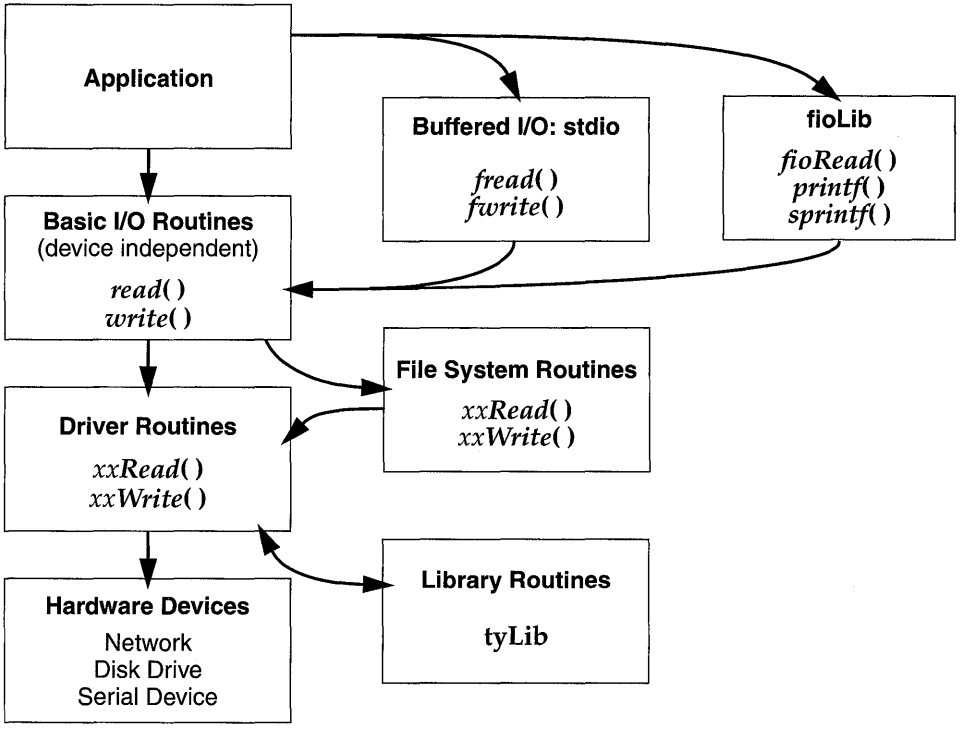
This chapter first describes the nature of *files* and *devices*, and the user view of basic and buffered I/O. The middle section discusses the details of some specific devices. The final section is a detailed discussion of the internal structure of the VxWorks I/O system.

Figure 3-1 diagrams the relationships between the different pieces of the VxWorks I/O system. All the elements of the I/O system are discussed in this chapter, except for file system routines, which are presented in *4. Local File Systems* in this manual.

3.2 Files, Devices, and Drivers

In VxWorks, applications access I/O devices by opening named *files*. A *file* can refer to one of two things:

Figure 3-1 **Overview of the VxWorks I/O System**



- An unstructured *"raw" device* such as a serial communications channel or an intertask pipe.
- A *logical file* on a structured, random-access device containing a file system.

Consider the following named files:

/usr/myfile **/pipe/mypipe** **/tyCo/0**

The first refers to a file called **myfile**, on a disk device called **/usr**. The second is a named pipe (by convention, pipe names begin with **/pipe**). The third refers to a physical serial channel. However, I/O can be done to or from any of these in the same way. Within VxWorks, they are all called *files*, even though they refer to very different physical objects.

Devices are handled by program modules called *drivers*. In general, using the I/O system does not require any further understanding of the implementation of

devices and drivers. Note, however, that the VxWorks I/O system gives drivers considerable flexibility in the way they handle each specific device. Drivers strive to follow the conventional user view presented here, but can differ in the specifics. See *3.7 Devices in VxWorks*, p.131.

Although all I/O is directed at named files, it can be done at two different levels: *basic* and *buffered*. The two differ in the way data is buffered and in the types of calls that can be made. These two levels are discussed in later sections.

3.2.1 File Names and the Default Device

A file name is specified as a character string. An unstructured device is specified with the device name. In the case of file system devices, the device name is followed by a file name. Thus the name **/tyCo/0** might name a particular serial I/O channel, and the name **DEV1:/file1** probably indicates the file **file1** on the **DEV1:** device.

When a file name is specified in an I/O call, the I/O system searches for a device with a name that matches at least an initial substring of the file name. The I/O function is then directed at this device.

If a matching device name cannot be found, then the I/O function is directed at a *default device*. You can set this default device to be any device in the system, including no device at all, in which case failure to match a device name returns an error.

Non-block devices are named when they are added to the I/O system, usually at system initialization time. Block devices are named when they are initialized for use with a specific file system. The VxWorks I/O system imposes no restrictions on the names given to devices. The I/O system does not interpret device or file names in any way, other than during the search for matching device and file names.

It is useful to adopt some naming conventions for device and file names: most device names begin with a slash (/), except non-NFS network devices and VxWorks DOS devices (dosFs).

By convention, NFS-based network devices are *mounted* with names that begin with a slash. For example:

```
/usr
```

Non-NFS network devices are named with the remote machine name followed by a colon. For example:

```
host:
```

The remainder of the name is the file name in the remote directory on the remote system.

File system devices using dosFs are often named with uppercase letters and/or digits followed by a colon. For example:

```
DEV1:
```

File names and directory names on dosFs devices are often separated by backslashes (\). These can be used interchangeably with forward slashes (/).

⚠ **NOTE:** Because device names are recognized by the I/O system using simple substring matching, a slash (/) should not be used alone as a device name.

3.3 Basic I/O

Basic I/O is the lowest level of I/O in VxWorks. The basic I/O interface is source-compatible with the I/O primitives in the standard C library. There are seven basic I/O calls, shown in the following table.

Table 3-1 **Basic I/O Routines**

Call	Description
creat( )	Create a file.
remove( )	Remove a file.
open( )	Open a file. (Optionally, create a file.)
close( )	Close a file.
read( )	Read a previously created or opened file.
write( )	Write a previously created or opened file.
ioctl( )	Perform special control functions on files or devices.

3.3.1 File Descriptors

At the basic I/O level, files are referred to by a *file descriptor*, or *fd*. An *fd* is a small integer returned by a call to ***open( )*** or ***creat( )***. The other basic I/O calls take an *fd* as a parameter to specify the intended file. An *fd* has no meaning discernible to the user; it is only a handle for the I/O system.

When a file is opened, an *fd* is allocated and returned. When the file is closed, the *fd* is deallocated. There are a finite number of *fd*s available in VxWorks. To avoid exceeding the system limit, it is important to close *fd*s that are no longer in use. The number of available *fd*s is specified in the initialization of the I/O system.

3.3.2 Standard Input, Standard Output, and Standard Error

Three file descriptors are reserved and have special meanings:

0 = standard input
1 = standard output
2 = standard error output

These *fd*s are never returned as the result of an ***open( )*** or ***creat( )***, but serve rather as indirect references that can be redirected to any other open *fd*.

These standard *fd*s are used to make tasks and modules independent of their actual I/O assignments. If a module sends its output to standard output (*fd* = 1), then its output can be redirected to any file or device, without altering the module.

VxWorks allows two levels of redirection. First, there is a global assignment of the three standard *fd*s. Second, individual tasks can override the global assignment of these *fd*s with their own assignments that apply only to that task.

Global Redirection

When VxWorks is initialized, the global assignments of the standard *fd*s are directed, by default, to the system console. When tasks are spawned, they initially have no task-specific *fd* assignments; instead, they use the global assignments.

The global assignments can be redirected using ***ioGlobalStdSet( )***. The parameters to this routine are the global standard *fd* to be redirected, and the *fd* to direct it to.

For example, the following call sets global standard output (*fd* = 1) to be the open file with a file descriptor of **fileFd**:

```
ioGlobalStdSet (1, fileFd);
```

All tasks in the system that do not have their own task-specific redirection write standard output to that file thereafter. For example, the task **tRlogind** calls ***ioGlobalStdSet( )*** to redirect I/O across the network during an **rlogin** session.

Task-Specific Redirection

The assignments for a specific task can be redirected using the routine ***ioTaskStdSet( )***. The parameters to this routine are the task ID (0 = self) of the task with the assignments to be redirected, the standard *fd* to be redirected, and the *fd* to direct it to. For example, a task can make the following call to write standard output to **fileFd**:

```
ioTaskStdSet (0, 1, fileFd);
```

All other tasks are unaffected by this redirection, and subsequent global redirections of standard output do not affect this task.

3.3.3 Open and Close

Before I/O can be performed to a device, an *fd* must be opened to that device by invoking the ***open( )*** routine (or ***creat( )***, as discussed in the next section). The arguments to ***open( )*** are the file name, the type of access, and, when necessary, the mode:

```
fd = open ("name", flags, mode);
```

The possible access flags are shown in Table 3-2.

Table 3-2 **File Access Flags**

Flag	Hex Value	Description
O_RDONLY	0	Open for reading only.
O_WRONLY	1	Open for writing only.
O_RDWR	2	Open for reading and writing.
O_CREAT	200	Create a new file.
O_TRUNC	400	Truncate the file.

The *mode* parameter is used in the following special cases to specify the mode (permission bits) of a file or to create subdirectories:

- In general, you can open only preexisting devices and files with ***open( )***. However, with NFS network, dosFs, and rt11Fs devices, you can also create files with ***open( )*** by or'ing **O_CREAT** with one of the access flags. In the case of NFS devices, ***open( )*** requires the third parameter specifying the mode of the file:

  ```
  fd = open ("name", O_CREAT | O_RDWR, 0644);
  ```

- With both dosFs and NFS devices, you can use the **O_CREAT** option to create a subdirectory by setting *mode* to **FSTAT_DIR**. Other uses of the mode parameter with dosFs devices are ignored.

The ***open( )*** routine, if successful, returns an *fd* (a small integer). This *fd* is then used in subsequent I/O calls to specify that file. The *fd* is a *global* identifier that is *not* task specific. One task can open a file, and then any other tasks can use the resulting *fd* (for example, pipes). The *fd* remains valid until ***close( )*** is invoked with that *fd*:

```
close (fd);
```

At that point, I/O to the file is flushed (completely written out) and the *fd* can no longer be used by any task. However, the same *fd* number can again be assigned by the I/O system in any subsequent ***open( )***.

When a task exits or is deleted, the files opened by that task are not automatically closed, because *fd*s are not task specific. Thus, it is recommended that tasks explicitly close all files when they are no longer required. As stated previously, there is a limit to the number of files that can be open at one time.

3.3.4 Create and Remove

File-oriented devices must be able to create and remove files as well as open existing files. The ***creat( )*** routine directs a file-oriented device to make a new file on the device and return a file descriptor for it. The arguments to ***creat( )*** are similar to those of ***open( )*** except that the file name specifies the name of the new file rather than an existing one; the ***creat( )*** routine returns an *fd* identifying the new file.

```
fd = creat ("name", flag);
```

The ***remove( )*** routine removes a named file on a file-oriented device:

```
remove ("name");
```

Do not remove files while they are open.

With non-file-system oriented device names, ***creat( )*** acts exactly like ***open( )***; however, ***remove( )*** has no effect.

3.3.5 Read and Write

After an *fd* is obtained by invoking ***open( )*** or ***creat( )***, tasks can read bytes from a file with ***read( )*** and write bytes to a file with ***write( )***. The arguments to ***read( )*** are the *fd*, the address of the buffer to receive input, and the maximum number of bytes to read:

```
nBytes = read (fd, &buffer, maxBytes);
```

The ***read( )*** routine waits for input to be available from the specified file, and returns the number of bytes actually read. For file-system devices, if the number of bytes read is less than the number requested, a subsequent ***read( )*** returns 0 (zero), indicating end-of-file. For non-file-system devices, the number of bytes read can be less than the number requested even if more bytes are available; a subsequent ***read( )*** may or may not return 0. In the case of serial devices and TCP sockets, repeated calls to ***read( )*** are sometimes necessary to read a specific number of bytes. (See the reference entry for ***fioRead( )*** in **fioLib**). A return value of **ERROR** (-1) indicates an unsuccessful read.

The arguments to ***write( )*** are the *fd*, the address of the buffer that contains the data to be output, and the number of bytes to be written:

```
actualBytes = write (fd, &buffer, nBytes);
```

The ***write( )*** routine ensures that all specified data is at least queued for output before returning to the caller, though the data may not yet have been written to the device (this is driver dependent). ***write( )*** returns the number of bytes written; if the number returned is not equal to the number requested, an error has occurred.

3.3.6 File Truncation

It is sometimes convenient to discard part of the data in a file. After a file is open for writing, you can use the ***ftruncate( )*** routine to truncate a file to a specified size. Its arguments are an *fd* and the desired length of the file:

```
status = ftruncate (fd, length);
```

If it succeeds in truncating the file, ***ftruncate( )*** returns **OK**. If the size specified is larger than the actual size of the file, or if the *fd* refers to a device that cannot be truncated, ***ftruncate( )*** returns **ERROR**, and sets **errno** to **EINVAL**.

The ***ftruncate( )*** routine is part of the POSIX 1003.1b standard, but this implementation is only partially POSIX-compliant: creation and modification times are not updated. This call is supported only by **dosFsLib**, the DOS-compatible file system library.

3.3.7 I/O Control

The *ioctl( )* routine is an open-ended mechanism for performing any I/O functions that do not fit the other basic I/O calls. Examples include determining how many bytes are currently available for input, setting device-specific options, obtaining information about a file system, and positioning random-access files to specific byte positions. The arguments to the *ioctl( )* routine are the *fd*, a code that identifies the control function requested, and an optional function-dependent argument:

```
result = ioctl (fd, function, arg);
```

For example, the following call uses the **FIOBAUDRATE** function to set the baud rate of a *tty* device to 9600:

```
status = ioctl (fd, FIOBAUDRATE, 9600);
```

The discussion of specific devices in *3.7 Devices in VxWorks*, p.131 summarizes the *ioctl( )* functions available for each device. The *ioctl( )* control codes are defined in **ioLib.h**. For more information, see the reference entries for specific device drivers.

3.3.8 Pending on Multiple File Descriptors: The Select Facility

The VxWorks *select* facility provides a UNIX- and Windows-compatible method for pending on multiple file descriptors. The library **selectLib** provides both task-level support, allowing tasks to wait for multiple devices to become active, and device driver support, giving drivers the ability to detect tasks that are pended while waiting for I/O on the device. To use this facility, the header file **selectLib.h** must be included in your application code.

Task-level support not only gives tasks the ability to simultaneously wait for I/O on multiple devices, but it also allows tasks to specify the maximum time to wait for I/O to become available. For an example of using the select facility to pend on multiple file descriptors, consider a client-server model in which the server is servicing both local and remote clients. The server task uses a pipe to communicate with local clients and a socket to communicate with remote clients. The server task must respond to clients as quickly as possible. If the server blocks waiting for a request on only one of the communication streams, it cannot service requests that come in on the other stream until it gets a request on the first stream. For example, if the server blocks waiting for a request to arrive in the socket, it cannot service requests that arrive in the pipe until a request arrives in the socket to unblock it. This can delay local tasks waiting to get their requests serviced. The select facility solves this problem by giving the server task the ability to monitor both the socket and the pipe and service requests as they come in, regardless of the communication stream used.

Tasks can block until data becomes available or the device is ready for writing. The ***select( )*** routine returns when one or more file descriptors are ready or a timeout has occurred. Using the ***select( )*** routine, a task specifies the file descriptors on which to wait for activity. Bit fields are used in the ***select( )*** call to specify the read and write file descriptors of interest. When ***select( )*** returns, the bit fields are modified to reflect the file descriptors that have become available. The macros for building and manipulating these bit fields are listed in Table 3-3.

Table 3-3 **Select Macros**

Macro	Function
FD_ZERO	Zeros all bits.
FD_SET	Sets bit corresponding to a specified file descriptor.
FD_CLR	Clears a specified bit.
FD_ISSET	Returns 1 if specified bit is set, otherwise returns 0.

Applications can use ***select( )*** with any character I/O devices that provide support for this facility (for example, pipes, serial devices, and sockets). For information on writing a device driver that supports ***select( )***, see *Implementing select( )*, p.163.

Example 3-1 **The Select Facility**

```
/* selServer.c - select example
 * In this example, a server task uses two pipes: one for normal-priority
 * requests, the other for high-priority requests. The server opens both
 * pipes and blocks while waiting for data to be available in at least one
 * of the pipes.
 */

#include "vxWorks.h"
#include "selectLib.h"
#include "fcntl.h"

#define MAX_FDS 2
#define MAX_DATA 1024
#define PIPEHI  "/pipe/highPriority"
#define PIPENORM "/pipe/normalPriority"

/*******************************************************************************
* selServer - reads data as it becomes available from two different pipes
*
* Opens two pipe fds, reading from whichever becomes available. The
* server code assumes the pipes have been created from either another
* task or the shell. To test this code from the shell do the following:
*  -> ld < selServer.o
*  -> pipeDevCreate ("/pipe/highPriority", 5, 1024)
```

```
*  -> pipeDevCreate ("/pipe/normalPriority", 5, 1024)
*  -> fdHi = open ("/pipe/highPriority", 1, 0)
*  -> fdNorm = open ("/pipe/normalPriority", 1, 0)
*  -> iosFdShow
*  -> sp selServer
*  -> i
* At this point you should see selServer's state as pended. You can now
* write to either pipe to make the selServer display your message.
*  -> write fdNorm, "Howdy", 6
*  -> write fdHi, "Urgent", 7
*/
STATUS selServer (void)
  {
  struct fd_set readFds;       /* bit mask of fds to read from */
  int      fds[MAX_FDS];       /* array of fds on which to pend */
  int      width;             /* number of fds on which to pend */
  int      i;                 /* index for fd array */
  char     buffer[MAX_DATA]; /* buffer for data that is read */

  /* open file descriptors */
  if ((fds[0] = open (PIPEHI, O_RDONLY, 0)) == ERROR)
    return (ERROR);
  if ((fds[1] = open (PIPENORM, O_RDONLY, 0)) == ERROR)
    return (ERROR);

  /* loop forever reading data and servicing clients */
  FOREVER
    {
    /* clear bits in read bit mask */
    FD_ZERO (&readFds);

    /* initialize bit mask */
    FD_SET (fds[0], &readFds);
    FD_SET (fds[1], &readFds);
    width = (fds[0] > fds[1]) ? fds[0] : fds[1];
    width++;

    /* pend, waiting for one or more fds to become ready */
    if (select (width, &readFds, NULL, NULL, NULL) == ERROR)
      return (ERROR);

    /* step through array and read from fds that are ready */
    for (i=0; i< MAX_FDS; i++)
      {
      /* check if this fd has data to read */
      if (FD_ISSET (fds[i], &readFds))
        {
        /* typically read from fd now that it is ready */
        read (fds[i], buffer, MAX_DATA);
        /* normally service request, for this example print it */
        printf ("SELSERVER Reading from %s: %s\n",
            (i == 0) ? PIPEHI : PIPENORM, buffer);
        }
      }
    }
  }
```

3.4 Buffered I/O: Stdio

The VxWorks I/O library provides a buffered I/O package that is compatible with the UNIX and Windows *stdio* package and provides full ANSI C support. To include the *stdio* package in the VxWorks system, define **INCLUDE_ANSI_STDIO** in **configAll.h**.

Note that the implementation of ***printf( )***, ***sprintf( )***, and ***sscanf( )***, traditionally considered part of the *stdio* package, is part of a different package in VxWorks. These routines are discussed in *3.5 Other Formatted I/O*, p.121.

3.4.1 Using Stdio

Although the VxWorks I/O system is efficient, some overhead is associated with each low-level call. First, the I/O system must dispatch from the device-independent user call (***read( )***, ***write( )***, and so on) to the driver-specific routine for that function. Second, most drivers invoke a mutual exclusion or queuing mechanism to prevent simultaneous requests by multiple users from interfering with each other.

Because the VxWorks primitives are fast, this overhead is quite small. However, an application processing a single character at a time from a file incurs that overhead for each character if it reads each character with a separate ***read( )*** call:

```
n = read (fd, &char, 1);
```

To make this type of I/O more efficient and flexible, the *stdio* package implements a buffering scheme in which data is read and written in large chunks and buffered privately. This buffering is transparent to the application; it is handled automatically by the *stdio* routines and macros. To access a file with *stdio*, a file is opened with ***fopen( )*** instead of ***open( )*** (many *stdio* calls begin with the letter *f*):

```
fp = fopen ("/usr/foo", "r");
```

The returned value, a *file pointer* (or *fp*) is a handle for the opened file and its associated buffers and pointers. An *fp* is actually a pointer to the associated data structure of type **FILE** (that is, it is declared as **FILE** *). By contrast, the low-level I/O routines identify a file with a *file descriptor* (*fd*), which is a small integer. In fact, the **FILE** structure pointed to by the *fp* contains the underlying *fd* of the open file.

An already open *fd* can be associated belatedly with a **FILE** buffer by calling ***fdopen( )***:

```
fp = fdopen (fd, "r");
```

After a file is opened with *fopen( )*, data can be read with *fread( )*, or a character at a time with *getc( )*, and data can be written with *fwrite( )*, or a character at a time with *putc( )*.

The routines and macros to get data into or out of a file are extremely efficient. They access the buffer with direct pointers that are incremented as data is read or written by the user. They pause to call the low-level read or write routines only when a read buffer is empty or a write buffer is full.

WARNING: The *stdio* buffers and pointers are *private* to a particular task. They are *not* interlocked with semaphores or any other mutual exclusion mechanism, because this defeats the point of an efficient private buffering scheme. Therefore, multiple tasks must not perform I/O to the same *stdio* **FILE** pointer at the same time.

3.4.2 Standard Input, Standard Output, and Standard Error

As discussed earlier in *3.3 Basic I/O*, p.112, there are three special file descriptors (0, 1, and 2) reserved for standard input, standard output, and standard error. There are three corresponding *stdio* **FILE** buffers that are automatically created when required; they are then associated with those file descriptors: *stdin*, *stdout*, and *stderr*. These can be used to do buffered I/O to the standard *fd*s.

3.5 Other Formatted I/O

3.5.1 Special Cases: *printf( )*, *sprintf( )*, and *sscanf( )*

The routines *printf( )*, *sprintf( )*, and *sscanf( )* are generally considered to be part of the standard *stdio* package. However, the VxWorks implementation of these routines, while functionally the same, does not use the *stdio* package. Instead, it uses a self-contained, formatted, non-buffered interface to the I/O system in the library **fioLib**. Note that these routines provide the functionality specified by ANSI; however, *printf( )* is not buffered.

Because these routines are implemented in this way, the full *stdio* package, which is optional, can be omitted from a VxWorks configuration without sacrificing their

availability. Applications requiring *printf*-style output that is buffered can still accomplish this by calling ***fprintf( )*** explicitly to *stdout*.

While ***sscanf( )*** is implemented in **fioLib** and can be used even if *stdio* is omitted, the same is not true of ***scanf( )***, which is implemented in the usual way in *stdio*.

3.5.2 Additional Routines: *printErr( )* and *fdprintf( )*

Additional routines in **fioLib** provide formatted but unbuffered output. The routine ***printErr( )*** is analogous to ***printf( )*** but outputs formatted strings to the standard error *fd* (2). The routine ***fdprintf( )*** outputs formatted strings to a specified *fd*.

3.5.3 Message Logging

Another higher-level I/O facility is provided by the library **logLib**, which allows formatted messages to be logged without having to do I/O in the current task's context, or when there is no task context. The message format and parameters are sent on a message queue to a logging task, which then formats and outputs the message. This is useful when messages must be logged from interrupt level, or when it is desirable not to delay the current task for I/O or use the current task's stack for message formatting (which can take up significant stack space). The message is displayed on the console unless otherwise redirected at system startup using ***logInit( )*** or dynamically using ***logFdSet( )***.

3.6 Asynchronous Input/Output

Asynchronous Input/Output (AIO) is the ability to perform input and output operations concurrently with ordinary internal processing. AIO enables you to decouple I/O operations from the activities of a particular task when these are logically independent.

The benefit of AIO is greater processing efficiency: it permits I/O operations to take place whenever resources are available, rather than making them await arbitrary events such as the completion of independent operations. AIO eliminates some of the unnecessary blocking of tasks that is caused by ordinary synchronous

I/O; this decreases contention for resources between input/output and internal processing, and expedites throughput.

The VxWorks AIO implementation meets the specification in the POSIX 1003.1b standard. To include AIO in your VxWorks configuration, define **INCLUDE_POSIX_AIO** and **INCLUDE_POSIX_AIO_SYSDRV** in **configAll.h**. The second configuration constant enables the auxiliary AIO system driver, required for asynchronous I/O on all current VxWorks devices.

3.6.1 The POSIX AIO Routines

The VxWorks library **aioPxLib** provides the POSIX AIO routines. To access a file asynchronously, open it with the *open( )* routine, like any other file. Thereafter, use the file descriptor returned by *open( )* in calls to the AIO routines. The POSIX AIO routines (and two associated non-POSIX routines) are listed in Table 3-4.

Table 3-4 **Asynchronous Input/Output Routines**

Function	Description
aioPxLibInit( )	Initialize the AIO library (non-POSIX).
aioShow( )	Display the outstanding AIO requests (non-POSIX).*
aio_read( )	Initiate an asynchronous read operation.
aio_write( )	Initiate an asynchronous write operation.
aio_listio( )	Initiate a list of up to **LIO_MAX** asynchronous I/O requests.
aio_error( )	Retrieve the error status of an AIO operation.
aio_return( )	Retrieve the return status of a completed AIO operation.
aio_cancel( )	Cancel a previously submitted AIO operation.
aio_suspend( )	Wait until an AIO operation is done, interrupted, or timed out.

* This function is not built into the Tornado shell. To use it from the Tornado shell, you must define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. Configuration* in this manual. When you invoke the function, its output is sent to the standard output device.

The default VxWorks initialization code calls ***aioPxLibInit( )*** automatically when **INCLUDE_POSIX_AIO** is defined in **configAll.h**. This routine takes one parameter, the maximum number of ***lio_listio( )*** calls that can be outstanding at one time. By

default this parameter is MAX_LIO_CALLS (defined in **configAll.h**). When the parameter is 0, the default value is taken from AIO_CLUST_MAX (defined in **h/private/aioPxLibP.h**).

The AIO system driver, **aioSysDrv**, is initialized by default with the routine *aioSysInit*( ) when both INCLUDE_POSIX_AIO and INCLUDE_POSIX_AIO_SYSDRV are defined. The purpose of **aioSysDrv** is to provide request queues independent of any particular device driver, so that you can use any VxWorks device driver with AIO.

The routine *aioSysInit*( ) takes three parameters: the number of AIO system tasks to spawn, and the priority and stack size for these system tasks. The number of AIO system tasks spawned equals the number of AIO requests that can be handled in parallel. The default initialization call uses three constants, all defined in **configAll.h**:

```
aioSysInit( MAX_AIO_SYS_TASKS, AIO_TASK_PRIORITY, AIO_TASK_STACK_SIZE )
```

When any of the parameters passed to *aioSysInit*( ) is 0, the corresponding value is taken from AIO_IO_TASKS_DFLT, AIO_IO_PRIO_DFLT, and AIO_IO_STACK_DFLT (all defined in **h/aioSysDrv.h**).

Table 3-5 lists the names of the constants defined in **configAll.h** for initialization routines called from **usrConfig.c**. It also shows the constants used within initialization routines when the parameters are 0, and where these constants are defined.

Table 3-5 **AIO Initialization Functions and Related Constants**

Initialization Function	configAll.h Constant	Def. Value	Header File Constant used when arg = 0	Def. Value	Header File
aioPxLibInit( )	MAX_LIO_CALLS	0	AIO_CLUST_MAX	100	h/private/aioPxLibP.h
aioSysInit( )	MAX_AIO_SYS_TASKS	0	AIO_IO_TASKS_DFLT	2	h/aioSysDrv.h
	AIO_TASK_PRIORITY	0	AIO_IO_PRIO_DFLT	50	h/aioSysDrv.h
	AIO_TASK_STACK_SIZE	0	AIO_IO_STACK_DFLT	0x7000	h/aioSysDrv.h

3.6.2 AIO Control Block

Each of the AIO calls takes an AIO control block (**aiocb**) as an argument to describe the AIO operation. The calling routine must allocate space for the control block, which is associated with a single AIO operation. No two concurrent AIO

operations can use the same control block; an attempt to do so yields undefined results.

The **aiocb** and the data buffers it references are used by the system while performing the associated request. Therefore, after you request an AIO operation, you must not modify the corresponding **aiocb** before calling *aio_return( )*; this function frees the **aiocb** for modification or reuse.

NOTE: If a routine allocates stack space for the **aiocb**, that routine must call *aio_return( )* to free the **aiocb** before returning.

The **aiocb** structure is defined in **aio.h**. It contains the following fields:

aio_fildes	file descriptor for I/O
aio_offset	offset from the beginning of the file
aio_buf	address of the buffer from/to which AIO is requested
aio_nbytes	number of bytes to read or write
aio_reqprio	priority reduction for this AIO request
aio_sigevent	signal to return on completion of an operation (optional)
aio_lio_opcode	operation to be performed by a *lio_listio( )* call
aio_sys	VxWorks-specific data (non-POSIX)

For full definitions and important additional information, see the reference entry for **aioPxLib**.

3.6.3 Using AIO

The routines *aio_read( )*, *aio_write( )*, or *lio_listio( )* initiate AIO operations. The last of these, *lio_listio( )*, allows you to submit a number of asynchronous requests (read and/or write) at one time. In general, the actual I/O (reads and writes) initiated by these routines does not happen immediately after the AIO request. For that reason, their return values do not reflect the outcome of the actual I/O operation, but only whether a request is successful—that is, whether the AIO routine is able to put the operation on a queue for eventual execution.

After the I/O operations themselves execute, they also generate return values that reflect the success or failure of the I/O. There are two routines that you can use to get information about the success or failure of the I/O operation: *aio_error( )* and *aio_return( )*. You can use *aio_error( )* to get the status of an AIO operation

(success, failure, or in progress), and *aio_return( )* to obtain the return values from the individual I/O operations. Until an AIO operation completes, its error status is **EINPROGRESS**. To cancel an AIO operation, call *aio_cancel( )*.

AIO with Periodic Checks for Completion

The following code uses a pipe for the asynchronous I/O operations. The example creates the pipe, submits an AIO read request, verifies that the read request is still in progress, and submits an AIO write request. Under normal circumstances, a synchronous read to an empty pipe blocks and the task does not execute the write, but in the case of AIO, we initiate the read request and continue. After the write request is submitted, the example task loops, checking the status of the AIO requests periodically until both the read and write complete. Because the AIO control blocks are on the stack, we must call *aio_return( )* before returning from *aioExample( )*.

Example 3-2 **Asynchronous I/O**

```
/* aioEx.c - example code for using asynchronous I/O */

/* includes */

#include "vxWorks.h"
#include "aio.h"
#include "errno.h"

/* defines */

#define BUFFER_SIZE 200

/*******************************************************************************
*
* aioExample - use AIO library
*
* This example shows the basic functions of the AIO library.
*
* RETURNS: OK if successful, otherwise ERROR.
*/

STATUS aioExample (void)
  {
  int       fd;
  static char   exFile [] = "/pipe/1stPipe";
  struct aiocb  aiocb_read; /* read aiocb */
  struct aiocb  aiocb_write; /* write aiocb */
  static char *  test_string = "testing 1 2 3";
  char      buffer [BUFFER_SIZE]; /* buffer for read aiocb */

  pipeDevCreate (exFile, 50, 100);
```

```
if ((fd = open (exFile, O_CREAT | O_TRUNC | O_RDWR, 0666)) ==
  ERROR)
{
printf ("aioExample: cannot open %s errno 0x%x\n", exFile, errno);
return (ERROR);
}

printf ("aioExample: Example file = %s\tFile descriptor = %d\n",
    exFile, fd);

/* initialize read and write aiocbs */
bzero ((char *) &aiocb_read, sizeof (struct aiocb));
bzero ((char *) buffer, sizeof (buffer));
aiocb_read.aio_fildes = fd;
aiocb_read.aio_buf = buffer;
aiocb_read.aio_nbytes = BUFFER_SIZE;
aiocb_read.aio_reqprio = 0;

bzero ((char *) &aiocb_write, sizeof (struct aiocb));
aiocb_write.aio_fildes = fd;
aiocb_write.aio_buf = test_string;
aiocb_write.aio_nbytes = strlen (test_string);
aiocb_write.aio_reqprio = 0;

/* initiate the read */
if (aio_read (&aiocb_read) == -1)
  printf ("aioExample: aio_read failed\n");

/* verify that it is in progress */
if (aio_error (&aiocb_read) == EINPROGRESS)
  printf ("aioExample: read is still in progress\n");

/* write to pipe - the read should be able to complete */
printf ("aioExample: getting ready to initiate the write\n");
if (aio_write (&aiocb_write) == -1)
  printf ("aioExample: aio_write failed\n");

/* wait til both read and write are complete */
while ((aio_error (&aiocb_read) == EINPROGRESS) ||
    (aio_error (&aiocb_write) == EINPROGRESS))
  taskDelay (1);

/* print out what was read */
printf ("aioExample: message = %s\n", buffer);

/* clean up */
if (aio_return (&aiocb_read) == -1)
  printf ("aioExample: aio_return for aiocb_read failed\n");
if (aio_return (&aiocb_write) == -1)
  printf ("aioExample: aio_return for aiocb_write failed\n");

close (fd);
return (OK);
}
```

Alternatives for Testing AIO Completion

A task can determine whether an AIO request is complete in any of the following ways:

- Check the result of *aio_error( )* periodically, as in the previous example, until the status of an AIO request is no longer **EINPROGRESS**.
- Use *aio_suspend( )* to suspend the task until the AIO request is complete.
- Use signals to be informed when the AIO request is complete.

The following example is similar to the preceding *aioExample( )*, except that it uses signals to be notified when the write is complete. If you test this from the shell, spawn the routine to run at a lower priority than the AIO system tasks to assure that the test routine does not block completion of the AIO request. (For details on the shell, see the *Tornado User's Guide: Shell*.)

Example 3-3 **Asynchronous I/O with Signals**

```
/* aioExSig.c - example code for using signals with asynchronous I/O */

/* includes */

#include "vxWorks.h"
#include "aio.h"
#include "errno.h"

/* defines */

#define BUFFER_SIZE   200
#define LIST_SIZE     1
#define EXAMPLE_SIG_NO  25 /* signal number */

/* forward declarations */

void mySigHandler (int sig, struct siginfo * info, void * pContext);

/*******************************************************************************
*
* aioExampleSig - use AIO library.
*
* This example shows the basic functions of the AIO library.
* Note if this is run from the shell it must be spawned. Use:
*  -> sp aioExampleSig
*
* RETURNS: OK if successful, otherwise ERROR.
*/

STATUS aioExampleSig (void)
  {
  int           fd;
```

```
static char       exFile [] = "/pipe/1stPipe";
struct aiocb      aiocb_read;      /* read aiocb */
static struct aiocb   aiocb_write;      /* write aiocb */
struct sigaction    action;        /* signal info */
static char *      test_string = "testing 1 2 3";
char          buffer [BUFFER_SIZE]; /* aiocb read buffer */

pipeDevCreate (exFile, 50, 100);

if ((fd = open (exFile, O_CREAT | O_TRUNC| O_RDWR, 0666)) == ERROR)
  {
  printf ("aioExample: cannot open %s errno 0x%x\n", exFile, errno);
  return (ERROR);
  }

printf ("aioExampleSig: Example file = %s\tFile descriptor = %d\n",
    exFile, fd);

/* set up signal handler for EXAMPLE_SIG_NO */

action.sa_sigaction = mySigHandler;
action.sa_flags = SA_SIGINFO;
sigemptyset (&action.sa_mask);
sigaction (EXAMPLE_SIG_NO, &action, NULL);

/* initialize read and write aiocbs */

bzero ((char *) &aiocb_read, sizeof (struct aiocb));
bzero ((char *) buffer, sizeof (buffer));
aiocb_read.aio_fildes = fd;
aiocb_read.aio_buf = buffer;
aiocb_read.aio_nbytes = BUFFER_SIZE;
aiocb_read.aio_reqprio = 0;

bzero ((char *) &aiocb_write, sizeof (struct aiocb));
aiocb_write.aio_fildes = fd;
aiocb_write.aio_buf = test_string;
aiocb_write.aio_nbytes = strlen (test_string);
aiocb_write.aio_reqprio = 0;

/* set up signal info */

aiocb_write.aio_sigevent.sigev_signo = EXAMPLE_SIG_NO;
aiocb_write.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
aiocb_write.aio_sigevent.sigev_value.sival_ptr =
                    (void *) &aiocb_write;

/* initiate the read */

if (aio_read (&aiocb_read) == -1)
  printf ("aioExampleSig: aio_read failed\n");

/* verify that it is in progress */

if (aio_error (&aiocb_read) == EINPROGRESS)
  printf ("aioExampleSig: read is still in progress\n");
```

```
  /* write to pipe - the read should be able to complete */

  printf ("aioExampleSig: getting ready to initiate the write\n");
  if (aio_write (&aiocb_write) == -1)
    printf ("aioExampleSig: aio_write failed\n");

  /* clean up */

  if (aio_return (&aiocb_read) == -1)
    printf ("aioExampleSig: aio_return for aiocb_read failed\n");
  else
    printf ("aioExampleSig: aio read message = %s\n",
         aiocb_read.aio_buf);

  close (fd);
  return (OK);
  }

void mySigHandler
  (
  int         sig,
  struct siginfo * info,
  void *      pContext
  )

  {
  /* print out what was read */

  printf ("mySigHandler: Got signal for aio write\n");

  /* write is complete so let's do cleanup for it here */

  if (aio_return (info->si_value.sival_ptr) == -1)
    {
    printf ("mySigHandler: aio_return for aiocb_write failed\n");
    printErrno (0);
    }
  }
```

3.7 Devices in VxWorks

The VxWorks I/O system is flexible, allowing specific device drivers to handle the seven I/O functions. All VxWorks device drivers follow the basic conventions outlined previously, but differ in specifics; this section describes those specifics.

Table 3-6 **Drivers Provided with VxWorks**

Module	Driver Description
ttyDrv	Terminal driver
ptyDrv	Pseudo-terminal driver
pipeDrv	Pipe driver
memDrv	Pseudo memory device driver
nfsDrv	NFS client driver
netDrv	Network driver for remote file access
ramDrv	RAM driver for creating a RAM disk
scsiLib	SCSI interface library
-	Other hardware-specific drivers

3.7.1 Serial I/O Devices (Terminal and Pseudo-Terminal Devices)

VxWorks provides terminal and pseudo-terminal device drivers (*tty* and *pty* drivers). The *tty* driver is for actual terminals; the *pty* driver is for processes that simulate terminals. These pseudo terminals are useful in applications such as remote login facilities.[1]

VxWorks serial I/O devices are buffered serial byte streams. Each device has a ring buffer (circular buffer) for both input and output. Reading from a *tty* device extracts bytes from the input ring. Writing to a *tty* device adds bytes to the output ring. The size of each ring buffer is specified when the device is created during system initialization.

1. For the remainder of this section, the term *tty* is used to indicate both *tty* and *pty* devices.

Tty Options

The *tty* devices have a full range of options that affect the behavior of the device. These options are selected by setting bits in the device option word using the ***ioctl( )*** routine with the **FIOSETOPTIONS** function (see *I/O Control Functions*, p.134). For example, to set all the *tty* options except **OPT_MON_TRAP**:

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL & ~OPT_MON_TRAP);
```

Table 3-7 is a summary of the available options. The listed names are defined in the header file **ioLib.h**. For more detailed information, see the reference entry for **tyLib**.

Table 3-7 **Tty Options**

Library	Description
OPT_LINE	Select *line mode*. (See *Raw Mode and Line Mode*, p.132.)
OPT_ECHO	Echo input characters to the output of the same channel.
OPT_CRMOD	Translate input **RETURN** characters into **NEWLINE** (\n); translate output **NEWLINE** into **RETURN-LINEFEED**.
OPT_TANDEM	Respond to X-on/X-off protocol (**CTRL+Q** and **CTRL+S**).
OPT_7_BIT	Strip the most significant bit from all input bytes.
OPT_MON_TRAP	Enable the special *ROM monitor trap* character, **CTRL+X** by default.
OPT_ABORT	Enable the special *target shell abort* character, **CTRL+C** by default. (Only useful if the target shell is configured into the system; see *9. Target Shell* in this manual for details.)
OPT_TERMINAL	Set all of the above option bits.
OPT_RAW	Set none of the above option bits.

Raw Mode and Line Mode

A *tty* device operates in one of two modes: *raw mode* (unbuffered) or *line mode*. Raw mode is the default. Line mode is selected by the **OPT_LINE** bit of the device option word (see *Tty Options*, p.132).

In *raw mode*, each input character is available to readers as soon as it is input from the device. Reading from a *tty* device in raw mode causes as many characters as

possible to be extracted from the input ring, up to the limit of the user's read buffer. Input cannot be modified except as directed by other *tty* option bits.

In *line mode*, all input characters are saved until a **NEWLINE** character is input; then the entire line of characters, including the **NEWLINE**, is made available in the ring at one time. Reading from a *tty* device in line mode causes characters up to the end of the next line to be extracted from the input ring, up to the limit of the user's read buffer. Input can be modified by the special characters **CTRL+H** (backspace), **CTRL+U** (line-delete), and **CTRL+D** (end-of-file), which are discussed in *Tty Special Characters*, p.133.

Tty Special Characters

The following special characters are enabled if the *tty* device operates in line mode, that is, with the **OPT_LINE** bit set:

- The backspace character, by default **CTRL+H**, causes successive previous characters to be deleted from the current line, up to the start of the line. It does this by echoing a backspace followed by a space, and then another backspace.
- The line-delete character, by default **CTRL+U**, deletes all the characters of the current line.
- The end-of-file (EOF) character, by default **CTRL+D**, causes the current line to become available in the input ring without a **NEWLINE** and without entering the EOF character itself. Thus if the EOF character is the first character typed on a line, reading that line returns a zero byte count, which is the usual indication of end-of-file.

The following characters have special effects if the *tty* device is operating with the corresponding option bit set:

- The flow control characters, **CTRL+Q** and **CTRL+S**, commonly known as *X-on/X-off protocol*. Receipt of a **CTRL+S** input character suspends output to that channel. Subsequent receipt of a **CTRL+Q** resumes the output. Conversely, when the VxWorks input buffer is almost full, a **CTRL+S** is output to signal the other side to suspend transmission. When the input buffer is empty enough, a **CTRL+Q** is output to signal the other side to resume transmission. X-on/X-off protocol is enabled by **OPT_TANDEM**.
- The *ROM monitor trap* character, by default **CTRL+X**. This character traps to the ROM-resident monitor program. Note that this is drastic. All normal VxWorks functioning is suspended, and the computer system is controlled entirely by the monitor. Depending on the particular monitor, it may or may not be

possible to restart VxWorks from the point of interruption. The monitor trap character is enabled by **OPT_MON_TRAP**.

- The special *target shell abort* character, by default **CTRL+C**. This character restarts the target shell if it gets stuck in an unfriendly routine, such as one that has taken an unavailable semaphore or is caught in an infinite loop. The target shell abort character is enabled by **OPT_ABORT**.

The characters for most of these functions can be changed using the **tyLib** routines shown in Table 3-8.

Table 3-8 **Tty Special Characters**

Character	Description	Modifier
CTRL+H	backspace (character delete)	***tyBackspaceSet( )***
CTRL+U	line delete	***tyDeleteLineSet( )***
CTRL+D	EOF (end of file)	***tyEOFSet( )***
CTRL+C	target shell abort	***tyAbortSet( )***
CTRL+X	trap to boot ROMs	***tyMonitorTrapSet( )***
CTRL+S	output suspend	N/A
CTRL+Q	output resume	N/A

I/O Control Functions

The *tty* devices respond to the ***ioctl( )*** functions in Table 3-9, defined in **ioLib.h**. For more information, see the reference entries for **tyLib**, **ttyDrv**, and ***ioctl( )***.

NOTE: To change the driver's hardware options (for example, the number of stop bits or parity bits), use the ***ioctl( )*** function **SIO_HW_OPTS_SET**. Because this command is not implemented in most drivers, you may need to add it to your BSP serial driver, which resides in **src/drv/sio**. The details of how to implement this command depend on your board's serial chip. The constants defined in the header file **h/sioLib.h** provide the POSIX definitions for setting the hardware options.

Table 3-9 **I/O Control Functions Supported by tyLib**

Function	Description
FIOBAUDRATE	Set the baud rate to the specified argument.
FIOCANCEL	Cancel a read or write.
FIOFLUSH	Discard all bytes in the input and output buffers.
FIOGETNAME	Get the file name of the *fd*.
FIOGETOPTIONS	Return the current device option word.
FIONREAD	Get the number of unread bytes in the input buffer.
FIONWRITE	Get the number of bytes in the output buffer.
FIOSETOPTIONS	Set the device option word.

3.7.2 Pipe Devices

Pipes are virtual devices by which tasks communicate with each other through the I/O system. Tasks write messages to pipes; these messages can then be read by other tasks. Pipe devices are managed by **pipeDrv** and use the kernel message queue facility to bear the actual message traffic.

Creating Pipes

Pipes are created by calling the pipe create routine:

```
status = pipeDevCreate ("/pipe/name", maxMsgs, maxLength);
```

The new pipe can have at most *maxMsgs* messages queued at a time. Tasks that write to a pipe that already has the maximum number of messages queued are delayed until a message is dequeued. Each message in the pipe can be at most *maxLength* bytes long; attempts to write longer messages result in an error.

Writing to Pipes from ISRs

VxWorks pipes are designed to allow ISRs to write to pipes in the same way as task-level code. Many VxWorks facilities cannot be used from ISRs, including I/O to devices other than pipes. However, ISRs can use pipes to communicate with tasks, which can then invoke such facilities.

ISRs write to a pipe using the *write( )* call. Tasks and ISRs can write to the same pipes. However, if the pipe is full, the message is discarded because the ISRs cannot pend. ISRs must not invoke any I/O function on pipes other than *write( )*.

I/O Control Functions

Pipe devices respond to the *ioctl( )* functions summarized in Table 3-10. The functions listed are defined in the header file **ioLib.h**. For more information, see the reference entries for **pipeDrv** and for *ioctl( )* in **ioLib**.

Table 3-10 **I/O Control Functions Supported by pipeDrv**

Function	Description
FIOFLUSH	Discard all messages in the pipe.
FIOGETNAME	Get the pipe name of the *fd*.
FIONMSGS	Get the number of messages remaining in the pipe.
FIONREAD	Get the size in bytes of the first message in the pipe.

3.7.3 Pseudo Memory Devices

The **memDrv** driver allows the I/O system to access memory directly as a pseudo-I/O device. Memory location and size are specified when the device is created. This feature is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs. This driver does not implement a file system as does **ramDrv**. The **ramDrv** driver must be given memory over which it has absolute control; whereas **memDrv** provides a high-level method of reading and writing bytes in absolute memory locations through I/O calls.

Installing the Memory Driver

The driver is first initialized and then the device is created:

```
STATUS memDrv
    (void)
STATUS memDevCreate
    (char * name, char * base, int length)
```

Memory for the device is an absolute memory location beginning at *base*. The *length* parameter indicates the size of the memory. For additional information on the memory driver, see the reference entries for **memDrv**, ***memDevCreate( )***, and ***memDrv( )***.

I/O Control Functions

The memory driver responds to the ***ioctl( )*** functions summarized in Table 3-11. The functions listed are defined in the header file **ioLib.h**. For more information, see the reference entries for **memDrv** and for ***ioctl( )*** in **ioLib**.

Table 3-11 **I/O Control Functions Supported by memDrv**

Function	Description
FIOSEEK	Set the current byte offset in the file.
FIOWHERE	Return the current byte position in the file.

3.7.4 Network File System (NFS) Devices

Network File System (NFS) devices allow files on remote hosts to be accessed with the NFS protocol. The NFS protocol specifies both *client* software, to read files from remote machines, and *server* software, to export files to remote machines.

The driver **nfsDrv** acts as a VxWorks NFS client to access files on any NFS server on the network. VxWorks also allows you to run an NFS server to export files to other systems; see *Allowing Remote Access to VxWorks Files through NFS*, p.288 in this manual.

Using NFS devices, you can create, open, and access remote files exactly as though they were on a file system on a local disk. This is called *network transparency.*

Mounting a Remote NFS File System from VxWorks

Access to a remote NFS file system is established by mounting that file system locally and creating an I/O device for it using ***nfsMount( )***. Its arguments are (1) the host name of the NFS server, (2) the name of the host file system, and (3) the local name for the file system.

For example, the following call mounts **/usr** of the host **mars** as **/vxusr** locally:

```
nfsMount ("mars", "/usr", "/vxusr");
```

This creates a VxWorks I/O device with the specified local name (**/vxusr**, in this example). If the local name is specified as **NULL**, the local name is the same as the remote name.

After a remote file system is mounted, the files are accessed as though the file system were local. Thus, after the previous example, opening the file **/vxusr/foo** opens the file **/usr/foo** on the host **mars**.

The remote file system must be *exported* by the system on which it actually resides. However, NFS servers can export only local file systems. Use the appropriate command on the server to see which file systems are local. NFS requires *authentication* parameters to identify the user making the remote access. To set these parameters, use the routines ***nfsAuthUnixSet( )*** and ***nfsAuthUnixPrompt( )***.

Define **INCLUDE_NFS** in **configAll.h** to include NFS client support in your VxWorks configuration.

The subject of exporting and mounting NFS file systems and authenticating access permissions is discussed in more detail in *Transparent Remote File Access with NFS*, p.286. See also the reference entries **nfsLib** and **nfsDrv**, and the NFS documentation from Sun Microsystems.

I/O Control Functions for NFS Clients

NFS client devices respond to the ***ioctl( )*** functions summarized in Table 3-12. The functions listed are defined in **ioLib.h**. For more information, see the reference entries for **nfsDrv** and for ***ioctl( )*** in **ioLib**.

3.7.5 Non-NFS Network Devices

VxWorks also supports network access to files on the remote host through the Remote Shell protocol (RSH) or the File Transfer Protocol (FTP). These implementations of network devices use the driver **netDrv**. When a remote file is opened using RSH or FTP, the entire file is copied into local memory. As a result, the largest file that can be opened is restricted by the available memory. Read and write operations are performed on the in-memory copy of the file. When closed, the file is copied back to the original remote file if it was modified.

Table 3-12 **I/O Control Functions Supported by nfsDrv**

Function	Description
FIOFSTATGET	Get file status information (directory entry data).
FIOGETNAME	Get the file name of the *fd*.
FIONREAD	Get the number of unread bytes in the file.
FIOREADDIR	Read the next directory entry.
FIOSEEK	Set the current byte offset in the file.
FIOSYNC	Flush data to a remote NFS file.
FIOWHERE	Return the current byte position in the file.

In general, NFS devices are preferable to RSH and FTP devices for performance and flexibility, because NFS does not copy the entire file into local memory. However, NFS is not supported by all host systems.

Creating Network Devices

To access files on a remote host using either RSH or FTP, a network device must first be created by calling the routine ***netDevCreate( )***. The arguments to ***netDevCreate( )*** are (1) the name of the device, (2) the name of the host the device accesses, and (3) which protocol to use: 0 (RSH) or 1 (FTP).

For example, the following call creates an RSH device called **mars:** that accesses the host **mars**. By convention, the name for a network device is the remote machine's name followed by a colon (**:**).

```
netDevCreate ("mars:", "mars", 0);
```

Files on a network device can be created, opened, and manipulated as if on a local disk. Thus, opening the file **mars:/usr/foo** actually opens **/usr/foo** on host **mars**.

Note that creating a network device allows access to any file or device on the remote system, while mounting an NFS file system allows access only to a specified file system.

For the files of a remote host to be accessible with RSH or FTP, permissions and user identification must be established on both the remote and local systems. Creating and configuring network devices is discussed in detail in *Transparent Remote File Access with RSH and FTP*, p.283 and in the reference entry for **netDrv**.

I/O Control Functions

RSH and FTP devices respond to the same ***ioctl( )*** functions as NFS devices except for **FIOSYNC** and **FIOREADDIR**. The functions are defined in the header file **ioLib.h**. For more information, see the reference entries for **netDrv** and ***ioctl( )***.

3.7.6 Block Devices

A *block device* is a device that is organized as a sequence of individually accessible blocks of data. The most common type of block device is a disk. In VxWorks, the term *block* refers to the smallest addressable unit on the device. For most disk devices, a VxWorks block corresponds to a *sector*, although terminology varies.

Block devices in VxWorks have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, block device support consists of low-level drivers that interact with a file system. The file system, in turn, interacts with the I/O system. This arrangement allows a single low-level driver to be used with various different file systems and reduces the number of I/O functions that must be supported in the driver. The internal implementation of low-level drivers for block devices is discussed in *3.9.4 Block Devices*, p.171.

File Systems

For use with block devices, VxWorks is supplied with file system libraries compatible with the MS-DOS (dosFs) and RT-11 (rt11Fs) file systems. In addition, there is a library for a simple raw disk file system (rawFs), which treats an entire disk much like a single large file. Also supplied is a file system that supports SCSI tape devices, which are organized so that individual blocks of data are read and written sequentially. Use of these file systems is discussed in *4. Local File Systems* in this manual. Also see the reference entries for **dosFsLib**, **rt11FsLib**, **rawFsLib**, and **tapeFsLib**.

RAM Disk Drivers

RAM drivers, as implemented in **ramDrv**, emulate disk devices but actually keep all data in memory. Memory location and "disk" size are specified when a RAM device is created by calling ***ramDevCreate( )***. This routine can be called repeatedly to create multiple RAM disks.

Memory for the RAM disk can be preallocated and the address passed to *ramDevCreate( )*, or memory can be automatically allocated from the system memory pool using *malloc( )*.

After the device is created, a name and file system (dosFs, rt11Fs, or rawFs) must be associated with it using the file system's device initialization routine or file system's make routine, for example, *dosFsDevInit( )* or *dosFsMkfs( )*. Information describing the device is passed to the file system in a **BLK_DEV** structure. A pointer to this structure is returned by the RAM disk creation routine.

In the following example, a 200KB RAM disk is created with automatically allocated memory, 512-byte sections, a single track, and no sector offset. The device is assigned the name **DEV1:** and initialized for use with dosFs.

```
BLK_DEV                 *pBlkDev;
DOS_VOL_DESC            *pVolDesc;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
pVolDesc = dosFsMkfs ("DEV1:", pBlkDev);
```

The *dosFsMkfs( )* routine calls *dosFsDevInit( )* with default parameters and initializes the file system on the disk by calling *ioctl( )* with the **FIODISKINIT**.

If the RAM disk memory already contains a disk image, the first argument to *ramDevCreate( )* is the address in memory, and the formatting arguments must be identical to those used when the image was created. For example:

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, NULL);
```

In this case, *dosFsDevInit( )* must be used instead, because the file system already exists on the disk and does not require re-initialization. This procedure is useful if a RAM disk is to be created at the same address used in a previous boot of VxWorks. The contents of the RAM disk are then preserved. Creating a RAM disk with rt11Fs using *rt11FsMkfs( )* and *rt11FsDevInit( )* follows these same procedures. For more information on RAM disk drivers, see the reference entry for **ramDrv**. For more information on file systems, see *4. Local File Systems*.

SCSI Drivers

SCSI is a standard peripheral interface that allows connection with a wide variety of hard disks, optical disks, floppy disks, and tape drives. SCSI block drivers are compatible with the dosFs and rt11Fs libraries, and offer several advantages for target configurations. They provide:

- local mass storage in non-networked environments
- faster I/O throughput than Ethernet networks

The SCSI-2 support in VxWorks supersedes previous SCSI support, although it offers the option of configuring the original SCSI functionality, now known as SCSI-1. With SCSI-2 enabled, the VxWorks environment can still handle SCSI-1 applications, such as file systems created under SCSI-1. However, applications that directly used SCSI-1 data structures defined in **scsiLib.h** may require modifications and recompilation for SCSI-2 compatibility.

The VxWorks SCSI implementation consists of two modules, one for the device-independent SCSI interface and one to support a specific SCSI controller. The **scsiLib** library provides routines that support the device-independent interface; device-specific libraries provide configuration routines that support specific controllers (for example, **wd33c93Lib** for the Western Digital WD33C93 device or **mb87030Lib** for the Fujitsu MB87030 device). There are also additional support routines for individual targets in **sysLib.c**.

Configuring SCSI Drivers

Constants associated with SCSI drivers are listed in Table 3-13. Define these in **config.h**.

Table 3-13 **SCSI Constants**

Constant	Description
INCLUDE_SCSI	Include SCSI interface.
INCLUDE_SCSI2	SCSI-2 extensions.
INCLUDE_SCSI_DMA	Enable DMA for SCSI.
INCLUDE_SCSI_BOOT	Allow booting from a SCSI device.
SCSI_AUTO_CONFIG	Auto-configure and locate all targets on a SCSI bus.
INCLUDE_DOSFS	Include the DOS file system.
INCLUDE_TAPEFS	Include the tape file system.

To enable SCSI functionality, define **INCLUDE_SCSI** in **config.h**. This enables SCSI-1. To enable SCSI-2, you must define, in addition to SCSI-1, the constants **INCLUDE_SCSI2** and (if you plan to use SCSI tape support) **INCLUDE_TAPEFS**. To enable automatic configuration of drivers, define **SCSI_AUTO_CONFIG** in **config.h**.

⚠ **NOTE:** Including SCSI-2 in your VxWorks image can significantly increase the image size. 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**, be sure to see *8.4.1 Scaling Down VxWorks*, p.447 and *Creating Bootable Applications* in the *Tornado User's Guide: Cross-Development* for information on how to resolve the problem.

Configuring the SCSI Bus ID

Each board in a SCSI-2 environment must define a unique SCSI bus ID for the SCSI initiator. SCSI-1 drivers, which support only a single initiator at a time, assume an initiator SCSI bus ID of 7. However, SCSI-2 supports multiple initiators, up to eight initiators and targets at one time. Therefore, to ensure a unique ID, choose a value in the range 0-7 to be passed as a parameter to the driver's initialization routine (for example, ***ncr710CtrlInitScsi2( )***) by the ***sysScsiInit( )*** routine in **sysScsi.c**. For more information, see the reference entry for the relevant driver initialization routine. If there are multiple boards on one SCSI bus, and all of these boards use the same BSP, then different versions of the BSP must be compiled for each board by assigning unique SCSI bus IDs.

ROM Size Adjustment for SCSI Boot

If **INCLUDE_SCSI_BOOT** is defined in **config.h**, larger ROMs may be required for some boards. If this is the case, the definition of **ROM_SIZE** in **Makefile** and in **config.h** should be changed to a size that suits the capabilities of the target hardware.

Structure of the SCSI Subsystem

The SCSI subsystem supports libraries and drivers for both SCSI-1 and SCSI-2. It consists of the following six libraries which are independent of any SCSI controller:

scsiLib	routines that provide the mechanism for switching SCSI requests to either the SCSI-1 library (**scsi1Lib**) or the SCSI-2 library (**scsi2Lib**), as configured by the board support package (BSP).
scsi1Lib	SCSI-1 library routines and interface, used when only **INCLUDE_SCSI** is defined (see *Configuring SCSI Drivers*, p.142.)
scsi2Lib	SCSI-2 library routines and all physical device creation and deletion routines.
scsiCommonLib	commands common to all types of SCSI devices.
scsiDirectLib	routines and commands for direct access devices (disks).

scsiSeqLib routines and commands for sequential access block devices (tapes).

Controller-independent support for the SCSI-2 functionality is divided into **scsi2Lib**, **scsiCommonLib**, **scsiDirectLib**, and **scsiSeqLib**. The interface to any of these SCSI-2 libraries can be accessed directly. However, **scsiSeqLib** is designed to be used in conjunction with tapeFs, while **scsiDirectLib** works with dosFs, rt11Fs, and rawFs. Applications written for SCSI-1 can be used with SCSI-2; however, SCSI-1 device drivers cannot.

VxWorks targets using SCSI interface controllers require a controller-specific device driver. These device drivers work in conjunction with the controller-independent SCSI libraries, and they provide controller configuration and initialization routines contained in controller-specific libraries. For example, the Western Digital WD33C93 SCSI controller is supported by the device driver libraries **wd33c93Lib**, **wd33c93Lib1**, and **wd33c93Lib2**. Routines tied to SCSI-1 (such as *wd33c93CtrlCreate*( )) and SCSI-2 (such as *wd33c93CtrlCreateScsi2*( )) are segregated into separate libraries to simplify configuration. There are also additional support routines for individual targets in **sysLib.c**.

Booting and Initialization

To boot from a SCSI device, see *4.2.21 Booting from a Local dosFs File System Using SCSI*, p.218.

After VxWorks is built with SCSI support, the system startup code initializes the SCSI interface by executing *sysScsiInit*( ) and *usrScsiConfig*( ) when the constant **INCLUDE_SCSI** is defined. The call to *sysScsiInit*( ) initializes the SCSI controller and sets up interrupt handling. The physical device configuration is specified in *usrScsiConfig*( ), which is in **src/config/usrScsi.c**. The routine contains an example of the calling sequence to declare a hypothetical configuration, including:

- definition of physical devices with *scsiPhysDevCreate*( )
- creation of logical partitions with *scsiBlkDevCreate*( )
- specification of a file system with either *dosFsDevInit*( ) or *rt11FsDevInit*( )

If you are not using **SCSI_AUTO_CONFIG**, modify *usrScsiConfig*( ) to reflect your actual configuration. For more information on the calls used in this routine, see the reference entries for *scsiPhysDevCreate*( ), *scsiBlkDevCreate*( ), *dosFsDevInit*( ), *rt11FsDevInit*( ), *dosFsMkfs*( ), and *rt11FsMkfs*( ).

Device-Specific Configuration Options

The SCSI libraries have the following default behaviors enabled:

- SCSI messages

- disconnects
- minimum period and maximum REQ/ACK offset
- tagged command queuing
- wide data transfer

Device-specific options do not need to be set if the device shares this default behavior. However, if you need to configure a device that diverges from these default characteristics, use ***scsiTargetOptionsSet( )*** to modify option values. These options are fields in the **SCSI_OPTIONS** structure, shown below. **SCSI_OPTIONS** is declared in **scsi2Lib.h**. You can choose to set some or all of these option values to suit your particular SCSI device and application.

```
typedef struct                          /* SCSI_OPTIONS - programmable options */
    {
    UINT      selTimeOut;               /* device selection time-out (us)      */
    BOOL      messages;                 /* FALSE => do not use SCSI messages   */
    BOOL      disconnect;               /* FALSE => do not use disconnect      */
    UINT8      maxOffset;               /* max sync xfer offset (0 => async.)  */
    UINT8      minPeriod;               /* min sync xfer period (x 4 ns)       */
    SCSI_TAG_TYPE tagType;              /* default tag type                    */
    UINT      maxTags;                 /* max cmd tags available (0 => untag  */
    UINT8      xferWidth;               /* wide data trnsfr width in SCSI units */
    } SCSI_OPTIONS;
```

There are numerous types of SCSI devices, each supporting its own mix of SCSI-2 features. To set device-specific options, define a **SCSI_OPTIONS** structure and assign the desired values to the structure's fields. After setting the appropriate fields, call ***scsiTargetOptionsSet( )*** to effect your selections. Example 3-5 illustrates one possible device configuration using **SCSI_OPTIONS**.

Call ***scsiTargetOptionsSet( )*** after initializing the SCSI subsystem, but before initializing the SCSI physical device. For more information about setting and implementing options, see the reference entry for ***scsiTargetOptionsSet( )***.

WARNING: Calling ***scsiTargetOptionsSet( )*** after the physical device has been initialized may lead to undefined behavior.

The SCSI subsystem performs each SCSI command request as a SCSI transaction. This requires the SCSI subsystem to select a device. Different SCSI devices require different amounts of time to respond to a selection; in some cases, the **selTimeOut** field may need to be altered from the default.

If a device does not support SCSI messages, the boolean field **messages** can be set to FALSE. Similarly, if a device does not support disconnect/reconnect, the boolean field **disconnect** can be set to FALSE.

The SCSI subsystem automatically tries to negotiate synchronous data transfer parameters. However, if a SCSI device does not support synchronous data transfer,

set the **maxOffset** field to 0. By default, the SCSI subsystem tries to negotiate the maximum possible REQ/ACK offset and the minimum possible data transfer period supported by the SCSI controller on the VxWorks target. This is done to maximize the speed of transfers between two devices. However, speed depends upon electrical characteristics, like cable length, cable quality, and device termination; therefore, it may be necessary to reduce the values of **maxOffset** or **minPeriod** for fast transfers.

The **tagType** field defines the type of tagged command queuing desired, using one of the following macros:

- **SCSI_TAG_UNTAGGED**
- **SCSI_TAG_SIMPLE**
- **SCSI_TAG_ORDERED**
- **SCSI_TAG_HEAD_OF_QUEUE**

For more information about the types of tagged command queuing available, see the ANSI X3T9-I/O Interface Specification *Small Computer System Interface (SCSI-2)*.

The **maxTags** field sets the maximum number of command tags available for a particular SCSI device.

Wide data transfers with a SCSI target device are automatically negotiated upon initialization by the SCSI subsystem. Wide data transfer parameters are always negotiated before synchronous data transfer parameters, as specified by the SCSI ANSI specification, because a wide negotiation resets any prior negotiation of synchronous parameters. However, if a SCSI device does not support wide parameters and there are problems initializing that device, you must set the **xferWidth** field to 0. By default, the SCSI subsystem tries to negotiate the maximum possible transfer width supported by the SCSI controller on the VxWorks target in order to maximize the default transfer speed between the two devices. For more information on the actual routine call, see the reference entry for ***scsiTargetOptionsSet( )***.

SCSI Configuration Examples

The following examples show some possible configurations for different SCSI devices. Example 3-4 is a simple block device configuration setup. Example 3-5 involves selecting special options and demonstrates the use of ***scsiTargetOptionsSet( )***. Example 3-6 configures a tape device and a tape file system. Example 3-7 configures a SCSI device for synchronous data transfer. Example 3-8 shows how to configure the SCSI bus ID. These examples can be embedded either in the ***usrScsiConfig( )*** routine or in a user-defined SCSI configuration function.

Example 3-4 **Configuring SCSI Drivers**

In the following example, ***usrScsiConfig( )*** was modified to reflect a new system configuration. The new configuration has a SCSI disk with a bus ID of 4 and a Logical Unit Number (LUN) of 0 (zero). The disk is configured with a dosFs file system (with a total size of 0x20000 blocks) and a rawFs file system (spanning the remainder of the disk). The following ***usrScsiConfig( )*** code reflects this modification.

```
/* configure Winchester at busId = 4, LUN = 0 */

if ((pSpd40 = scsiPhysDevCreate (pSysScsiCtrl, 4, 0, 0, NONE, 0, 0, 0))
        == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    }
else
    {
    /* create block devices - one for dosFs and one for rawFs */

    if (((pSbd0 = scsiBlkDevCreate (pSpd40, 0x20000, 0)) == NULL) ||
        ((pSbd1 = scsiBlkDevCreate (pSpd40, 0, 0x20000)) == NULL))
        {
        return (ERROR);
        }

    /* initialize both dosFs and rawFs file systems */

    if ((dosFsDevInit ("/sd0/", pSbd0, NULL) == NULL) ||
        (rawFsDevInit ("/sd1/", pSbd1) == NULL))
        {
        return (ERROR);
        }
    }
```

If problems with your configuration occur, insert the following lines at the beginning of ***usrScsiConfig( )*** to obtain further information on SCSI bus activity.

```
#if FALSE
scsiDebug = TRUE;
scsiIntsDebug = TRUE;
#endif
```

Do not declare the global variables **scsiDebug** and **scsiIntsDebug** locally. They can be set or reset from the shell; see the *Tornado User's Guide: Shell* for details.

Example 3-5 **Configuring a SCSI Disk Drive with Asynchronous Data Transfer and No Tagged Command Queuing**

In this example, a SCSI disk device is configured without support for synchronous data transfer and tagged command queuing. The *scsiTargetOptionsSet( )* routine is used to turn off these features. The SCSI ID of this disk device is 2, and the LUN is 0:

```
int             which;
SCSI_OPTIONS    option;
int             devBusId;

devBusId = 2;
which = SCSI_SET_OPT_XFER_PARAMS | SCSI_SET_OPT_TAG_PARAMS;
option.maxOffset = SCSI_SYNC_XFER_ASYNC_OFFSET;
                                        /* => 0 defined in scsi2Lib.h */
option.minPeriod = SCSI_SYNC_XFER_MIN_PERIOD;  /* defined in scsi2Lib.h */
option.tagType = SCSI_TAG_UNTAGGED;      /* defined in scsi2Lib.h */
option.maxTag = SCSI_MAX_TAGS;

if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId, &option, which) == ERROR)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n", 0, 0, 0, 0,
        0, 0);
    return (ERROR);
    }

/* configure SCSI disk drive at busId = devBusId, LUN = 0 */

if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE, 0, 0,
        0)) == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }
```

Example 3-6 **Working with Tape Devices**

SCSI tape devices can be controlled using common commands from **scsiCommonLib** and sequential commands from **scsiSeqLib**. These commands use a pointer to a SCSI sequential device structure, **SEQ_DEV**, defined in **seqIo.h**. For more information on controlling SCSI tape devices, see the reference entries for these libraries.

This example configures a SCSI tape device whose bus ID is 5 and whose LUN is 0. It includes commands to create a physical device pointer, set up a sequential device, and initialize a tapeFs device.

```
/* configure Exabyte 8mm tape drive at busId = 5, LUN = 0 */

if ((pSpd50 = scsiPhysDevCreate (pSysScsiCtrl, 5, 0, 0, NONE, 0, 0, 0))
        == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }

/* configure the sequential device for this physical device */

if ((pSd0 = scsiSeqDevCreate (pSpd50)) == (SEQ_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiSeqDevCreate failed.\n");
        return (ERROR);
    }

/* setup the tape device configuration */

pTapeConfig = (TAPE_CONFIG *) calloc (sizeof (TAPE_CONFIG), 1);
pTapeConfig->rewind = TRUE;    /* this is a rewind device */
pTapeConfig->blkSize = 512;    /* uses 512 byte fixed blocks */

/* initialize a tapeFs device */

if (tapeFsDevInit ("/tape1", pSd0, pTapeConfig) == NULL)
    {
    return (ERROR);
    }

/* rewind the physical device using scsiSeqLib interface directly*/

if (scsiRewind (pSd0) == ERROR)
    {
    return (ERROR);
    }
```

Example 3-7 **Configuring a SCSI Disk for Synchronous Data Transfer with Non-Default Offset and Period Values**

In this example, a SCSI disk drive is configured with support for synchronous data transfer. The offset and period values are user-defined and differ from the driver default values.The chosen period is 25, defined in SCSI units of 4 ns. Thus the period is actually 4 * 25 = 100 ns. The synchronous offset is chosen to be 2. Note that you may need to adjust the values depending on your hardware environment.

```
int                     which;
SCSI_OPTIONS            option;
int                     devBusId;

devBusId = 2;

    which = SCSI_SET_IPT_XFER_PARAMS;
    option.maxOffset = 2;
    option.minPeriod = 25;

    if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId &option, which) ==
        ERROR)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n",
                        0, 0, 0, 0, 0, 0)
        return (ERROR);
        }

    /* configure SCSI disk drive at busId = devBusId, LUN = 0 */

    if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE,
                                     0, 0, 0)) == (SCSI_PHYS_DEV *) NULL)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n")
        return (ERROR);
        }
```

Example 3-8 **Changing the Bus ID of the SCSI Controller**

To change the bus ID of the SCSI controller, modify ***sysScsiInit( )*** in **sysScsi.c**. Set the SCSI bus ID to a value between 0 and 7 in the call to ***xxxCtrlInitScsi2( )*** (where *xxx* is the controller name); the default bus ID for the SCSI controller is 7.

Troubleshooting

- **Incompatibilities Between SCSI-1 and SCSI-2**

  Applications written for SCSI-1 may not execute for SCSI-2 because data structures in **scsi2Lib.h**, such as **SCSI_TRANSACTION** and **SCSI_PHYS_DEV**, have changed. This applies only if the application used these structures directly.

  If this is the case, you can choose to configure only the SCSI-1 level of support, or you can modify your application according to the data structures in **scsi2Lib.h**. In order to set new fields in the modified structure, some applications may simply need to be recompiled, and some applications will have to be modified and then recompiled.

- **SCSI Bus Failure**

  If your SCSI bus hangs, it could be for a variety of reasons. Some of the more common are:

  - Your cable has a defect. This is the most common cause of failure.
  - The cable exceeds the cumulative maximum length of 6 meters specified in the SCSI-2 standard, thus changing the electrical characteristics of the SCSI signals.
  - The bus is not terminated correctly. Consider providing termination power at both ends of the cable, as defined in the SCSI-2 ANSI specification.
  - The minimum transfer period is insufficient or the REQ/ACK offset is too great. Use ***scsiTargetOptionsSet( )*** to set appropriate values for these options.
  - The driver is trying to negotiate wide data transfers on a device that does not support them. In rejecting wide transfers, the device-specific driver cannot handle this phase mismatch. Use ***scsiTargetOptionsSet( )*** to set the appropriate value for the **xferWidth** field for that particular SCSI device.

3.7.7 Sockets

In VxWorks, the underlying basis of network communications is *sockets*. A socket is an endpoint for communication between tasks; data is sent from one socket to another. Sockets are not created or opened using the standard I/O functions. Instead they are created by calling ***socket( )***, and connected and accessed using other routines in **sockLib**. However, after a *stream* socket (using TCP) is created and connected, it can be accessed as a standard I/O device, using ***read( )***, ***write( )***, ***ioctl( )***, and ***close( )***. The value returned by ***socket( )*** as the socket handle is in fact an I/O system *fd*.

VxWorks socket routines are source-compatible with the BSD 4.3 UNIX socket functions and the Windows Sockets (Winsock 1.1) networking standard. Use of these routines is discussed in *5.2.6 Sockets*, p.251.

3.8 Differences Between VxWorks and Host System I/O

Most commonplace uses of I/O in VxWorks are completely source-compatible with I/O in UNIX and Windows. However, note the following differences:

- **Device Configuration.** In VxWorks, device drivers can be installed and removed dynamically.
- **File Descriptors.** In UNIX and Windows, *fd*s are unique to each process. In VxWorks, *fd*s are global entities, accessible by any task, except for standard input, standard output, and standard error (0, 1, and 2), which can be task specific.
- **I/O Control.** The specific parameters passed to ***ioctl( )*** functions can differ between UNIX and VxWorks.
- **Driver Routines.** In UNIX, device drivers execute in system mode and are not preemptible. In VxWorks, driver routines are in fact preemptible because they execute within the context of the task that invoked them.

3.9 Internal Structure

The VxWorks I/O system is different from most in the way the work of performing user I/O requests is apportioned between the device-independent I/O system and the device drivers themselves.

In many systems, the device driver supplies a few routines to perform low-level I/O functions such as inputting or outputting a sequence of bytes to character-oriented devices. The higher-level protocols, such as communications protocols on character-oriented devices, are implemented in the device-independent part of the I/O system. The user requests are heavily processed by the I/O system before the driver routines get control.

While this approach is designed to make it easy to implement drivers and to ensure that devices behave as much alike as possible, it has several drawbacks. The driver writer is often seriously hampered in implementing alternative protocols that are not provided by the existing I/O system. In a real-time system, it is sometimes desirable to bypass the standard protocols altogether for certain devices where throughput is critical, or where the device does not fit the standard model.

In the VxWorks I/O system, minimal processing is done on user I/O requests before control is given to the device driver. Instead, the VxWorks I/O system acts as a switch to route user requests to appropriate driver-supplied routines. Each driver can then process the raw user requests as appropriate to its devices. In addition, however, several high-level subroutine libraries are available to driver writers that implement standard protocols for both character- and block-oriented devices. Thus the VxWorks I/O system gives you the best of both worlds: while it is easy to write a standard driver for most devices with only a few pages of device-specific code, driver writers are free to execute the user requests in nonstandard ways where appropriate.

There are two fundamental types of device: *block* and *character* (or *non-block*; see Figure 3-8). Block devices are used for storing file systems. They are random access devices where data is transferred in blocks. Examples of block devices include hard and floppy disks. Character devices are any device that does not fall in the block category. Examples of character devices include serial and graphical input devices, for example, terminals and graphics tablets.

As discussed in earlier sections, the three main elements of the VxWorks I/O system are drivers, devices, and files. The following sections describe these elements in detail. The discussion focuses on character drivers; however, much of it is applicable for block devices. Because block drivers must interact with

VxWorks file systems, they use a slightly different organization; see *3.9.4 Block Devices*, p.171.

Example 3-9 shows the abbreviated code for a hypothetical driver that is used as an example throughout the following discussions. This example driver is typical of drivers for character-oriented devices.

In VxWorks, each driver has a short, unique abbreviation, such as **net** or **tty**, which is used as a prefix for each of its routines. The abbreviation for the example driver is *xx*.

Example 3-9 **Hypothetical Driver**

```
/*******************************************************************************
* xxDrv - driver initialization routine
*
* xxDrv() initializes the driver. It installs the driver via iosDrvInstall.
* It may allocate data structures, connect ISRs, and initialize hardware.
*/

STATUS xxDrv ()
  {
  xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
  (void) intConnect (intvec, xxInterrupt, ...);
  ...
  }

/*******************************************************************************
* xxDevCreate - device creation routine
*
* Called to add a device called <name> to be serviced by this driver. Other
* driver-dependent arguments may include buffer sizes, device addresses...
* The routine adds the device to the I/O system by calling iosDevAdd.
* It may also allocate and initialize data structures for the device,
* initialize semaphores, initialize device hardware, and so on.
*/

STATUS xxDevCreate (name, ...)
  char * name;
  ...
  {
  status = iosDevAdd (xxDev, name, xxDrvNum);
  ...
  }

/*******************************************************************************
* The following routines implement the basic I/O functions. The xxOpen()
* return value is meaningful only to this driver, and is passed back as an
* argument to the other I/O routines.
*/
```

```
int xxOpen (xxDev, remainder, mode)
  XXDEV * xxDev;
  char * remainder;
  int mode;
  {
  /* serial devices should have no file name part */

  if (remainder[0] != 0)
    return (ERROR);
  else
    return ((int) xxDev);
  }

int xxRead (xxDev, buffer, nBytes)
  XXDEV * xxDev;
  char * buffer;
  int nBytes;
  ...
int xxWrite (xxDev, buffer, nBytes)
  ...
int xxIoctl (xxDev, requestCode, arg)
  ...

/*******************************************************************************
* xxInterrupt - interrupt service routine
*
* Most drivers have routines that handle interrupts from the devices
* serviced by the driver. These routines are connected to the interrupts
* by calling intConnect (usually in xxDrv above). They can receive a
* single argument, specified in the call to intConnect (see intLib).
*/

VOID xxInterrupt (arg)
  ...
```

3.9.1 Drivers

A driver for a non-block device implements the seven basic I/O functions—*creat*( ), *remove*( ), *open*( ), *close*( ), *read*( ), *write*( ), and *ioctl*( )—for a particular kind of device. In general, this type of driver has routines that implement each of these functions, although some of the routines can be omitted if the functions are not operative with that device.

Drivers can optionally allow tasks to wait for activity on multiple file descriptors. This is implemented using the driver's ***ioctl*( )** routine; see *Implementing select( )*, p.163.

A driver for a block device interfaces with a file system, rather than directly with the I/O system. The file system in turn implements most I/O functions. The driver

need only supply routines to read and write blocks, reset the device, perform I/O control, and check device status. Drivers for block devices have a number of special requirements that are discussed in *3.9.4 Block Devices*, p.171.

When the user invokes one of the basic I/O functions, the I/O system routes the request to the appropriate routine of a specific driver, as detailed in the following sections. The driver's routine runs in the calling task's context, as though it were called directly from the application. Thus, the driver is free to use any facilities normally available to tasks, including I/O to other devices. This means that most drivers have to use some mechanism to provide mutual exclusion to critical regions of code. The usual mechanism is the semaphore facility provided in **semLib**.

In addition to the routines that implement the seven basic I/O functions, drivers also have three other routines:

- An initialization routine that installs the driver in the I/O system, connects to any interrupts used by the devices serviced by the driver, and performs any necessary hardware initialization (typically named *xxDrv*( )).
- A routine to add devices that are to be serviced by the driver (typically named *xxDevCreate*( )) to the I/O system.
- Interrupt-level routines that are connected to the interrupts of the devices serviced by the driver.

The Driver Table and Installing Drivers

The function of the I/O system is to route user I/O requests to the appropriate routine of the appropriate driver. The I/O system does this by maintaining a table that contains the address of each routine for each driver. Drivers are installed dynamically by calling the I/O system internal routine ***iosDrvInstall*( )**. The arguments to this routine are the addresses of the seven I/O routines for the new driver. The ***iosDrvInstall*( )** routine enters these addresses in a free slot in the driver table and returns the index of this slot. This index is known as the *driver number* and is used subsequently to associate particular devices with the driver.

Null (0) addresses can be specified for some of the seven routines. This indicates that the driver does not process those functions. For non-file-system drivers, ***close*( )** and ***remove*( )** often do nothing as far as the driver is concerned.

VxWorks file systems (**dosFsLib**, **rt11FsLib**, and **rawFsLib**) contain their own entries in the driver table, which are created when the file system library is initialized.

Example of Installing a Driver

Figure 3-2 shows the actions taken by the example driver and by the I/O system when the initialization routine *xxDrv*( ) runs.

[1] The driver calls ***iosDrvInstall*( )**, specifying the addresses of the driver's routines for the seven basic I/O functions.

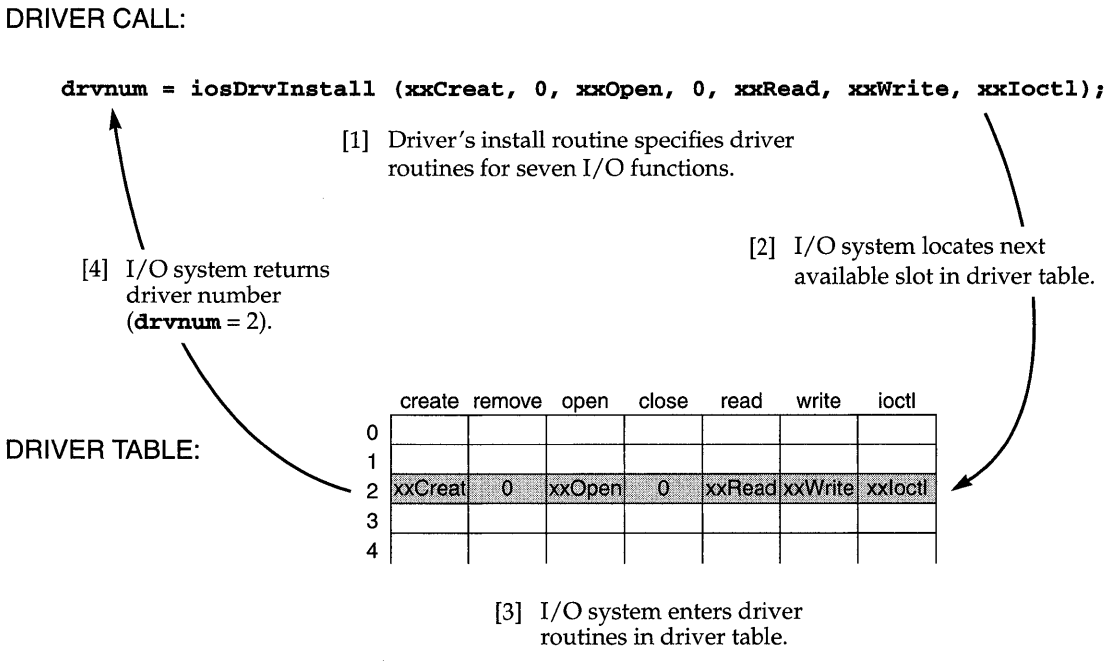
The I/O system:

[2] Locates the next available slot in the driver table, in this case slot 2.

[3] Enters the addresses of the driver routines in the driver table.

[4] Returns the slot number as the driver number of the newly installed driver.

Figure 3-2 **Example – Driver Initialization for Non-Block Devices**



3.9.2 Devices

Some drivers are capable of servicing many instances of a particular kind of device. For example, a single driver for a serial communications device can often handle many separate channels that differ only in a few parameters, such as device address.

In the VxWorks I/O system, devices are defined by a data structure called a *device header* (**DEV_HDR**). This data structure contains the device name string and the driver number for the driver that services this device. The device headers for all the devices in the system are kept in a memory-resident linked list called the *device list*. The device header is the initial part of a larger structure determined by the individual drivers. This larger structure, called a *device descriptor*, contains additional device-specific data such as device addresses, buffers, and semaphores.

The Device List and Adding Devices

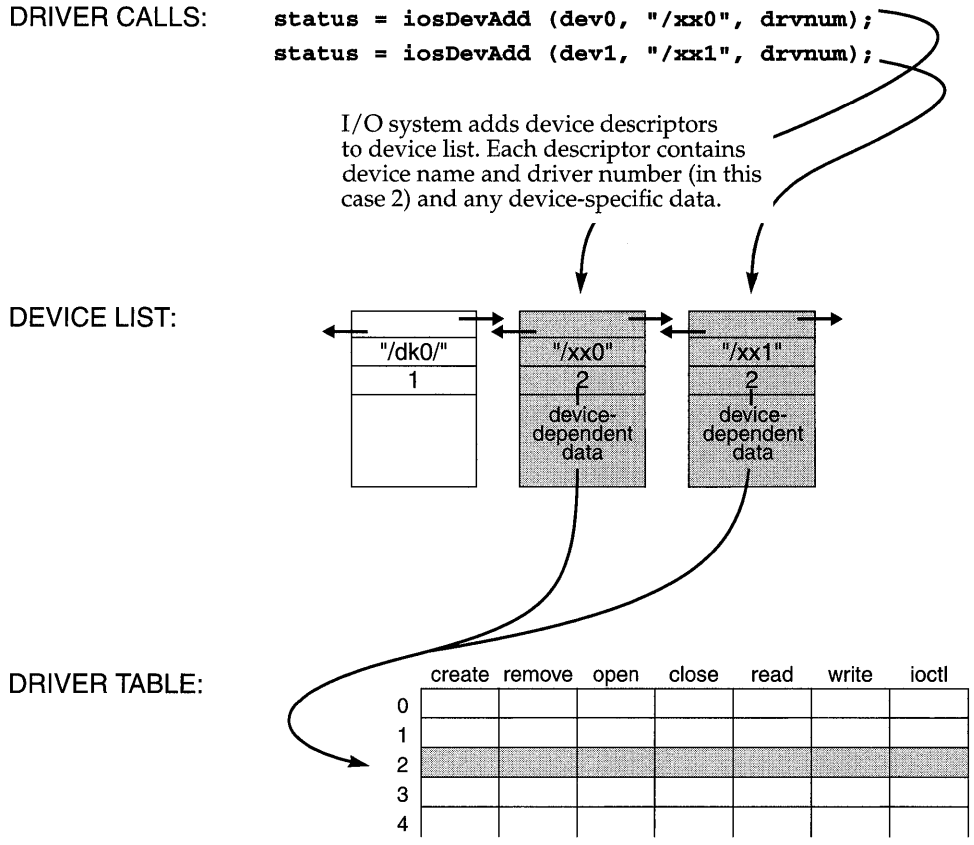
Non-block devices are added to the I/O system dynamically by calling the internal I/O routine ***iosDevAdd( )***. The arguments to ***iosDevAdd( )*** are the address of the device descriptor for the new device, the device's name, and the driver number of the driver that services the device. The device descriptor specified by the driver can contain any necessary device-dependent information, as long as it begins with a device header. The driver does not need to fill in the device header, only the device-dependent information. The ***iosDevAdd( )*** routine enters the specified device name and the driver number in the device header and adds it to the system device list.

To add a block device to the I/O system, call the device initialization routine for the file system required on that device (***dosFsDevInit( )***, ***rt11FsDevInit( )***, or ***rawFsDevInit( )***). The device initialization routine then calls ***iosDevAdd( )*** automatically.

Example of Adding Devices

In Figure 3-3, the example driver's device creation routine *xxDevCreate( )* adds devices to the I/O system by calling ***iosDevAdd( )***.

Figure 3-3 **Example – Addition of Devices to I/O System**



3.9.3 File Descriptors

Several *fd*s can be open to a single device at one time. A device driver can maintain additional information associated with an *fd* beyond the I/O system's device information. In particular, devices on which multiple files can be open at one time have file-specific information (for example, file offset) associated with each *fd*. You can also have several *fd*s open to a non-block device, such as a *tty*; typically there is no additional information, and thus writing on any of the *fd*s produces identical results.

The Fd Table

Files are opened with *open( )* (or *creat( )*). The I/O system searches the device list for a device name that matches the file name (or an initial substring) specified by the caller. If a match is found, the I/O system uses the driver number contained in the corresponding device header to locate and call the driver's open routine in the driver table.

The I/O system must establish an association between the file descriptor used by the caller in subsequent I/O calls, and the driver that services it. Additionally, the driver must associate some data structure per descriptor. In the case of non-block devices, this is usually the device descriptor that was located by the I/O system.

The I/O system maintains these associations in a table called the *fd table*. This table contains the driver number and an additional driver-determined 4-byte value. The driver value is the internal descriptor returned by the driver's open routine, and can be any nonnegative value the driver requires to identify the file. In subsequent calls to the driver's other I/O functions (*read( )*, *write( )*, *ioctl( )*, and *close( )*), this value is supplied to the driver in place of the *fd* in the application-level I/O call.

Example of Opening a File

In Figure 3-4 and Figure 3-5, a user calls *open( )* to open the file */xx0*. The I/O system takes the following series of actions:

[1] It searches the device list for a device name that matches the specified file name (or an initial substring). In this case, a complete device name matches.

[2] It reserves a slot in the *fd* table, which is used if the open is successful.

[3] It then looks up the address of the driver's open routine, *xxOpen( )*, and calls that routine. Note that the arguments to *xxOpen( )* are transformed by the I/O system from the user's original arguments to *open( )*. The first argument to *xxOpen( )* is a pointer to the device descriptor the I/O system located in the full file name search. The next parameter is the *remainder* of the file name specified by the user, after removing the initial substring that matched the device name. In this case, because the device name matched the entire file name, the remainder passed to the driver is a null string. The driver is free to interpret this remainder in any way it wants. In the case of block devices, this remainder is the name of a file on the device. In the case of non-block devices like this one, it is usually an error for the remainder to be anything *but* the null string. The last parameter is the file access flag, in this case **O_RDONLY**; that is, the file is opened for reading only.

Figure 3-4 **Example: Call to I/O Routine *open*( ) [*Part 1*]**

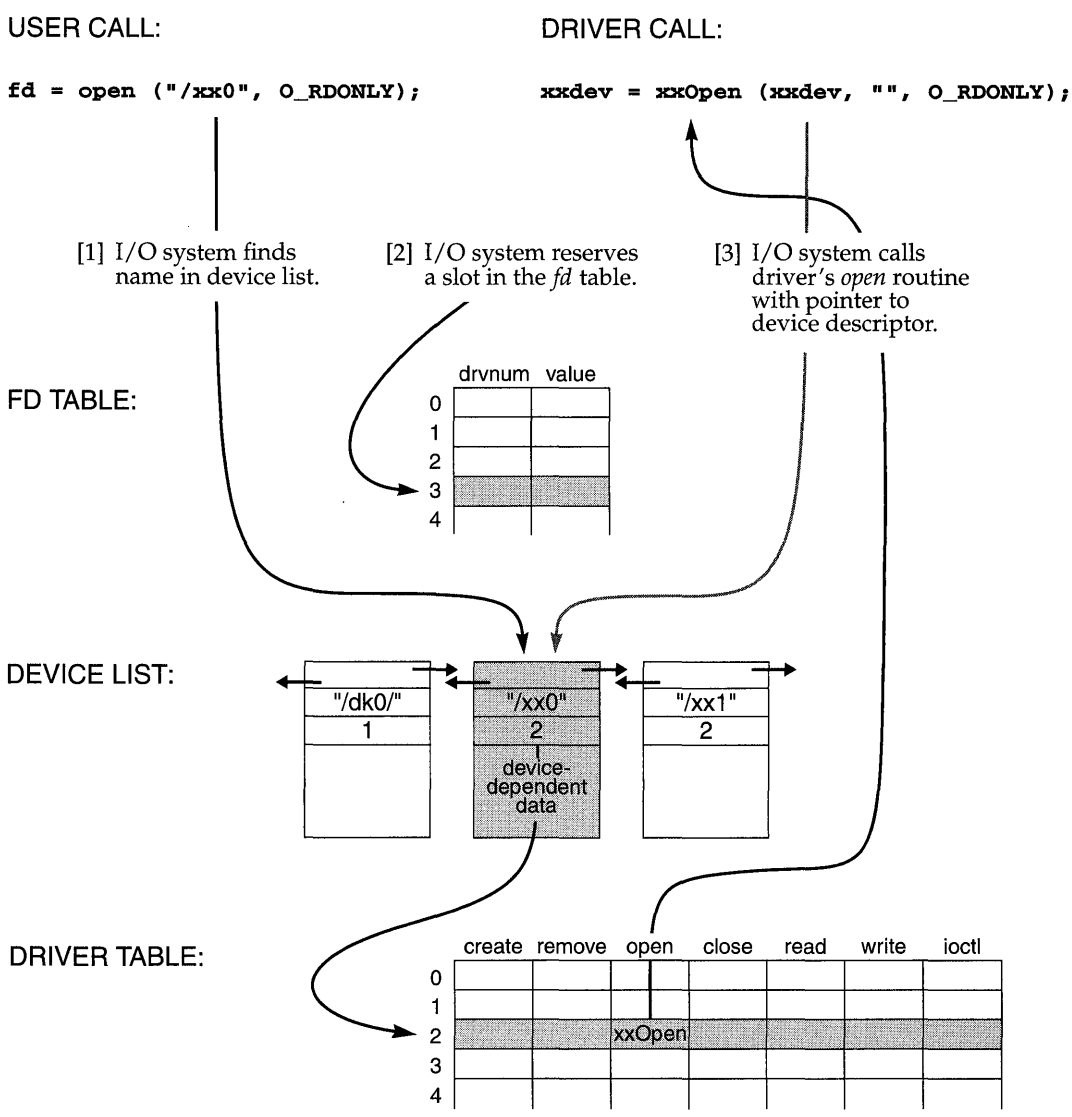
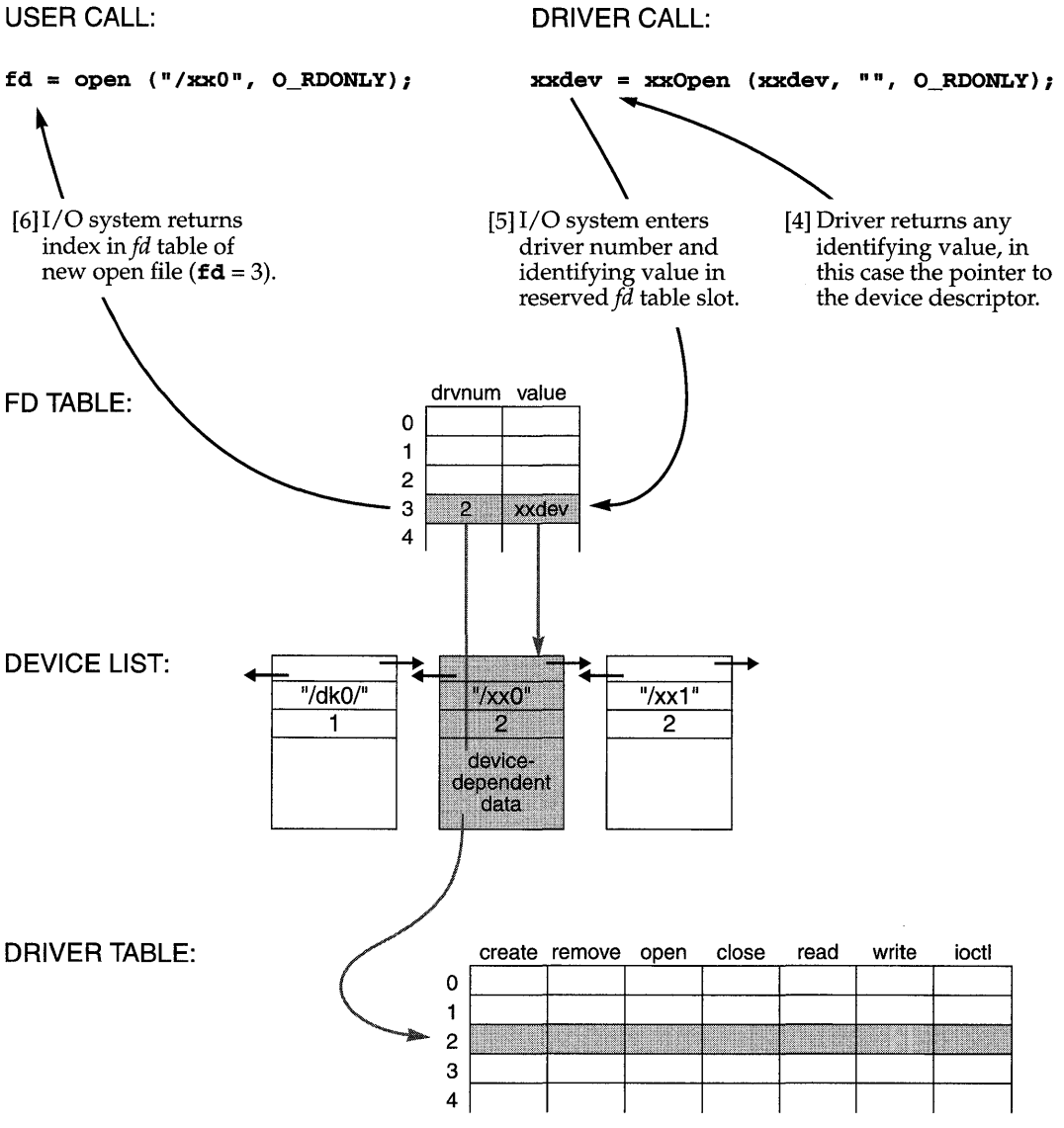


Figure 3-5 **Example: Call to I/O Routine *open*( ) [*Part 2*]**



[4] It executes *xxOpen( )*, which returns a value that subsequently identifies the newly opened file. In this case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the file being opened. Note that if the driver returns only the device descriptor, the driver cannot distinguish multiple files opened to the same device. In the case of non-block device drivers, this is usually appropriate.

[5] The I/O system then enters the driver number and the value returned by *xxOpen( )* in the reserved slot in the *fd* table. Again, the value entered in the *fd* table has meaning only for the driver, and is arbitrary as far as the I/O system is concerned.

[6] Finally, it returns to the user the index of the slot in the *fd* table, in this case 3.

Example of Reading Data from the File

In Figure 3-6, the user calls *read( )* to obtain input data from the file. The specified *fd* is the index into the *fd* table for this file. The I/O system uses the driver number contained in the table to locate the driver's read routine, *xxRead( )*. The I/O system calls *xxRead( )*, passing it the identifying value in the *fd* table that was returned by the driver's open routine, *xxOpen( )*. Again, in this case the value is the pointer to the device descriptor. The driver's read routine then does whatever is necessary to read data from the device.

The process for user calls to *write( )* and *ioctl( )* follow the same procedure.

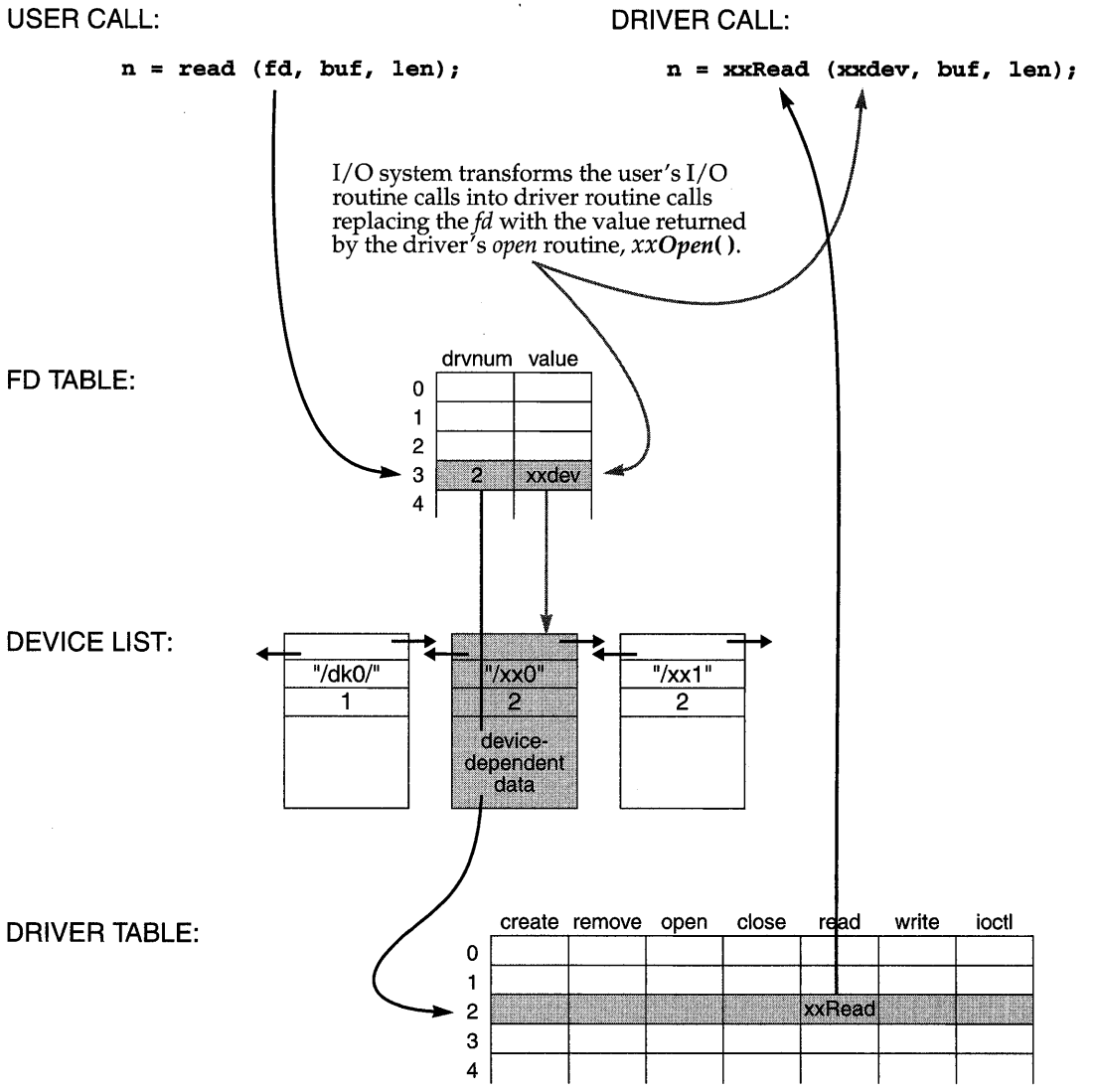
Example of Closing a File

The user terminates the use of a file by calling *close( )*. As in the case of *read( )*, the I/O system uses the driver number contained in the *fd* table to locate the driver's close routine. In the example driver, no close routine is specified; thus no driver routines are called. Instead, the I/O system marks the slot in the *fd* table as being available. Any subsequent references to that *fd* cause an error. However, subsequent calls to *open( )* can reuse that slot.

Implementing select( )

Supporting *select( )* in your driver allows tasks to wait for input from multiple devices or to specify a maximum time to wait for the device to become ready for I/O. Writing a driver that supports *select( )* is simple, because most of the

Figure 3-6 **Example: Call to I/O Routine *read*( )**



functionality is provided in **selectLib**. You might want your driver to support ***select( )*** if any of the following is appropriate for the device:

- The tasks want to specify a timeout to wait for I/O from the device. For example, a task might want to time out on a UDP socket if the packet never arrives.
- The driver supports multiple devices, and the tasks want to wait simultaneously for any number of them. For example, multiple pipes might be used for different data priorities.
- The tasks want to wait for I/O from the device while also waiting for I/O from another device. For example, a server task might use both pipes and sockets.

To implement ***select( )***, the driver must keep a list of tasks waiting for device activity. When the device becomes ready, the driver unblocks all the tasks waiting on the device.

For a device driver to support ***select( )***, it must declare a **SEL_WAKEUP_LIST** structure (typically declared as part of the device descriptor structure) and initialize it by calling ***selWakeupListInit( )***. This is done in the driver's *xxDevCreate( )* routine. When a task calls ***select( )***, **selectLib** calls the driver's ***ioctl( )*** routine with the function **FIOSELECT** or **FIOUNSELECT**. If ***ioctl( )*** is called with **FIOSELECT**, the driver must do the following:

1. Add the **SEL_WAKEUP_NODE** (provided as the third argument of ***ioctl( )***) to the **SEL_WAKEUP_LIST** by calling ***selNodeAdd( )***.
2. Use the routine ***selWakeupType( )*** to check whether the task is waiting for data to read from the device (**SELREAD**) or if the device is ready to be written (**SELWRITE**).
3. If the device is ready (for reading or writing as determined by ***selWakeupType( )***), the driver calls the routine ***selWakeup( )*** to make sure that the ***select( )*** call in the task does not pend. This avoids the situation where the task is blocked but the device is ready.

If ***ioctl( )*** is called with **FIOUNSELECT**, the driver calls ***selNodeDelete( )*** to remove the provided **SEL_WAKEUP_NODE** from the wakeup list.

When the device becomes available, ***selWakeupAll( )*** is used to unblock all the tasks waiting on this device. Although this typically occurs in the driver's ISR, it can also occur elsewhere. For example, a pipe driver might call ***selWakeupAll( )*** from its *xxRead( )* routine to unblock all the tasks waiting to write, now that there is room in the pipe to store the data. Similarly the pipe's *xxWrite( )* routine might call ***selWakeupAll( )*** to unblock all the tasks waiting to read, now that there is data in the pipe.

Example 3-10 **Driver Code Using the Select Facility**

```
/* This code fragment shows how a driver might support select(). In this
 * example, the driver unblocks tasks waiting for the device to become ready
 * in its interrupt service routine.
 */

/* arkLib.h - header file for ark driver */

typedef struct     /* ARK_DEV */
    {
    DEV_HDR     devHdr;                 /* ark device header */
    BOOL        arkDataAvailable;       /* data is available to read */
    BOOL        arkRdyForWriting;       /* device is ready to write */
    SEL_WAKEUP_LIST selWakeupList;      /* list of tasks pended in select */
    } ARK_DEV;

/* arkDrv.c - code fragments for supporting select() in a driver */

#include "vxWorks.h"
#include "selectLib.h"

STATUS arkDevCreate
    (
    char *  name,                       /* name of ark to create */
    int     number,                     /* number of arks to create */
    int     aCount                      /* number of animals to live on ark */
    )

    {
    ARK_DEV * pArkDev;                  /* pointer to ark device descriptor */

    ... your driver code ...

    /* allocate memory for ARK_DEV */

    pArkDev = (ARK_DEV *) malloc ((unsigned) sizeof (ARK_DEV + number * aCount));

    ... your driver code ...

    /* initialize wakeup list */

    selWakeupListInit (&pArkDev->selWakeupList);

    ... your driver code ...

    }

STATUS arkIoctl
    (
    ARK_DEV * pArkDev,                  /* pointer to ark device descriptor */
    int     request,                    /* ioctl function */
    int *   arg                         /* where to send answer */
    )
```

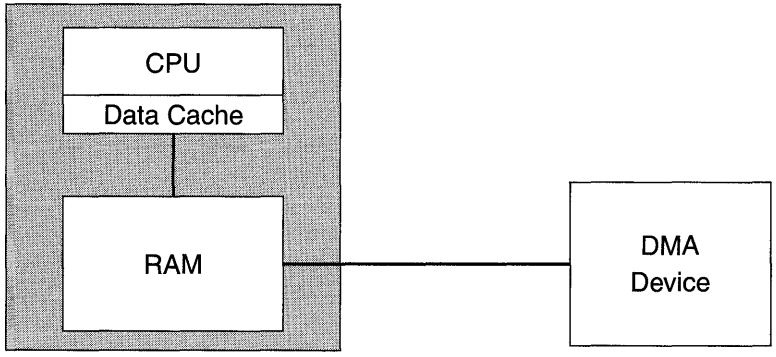
```
    {
    ... your driver code ...

    switch (request)
        {

        ... your driver code ...

        case FIOSELECT:

            /* add node to wakeup list */

            selNodeAdd (&pArkDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELREAD
                && pArkDev->arkDataAvailable)

            /* data available, make sure task does not pend */

            selWakeup ((SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELWRITE
                && pArkDev->arkRdyForWriting)

            /* device ready for writing, make sure task does not pend */

            selWakeup ((SEL_WAKEUP_NODE *) arg);

        case FIOUNSELECT:

            /* delete node from wakeup list */

            selNodeDelete (&pArkDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            ... your driver code ...
        }
    }

void arkISR
    (
    ARK_DEV *pArkDev;
    )

    {
    ... your driver code ...

    /* if there is data available to read, wake up all pending tasks */

    if (pArkDev->arkDataAvailable)
        selWakeupAll (&pArkDev->selWakeupList, SELREAD);

    /* if the device is ready to write, wake up all pending tasks */

    if (pArkDev->arkRdyForWriting)
        selWakeupAll (&pArkDev->selWakeupList, SELWRITE);
    }
```

Cache Coherency

Drivers written for boards with caches must guarantee *cache coherency*. Cache coherency means data in the cache must be in sync, or coherent, with data in RAM. The data cache and RAM can get out of sync any time there is asynchronous access to RAM (for example, DMA device access or VMEbus access). Data caches are used to increase performance by reducing the number of memory accesses. Figure 3-7 shows the relationships between the CPU, data cache, RAM, and a DMA device.

Data caches can operate in one of two modes: *writethrough* and *copyback*. Write-through mode writes data to both the cache and RAM; this guarantees cache coherency on output but not input. Copyback mode writes the data only to the cache; this makes cache coherency an issue for both input and output of data.

Figure 3-7 **Cache Coherency**



If a CPU writes data to RAM that is destined for a DMA device, the data can first be written to the data cache. When the DMA device transfers the data from RAM, there is no guarantee that the data in RAM was updated with the data in the cache. Thus, the data output to the device may not be the most recent—the new data may still be sitting in the cache. This data incoherency can be solved by making sure the data cache is flushed to RAM before the data is transferred to the DMA device.

If a CPU reads data from RAM that originated from a DMA device, the data read can be from the cache buffer (if the cache buffer for this data is not marked invalid) and not the data just transferred from the device to RAM. The solution to this data incoherency is to make sure that the cache buffer is marked invalid so that the data is read from RAM and not from the cache.

Drivers can solve the cache coherency problem either by allocating cache-safe buffers (buffers that are marked non-cacheable) or flushing and invalidating cache

entries any time the data is written to or read from the device. Allocating cache-safe buffers is useful for static buffers; however, this typically requires MMU support. Non-cacheable buffers that are allocated and freed frequently (dynamic buffers) can result in large amounts of memory being marked non-cacheable. An alternative to using non-cacheable buffers is to flush and invalidate cache entries manually; this allows dynamic buffers to be kept coherent.

The routines ***cacheFlush( )*** and ***cacheInvalidate( )*** are used to manually flush and invalidate cache buffers. Before a device reads the data, flush the data from the cache to RAM using ***cacheFlush( )*** to ensure the device reads current data. After the device has written the data into RAM, invalidate the cache entry with ***cacheInvalidate( )***. This guarantees that when the data is read by the CPU, the cache is updated with the new data in RAM.

Example 3-11 **DMA Transfer Routine**

```
/* This a sample DMA transfer routine. Before programming the device to
 * output the data to the device, it flushes the cache by calling
 * cacheFlush(). On a read, after the device has transferred the data, the
 * cache entry must be invalidated using cacheInvalidate().
 */

#include "vxWorks.h"
#include "cacheLib.h"
#include "fcntl.h"
#include "example.h"
void exampleDmaTransfer    /* 1 = READ, 0 = WRITE */
    (
    UINT8 *pExampleBuf,
    int exampleBufLen,
    int xferDirection
    )
    {
    if (xferDirection == 1)
        {
        myDevToBuf (pExampleBuf);
        cacheInvalidate (DATA_CACHE, pExampleBuf, exampleBufLen);
        }
    else
        {
        cacheFlush (DATA_CACHE, pExampleBuf, exampleBufLen);
        myBufToDev (pExampleBuf);
        }
    }
```

It is possible to make a driver more efficient by combining cache-safe buffer allocation and cache-entry flushing or invalidation. The idea is to flush or invalidate a cache entry only when absolutely necessary. To address issues of cache coherency for static buffers, use ***cacheDmaMalloc( )***. This routine initializes a **CACHE_FUNCS** structure (defined in **cacheLib.h**) to point to flush and invalidate

routines that can be used to keep the cache coherent. The macros **CACHE_DMA_FLUSH** and **CACHE_DMA_INVALIDATE** use this structure to optimize the calling of the flush and invalidate routines. If the corresponding function pointer in the **CACHE_FUNCS** structure is **NULL**, no unnecessary flush/invalidate routines are called because it is assumed that the buffer is cache coherent (hence it is not necessary to flush/invalidate the cache entry manually).

Some architectures allow the virtual address to be different from the physical address seen by the device; see *7.3 Virtual Memory Configuration*, p.408 in this manual. In this situation, the driver code uses a virtual address and the device uses a physical address. Whenever a device is given an address, it must be a physical address. Whenever the driver accesses the memory, it uses the virtual address. The driver translates the address using the following macros: **CACHE_DMA_PHYS_TO_VIRT** (to translate a physical address to a virtual one) and **CACHE_DMA_VIRT_TO_PHYS** (to translate a virtual address to a physical one).

Example 3-12 **Address-Translation Driver**

```
/* The following code is an example of a driver that performs address
 * translations. It attempts to allocate a cache-safe buffer, fill it, and
 * then write it out to the device. It uses CACHE_DMA_FLUSH to make sure
 * the data is current. The driver then reads in new data and uses
 * CACHE_DMA_INVALIDATE to guarantee cache coherency.
 */

#include "vxWorks.h"
#include "cacheLib.h"
#include "myExample.h"
STATUS myDmaExample (void)
    {
    void * pMyBuf;
    void * pPhysAddr;

    /* allocate cache safe buffers if possible */

    if ((pMyBuf = cacheDmaMalloc (MY_BUF_SIZE)) == NULL)
    return (ERROR);

    ... fill buffer with useful information ...

    /* flush cache entry before data is written to device */

    CACHE_DMA_FLUSH (pMyBuf, MY_BUF_SIZE);

    /* convert virtual address to physical */

    pPhysAddr = CACHE_DMA_VIRT_TO_PHYS (pMyBuf);

    /* program device to read data from RAM */

    myBufToDev (pPhysAddr);
```

```
... wait for DMA to complete ...

... ready to read new data ...

/* program device to write data to RAM */

myDevToBuf (pPhysAddr);

... wait for transfer to complete ...

/* convert physical to virtual address */

pMyBuf = CACHE_DMA_PHYS_TO_VIRT (pPhysAddr);

/* invalidate buffer */

CACHE_DMA_INVALIDATE (pMyBuf, MY_BUF_SIZE);

... use data ...

/* when done free memory */

if (cacheDmaFree (pMyBuf) == ERROR)
    return (ERROR);

return (OK);
}
```

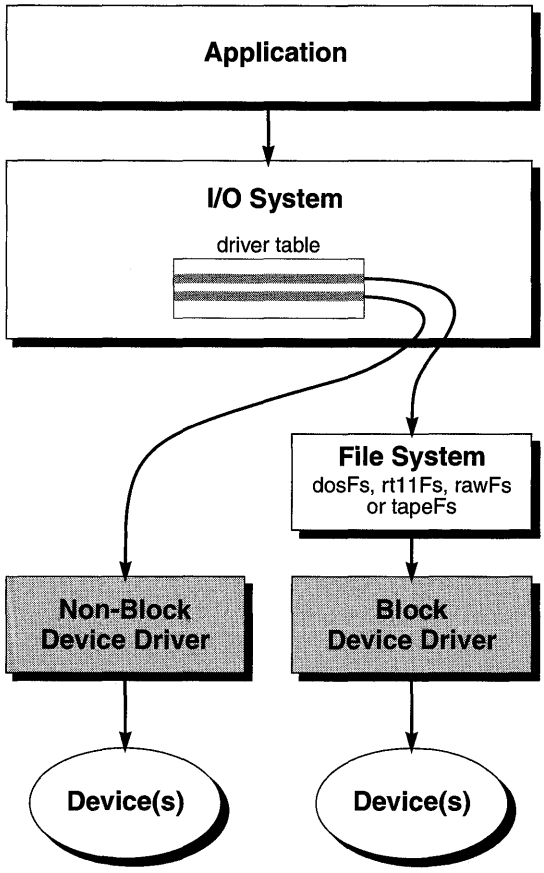
3.9.4 Block Devices

General Implementation

In VxWorks, block devices have a slightly different interface than other I/O devices. Rather than interacting directly with the I/O system, block device drivers interact with a file system. The file system, in turn, interacts with the I/O system. Direct access block devices have been supported since SCSI-1 and are used compatibly with dosFs, rt11Fs, and rawFs. In addition, VxWorks supports SCSI-2 sequential devices, which are organized so individual blocks of data are read and written sequentially. When data blocks are written, they are added sequentially at the end of the written medium; that is, data blocks cannot be replaced in the middle of the medium. However, data blocks can be accessed individually for reading throughout the medium. This process of accessing data on a sequential medium differs from that of other block devices.

Figure 3-8 shows a layered model of I/O for both block and non-block (character) devices. This layered arrangement allows the same block device driver to be used

Figure 3-8 **Non-Block Devices vs. Block Devices**



with different file systems, and reduces the number of I/O functions that must be supported in the driver.

A device driver for a block device must provide a means for creating a logical block device structure, a **BLK_DEV** for direct access block devices or a **SEQ_DEV** for sequential block devices. The **BLK_DEV/SEQ_DEV** structure describes the device in a generic fashion, specifying only those common characteristics that must be known to a file system being used with the device. Fields within the structures specify various physical configuration variables for the device—for example, block size, or total number of blocks. Other fields in the structures specify routines within the device driver that are to be used for manipulating the device (reading

blocks, writing blocks, doing I/O control functions, resetting the device, and checking device status). The **BLK_DEV/SEQ_DEV** structures also contain fields used by the driver to indicate certain conditions (for example, a disk change) to the file system.

When the driver creates the block device, the device has no name or file system associated with it. These are assigned during the device initialization routine for the chosen file system (for example, ***dosFsDevInit( )***, ***rt11FsDevInit( )*** or ***tapeFsDevInit( )***).

The low-level device driver for a block device is not installed in the I/O system driver table, unlike non-block device drivers. Instead, each file system in the VxWorks system is installed in the driver table as a "driver." Each file system has only one entry in the table, even though several different low-level device drivers can have devices served by that file system.

After a device is initialized for use with a particular file system, all I/O operations for the device are routed through that file system. To perform specific device operations, the file system in turn calls the routines in the specified **BLK_DEV** or **SEQ_DEV** structure.

A driver for a block device must provide the interface between the device and VxWorks. There is a specific set of functions required by VxWorks; individual devices vary based on what additional functions must be provided. The user manual for the device being used, as well as any other drivers for the device, is invaluable in creating the VxWorks driver. The following sections describe the components necessary to build low-level block device drivers that adhere to the standard interface for VxWorks file systems.

Low-Level Driver Initialization Routine

The driver normally requires a general initialization routine. This routine performs all operations that are done one time only, as opposed to operations that must be performed for each device served by the driver. As a general guideline, the operations in the initialization routine affect the whole device controller, while later operations affect only specific devices.

Common operations in block device driver initialization routines include:

- initializing hardware
- allocating and initializing data structures
- creating semaphores
- initializing interrupt vectors
- enabling interrupts

The operations performed in the initialization routine are entirely specific to the device (controller) being used; VxWorks has no requirements for a driver initialization routine.

Unlike non-block device drivers, the driver initialization routine does not call ***iosDrvInstall( )*** to install the driver in the I/O system driver table. Instead, the file system installs itself as a "driver" and routes calls to the actual driver using the routine addresses placed in the block device structure, **BLK_DEV** or **SEQ_DEV** (see *Device Creation Routine*, p.174).

Device Creation Routine

The driver must provide a routine to create (define) a logical disk or sequential device. A logical disk device may be only a portion of a larger physical device. If this is the case, the device driver must keep track of any block offset values or other means of identifying the physical area corresponding to the logical device. VxWorks file systems always use block numbers beginning with zero for the start of a device. A sequential access device can be either of variable block size or fixed block size. Most applications use devices of fixed block size.

The device creation routine generally allocates a device descriptor structure that the driver uses to manage the device. The first item in this device descriptor must be a VxWorks block device structure (**BLK_DEV** or **SEQ_DEV**). It must appear first because its address is passed by the file system during calls to the driver; having the **BLK_DEV** or **SEQ_DEV** as the first item permits also using this address to identify the device descriptor.

The device creation routine must initialize the fields within the **BLK_DEV** or **SEQ_DEV** structure. The **BLK_DEV** fields and their initialization values are shown in Table 3-14. The **SEQ_DEV** fields and their initialization values are shown in Table 3-15.

The device creation routine returns the address of the **BLK_DEV** or **SEQ_DEV** structure. This address is then passed during the file system device initialization call to identify the device.

Unlike non-block device drivers, the device creation routine for a block device does not call ***iosDevAdd( )*** to install the device in the I/O system device table. Instead, this is done by the file system's device initialization routine.

Table 3-14 **Fields in the BLK_DEV Structure**

Field	Value
bd_blkRd	Address of the driver routine that reads blocks from the device.
bd_blkWrt	Address of the driver routine that writes blocks to the device.
bd_ioctl	Address of the driver routine that performs device I/O control.
bd_reset	Address of the driver routine that resets the device (**NULL** if none).
bd_statusChk	Address of the driver routine that checks disk status (**NULL** if none).
bd_removable	TRUE if the device is removable (for example, a floppy disk); FALSE otherwise.
bd_nBlocks	Total number of blocks on the device.
bd_bytesPerBlk	Number of bytes per block on the device.
bd_blksPerTrack	Number of blocks per track on the device.
bd_nHeads	Number of heads (surfaces).
bd_retry	Number of times to retry failed reads or writes.
bd_mode	Device mode (write-protect status); generally set to **O_RDWR**.
bd_readyChanged	TRUE if the device ready status has changed; initialize to TRUE to cause the disk to be mounted.

Table 3-15 **Fields in the SEQ_DEV Structure**

Field	Value
sd_seqRd	Address of the driver routine that reads blocks from the device.
sd_seqWrt	Address of the driver routine that writes blocks to the device.
sd_ioctl	Address of the driver routine that performs device I/O control.
sd_seqWrtFileMarks	Address of the driver routine that writes file marks to the device.
sd_rewind	Address of the driver routine that rewinds the sequential device.
sd_reserve	Address of the driver routine that reserves a sequential device.
sd_release	Address of the driver routine that releases a sequential device.
sd_readBlkLim	Address of the driver routine that reads the data block limits from the sequential device.
sd_load	Address of the driver routine that either loads or unloads a sequential device.
sd_space	Address of the driver routine that moves (spaces) the medium forward or backward to end-of-file or end-of-record markers.

Table 3-15 **Fields in the SEQ_DEV Structure** *(Continued)*

Field	Value
sd_erase	Address of the driver routine that erases a sequential device.
sd_reset	Address of the driver routine that resets the device (**NULL** if none).
sd_statusChk	Address of the driver routine that checks sequential device status (**NULL** if none).
sd_blkSize	Block size of sequential blocks for the device. A block size of 0 means that variable block sizes are used.
sd_mode	Device mode (write protect status).
sd_readyChanged	TRUE if the device ready status has changed; initialize to TRUE to cause the sequential device to be mounted.
sd_maxVarBlockLimit	Maximum block size for a variable block.
sd_density	Density of sequential access media.

Read Routine (Direct-Access Devices)

The driver must supply a routine to read one or more blocks from the device. For a direct access device, the read-blocks routine must have the following arguments and result:

```
STATUS xxBlkRd
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int       startBlk,   /* starting block to read */
    int       numBlks,    /* number of blocks to read */
    char *    pBuf        /* pointer to buffer to receive data */
    )
```

In this and following examples, the routine names begin with *xx*. These names are for illustration only, and do not have to be used by your device driver. VxWorks references the routines by address only; the name can be anything.

pDev a pointer to the driver's device descriptor structure, represented here by the symbolic name **DEVICE**. (Actually, the file system passes the address of the corresponding **BLK_DEV** structure; these are equivalent, because the **BLK_DEV** is the first item in the device descriptor.) This identifies the device.

startBlk the starting block number to be read from the device. The file system always uses block numbers beginning with zero for the start of the

device. Any offset value used for this logical device must be added in by the driver.

numBlks the number of blocks to be read. If the underlying device hardware does not support multiple-block reads, the driver routine must do the necessary looping to emulate this ability.

pBuf the address where data read from the disk is to be copied.

The read routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

Read Routine (Sequential Devices)

The driver must supply a routine to read a specified number of bytes from the device. The bytes being read are always assumed to be read from the current location of the read/write head on the media. The read routine must have the following arguments and result:

```
STATUS xxSeqRd
    (
    DEVICE *  pDev,          /* pointer to device descriptor */
    int       numBytes,      /* number of bytes to read */
    char *    buffer,        /* pointer to buffer to receive data */
    BOOL      fixed          /* TRUE => fixed block size */
    )
```

pDev a pointer to the driver's device descriptor structure, represented here by the symbolic name **DEVICE**. (Actually, the file system passes the address of the corresponding **SEQ_DEV** structure; these are equivalent, because the **SEQ_DEV** structure is the first item in the device descriptor.) This identifies the device.

numBytes the number of bytes to be read.

buffer the buffer into which *numBytes* of data are read.

fixed specifies whether the read routine reads fixed-sized blocks from the sequential device or variable-sized blocks, as specified by the file system. If *fixed* is TRUE, then fixed sized blocks are used.

The read routine returns **OK** if the transfer is completed successfully, or **ERROR** if a problem occurs.

Write Routine (Direct-Access Devices)

The driver must supply a routine to write one or more blocks to the device. The definition of this routine closely parallels that of the read routine. For direct-access devices, the write routine is as follows:

```
STATUS xxBlkWrt
    (
    DEVICE *  pDev,      /* pointer to device descriptor */
    int       startBlk,  /* starting block for write */
    int       numBlks,   /* number of blocks to write */
    char *    pBuf       /* ptr to buffer of data to write */
    )
```

pDev a pointer to the driver's device descriptor structure.

startBlk the starting block number to be written to the device.

numBlks the number of blocks to be written. If the underlying device hardware does not support multiple-block writes, the driver routine must do the necessary looping to emulate this ability.

pBuf the address of the data to be written to the disk.

The write routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

Write Routine (Sequential Devices)

The driver must supply a routine to write a specified number of bytes to the device. The bytes being written are always assumed to be written to the current location of the read/write head on the media. For sequential devices, the write routine is as follows:

```
STATUS xxWrtTape
    (
    DEVICE *  pDev,       /* ptr to SCSI sequential device info */
    int       numBytes,   /* total bytes or blocks to be written */
    char *    buffer,     /* ptr to input data buffer      */
    BOOL      fixed       /* TRUE => fixed block size */
    )
```

pDev a pointer to the driver's device descriptor structure.

numBytes the number of bytes to be written.

buffer the buffer from which *numBytes* of data are written.

fixed specifies whether the write routine reads fixed-sized blocks from the sequential device or variable-sized blocks, as specified by the file system. If *fixed* is TRUE, then fixed sized blocks are used.

The write routine returns **OK** if the transfer is successful, or **ERROR** if a problem occurs.

I/O Control Routine

The driver must provide a routine that can handle I/O control requests. In VxWorks, most I/O operations beyond basic file handling are implemented through ***ioctl*( )** functions. The majority of these are handled directly by the file system. However, if the file system does not recognize a request, that request is passed to the driver's I/O control routine.

Define the driver's I/O control routine as follows:

```
STATUS xxIoctl
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int       funcCode,   /* ioctl() function code */
    int       arg         /* function-specific argument */
    )
```

pDev a pointer to the driver's device descriptor structure.

funcCode the requested ***ioctl*( )** function. Standard VxWorks I/O control functions are defined in the include file **ioLib.h**. Other user-defined function code values can be used as required by your device driver. The I/O control functions supported by the dosFs, rt11Fs, rawFs, and tapeFs are summarized in *4. Local File Systems* in this manual.

arg specific to the particular ***ioctl*( )** function requested. Not all ***ioctl*( )** functions use this argument.

The driver's I/O control routine typically takes the form of a multi-way switch statement, based on the function code. The driver's I/O control routine must supply a default case for function code requests it does not recognize. For such requests, the I/O control routine sets **errno** to **S_ioLib_UNKNOWN_REQUEST** and returns **ERROR**.

The driver's I/O control routine returns **OK** if it handled the request successfully; otherwise, it returns **ERROR**.

Device-Reset Routine

The driver usually supplies a routine to reset a specific device, but it is not required. This routine is called when a VxWorks file system first mounts a disk or tape, and again during retry operations when a read or write fails.

Declare the driver's device-reset routine as follows:

```
STATUS xxReset
    (
    DEVICE *  pDev
    )
```

pDev a pointer to the driver's device descriptor structure.

When called, this routine resets the device and controller. Do not reset other devices, if it can be avoided. The routine returns **OK** if the driver succeeded in resetting the device; otherwise, it returns **ERROR**.

If no reset operation is required for the device, this routine can be omitted. In this case, the device-creation routine sets the *xx*_**reset** field in the **BLK_DEV** or **SEQ_DEV** structure to **NULL**.

In this and following examples, the names of fields in the **BLK_DEV** and **SEQ_DEV** structures are parallel except for the initial letters **bd_** or **sd_**. In these cases, the initial letters are represented by *xx_*, as in the *xx*_**reset** field to represent both the **bd_reset** field and the **sd_reset** field.

Status-Check Routine

If the driver provides a routine to check device status or perform other preliminary operations, the file system calls this routine at the beginning of each ***open*( )** or ***creat*( )** on the device.

Define the status-check routine as follows:

```
STATUS xxStatusChk
    (
    DEVICE *  pDev    /* pointer to device descriptor */
    )
```

pDev a pointer to the driver's device descriptor structure.

The routine returns **OK** if the open or create operation can continue. If it detects a problem with the device, it sets **errno** to some value indicating the problem, and returns **ERROR**. If **ERROR** is returned, the file system does not continue the operation.

A primary use of the status-check routine is to check for a disk change on devices that do not detect the change until after a new disk is inserted. If the routine determines that a new disk is present, it sets the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE and returns **OK** so that the open or create operation can continue. The new disk is then mounted automatically by the file system. (See *Change in Ready Status*, p.181.)

Similarly, the status check routine can be used to check for a tape change. This routine determines whether a new tape has been inserted. If a new tape is present, the routine sets the **sd_readyChanged** field in the **SEQ_DEV** structure to TRUE and returns **OK** so that the open or create operation can continue. The device driver should not be able to unload a tape, nor should you physically eject a tape, while a file descriptor is open on the tape device.

If the device driver requires no status-check routine, the device-creation routine sets the *xx*_**statusChk** field in the **BLK_DEV** or **SEQ_DEV** structure to **NULL**.

Write-Protected Media

The device driver may detect that the disk or tape in place is write-protected. If this is the case, the driver sets the *xx*_**mode** field in the **BLK_DEV** or **SEQ_DEV** structure to **O_RDONLY**. This can be done at any time (even after the device is initialized for use with the file system). The file system checks this value and does not allow writes to the device until the *xx*_**mode** field is changed (to **O_RDWR** or **O_WRONLY**) or the file system's mode change routine (for example, ***dosFsModeChange*( )**) is called to change the mode. (The *xx*_**mode** field is changed automatically if the file system's mode change routine is used.)

Change in Ready Status

The driver informs the file system whenever a change in the device's ready status is recognized. This can be the changing of a floppy disk, changing of the tape medium, or any other situation that makes it advisable for the file system to remount the disk.

To announce a change in ready status, the driver sets the *xx*_**readyChanged** field in the **BLK_DEV** or **SEQ_DEV** structure to TRUE. This is recognized by the file system, which remounts the disk during the next I/O initiated on the disk. The file system then sets the *xx*_**readyChanged** field to FALSE. The *xx*_**readyChanged** field is never cleared by the device driver.

Setting *xx*_**readyChanged** to TRUE has the same effect as calling the file system's ready-change routine (for example, ***dosFsReadyChange*( )**) or calling ***ioctl*( )** with the **FIODISKCHANGE** function code.

An optional status-check routine (see *Status-Check Routine*, p.180) can provide a convenient mechanism for asserting a ready-change, particularly for devices that cannot detect a disk change until after the new disk is inserted. If the status-check routine detects that a new disk is present, it sets *xx*_**readyChanged** to TRUE. This routine is called by the file system at the beginning of each open or create operation.

Write-File-Marks Routine (Sequential Devices)

The sequential driver must provide a routine that can write file marks onto the tape device. The write file marks routine must have the following arguments

```
STATUS xxWrtFileMarks
    (
    DEVICE *  pDev,        /* pointer to device descriptor */
    int       numMarks,    /* number of file marks to write */
    BOOL      shortMark    /* short or long file marks */
    )
```

pDev a pointer to the driver's device descriptor structure.

numMarks the number of file marks to be written sequentially.

shortMark the type of file mark (short or long). If *shortMark is* TRUE, short marks are written.

The write file marks routine returns **OK** if the file marks are written correctly on the tape device; otherwise, it returns **ERROR**.

Rewind Routine (Sequential Devices)

The sequential driver must provide a rewind routine in order to rewind tapes in the tape device. The rewind routine is defined as follows:

```
STATUS xxRewind
    (
    DEVICE *  pDev  /* pointer to device descriptor */
    )
```

pDev a pointer to the driver's device descriptor structure.

When called, this routine rewinds the tape in the tape device. The routine returns **OK** if completion is successful; otherwise, it returns **ERROR**.

Reserve Routine (Sequential Devices)

The sequential driver can provide a reserve routine that reserves the physical tape device for exclusive access by the host that is executing the reserve routine. The tape device remains reserved until it is released by that host, using a release routine, or by some external stimulus.

The reserve routine is defined as follows:

```
STATUS xxReserve
    (
    DEVICE *  pDev  /* pointer to device descriptor */
    )
```

pDev a pointer to the driver's device descriptor structure.

If a tape device is reserved successfully, the reserve routine returns **OK**. However, if the tape device cannot be reserved or an error occurs, it returns **ERROR**.

Release Routine (Sequential Devices)

This routine releases the exclusive access that a host has on a tape device. The tape device is then free to be reserved again by the same host or some other host. This routine is the opposite of the reserve routine and must be provided by the driver if the reserve routine is provided.

The release routine is defined as follows:

```
STATUS xxReset
    (
    DEVICE *  pDev  /* pointer to device descriptor */
    )
```

pDev a pointer to the driver's device descriptor structure.

If the tape device is released successfully, this routine returns **OK**. However, if the tape device cannot be released or an error occurs, this routine returns **ERROR**.

Read-Block-Limits Routine (Sequential Devices)

The read-block-limits routine can poll a tape device for its physical block limits. These block limits are then passed back to the file system so the file system can decide the range of block sizes to be provided to a user.

The read-block-limits routine is defined as follows:

```
STATUS xxReadBlkLim
    (
    DEVICE *  pDev,          /* pointer to device descriptor */
    int       *maxBlkLimit,  /* maximum block size for device */
    int       *minBlkLimit   /* minimum block size for device */
    )
```

pDev a pointer to the driver's device descriptor structure.

maxBlkLimit
returns the maximum block size that the tape device can handle to the calling tape file system.

minBlkLimit
returns the minimum block size that the tape device can handle.

The routine returns **OK** if no error occurred while acquiring the block limits; otherwise, it returns **ERROR**.

Load/Unload Routine (Sequential Devices)

The sequential device driver must provide a load/unload routine in order to mount or unmount tape volumes from a physical tape device. Loading means that a volume is being mounted by the file system. This is usually done upon an ***open*( )** or a ***creat*( )**. However, a device should be unloaded or unmounted only when the file system wants to eject the tape volume from the tape device.

The load/unload routine is defined as follows:

```
STATUS xxLoad
    (
    DEVICE *  pDev,  /* pointer to device descriptor */
    BOOL      load   /* load or unload device */
    )
```

pDev a pointer to the driver's device descriptor structure.

load a boolean variable that determines if the tape is loaded or unloaded. If *load* is TRUE, the tape is loaded. If *load* is FALSE, the tape is unloaded.

The load/unload routine returns **OK** if the load or unload operation ends successfully; otherwise, it returns **ERROR**.

Space Routine (Sequential Devices)

The sequential device driver must provide a space routine that moves, or spaces, the tape medium forward or backward. The amount of distance that the tape spaces depends on the kind of search that must be performed. In general, tapes can be searched by end-of-record marks, end-of-file marks, or other types of device-specific markers.

The basic definition of the space routine is as follows; however, other arguments can be added to the definition:

```
STATUS xxSpace
    (
    DEVICE *  pDev,       /* pointer to device descriptor */
    int       count,      /* number of spaces */
    int       spaceCode   /* type of space */
    )
```

pDev a pointer to the driver's device descriptor structure.

count specifies the direction of search. A positive *count* value represents forward movement of the tape device from its current location (forward space); a negative *count* value represents a reverse movement (back space).

spaceCode defines the type of space mark that the tape device searches for on the tape medium. The basic types of space marks are end-of-record and end-of-file. However, different tape devices may support more sophisticated kinds of space marks designed for more efficient maneuvering of the medium by the tape device.

If the device is able to space in the specified direction by the specified count and space code, the routine returns **OK**; if these conditions cannot be met, it returns **ERROR**.

Erase Routine (Sequential Devices)

The sequential driver must provide a routine that allows a tape to be erased. The erase routine is defined as follows:

```
STATUS xxErase
    (
    DEVICE *  pDev  /* pointer to device descriptor */
    )
```

pDev a pointer to the driver's device descriptor structure.

The routine returns **OK** if the tape is erased; otherwise, it returns **ERROR**.

3.9.5 Driver Support Libraries

The subroutine libraries in Table 3-16 may assist in the writing of device drivers. Using these libraries, drivers for most devices that follow standard protocols can be written with only a few pages of device-dependent code. See the reference entry for each library for details.

Table 3-16 **VxWorks Driver Support Routines**

Library	Description
errnoLib	Error status library
ftpLib	ARPA File Transfer Protocol library
ioLib	I/O interface library
iosLib	I/O system library
intLib	Interrupt support subroutine library
remLib	Remote command library
rngLib	Ring buffer subroutine library
ttyDrv	Terminal driver
wdLib	Watchdog timer subroutine library

4 Local File Systems

List of Tables

List of Figures

List of Examples

4.1 Introduction

This chapter discusses the organization, configuration, and use of VxWorks file systems. VxWorks provides two local file systems appropriate for real-time use with block devices (disks): one is compatible with MS-DOS file systems and the other with the RT-11 file system. The support libraries for these file systems are **dosFsLib** and **rt11FsLib**. VxWorks also provides a simple *raw file system*, which treats an entire disk much like a single large file. The support library for this "file system" is **rawFsLib**. In addition, VxWorks provides a file system for tape devices that do not use a standard file or directory structure on tape. The tape volume is treated much like a raw device where the entire volume is a large file. The support library for this file system is **tapeFsLib**.

In VxWorks, the file system is not tied to a specific type of block device or its driver. VxWorks block devices all use a standard interface so that file systems can be freely mixed with device drivers. Alternatively, you can write your own file systems that can be used by drivers in the same way, by following the same standard interfaces between the file system, the driver, and the I/O system. VxWorks I/O architecture makes it possible to have multiple file systems, even of different types, in a single VxWorks system. The block device interface is discussed in *3.9.4 Block Devices*, p.171.

4.2 MS-DOS-Compatible File System: dosFs

Diskettes formatted using the dosFs file system are compatible with MS-DOS diskettes up to and including release 6.2. Hard disks initialized by the two file systems have slightly different formats. However, the data itself is compatible and dosFs can be configured to use a disk formatted by MS-DOS.

The dosFs file system offers considerable flexibility appropriate to the varying demands of real-time applications. Major features include:

- A hierarchical arrangement of files and directories, allowing efficient organization and permitting an arbitrary number of files to be created on a volume.
- A choice of contiguous or non-contiguous files on a per-file basis. Non-contiguous files result in more efficient use of available disk space, while contiguous files offer enhanced performance.
- Compatibility with widely available storage and retrieval media. Diskettes created with VxWorks (that do not use dosFs extended filenames) and MS-DOS PCs and other systems can be freely interchanged. Hard disks are compatible if the partition table is accounted for.
- The ability to boot VxWorks from any local SCSI device that has a dosFs file system.
- The ability to use longer file names than the 8-character filename plus 3-character extension (8.3) convention allowed by MS-DOS.
- NFS (Network File System) support.

4.2.1 Disk Organization

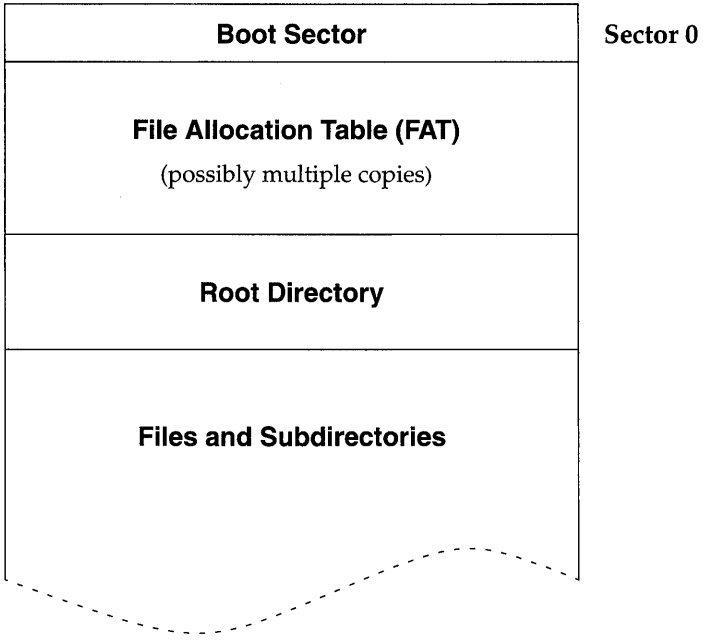
The MS-DOS/dosFs file system provides the means for organizing disk data in a flexible manner. It maintains a hierarchical set of named directories, each containing files or other directories. Files can be appended; as they expand, new disk space is allocated automatically. The disk space allocated to a file is not necessarily contiguous, which results in a minimum of wasted space. However, to enhance its real-time performance, the dosFs file system allows contiguous space to be pre-allocated to files individually, thereby minimizing seek operations and providing more deterministic behavior.

The general organization of an MS-DOS/dosFs file system is shown in Figure 4-1 and the various elements are discussed in the following sections.

Clusters

The disk space allocated to a file in an MS-DOS/dosFs file system consists of one or more disk *clusters*. A cluster is a set of contiguous disk sectors.[1] For floppy disks, two sectors generally make up a cluster; for fixed disks, there can be more sectors

Figure 4-1 **MS-DOS Disk Organization**



NOTE: If the number of reserved sectors (**dosvc_nResrvd**) is greater than 1, the first FAT copy does not immediately follow the boot sector.

per cluster. A cluster is the smallest amount of disk space the file system can allocate at a time. A large number of sectors per cluster allows a larger disk to be described in a fixed-size File Allocation Table (FAT; see *File Allocation Table*, p.194), but can result in wasted disk space.

Boot Sector

The first sector on an MS-DOS/dosFs hard disk or diskette is called the *boot sector.* This sector contains a variety of configuration data. Some of the data fields

1. In this and subsequent sections covering the dosFs file system, the term *sector* refers to the minimum addressable unit on a disk, because this is the term used by most MS-DOS documentation. In VxWorks, the units are normally referred to as *blocks*, and a disk device is called a *block device*.

describe the physical properties of the disk (such as the total number of sectors), and other fields describe file system variables (such as the size of the root directory).

The boot sector information is written to a disk when it is initialized. The dosFs file system can use diskettes that are initialized on another system (for example, using the **FORMAT** utility on an MS-DOS PC), or VxWorks can initialize the diskette, using the **FIODISKINIT** function of the ***ioctl( )*** call.

As the MS-DOS standard has evolved, various fields have been added to the boot sector definition. Disks initialized under VxWorks use the boot sector fields defined by MS-DOS version 5.0.

When MS-DOS initializes a hard disk, it writes a *partition table* in addition to the boot sector. VxWorks does not create such a table. Therefore hard disks initialized by the two systems are not identical. VxWorks can read files from a disk formatted by MS-DOS if the block offset parameter in the device creation routine points beyond the partition table to the first byte of the data area.

File Allocation Table

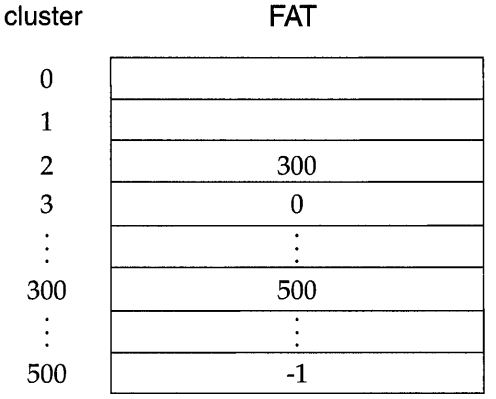
Each MS-DOS/dosFs volume contains a File Allocation Table (FAT). The FAT contains an entry for each cluster on the disk that can be allocated to a file or directory. When a cluster is unused (available for allocation), its entry is zero. If a cluster is allocated to a file, its entry is the cluster number of the next portion of the file. If a cluster is the last in a file, its entry is -1. Thus, the representation of a file (or directory) consists of a linked list of FAT entries. In the example shown in Figure 4-2, one file consists of clusters 2, 300, and 500. Cluster 3 is unused.

The FAT uses either 12 or 16 bits per entry. Disk volumes that contain up to 4085 clusters use 12-bit entries; disks with more than 4085 clusters use 16-bit entries. The entries (particularly 12-bit entries) are encoded in a specific manner, done originally to take advantage of the Intel 8088 architecture. However, all FAT handling is done by the dosFs file system; thus the encoding and decoding is of no concern to VxWorks applications.

A volume typically contains multiple copies of the FAT. This redundancy allows data recovery in the event of a media error in the first FAT copy.

⚠ **NOTE:** The dosFs file system maintains multiple FAT copies if that is the specified configuration; however, the copies are not automatically used in the event of an error.

Figure 4-2 **FAT Entries**



The size of the FAT and the number of FAT copies are determined by fields in the boot sector. For disks initialized using the dosFs file system, these parameters are specified during the ***dosFsDevInit( )*** call by setting fields in the volume configuration structure, **DOS_VOL_CONFIG**.

Root Directory

Each MS-DOS/dosFs volume contains a root directory. The root directory always occupies a set of contiguous disk sectors immediately following the FAT copies. The disk area occupied by the root directory is not described by entries in the FAT.

The root directory is of a fixed size; this size is specified by a field in the boot sector as the maximum allowed number of directory entries. For disks initialized using the dosFs file system, this size is specified during the ***dosFsDevInit( )*** call, by setting a field in the volume configuration structure, **DOS_VOL_CONFIG**.

Because the root directory has a fixed size, an error is returned if the directory is full and an attempt is made to add entries to it.

For more information on the contents of the directory entry, see *4.2.13 Directory Entries*, p.207.

Subdirectories

In addition to the root directory, MS-DOS/dosFs volumes sometimes contain a hierarchy of subdirectories. Like the root directory, subdirectories contain entries

for files and other subdirectories; however, in other ways they differ from the root directory and resemble files:

- First, each subdirectory is described by an entry in another directory, as is a file. Such a directory entry has a bit set in the file-attribute byte to indicate that it describes a subdirectory. Also, subdirectories, unlike the root directory, have user-assigned names.
- Second, the disk space allocated to a subdirectory is composed of a set of disk clusters, linked by FAT entries. This means that a subdirectory can grow as entries are added to it, and that the subdirectory is not necessarily made up of contiguous clusters. The root directory, unlike subdirectories, can be made up of any number of sectors, not necessarily equal to a whole number of clusters.
- Third, subdirectories always contain two special entries. The "." entry refers to the subdirectory itself, while the ".." entry refers to the subdirectory's parent directory. The root directory does not contain these special entries.

Files

The disk space allocated to a file in the MS-DOS/dosFs file system is a set of clusters that are chained together through entries in the FAT. A file is not necessarily made up of contiguous clusters; the various clusters can be located anywhere on the disk and in any order.

Each file has a descriptive entry in the directory where it resides. This entry contains the file's name, size, last modification date and time, and a field giving several important attributes (read-only, system, hidden, modified since last archived). It also contains the starting cluster number for the file; subsequent clusters are located using the FAT.

Volume Label

An MS-DOS/dosFs disk can have a *volume label* associated with it. The volume label is a special entry in the root directory. Rather than containing the name of a file or subdirectory, the volume label entry contains a string used to identify the volume. This string can contain up to 11 characters. The volume label entry is identified by a special value of the file-attribute byte in the directory entry.

Note that a volume label entry is not reported using ***ls*( )**. However, it does occupy one of the fixed number of entries in the root directory.

The volume label can be added to a dosFs volume by using the *ioctl( )* call with the **FIOLABELSET** function. This adds a label entry to the volume's root directory if none exists or changes the label string in an existing volume label entry. The volume label entry takes up one of the fixed number of root directory entries; attempting to add an entry when the root directory is full results in an error.

The current volume label string for a volume can be obtained by calling the *ioctl( )* call with the **FIOLABELGET** function. If the volume has no label, this call returns **ERROR** and sets **errno** to **S_dosFsLib_NO_LABEL**.

Disks initialized under VxWorks or under MS-DOS 5.0 (or later) also contain the volume label string within a boot sector field.

4.2.2 Initializing the dosFs File System

Note that before any other operations can be performed, the dosFs file system library, **dosFsLib**, must be initialized by calling *dosFsInit( )*. This routine takes a single parameter, the maximum number of dosFs file descriptors that can be open at one time. That number of file descriptors is allocated during initialization; a descriptor is used each time your application opens a file, directory, or the file system device.

The *dosFsInit( )* routine also makes an entry for the file system in the I/O system driver table (with *iosDrvInstall( )*). This entry specifies entry points for dosFs file operations and is used for all devices that use the dosFs file system. The driver number assigned to the dosFs file system is recorded in a global variable **dosFsDrvNum**.

The *dosFsInit( )* routine is normally called by the *usrRoot( )* task after starting the VxWorks system. To use this initialization, define **INCLUDE_DOSFS** in the configuration file **configAll.h**, and set **NUM_DOSFS_FILES** to the desired maximum open file count.

4.2.3 Initializing a Device for Use with dosFs

After the dosFs file system is initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine (*xxDevCreate( )*). The driver routine returns a pointer to a block device descriptor structure (**BLK_DEV**). The **BLK_DEV** structure describes the physical aspects of the device and specifies the routines that the device driver provides to a file system. For more information on block devices, see *3.9.4 Block Devices*, p.171.

Immediately after its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with the dosFs file system, the already-created block device must be associated with dosFs and a name must be assigned to it. This is done with the ***dosFsDevInit( )*** routine. Its parameters are the name to be used to identify the device, a pointer to the block device descriptor structure (**BLK_DEV**), and a pointer to the volume configuration structure **DOS_VOL_CONFIG** (see *4.2.4 Volume Configuration*, p.199). For example:

```
DOS_VOL_DESC *pVolDesc;
DOS_VOL_CONFIG configStruct;
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, &configStruct);
```

The ***dosFsDevInit( )*** call performs the following tasks:

- Assigns the specified name to the device and enters the device in the I/O system device table (with ***iosDevAdd( )***).
- Allocates and initializes the file system's volume descriptor for the device.
- Returns a pointer to the volume descriptor. This pointer is subsequently used to identify the volume during certain file system calls.

Initializing the device for use with dosFs does not format the disk, nor does it initialize the disk with MS-DOS structures (root directory, FAT, and so on). This permits using ***dosFsDevInit( )*** with disks that already have data in an existing MS-DOS file system; see *4.2.6 Using an Already Initialized Disk*, p.204. Formatting and DOS disk initialization can be done using the ***ioctl( )*** functions **FIODISKFORMAT** and **FIODISKINIT**, respectively.

The ***dosFsMkfs( )*** call provides an easier method of initializing a dosFs device; it does the following:

- Provides a set of default configuration values.
- Calls ***dosFsDevInit( )***.
- Initializes the disk structures using ***ioctl( )*** with the **FIODISKINIT** function.

The routine ***dosFsMkfs( )*** by default does not enable any dosFs-specific volume options (**DOS_OPT_CHANGENOWARN**, **DOS_OPT_AUTOSYNC**, **DOS_OPT_LONGNAMES**, **DOS_OPT_LOWERCASE**, or **DOS_OPT_EXPORT**). To enable any combination of these options, use ***dosFsMkfsOptionsSet( )*** before calling ***dosFsMkfs( )*** to initialize the disk. For more information on the default configuration values, see the manual entry for ***dosFsMkfs( )***.

4.2.4 Volume Configuration

The volume configuration structure, **DOS_VOL_CONFIG**, is used during the ***dosFsDevInit( )*** call. This structure contains various dosFs file system variables describing the layout of data on the disk. Most of the fields in the structure correspond to those in the boot sector. Table 4-1 lists the fields in the **DOS_VOL_CONFIG** structure.

Table 4-1 **DOS_VOL_CONFIG Fields**

Field	Description
dosvc_mediaByte	Media-descriptor byte
dosvc_secPerClust	Number of sectors per cluster
dosvc_nResrvd	Number of reserved sectors that precede the first FAT copy; the minimum is 1 (the boot sector)
dosvc_nFats	Number of FAT copies
dosvc_secPerFat	Number of sectors per FAT copy
dosvc_maxRootEnts	Maximum number of entries in root directory
dosvc_nHidden	Number of hidden sectors, normally 0
dosvc_options	VxWorks-specific file system options
dosvc_reserved	Reserved for future use by Wind River Systems

Calling ***dosFsConfigInit( )***is a convenient way to initialize **DOS_VOL_CONFIG**. It takes the configuration variables as parameters and fills in the structure. This is useful for initializing devices interactively from the Tornado shell (see the *Tornado User's Guide: Shell*). The **DOS_VOL_CONFIG** structure must be allocated *before* ***dosFsConfigInit( )*** is called.

DOS_VOL_CONFIG *Fields*

All but the last two **DOS_VOL_CONFIG** fields in Table 4-1 describe standard MS-DOS characteristics. The field **dosvc_options** is specific to the dosFs file system. Possible options for this field are shown in Table 4-2.

Table 4-2 **dosFs Volume Options**

Option	Hex Value	Description
DOS_OPT_CHANGENOWARN	0x1	Disk may be changed without warning.
DOS_OPT_AUTOSYNC	0x2	Synchronize disk during I/O.
DOS_OPT_LONGNAMES	0x4	Use case-sensitive file names not restricted to 8.3 convention.
DOS_OPT_EXPORT	0x8	Allow exporting using NFS.
DOS_OPT_LOWERCASE	0x40	Use lower case filenames on disk.

The first two options specify the action used to synchronize the disk buffers with the physical device. The remaining options involve extensions to dosFs capabilities.

DOS_OPT_CHANGENOWARN

Set this option if the device is a disk that can be replaced without being unmounted or having its change in ready-status declared. In this situation, check the disk regularly to determine whether it has changed. This causes significant overhead; thus, we recommend that you provide a mechanism that always synchronizes and unmounts a disk before it is removed, or at least announces a change in ready-status. If such a mechanism is in place, or if the disk is not removable, do not set this option. Auto-sync mode is enabled automatically when **DOS_OPT_CHANGENOWARN** is set (see the description for **DOS_OPT_AUTOSYNC**, next). For more information on **DOS_OPT_CHANGENOWARN**, see *4.2.17 Changing Disks*, p.211.

DOS_OPT_AUTOSYNC

Set this option to assure that directory and FAT data in the disk's buffers are written to the physical device as soon as possible after modification, rather than only when the file is closed. This can be desirable in situations where it is important that data be stored on the physical medium as soon as possible so as to avoid loss in the event of a system crash. There is a significant performance penalty incurred when using auto-sync mode; limit its use, therefore, to circumstances where there is a threat to data integrity.

However, **DOS_OPT_AUTOSYNC** does not make dosFs automatically write data to disk immediately after every ***write*( )**; doing so implies an extreme performance penalty. If your application requires this effect, use the ***ioctl*( )** function **FIOFLUSH** after every call to ***write*( )**.

Note that auto-sync mode is automatically enabled whenever **DOS_OPT_CHANGENOWARN** is set. For more information on auto-sync mode, see *4.2.17 Changing Disks*, p.211.

DOS_OPT_LONGNAMES
Set this option to allow the use of case-sensitive file names, with name lengths not restricted to MS-DOS's 8.3 convention. For more information on this option, see *4.2.18 Long Name Support*, p.214.

DOS_OPT_EXPORT
Set this option to initialize file systems that you intend to export using NFS. With this option, dosFs initialization creates additional in-memory data structures that are required to support the NFS protocol. While this option is necessary to initialize a file system that can be exported, it does not actually export the file system. See *Allowing Remote Access to VxWorks Files through NFS*, p.288.

DOS_OPT_LOWERCASE
Set this option to force filenames created by dosFs to use lowercase alphabetical characters. (Normally, filenames are created using uppercase characters, unless the **DOS_OPT_LONGNAMES** option is enabled.) This option may be required if the dosFs volume is mounted by a PC-based NFS client. This option has no effect if **DOS_OPT_LONGNAMES** is also specified.

Calculating Configuration Values

The values for **dosvc_secPerClust** and **dosvc_secPerFat** in the **DOS_VOL_CONFIG** structure must be calculated based on the particular device being used.

dosvc_secPerClust
This field specifies how many contiguous disk sectors make up a single cluster. Because a cluster is the smallest amount of disk space that can be allocated at a time, the size of a cluster determines how finely the disk allocation can be controlled. A large number of sectors per cluster causes more sectors to be allocated at a time and reduces the overall efficiency of disk space usage. For this reason, it is generally preferable to use the smallest possible number of sectors per cluster, although having less than two sectors per cluster is generally not necessary.

The maximum size of a FAT entry is 16 bits; thus, there is a maximum of 65,536 (64KB, or 0x10000) clusters that can be described. This is therefore the maximum number of clusters for a device. To determine the

appropriate number of sectors per cluster, divide the total number of sectors on the disk (the **bd_nBlocks** field in the device's **BLK_DEV** structure) by 0x10000 (64KB). Round up the resulting value to the next whole number. The final result is the number of sectors per cluster; place this value in the **dosvc_secPerClust** field in the **DOS_VOL_CONFIG** structure.

dosvc_secPerFat

This field specifies the number of sectors required on the disk for each copy of the FAT. To calculate this value, first determine the total number of clusters on the disk. The total number of clusters is equal to the total number of sectors (**bd_nBlocks** in the **BLK_DEV** structure) divided by the number of sectors per cluster. As mentioned previously, the maximum number of clusters on a disk is 64KB.

The cluster count must then be multiplied by the size of each FAT entry: if the total number of clusters is 4085 or less, each FAT entry requires 12 bits (1½ bytes); if the number of clusters is greater than 4085, each FAT entry requires 16 bits (2 bytes). The result of this multiplication is the total number of bytes required by each copy of the FAT. This byte count is then divided by the size of each sector (the **bd_bytesPerBlk** field in the **BLK_DEV** structure) to determine the number of sectors required for each FAT copy; if there is any remainder, add one (1) to the result. Place this final value in the **dosvc_secPerFat** field.

Assuming 512-byte sectors, the largest possible FAT (with entries describing 64KB clusters) occupies 256 sectors per copy, calculated as follows:

$$\frac{64\text{KB entries} \times 2\ \text{bytes/entry}}{512\ \text{bytes/sector}} = 256\ \text{sectors}$$

Standard Disk Configurations

For floppy disks, a number of standard disk configurations are used in MS-DOS systems. In general, these are uniquely identified by the media-descriptor byte value (at least for a given size of floppy disk), although some manufacturers have used duplicate values for different formats. Some widely used configurations are summarized in Table 4-3.

Fixed disks do not use standard disk configurations because they are rerely attached to a foreign system. Usually fixed disks use a media format byte of 0xF8.

Table 4-3 **MS-DOS Floppy Disk Configurations**

Capacity	160KB	180KB	320KB	360KB	1.2MB	720KB	1.44MB
Size	5.25"	5.25"	5.25"	5.25"	5.25"	3.5"	3.5"
Sides	1	1	2	2	2	2	2
Tracks	40	40	40	40	80	80	80
Sectors/Track	8	9	8	9	15	9	18
Bytes/Sector	512	512	512	512	512	512	512
secPerClust	1	1	2	2	1	2	1
nResrvd	1	1	1	1	1	1	1
nFats	2	2	2	2	2	2	2
maxRootEnts	64	64	112	112	224	112	224
mediaByte	0xFE	0xFC	0xFF	0xFD	0xF9	0xF9	0xF0
secPerFat	1	2	1	2	7	3	9
nHidden	0	0	0	0	0	0	0

4.2.5 Changes In Volume Configuration

As mentioned previously, various disk configuration parameters are specified when the dosFs file system device is first initialized using ***dosFsDevInit( )***. These parameters are kept in the volume descriptor, **DOS_VOL_DESC**, for the device. However, it is possible for a disk with different parameter values to be placed in a drive after the device is already initialized. If another disk is substituted for the one with the configuration parameters that were last entered into the volume descriptor, the configuration parameters of the new disk must be obtained before it can be used.

When a disk is mounted, the boot sector information is read from the disk. This data is used to update the configuration data in the volume descriptor. Note that this happens the first time the disk is accessed, and again after the volume is unmounted (using ***dosFsVolUnmount( )***) or a ready-change operation is performed. For more information, see *4.2.17 Changing Disks*, p.211.

This automatic re-initialization of the configuration data has an important implication. The volume descriptor data is used when initializing a disk (with

FIODISKINIT); thus, the disk is initialized with the configuration of the most recently mounted disk, regardless of the original specification during ***dosFsDevInit*( )**. Therefore, we recommend that you use **FIODISKINIT** immediately after ***dosFsDevInit*( )**, before any disk is mounted. (The device is opened in raw mode, the **FIODISKINIT** ***ioctl*( )** function is performed, and the device is closed.)

4.2.6 Using an Already Initialized Disk

If you are using a disk that is already initialized with an MS-DOS boot sector, FAT, and root directory (for example, by using the FORMAT utility in MS-DOS), it is not necessary to provide the volume configuration data during ***dosFsDevInit*( )**.

You can omit the MS-DOS configuration data by specifying a **NULL** pointer instead of the address of a **DOS_VOL_CONFIG** structure during ***dosFsDevInit*( )**. However, only use this method if you are sure that the first use of the volume is with a properly formatted and initialized disk.

When mounting an already initialized disk, all standard MS-DOS configuration values are obtained from the disk's boot sector. However, the options that are specific to dosFs must be determined differently.

Disks that are already initialized with the **DOS_OPT_LONGNAMES** (case-sensitive file names not restricted to 8.3 convention) option are recognized automatically by a specific volume ID string that is placed in the boot sector.

The **DOS_OPT_CHANGENOWARN**, **DOS_OPT_AUTOSYNC**, **DOS_OPT_LOWERCASE**, and **DOS_OPT_EXPORT** options are recorded only in memory, not on disk. Therefore they cannot be detected when you initialize a disk with **NULL** in place of the **DOS_VOL_CONFIG** structure pointer; you must re-enable them each time you mount a disk. You can set default values for these options with the ***dosFsDevInitOptionsSet*( )** routine: the defaults apply to any dosFs file systems you initialize with ***dosFsDevInit*( )** thereafter, unless you supply explicit **DOS_VOL_CONFIG** information.

You can also enable the **DOS_OPT_CHANGENOWARN** and **DOS_OPT_AUTOSYNC** options dynamically during disk operation, rather than during initialization, with the ***dosFsVolOptionsSet*( )** routine.

4.2.7 Accessing Volume Configuration Information

Disk configuration information is available using *dosFsConfigShow( )*[2] and *dosFsConfigGet( )* from the Tornado shell. See the *Tornado User's Guide: Shell.*

Use *dosFsConfigShow( )* to display configuration information such as the largest contiguous area and the device name. For example:

```
-> dosFsConfigShow "/RAM1/"
value = 0 = 0x0
```

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

```
device name:                    /RAM1/
total number of sectors:        400
bytes per sector:               512
media byte:                     0xf0
# of sectors per cluster:       2
# of reserved sectors:          1
# of FAT tables:                2
# of sectors per FAT:           1
max # of root dir entries:      112
# of hidden sectors:            0
removable medium:               FALSE
disk change w/out warning:      not enabled
auto-sync mode:                 not enabled
long file names:                not enabled
exportable file system:         not enabled
volume mode:                    O_RDWR (read/write)
available space:                199680 bytes
max avail. contig space:        199680 bytes
```

The *dosFsConfigGet( )* routine stores the disk configuration information in a **DOS_VOL_CONFIG** structure. This can be useful if you have a pre-existing disk and want to initialize a new disk with the same parameters, or if you initialized the dosFs file system on the disk using *dosFsMkfs( )* and need to obtain the actual configuration values that were calculated.

4.2.8 Mounting Volumes

A disk volume is *mounted* automatically, generally during the first *open( )* or *creat( )* operation for a file or directory on the disk. (Certain *ioctl( )* calls also cause the disk to be mounted.) If a NULL pointer is specified instead of the address of a **DOS_VOL_CONFIG** structure during the *dosFsDevInit( )* call, the disk is mounted immediately to obtain the configuration values.

---

2. If **INCLUDE_SHOW_ROUTINES** is defined in the VxWorks configuration; see *8. Configuration.*

When a disk is mounted, the boot sector, FAT, and directory data are read from the disk. The volume descriptor, **DOS_VOL_DESC**, is updated to reflect the configuration of the newly mounted disk.

Automatic mounting occurs on the first file access following ***dosFsVolUnmount( )*** or a ready-change operation (see *4.2.17 Changing Disks*, p.211), or periodically if the disk is defined during the ***dosFsDevInit( )*** call with the option **DOS_OPT_CHANGENOWARN** set. Automatic mounting does not occur when a disk is opened in raw mode; see *4.2.10 Opening the Whole Device (Raw Mode)*, p.206.

4.2.9 File I/O

Files on a dosFs file system device are created, deleted, written, and read using the standard VxWorks I/O routines: ***creat( )***, ***remove( )***, ***write( )***, and ***read( )***. See *3.3 Basic I/O*, p.112 for more information.

4.2.10 Opening the Whole Device (Raw Mode)

It is possible to open an entire dosFs volume. This is done by specifying only the device name during the ***open( )*** or ***creat( )*** call. A file descriptor is returned, as when a regular file is opened; however, operations on that file descriptor affect the entire device. Opening the entire volume in this manner is called *raw mode*.

The most common reason for opening the entire device is to obtain a file descriptor for an ***ioctl( )*** function that does not pertain to an individual file. An example is the **FIONFREE** function, which returns the number of available bytes on the volume. However, for many of these functions, the file descriptor can be any open file descriptor to the volume, even one for a specific file.

When a disk is initialized with MS-DOS data structures (boot sector, empty root directory, FAT), open the device in raw mode. The ***ioctl( )*** function **FIODISKINIT** performs the initialization.

You can both read and write data on a disk in raw mode. In this mode, the entire disk data area (that is, the disk portion following the boot sector, root directory, and FAT) is treated much like a single large file. No directory entry is made to describe any data written using raw mode.

For low-level I/O to an entire device, including the area used by MS-DOS data structures, see *4.4 Raw File System: rawFs*, p.225 and the reference entry for **rawFsLib**.

4.2.11 Creating Subdirectories

Subdirectories can be created in any directory at any time, except in the root directory if it has reached its maximum entry count. Subdirectories can be created in two ways:

1. Using ***ioctl( )*** with the **FIOMKDIR** function: The name of the directory to be created is passed as a parameter to ***ioctl( )***. The file descriptor used for the ***ioctl( )*** call is acquired either through opening the entire volume (raw mode), a regular file, or another directory on the volume.
2. Using ***open( )***: To create a directory, the **O_CREAT** option must be set in the *flags* parameter to open, and the **FSTAT_DIR** option must be set in the *mode* parameter. The ***open( )*** call returns a file descriptor that describes the new directory. Use this file descriptor for reading only and close it when it is no longer needed.

When creating a directory using either method, the new directory name must be specified. This name can be either a full path name or a path name relative to the current working directory.

4.2.12 Removing Subdirectories

A directory that is to be deleted must be empty (except for the "." and ".." entries). The root directory can never be deleted. There are two methods for removing directories:

1. Using ***ioctl( )*** call with the **FIORMDIR** function, specifying the name of the directory. Again, the file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
2. Using the ***remove( )*** function, specifying the name of the directory.

4.2.13 Directory Entries

Each dosFs directory contains a set of entries describing its files and immediate subdirectories. Each entry contains the following information about a file or subdirectory:

file name — an 8-byte string (padded with spaces, if necessary) specifying the base name of the file. (Names can be up to 40 characters; for details see *4.2.18 Long Name Support*, p.214.)

file extension	a 3-byte string (space-padded) specifying an optional extension to the file or subdirectory name. (If case-sensitive file names not restricted to the 8.3 convention are selected, the extension concept is not applicable.)
file attribute	a one-byte field specifying file characteristics; see *4.2.15 File Attributes*, p.208.
time	the encoded creation or modification time for the file.
date	the encoded creation or modification date for the file.
cluster number	the number of the starting cluster within the file. Subsequent clusters are found by searching the FAT.
file size	the size of the file, in bytes. This field is always 0 for entries describing subdirectories.

4.2.14 Reading Directory Entries

Directories on dosFs volumes can be searched using the ***opendir( )***, ***readdir( )***, ***rewinddir( )***, and ***closedir( )*** routines. These calls can be used to determine the names of files and subdirectories.

To obtain more detailed information about a specific file, use the ***fstat( )*** or ***stat( )*** function. Along with standard file information, the structure used by these routines also returns the file-attribute byte from a directory entry.

For more information, see the manual entry for **dirLib**.

4.2.15 File Attributes

The file-attribute byte in a dosFs directory entry consists of a set of flag bits, each indicating a particular file characteristic. The characteristics described by the file-attribute byte are shown in Table 4-4.

DOS_ATTR_RDONLY is checked when a file is opened for **O_WRONLY** or **O_RDWR**. If the flag is set, ***open( )*** returns **ERROR** and sets **errno** to **S_dosFsLib_READ_ONLY**.

Table 4-4 **Flags in the File-Attribute Byte**

VxWorks Flag Name	Hex value	Description
DOS_ATTR_RDONLY	0x01	read-only file
DOS_ATTR_HIDDEN	0x02	hidden file
DOS_ATTR_SYSTEM	0x04	system file
DOS_ATTR_VOL_LABEL	0x08	volume label
DOS_ATTR_DIRECTORY	0x10	subdirectory
DOS_ATTR_ARCHIVE	0x20	file is subject to archiving

NOTE: The MS-DOS hidden file and system file flags, **DOS_ATTR_HIDDEN** and **DOS_ATTR_SYSTEM**, are ignored by **dosFsLib**. If present, they are kept intact, but they produce no special handling (for example, entries with these flags are reported when searching directories).

The volume label flag, **DOS_ATTR_VOL_LABEL**, indicates that a directory entry contains the dosFs volume label for the disk. A label is not required. If used, there can be only one volume label entry per volume, in the root directory. The volume label entry is not reported when reading the contents of a directory (using ***readdir*( )**). It can only be determined using the ***ioctl*( )** function **FIOLABELGET**. The volume label can be set (or reset) to any string of 11 or fewer characters, using the ***ioctl*( )** function **FIOLABELSET**. Any file descriptor open to the volume can be used during these ***ioctl*( )** calls.

The directory flag, **DOS_ATTR_DIRECTORY**, indicates that this entry is not a regular file, but a subdirectory.

The archive flag, **DOS_ATTR_ARCHIVE**, is set when a file is created or modified. This flag is intended for use by other programs that search a volume for modified files and selectively archive them. Such a program must clear the archive flag since VxWorks does not.

All the flags in the attribute byte, except the directory and volume label flags, can be set or cleared using the ***ioctl*( )** function **FIOATTRIBSET**. This function is called after the opening of the specific file with the attributes to be changed. The attribute-byte value specified in the **FIOATTRIBSET** call is copied directly; to preserve existing flag settings, determine the current attributes using ***stat*( )** or ***fstat*( )**, then change them using bitwise *and* and *or* operations.

Example 4-1 **Setting DosFs File Attributes**

This example makes a dosFs file read-only, and leaves other attributes intact.

```
#include "vxWorks.h"
#include "ioLib.h"
#include "dosFsLib.h"
#include "sys/stat.h"
#include "fcntl.h"

STATUS changeAttributes (void)
    {
    int         fd;
    struct stat statStruct;

    /* open file */

    if ((fd = open ("file", O_RDONLY, 0)) == ERROR)
        return (ERROR);

    /* get directory entry data */

    if (fstat (fd, &statStruct) == ERROR)
        return (ERROR);

    /* set read-only flag on file */

    if (ioctl (fd, FIOATTRIBSET, (statStruct.st_attrib | DOS_ATTR_RDONLY))
        == ERROR)
        return (ERROR);

    /* close file */

    close (fd);
    }
```

4.2.16 File Date and Time

Directory entries contain a time and date for each file or directory. This time is set when the file is created, and it is updated when a file that was modified is closed. Entries describing subdirectories are not updated—they always contain the creation date and time for the subdirectory.

The **dosFsLib** library maintains the date and time in an internal structure. While there is currently no mechanism for automatically advancing the date or time, two different methods for setting the date and time are provided.

The first method involves using two routines, *dosFsDateSet( )* and *dosFsTimeSet( )*. The following examples illustrate their use:

```
dosFsDateSet (1990, 12, 25);        /* set date to Dec-25-1990 */
dosFsTimeSet (14, 30, 22);          /* set time to 14:30:22    */
```

These routines must be called periodically to update the time and date values.

The second method requires a user-supplied hook routine. If a time and date hook routine is installed using ***dosFsDateTimeInstall( )***, that routine is called whenever **dosFsLib** requires the current date and time. You can use this to take advantage of hardware time-of-day clocks that can be read to obtain the current time. It can also be used with other applications that maintain actual time and date.

Define the date/time hook routine as follows (the name ***dateTimeHook*** is an example; the actual routine name can be anything):

```
void dateTimeHook
    (
    DOS_DATE_TIME * pDateTime    /* ptr to dosFs date & time struct */
    )
```

On entry to the hook routine, the **DOS_DATE_TIME** structure contains the last time and date set in **dosFsLib**. Next, the hook routine fills the structure with the correct values for the current time and date. Unchanged fields in the structure retain their previous values.

The MS-DOS specification provides only for 2-second granularity in file time-stamps. If the number of seconds in the time specified during ***dosFsTimeSet( )*** or the date/time hook routine is odd, it is rounded down to the next even number.

The date and time used by **dosFsLib** is initially Jan-01-1980, 00:00:00.

4.2.17 Changing Disks

To increase performance, the dosFs file system keeps in memory copies of directory entries and the file allocation table (FAT) for each mounted volume. While this greatly speeds up access to files, it requires that **dosFsLib** be notified when removable disks are changed (for example, when floppies are swapped). Two different notification methods are provided: (1) ***dosFsVolUnmount( )*** and (2) the ready-change mechanism. The following sections are not generally applicable for non-removable media (although ***dosFsVolUnmount( )*** can be useful in system shutdown situations).

Unmounting Volumes

The preferred method of announcing a disk change is to call ***dosFsVolUnmount( )*** prior to removing the disk. This call flushes all modified data structures to disk if possible (see *Synchronizing Volumes*, p.213) and also marks any open file descriptors as obsolete. During the next I/O operation, the disk is remounted. The ***ioctl( )*** call can also be used to initiate ***dosFsVolUnmount( )***, by specifying the

FIOUNMOUNT function code. Any open file descriptor to the device can be used in the ***ioctl( )*** call.

Subsequent attempts to use obsolete file descriptors for I/O operations return an **S_dosFsLib_FD_OBSOLETE** error. To free such file descriptors, use ***close( )***, as usual. This returns **S_dosFsLib_FD_OBSOLETE** as well, but it successfully frees the descriptor. File descriptors acquired when opening the entire volume (raw mode) are not marked as obsolete during ***dosFsVolUnmount( )*** and can still be used.

ISRs must not call ***dosFsVolUnmount( )*** directly, because it is possible for the call to pend while the device becomes available. The ISR can instead give a semaphore that prompts a task to unmount the volume. (Note that ***dosFsReadyChange( )*** can be called directly from ISRs; see *Announcing Disk Changes with Ready-Change*, p.212.)

When ***dosFsVolUnmount( )*** is called, it attempts to write buffered data out to the disk. Its use is therefore inappropriate for situations where the disk-change notification does not occur until a new disk is inserted, because the old buffered data would be written to the new disk. In this case, use ***dosFsReadyChange( )***, which is described in *Announcing Disk Changes with Ready-Change*, p.212.

If ***dosFsVolUnmount( )*** is called after the disk is physically removed, the data-flushing portion of its operation fails. However, the file descriptors are still marked as obsolete and the disk is marked as requiring remounting. In this situation, ***dosFsVolUnmount( )*** does *not* return an error. To avoid lost data, explicitly synchronize the disk before removing it (see *Synchronizing Volumes*, p.213).

Announcing Disk Changes with Ready-Change

The second method of informing **dosFsLib** that a disk change is taking place is with the *ready-change* mechanism. A change in the disk's ready-status is interpreted by **dosFsLib** as indicating that the disk must be remounted before the next I/O operation.

There are three ways to announce a ready-change:

- By calling ***dosFsReadyChange( )*** directly.
- By calling ***ioctl( )*** with the **FIODISKCHANGE** function.
- By having the device driver set the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE; this has the same effect as notifying **dosFsLib** directly.

The ready-change mechanism does not provide the ability to flush data structures to the disk. It merely marks the volume as needing remounting. Thus, buffered

data (data written to files, directory entries, or FAT changes) can be lost. This can be avoided by synchronizing the disk before asserting ready-change (see *Synchronizing Volumes*, p.213). The combination of synchronizing and asserting ready-change provides all the functionality of ***dosFsVolUnmount( )***, except for marking file descriptors as obsolete.

Ready-change can be used in ISRs, because it does not attempt to flush data or perform other operations that could cause delay.

The block device driver status-check routine (identified by the **bd_statusChk** field in the **BLK_DEV** structure) can be useful for asserting ready-change for devices that detect a disk change only after the new disk is inserted. This routine is called at the beginning of each ***open( )*** or ***creat( )*** operation, before the file system checks for ready-change. See *3.9.4 Block Devices*, p.171.

Disks with No Change Notification

If it is not possible for ***dosFsVolUnmount( )*** to be called or a ready-change to be announced, then each time the disk is changed, the device must be specially identified when it is initialized for use with the file system. This is done by setting **DOS_OPT_CHANGENOWARN** in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure when calling ***dosFsDevInit( )***; see *4.2.4 Volume Configuration*, p.199.

This configuration option results in a significant performance penalty, because the disk configuration data must be read in regularly from the physical disk (in case it was removed and a new one inserted). In addition, setting **DOS_OPT_CHANGENOWARN** also enables auto-sync mode; see *Auto-Sync Mode*, p.214. Note that all that is required for disk change notification is that either the ***dosFsVolUnmount( )*** call or ready-change be issued each time the disk is changed. It is not necessary that it be called from the device driver or an ISR. For example, if your application provided a user interface through which an operator could enter a command resulting in an ***dosFsVolUnmount( )*** call before removing the disk, that would be sufficient, and **DOS_OPT_CHANGENOWARN** does not need to be set. However, it is important that the operator follow such a procedure strictly.

Synchronizing Volumes

When a disk is *synchronized*, all modified buffered data is physically written to the disk, so that the disk is up to date. This includes data written to files, updated directory information, and the FAT.

To avoid loss of data, synchronize a disk before removing it. You may need to explicitly synchronize a disk, depending on when (or if) ***dosFsVolUnmount( )*** is called. If your application does not call this routine, or it is called after the disk is removed, use ***ioctl( )*** to explicitly write the data to the device.

When ***dosFsVolUnmount( )*** is called, an attempt is made to synchronize the device before unmounting. If the disk is still present and writable at the time of the call, synchronization takes place, and no further action is required to protect the integrity of the data written to it before it is dismounted. However, if the ***dosFsVolUnmount( )*** call is made after a disk is removed, it is obviously too late to synchronize, and ***dosFsVolUnmount( )*** discards the buffered data.

To explicitly synchronize a disk before it is removed, use ***ioctl( )*** specifying the **FIOSYNC** function. (This could be done in response to an operator command.) Do this if the ***dosFsVolUnmount( )*** call is made after a disk is removed or if the routine ***dosFsVolUnmount( )*** is never called. The file descriptor used during the ***ioctl( )*** call is obtained when the whole volume (raw mode) is opened.

Auto-Sync Mode

dosFsLib provides a modified mode of synchronization called *auto-sync*. When this option is enabled, data for modified directories and the FAT are physically written to these devices as soon as they are logically altered. (Otherwise, such changes are not necessarily written out until the involved file is closed.)

Auto-sync mode is enabled by setting **DOS_OPT_AUTOSYNC** in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure when ***dosFsDevInit( )*** is called; see *4.2.4 Volume Configuration*, p.199. Auto-sync mode is automatically enabled if the volume does not have disk change notification (that is, if **DOS_OPT_CHANGENOWARN** is set by ***dosFsDevInit( )***).

Auto-sync results in a performance penalty, but it provides the highest level of data security, because it minimizes the period during which directory and FAT data are not up to date on the disk. Auto-sync is often desirable for applications where data integrity is threatened by events such as a system crash.

4.2.18 Long Name Support

The dosFs long name support allows the use of case-sensitive file names longer than MS-DOS's 8.3 convention. These names can be up to 40 characters long and can be made up of any ASCII characters. In addition, a dot (.), which in MS-DOS indicates a file-name extension, has no special significance.

Long name support is enabled by setting **DOS_OPT_LONGNAMES** in the **dosvc_options** field of the **DOS_VOL_CONFIG** structure when calling ***dosFsDevInit( )***.

WARNING: If you use this feature, the disk is no longer MS-DOS compatible. Use long name support only for storing data local to VxWorks, on a disk that is initialized on a VxWorks system using ***dosFsDevInit( )*** or ***dosFsMkfs( )***.

4.2.19 Contiguous File Support

The dosFs file system provides efficient handling of *contiguous files*. A contiguous file is made up of a series of consecutive disk sectors. This capability includes both the allocation of contiguous space to a specified file (or directory) and optimized access to such a file.

To allocate a contiguous area to a file, first create the file in the normal fashion, using ***open( )*** or ***creat( )***. Then use the file descriptor returned during the creation of the file to make the ***ioctl( )*** call, specifying the **FIOCONTIG** function. The parameter to ***ioctl( )*** with the **FIOCONTIG** function is the size of the requested contiguous area, in bytes. The FAT is searched for a suitable section of the disk, and if found, it is assigned to the file. (If there is no contiguous area on the volume large enough to satisfy the request, an error is returned.) The file can then be closed, or it can be used for further I/O operations.

Example 4-2 **Creating a DosFs Contiguous File**

This example creates a dosFs file and allocates 0x10000 contiguous bytes to it.

```
#include "vxWorks.h"
#include "ioLib.h"
#include "fcntl.h"

STATUS fileContigTest (void)
    {
    int fd;
    STATUS status;

    /* open file */

    if ((fd = creat ("file", O_RDWR)) == ERROR)
        return (ERROR);

    /* get contiguous area */

    status = ioctl (fd, FIOCONTIG, 0x10000);
    if (status != OK)
```

```
    /* do error handling */
      printf ("ERROR");
  /* use file */
  /* close file */
  close (fd);
  }
```

It is also possible to request the largest available contiguous space. Use **CONTIG_MAX** for the size of the contiguous area. For example:

```
status = ioctl (fd, FIOCONTIG, CONTIG_MAX);
```

It is important that the file descriptor used for the ***ioctl*( )** call be the only descriptor open to the file. Furthermore, because a file can be assigned a different area of the disk than is originally allocated, perform the ***ioctl*( ) FIOCONTIG** operation before any data is written to the file.

To deallocate unused reserved bytes, use the POSIX-compatible routine ***ftruncate*( )** or the ***ioctl*( )** function **FIOTRUNC**.

Subdirectories can also be allocated a contiguous disk area in the same manner. If the directory is created using the ***ioctl*( )** function **FIOMKDIR**, it must be explicitly opened to obtain a file descriptor to it; if the directory is created using options to ***open*( )**, the returned file descriptor from that call can be used. A directory must be empty (except for the "." and ".." entries) when it has contiguous space allocated to it.

When any file is opened, it is checked for contiguity. If a file is recognized as contiguous, a more efficient technique for locating specific sections of the file is used, rather than following cluster chains in the FAT, as must be done for fragmented files. This enhanced handling of contiguous files takes place regardless of whether the space is explicitly allocated using **FIOCONTIG**.

To find the maximum contiguous area on a device, use the ***ioctl*( )** function **FIONCONTIG**. This information can also be displayed by ***dosFsConfigShow*( )** if **INCLUDE_SHOW_ROUTINES** is defined in the VxWorks configuration; see *8. Configuration*.

Example 4-3 **Finding the Maximum Contiguous Area on a DosFs Device**

In this example, the size (in bytes) of the largest contiguous area is copied to the integer pointed to by the third parameter to ***ioctl*( )** (*count*).

```
#include "vxWorks.h"
#include "fcntl.h"
#include "ioLib.h"
```

```
STATUS contigTest (void)
    {
    int count;
    int fd;

    /* open device in raw mode */
    if ((fd = open ("/DEV1/", O_RDONLY, 0)) == ERROR)
        return (ERROR);

    /* find max contiguous area */
    ioctl (fd, FIONCONTIG, &count);

    /* close device and display size of largest contiguous area */
    close (fd);
    printf ("largest contiguous area = %d\n", count);
    }
```


4.2.20 *I/O Control Functions Supported by* dosFsLib

The dosFs file system supports the *ioctl( )* functions listed in Table 4-5. These functions are defined in the header file **ioLib.h**. For more information, see the manual entries for **dosFsLib** and for *ioctl( )* in **ioLib**.

Table 4-5 **I/O Control Functions Supported by dosFsLib**

Function	Decimal Value	Description
FIOATTRIBSET	35	Set the file-attribute byte in the dosFs directory entry.
FIOCONTIG	36	Allocate contiguous disk space for a file or directory.
FIODISKCHANGE	13	Announce a media change.
FIODISKFORMAT	5	Format the disk (device driver function).
FIODISKINIT	6	Initialize a dosFs file system on a disk volume.
FIOFLUSH	2	Flush the file output buffer.
FIOFSTATGET	38	Get file status information (directory entry data).
FIOGETNAME	18	Get the file name of the *fd*.
FIOLABELGET	33	Get the volume label.
FIOLABELSET	34	Set the volume label.
FIOMKDIR	31	Create a new directory.

Table 4-5 **I/O Control Functions Supported by dosFsLib** *(Continued)*

Function	Decimal Value	Description
FIONCONTIG	41	Get the size of the maximum contiguous area on a device.
FIONFREE	30	Get the number of free bytes on the volume.
FIONREAD	1	Get the number of unread bytes in a file.
FIOREADDIR	37	Read the next directory entry.
FIORENAME	10	Rename a file or directory.
FIORMDIR	32	Remove a directory.
FIOSEEK	7	Set the current byte offset in a file.
FIOSYNC	21	Same as **FIOFLUSH**, but also re-reads buffered file data.
FIOTRUNC	42	Truncate a file to a specified length.
FIOUNMOUNT	39	Unmount a disk volume.
FIOWHERE	8	Return the current byte position in a file.

4.2.21 Booting from a Local dosFs File System Using SCSI

VxWorks can be booted from a local SCSI device. Before you can boot from SCSI, you must make new boot ROMs that contain the SCSI library. Define the constants **INCLUDE_SCSI** and **INCLUDE_SCSI_BOOT** in **config.h** and rebuild **bootrom.hex** (see the *Tornado User's Guide: Cross-Development*).

After burning the SCSI boot ROMs, you can prepare the dosFs file system for use as a boot device. The simplest way to do this is to partition the SCSI device so that a dosFs file system starts at block 0. You can then make the new **vxWorks** image, place it on your SCSI boot device, and boot the new VxWorks system. These steps are shown in more detail below.

WARNING: For use as a boot device, the directory name for the dosFs file system must begin and end with slashes (as with **/sd0/** used in the following example). This is an exception to the usual naming convention for dosFs file systems.

1. Create the SCSI device using ***scsiPhysDevCreate( )*** (see *SCSI Drivers*, p.141), and initialize the disk with a dosFs file system (see *4.2.2 Initializing the dosFs File System*, p.197). Modify the file **src/config/usrScsiConfig.c** to reflect your

SCSI configuration. The following example creates a SCSI device with a dosFs file system spanning the full device:

```
pPhysDev = scsiPhysDevCreate (pSysScsiCtrl, 2, 0, 0, -1, 0, 0, 0);
pBlkDev = scsiBlkDevCreate (pPhysDev, 0, 0);
dosFsDevInit ("/sd0/", pBlkDev, 0);
```

2. Remake VxWorks and copy the new kernel to the drive:[3]

```
-> copy "unixHost:/usr/wind/target/config/bspname/vxWorks", \
"/sd0/vxWorks"
```

3. Reboot the system, and then change the boot parameters. Boot device parameters for SCSI devices follow this format:

   scsi=*id,lun*

   where *id* is the SCSI ID of the boot device, and *lun* is its Logical Unit Number (LUN). To enable use of the network, include the on-board Ethernet device (for example, **ln** for LANCE) in the *other* field. The following example boots from a SCSI device with a SCSI ID of 2 and a LUN of 0.

```
[VxWorks Boot]: @
boot device            : scsi=2,0
processor number       : 0
host name              : host
file name              : /sd0/vxWorks
inet on ethernet (e)   : 147.11.1.222:ffffff00
host inet (h)          : 147.11.1.3
user (u)               : jane
flags (f)              : 0x0
target name (tn)       : t222
other                  : ln
Attaching to scsi device... done.
Loading /sd0/vxWorks... 378060 + 27484 + 21544
Starting at 0x1000...
```

3. If you are using the target shell and **INCLUDE_NET_SYM_TBL** is defined in your VxWorks configuration, you must also copy the symbol table to the drive, as follows:
   `-> copy "unixHost:/usr/wind/target/config/bspname/vxWorks.sym", "/sd0/vxWorks.sym"`

4.3 RT-11-Compatible File System: rt11Fs

VxWorks provides the file system rt11Fs, which is compatible with the RT-11 file system. It is provided primarily for compatibility with earlier versions of VxWorks. Normally, the dosFs file system is the preferred choice, because it offers such enhancements as optional contiguous file allocation, flexible file naming, and so on.

WARNING: The rt11Fs file system is considered obsolescent. In a future release of VxWorks, rt11Fs may not be supported.

4.3.1 Disk Organization

The rtllFs file system uses a simple disk organization. Although this simplicity results in some loss of flexibility, rt11Fs is suitable for many real-time applications.

The rt11Fs file system maintains only *contiguous files*. A contiguous file consists of a series of disk sectors that are consecutive. Contiguous files are well-suited to real-time applications because little time is spent locating specific portions of a file. The disadvantage of using contiguous files exclusively is that a disk can gradually become fragmented, reducing the efficiency of the disk space allocation.

The rt11Fs disk format uses a single directory to describe all files on the disk. The size of this directory is limited to a fixed number of directory entries. Along with regular files, unused areas of the disk are also described by special directory entries. These special entries are used to keep track of individual sections of free space on the disk.

4.3.2 Initializing the rt11Fs File System

Before any other operations can be performed, the rt11Fs file system library, **rt11FsLib**, must be initialized by calling ***rt11FsInit( )***. This routine takes a single parameter, the maximum number of rt11Fs file descriptors that can be open at one time. This count is used to allocate a set of descriptors; a descriptor is used each time a file or an rt11Fs device is opened.

The ***rt11FsInit( )*** routine also makes an entry for the rt11Fs file system in the I/O system driver table (with ***iosDrvInstall( )***). This entry specifies entry points for the rt11Fs file operations and is used for all devices that use the rt11Fs file system. The driver number assigned to the rt11Fs file systems is placed in a global variable **rt11FsDrvNum**.

The *rt11FsInit( )* routine is normally called by the *usrRoot( )* task after starting the VxWorks system. To use this initialization, make sure the symbol **INCLUDE_RT11FS** is defined in the configuration file **configAll.h**, and set **NUM_RT11FS_FILES** to the desired maximum open file count.

4.3.3 Initializing a Device for Use with rt11Fs

After the rt11Fs file system is initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine (*xxDevCreate( )*). The driver routine returns a pointer to a block device descriptor structure (**BLK_DEV**). The **BLK_DEV** structure describes the physical aspects of the device and specifies the routines in the device driver that a file system can call. For more information about block devices, see *3.9.4 Block Devices*, p.171.

Immediately after its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with rt11Fs, the already-created block device must be associated with rt11Fs and must have a name assigned to it. This is done with *rt11FsDevInit( )*. Its parameters are:

- the name to be used to identify the device
- a pointer to the **BLK_DEV** structure
- a boolean value indicating whether the disk uses standard RT-11 skew and interleave
- the number of entries to be used in the disk directory (in some cases, the actual number used is greater than the number specified)
- a boolean value indicating whether this disk is subject to being changed without notification to the file system

For example:

```
RT_VOL_DESC  *pVolDesc;
pVolDesc = rt11FsDevInit ("DEV1:", pBlkDev, rtFmt, nEntries, changeNoWarn);
```

The *rt11FsDevInit( )* call assigns the specified name to the device and enters the device in the I/O system device table (with *iosDevAdd( )*). It also allocates and initializes the file system's volume descriptor for the device. It returns a pointer to the volume descriptor to the caller; this pointer is used to identify the volume during some file system calls.

Note that initializing the device for use with the rt11Fs file system does not format the disk, nor does it initialize the rt11Fs disk directory. These are done using *ioctl( )* with the functions **FIODISKFORMAT** and **FIODISKINIT**, respectively.

4.3.4 Mounting Volumes

A disk volume is *mounted* automatically, generally during the first ***open( )*** or ***creat( )*** for a file or directory on the disk. (Certain ***ioctl( )*** functions also cause the disk to be mounted.) When a disk is mounted, the directory data is read it.

Automatic mounting reoccurs on the first file access following a ready-change operation (see *4.3.8 Changing Disks*, p.223) or periodically if the disk is defined during the ***rt11FsDevInit( )*** call with the **changeNoWarn** parameter set to TRUE. Automatic mounting does not occur when a disk is opened in raw mode. For more information, see *4.3.6 Opening the Whole Device (Raw Mode)*, p.222.

4.3.5 File I/O

Files on an rt11Fs file system device are created, deleted, written, and read using the standard VxWorks I/O routines: ***creat( )***, ***remove( )***, ***write( )***, and ***read( )***. The size of an rt11Fs file is determined during its initial ***open( )*** or ***creat( )***. Once closed, additional space cannot be allocated to the file. For more information, see *3.3 Basic I/O*, p.112.

4.3.6 Opening the Whole Device (Raw Mode)

It is possible to open an entire rt11Fs volume by specifying only the device name during the ***open( )*** or ***creat( )*** call. A file descriptor is returned, as when opening a regular file; however, operations on that file descriptor affect the entire device. Opening the entire volume in this manner is called *raw mode*.

The most common reason for opening the entire device is to obtain a file descriptor to perform an ***ioctl( )*** function that does not pertain to an individual file. An example is the **FIOSQUEEZE** function, which combines fragmented free space across the entire volume.

When a disk is initialized with an rt11Fs directory, open the device in raw mode. The ***ioctl( )*** function **FIODISKINIT** performs the initialization.

A disk can be read or written in raw mode. In this case, the entire disk area is treated much like a single large file. No directory entry is made to describe any data written using raw mode, and care must be taken to avoid overwriting the regular rt11Fs directory at the beginning of the disk. This type of I/O is also provided by **rawFsLib**.

4.3.7 Reclaiming Fragmented Free Disk Space

As previously mentioned, the contiguous file allocation scheme used by the rt11Fs file system can gradually result in disk fragmentation. In this situation, the available free space on the disk is scattered in a number of small chunks. This reduces the ability of the system to create new files.

To correct this condition, **rt11FsLib** includes the *ioctl( )* function **FIOSQUEEZE**. This routine moves files so that the free space is combined at the end of the disk. When you call *ioctl( )* with **FIOSQUEEZE**, it is critical that there be no open files on the device. With large disks, this call may require considerable time to execute.

4.3.8 Changing Disks

To increase performance, rt11Fs keeps copies of directory entries for each volume in memory. While this greatly speeds up access to files, it requires that **rt11FsLib** be notified when removable disks are changed (for example, when floppies are swapped). This notification is provided by the ready-change mechanism.

Announcing Disk Changes with Ready-Change

A change in ready-status is interpreted by **rt11FsLib** to mean that the disk must be remounted during the next I/O operation. There are three ways to announce a ready-change:

- By calling *rt11FsReadyChange( )* directly.
- By calling *ioctl( )* with **FIODISKCHANGE**.
- By having the device driver set the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE; this has the same effect as notifying **rt11FsLib** directly.

The ready-change announcement does not cause buffered data to be flushed to the disk; it merely marks the volume as needing remounting. As a result, data written to files or directory entry changes can be lost. To avoid this loss of data, close all files on the volume before changing the disk.

Ready-change can be used in ISRs, because it does not attempt to flush data or perform other operations that could cause delay.

The block device driver status-check routine (identified by the **bd_statusChk** field in the **BLK_DEV** structure) can be useful for asserting ready-change for devices that only detect a disk change after the new disk is inserted. This routine is called at the start of each *open( )* or *creat( )*, before the file system checks for ready-change.

Disks with No Change Notification

If it is not possible for a ready-change to be announced each time the disk is changed, the device must be specially identified when it is initialized for use with the file system. This is done by setting the **changeNoWarn** parameter to TRUE when calling *rt11FsDevInit( )*.

When this parameter is defined as TRUE, the disk is checked regularly to obtain the current directory information (in case the disk is removed and a new one inserted). As a result, this option causes a significant loss in performance.

Table 4-6 **I/O Control Functions Supported by rt11FsLib**

Function	Decimal Value	Description
FIODIRENTRY	9	Get information about specified device directory entries.
FIODISKCHANGE	13	Announce a media change.
FIODISKFORMAT	5	Format the disk.
FIODISKINIT	6	Initialize an rt11Fs file system on a disk volume.
FIOFLUSH	2	Flush the file output buffer.
FIOFSTATGET	38	Get file status information (directory entry data).
FIOGETNAME	18	Get the file name of the *fd*.
FIONREAD	1	Get the number of unread bytes in a file.
FIOREADDIR	37	Read the next directory entry.
FIORENAME	10	Rename a file.
FIOSEEK	7	Reset the current byte offset in a file.
FIOSQUEEZE	15	Coalesce fragmented free space on an rt11Fs volume.
FIOWHERE	8	Return the current byte position in a file.

4.3.9 I/O Control Functions Supported by rt11FsLib

The rt11Fs file system supports the *ioctl( )* functions shown in Table 4-6. The functions listed are defined in the header file **ioLib.h.** For more information, see the manual entries for **rt11FsLib** and for *ioctl( )* in **ioLib**.

4.4 Raw File System: rawFs

VxWorks provides a minimal "file system," rawFs, for use in systems that require only the most basic disk I/O functions. The rawFs file system, implemented in **rawFsLib**, treats the entire disk volume much like a single large file. Although the dosFs and rt11Fs file systems do provide this ability to varying degrees, the rawFs file system offers advantages in size and performance if more complex functions are not required.

4.4.1 Disk Organization

As mentioned previously, rawFs imposes no organization of the data on the disk.

The rawFs file system maintains no directory information; thus there is no division of the disk area into specific files, and no file names are used. All *open( )* operations on rawFs devices specify only the device name; no additional file names are allowed.

The entire disk area is available to any file descriptor that is open for the device. All read and write operations to the disk use a byte-offset relative to the start of the first block on the disk.

4.4.2 Initializing the rawFs File System

Before any other operations can be performed, the rawFs library, **rawFsLib**, must be initialized by calling *rawFsInit( )*. This routine takes a single parameter, the maximum number of rawFs file descriptors that can be open at one time. This count is used to allocate a set of descriptors; a descriptor is used each time a rawFs device is opened.

The *rawFsInit( )* routine also makes an entry for the rawFs file system in the I/O system driver table (with *iosDrvInstall( )*). This entry specifies the entry points for rawFs file operations and is for all devices that use the rawFs file system. The driver number assigned to the rawFs file systems is placed in a global variable **rawFsDrvNum**.

The *rawFsInit( )* routine is normally called by the *usrRoot( )* task after starting the VxWorks system. To use this initialization, define the symbol **INCLUDE_RAWFS** in **configAll.h**, and set **NUM_RAWFS_FILES** to the desired maximum open file descriptor count.

4.4.3 Initializing a Device for Use with the rawFs File System

After the rawFs file system is initialized, the next step is to create one or more devices. Devices are created by the device driver's device creation routine (*xxDevCreate*( )). The driver routine returns a pointer to a block device descriptor structure (**BLK_DEV**). The **BLK_DEV** structure describes the physical aspects of the device and specifies the routines in the device driver that a file system can call. For more information on block devices, see *3.9.4 Block Devices*, p.171.

Immediately after its creation, the block device has neither a name nor a file system associated with it. To initialize a block device for use with rawFs, the already-created block device must be associated with rawFs and a name must be assigned to it. This is done with the *rawFsDevInit*( ) routine. Its parameters are the name to be used to identify the device and a pointer to the block device descriptor structure (**BLK_DEV**):

```
RAW_VOL_DESC *pVolDesc;
BLK_DEV      *pBlkDev;
pVolDesc = rawFsDevInit ("DEV1:", pBlkDev);
```

The *rawFsDevInit*( ) call assigns the specified name to the device and enters the device in the I/O system device table (with *iosDevAdd*( )). It also allocates and initializes the file system's volume descriptor for the device. It returns a pointer to the volume descriptor to the caller; this pointer is used to identify the volume during certain file system calls.

Note that initializing the device for use with rawFs does not format the disk. That is done using an *ioctl*( ) call with the **FIODISKFORMAT** function.

No disk initialization (**FIODISKINIT**) is required, because there are no file system structures on the disk. Note, however, that rawFs accepts that *ioctl*( ) function code for compatibility with other file systems; in such cases, it performs no action and always returns **OK**.

4.4.4 Mounting Volumes

A disk volume is *mounted* automatically, generally during the first *open*( ) or *creat*( ) operation. (Certain *ioctl*( ) functions also cause the disk to be mounted.) The volume is again mounted automatically on the first disk access following a ready-change operation (see *4.4.6 Changing Disks*, p.227).

4.4.5 File I/O

To begin I/O to a rawFs device, first open the device using the standard ***open( )*** function. (The ***creat( )*** function can be used instead, although nothing is actually "created.") Data on the rawFs device is written and read using the standard I/O routines ***write( )*** and ***read( )***. For more information, see *3.3 Basic I/O*, p.112.

The character pointer associated with a file descriptor (that is, the byte offset where reads and writes take place) can be set by using ***ioctl( )*** with the **FIOSEEK** function.

Multiple file descriptors can be open simultaneously for a single device. These must be carefully managed to avoid modifying data that is also being used by another file descriptor. In most cases, such multiple open descriptors use **FIOSEEK** to set their character pointers to separate disk areas.

4.4.6 Changing Disks

The rawFs file system must be notified when removable disks are changed (for example, when floppies are swapped). Two different notification methods are provided: (1) ***rawFsVolUnmount( )*** and (2) the ready-change mechanism.

Unmounting Volumes

The first method of announcing a disk change is to call ***rawFsVolUnmount( )*** prior to removing the disk. This call flushes all modified file descriptor buffers if possible (see *Synchronizing Volumes*, p.229) and also marks any open file descriptors as obsolete. The next I/O operation remounts the disk. Calling ***ioctl( )*** with **FIOUNMOUNT** is equivalent to using ***rawFsVolUnmount( )***. Any open file descriptor to the device can be used in the ***ioctl( )*** call.

Attempts to use obsolete file descriptors for further I/O operations produce an **S_rawFsLib_FD_OBSOLETE** error. To free an obsolete descriptor, use ***close( )***, as usual. This frees the descriptor even though it produces the same error.

ISRs must not call ***rawFsVolUnmount( )*** directly, because the call can pend while the device becomes available. The ISR can instead give a semaphore that prompts a task to unmount the volume. (Note that ***rawFsReadyChange( )*** can be called directly from ISRs; see *Announcing Disk Changes with Ready-Change*, p.228.)

When ***rawFsVolUnmount( )*** is called, it attempts to write buffered data out to the disk. Its use is therefore inappropriate for situations where the disk-change notification does not occur until a new disk is inserted, because the old buffered

data would be written to the new disk. In this case, use ***rawFsReadyChange( )***, which is described in *Announcing Disk Changes with Ready-Change*, p.228.

If ***rawFsVolUnmount( )*** is called after the disk is physically removed, the data flushing portion of its operation fails. However, the file descriptors are still marked as obsolete, and the disk is marked as requiring remounting. An error is *not* returned by ***rawFsVolUnmount( )***; to avoid lost data in this situation, explicitly synchronize the disk before removing it (see *Synchronizing Volumes*, p.229).

Announcing Disk Changes with Ready-Change

The second method of announcing that a disk change is taking place is with the *ready-change* mechanism. A change in the disk's ready-status is interpreted by **rawFsLib** to indicate that the disk must be remounted during the next I/O call.

There are three ways to announce a ready-change:

- By calling ***rawFsReadyChange( )*** directly.
- By calling ***ioctl( )*** with **FIODISKCHANGE**.
- By having the device driver set the **bd_readyChanged** field in the **BLK_DEV** structure to TRUE; this has the same effect as notifying **rawFsLib** directly.

The ready-change announcement does not cause buffered data to be flushed to the disk. It merely marks the volume as needing remounting. As a result, data written to files can be lost. This can be avoided by synchronizing the disk before asserting ready-change. The combination of synchronizing and asserting ready-change provides all the functionality of ***rawFsVolUnmount( )*** except for marking file descriptors as obsolete.

Ready-change can be used in ISRs, because it does not attempt to flush data or perform other operations that could cause delay.

The block device driver status-check routine (identified by the **bd_statusChk** field in the **BLK_DEV** structure)is useful for asserting ready-change for devices that only detect a disk change after the new disk is inserted. This routine is called at the beginning of each ***open( )*** or ***creat( )***, before the file system checks for ready-change.

Disks with No Change Notification

If it is not possible for a ready-change to be announced each time the disk is changed, close all file descriptors for the volume before changing the disk.

Synchronizing Volumes

When a disk is *synchronized*, all buffered data that is modified is written to the physical device so that the disk is up to date. For the rawFs file system, the only such data is that contained in open file descriptor buffers.

To avoid loss of data, synchronize a disk before removing it. You may need to explicitly synchronize a disk, depending on when (or if) the ***rawFsVolUnmount( )*** call is issued.

When ***rawFsVolUnmount( )*** is called, an attempt is made to synchronize the device before unmounting. If this disk is still present and writable at the time of the call, synchronization takes place automatically; there is no need to synchronize the disk explicitly.

However, if the ***rawFsVolUnmount( )*** call is made after a disk is removed, it is obviously too late to synchronize, and ***rawFsVolUnmount( )*** discards the buffered data. Therefore, make a separate ***ioctl( )*** call with the **FIOSYNC** function before removing the disk. (For example, this could be done in response to an operator command.) Any open file descriptor to the device can be used during the ***ioctl( )*** call. This call writes all modified file descriptor buffers for the device out to the disk.

4.4.7 *I/O Control Functions Supported by* rawFsLib

The rawFs file system supports the ***ioctl( )*** functions shown in Table 4-7. The functions listed are defined in the header file **ioLib.h**. For more information, see the manual entries for **rawFsLib** and for ***ioctl( )*** in **ioLib**.

Table 4-7 **I/O Control Functions Supported by rawFsLib**

Function	Decimal Value	Description
FIODISKCHANGE	13	Announce a media change.
FIODISKFORMAT	5	Format the disk (device driver function).
FIODISKINIT	6	Initialize a rawFs file system on a disk volume (not required).
FIOFLUSH	2	Same as **FIOSYNC**.
FIOGETNAME	18	Get the file name of the *fd*.
FIONREAD	1	Get the number of unread bytes on the device.

Table 4-7 **I/O Control Functions Supported by rawFsLib**

Function	Decimal Value	Description
FIOSEEK	7	Set the current byte offset on the device.
FIOSYNC	21	Write out all modified file descriptor buffers.
FIOUNMOUNT	39	Unmount a disk volume.
FIOWHERE	8	Return the current byte position on the device.

4.5 Tape File System: tapeFs

The tapeFs library, **tapeFsLib**, provides basic services for tape devices that do not use a standard file or directory structure on tape. The tape volume is treated much like a raw device where the entire volume is a large file. Any data organization on this large file is the responsibility of a higher-level layer.

4.5.1 Tape Organization

The tapeFs file system imposes no organization of the data on the tape volume. It maintains no directory information; there is no division of the tape area into specific files; and no file names are used. An ***open( )*** operation on the tapeFs device specifies only the device name; no additional file names are allowed.

The entire tape area is available to any file descriptor open for the device. All read and write operations to the tape use a location offset relative to the current location of the tape head. When a file is configured as a rewind device and first opened, tape operations begin at the beginning-of-medium (BOM); see *Initializing a Device for Use with the tapeFs File System*, p.231. Thereafter, all operations occur relative to where the tape head is located at that instant of time. No location information, as such, is maintained by tapeFs.

4.5.2 Using the tapeFs File System

Before tapeFs can be used, it must be configured by defining **INCLUDE_TAPEFS** in the BSP file **config.h**. Note that the tape file system must be configured with SCSI-2 enabled. See *Configuring SCSI Drivers*, p.142 for configuration details.

Once the tape file system has been configured, you must initialize it and then define a tape device. Once the device is initialized, the physical tape device is available to the tape file system and normal I/O system operations can be performed.

Initializing the tapeFs File System

The tapeFs library, **tapeFsLib**, is initialized by calling ***tapeFsInit*( )**. Each tape file system can handle multiple tape devices. However, each tape device is allowed only one file descriptor. Thus you cannot open two files on the same tape device.

The ***tapeFsInit*( )** routine also makes an entry for the tapeFs file system in the I/O system driver table (with ***iosDrvInstall*( )**). This entry specifies function pointers to carry out tapeFs file operations on devices that use the tapeFs file system. The driver number assigned to the tapeFs file system is placed in a global variable, **tapeFsDrvNum**.

When initializing a tape device, ***tapeFsInit*( )** is called automatically if ***tapeFsDevInit*( )** is called; thus, the tape file system does not require explicit initialization.

Initializing a Device for Use with the tapeFs File System

Once the tapeFs file system has been initialized, the next step is to create one or more devices that can be used with it. This is done using the sequential device creation routine, ***scsiSeqDevCreate*( )**. The driver routine returns a pointer to a sequential device descriptor structure, **SEQ_DEV**. The **SEQ_DEV** structure describes the physical aspects of the device and specifies the routines in the device driver that tapeFs can call. For more information on sequential devices, see the manual entry for ***scsiSeqDevCreate*( )**, *Configuring SCSI Drivers*, p.142, *3.9.4 Block Devices*, p.171, and Example 3-6.

Immediately after its creation, the sequential device has neither a name nor a file system associated with it. To initialize a sequential device for use with tapeFs, call ***tapeFsDevInit*( )** to assign a name and declare a file system. Its parameters are the volume name, for identifying the device; a pointer to **SEQ_DEV**, the sequential

device descriptor structure; and a pointer to an initialized tape configuration structure **TAPE_CONFIG**. This structure has the following form:

```
typedef struct  /* TAPE_CONFIG tape device config structure */
    {
    int blkSize;            /* block size; 0 => var. block size */
    BOOL rewind;            /* TRUE => a rewind device; FALSE => no rewind */
    int numFileMarks;       /* not used */
    int density;            /* not used */
    } TAPE_CONFIG;
```

In the preceding definition of **TAPE_CONFIG**, only two fields, **blkSize** and **rewind**, are currently in use. If **rewind** is TRUE, then a tape device is rewound to the beginning-of-medium (BOM) upon closing a file with *close*( ). However, if **rewind** is FALSE, then closing a file has no effect on the position of the read/write head on the tape medium.

For more information on initializing a tapeFs device, see the reference entry for *tapeFsDevInit*( ).

The **blkSize** field specifies the block size of the physical tape device. Having set the block size, each read or write operation has a transfer unit of **blkSize**. Tape devices can perform fixed or variable block transfers, a distinction also captured in the **blkSize** field.

Fixed Block and Variable Block Devices

A tape file system can be created for fixed block size transfers or variable block size transfers, depending on the capabilities of the underlying physical device. The type of data transfer (fixed block or variable block) is usually decided when the tape device is being created in the file system, that is, before the call to *tapeFsDevInit*( ). A block size of zero represents variable block size data transfers.

Once the block size has been set for a particular tape device, it is usually not modified. To modify the block size, use the *ioctl*( ) functions **FIOBLKSIZESET** and **FIOBLKSIZEGET** to set and get the block size on the physical device.

Note that for fixed block transfers, the tape file system buffers a block of data. If the block size of the physical device is changed after a file is opened, the file should first be closed and then re-opened in order for the new block size to take effect.

Example 4-4 **Tape Device Configuration**

There are many ways to configure a tape device. In this code example, a tape device is configured with a block size of 512 bytes and the option to rewind the device at the end of operations.

```
/* global variables assigned elsewhere */

SCSI_PHYS_DEV *   pScsiPhysDev;

/* local variable declarations */

TAPE_VOL_DESC *  pTapeVol;
SEQ_DEV  *       pSeqDev;
TAPE_CONFIG      pTapeConfig;

/* initialization code */

pTapeConfig.blkSize  = 512;
pTapeConfig.rewind   = TRUE;
pSeqDev    = scsiSeqDevCreate (pScsiPhysDev);
pTapeVol   = tapeFsDevInit ("/tape1", pSeqDev, pTapeConfig);
```

The *tapeFsDevInit( )* call assigns the specified name to the device and enters the device in the I/O system device table (with *iosDevAdd( )*). The return value of this routine is a pointer to a volume descriptor structure that contains volume-specific configuration and state information.

Mounting Volumes

A tape volume is *mounted* automatically during the *open( )* operation. There is no specific mount operation, that is, the mount is implicit in the *open( )* operation.

Modes of Operation

The tapeFs tape volumes can be operated in only one of two modes: read-only (**O_RDONLY**) or write-only (**O_WRONLY**). There is no read-write mode. The mode of operation is defined when the file is opened using *open( )*.

File I/O

To begin I/O to a tapeFs device, the device is first opened using *open( )*. Data on the tapeFs device is written and read using the standard I/O routines *write( )* and *read( )*. For more information, see *3.7.6 Block Devices*, p.140.

End-of-file markers can be written using *ioctl( )* with the **MTWEOF** function. For more information, see *I/O Control Functions Supported by tapeFsLib*, p.234.

Changing Tapes

The tapeFs file system should be notified when removable media are changed (for example, when tapes are swapped). The ***tapeFsVolUnmount( )*** routine controls the mechanism to unmount a tape volume.

A tape should be unmounted before it is removed. Prior to unmounting a tape volume, an open file descriptor must be closed. Closing an open file flushes any buffered data to the tape, thus synchronizing the file system with the data on the tape. To flush or synchronize data, call ***ioctl( )*** with the **FIOFLUSH** or **FIOSYNC** functions, prior to closing the file descriptor.

After closing any open file, call ***tapeFsVolUnmount( )*** before removing the tape. Once a tape has been unmounted, the next I/O operation must remount the tape using ***open( )***.

Interrupt handlers must not call ***tapeFsVolUnmount( )*** directly, because it is possible for the call to pend while the device becomes available. The interrupt handler can instead give a semaphore that prompts a task to unmount the volume.

I/O Control Functions Supported by tapeFsLib

The tapeFs file system supports the ***ioctl( )*** functions shown in Table 4-8. The functions listed are defined in the header files **ioLib.h**, **seqIo.h**, and **tapeFsLib.h.** For more information, see the reference entries for **tapeFsLib**, **ioLib**, and ***ioctl( )***.

Table 4-8 **I/O Control Functions Supported by tapeFsLib**

Function	Value	Meaning
FIOFLUSH	2	Write out all modified file descriptor buffers.
FIOSYNC	21	Same as **FIOFLUSH**.
FIOBLKSIZEGET	1001	Get the actual block size of the tape device by issuing a driver command to it. Check this value with that set in the **SEQ_DEV** data structure.
FIOBLKSIZESET	1000	Set the block size of the tape device on the device and in the **SEQ_DEV** data structure.
MTIOCTOP	1005	Perform a UNIX-like **MTIO** operation to the tape device. The type of operation and operation count is set in an **MTIO** structure passed to the ***ioctl( )*** routine. The **MTIO** operations are defined in Table 4-9.

The **MTIOCTOP** operation is compatible with the UNIX **MTIOCTOP** operation. The argument passed to ***ioctl( )*** with **MTIOCTOP** is a pointer to an **MTOP** structure that contains the following two fields:

```
typedef struct mtop
    {
    short mt_op;    /* operation */
    int  mt_count; /* number of operations */
    } MTOP;
```

The **mt_op** field contains the type of **MTIOCTOP** operation to perform. These operations are defined in Table 4-9. The **mt_count** field contains the number of times the operation defined in **mt_op** should be performed.

Table 4-9 **MTIOCTOP Operations**

Function	Value	Meaning
MTWEOF	0	Write an end-of-file record or "file mark."
MTFSF	1	Forward space over file mark.
MTBSF	2	Backward space over file mark.
MTFSR	3	Forward space over data block.
MTBSR	4	Backward space over data block.
MTREW	5	Rewind the tape device to the beginning-of-medium.
MTOFFL	6	Rewind and put the drive offline.
MTNOP	7	No operation, sets the status in the **SEQ_DEV** structure only.
MTRETEN	8	Re-tension the tape (cartridge tape only).
MTERASE	9	Erase the entire tape.
MTEOM	10	Position tape to end-of-media.
MTNBSF	11	Backward space file to beginning-of-medium.

5 Network

List of Tables

List of Figures

List of Examples

5.1 Introduction

The VxWorks network is the link that connects VxWorks systems with other VxWorks systems and many other kinds of hosts. The VxWorks network is fully compatible with the 4.3 BSD Tahoe UNIX network facilities. VxWorks is also compatible with the Network File System (NFS) designed by Sun Microsystems.

This link provides a seamless environment between development hosts and VxWorks target systems. Remote file access allows VxWorks tasks to access files on other systems across the network. Remote procedure calls allow a task on one machine to invoke procedures that actually run on another machine. If you are using the target shell, you can use **rlogin** and **telnet** to access the shell from your host development system; see *9. Target Shell* for information.

This chapter gives an overview of the components comprising the VxWorks network, and of the procedures necessary to configure the network.

NOTE: This chapter addresses VxWorks networking facilities in a general way, but it also describes some of the specific configuration information you must know if you are using a UNIX development host. We can provide this information because UNIX has a standard interface to a standard set of networking facilities. However, Windows systems do not: the availability of networking facilities on a Windows host depends on the version of Windows and on the networking software package you are using. Thus, if you are using a Windows development host, consult your Windows and networking software documentation for complete networking information.

5.2 Network Components

The hierarchy of VxWorks network components is shown in Figure 5-1. At the lowest level, VxWorks typically uses Ethernet as the basic transmission medium. VxWorks can also use serial lines for long-distance connections or shared memory on a common backplane in more closely coupled environments. On top of the transmission media, VxWorks uses the Internet protocols TCP/IP and UDP/IP to transport data between processes running under either VxWorks or the host development system.

Using Internet protocols, VxWorks makes several types of network facilities available:

- **Sockets** allow communications between tasks, running either under VxWorks or the host development system.
- **Remote Procedure Calls (RPC)** allow a task on one machine to invoke procedures that run on other machines. Both the calling task and called procedure can run under either VxWorks or the remote development system.
- **Remote File Access** allows VxWorks tasks to access host files remotely, with the Network File System (NFS), remote shell (RSH), Internet File Transfer Protocol (FTP), or Trivial File Transfer Protocol (TFTP).
- **File Export** allows remote systems that have NFS clients to use files maintained on VxWorks dosFs file systems.
- **Remote Command Execution** allows VxWorks tasks to invoke commands on a host development system over the network.

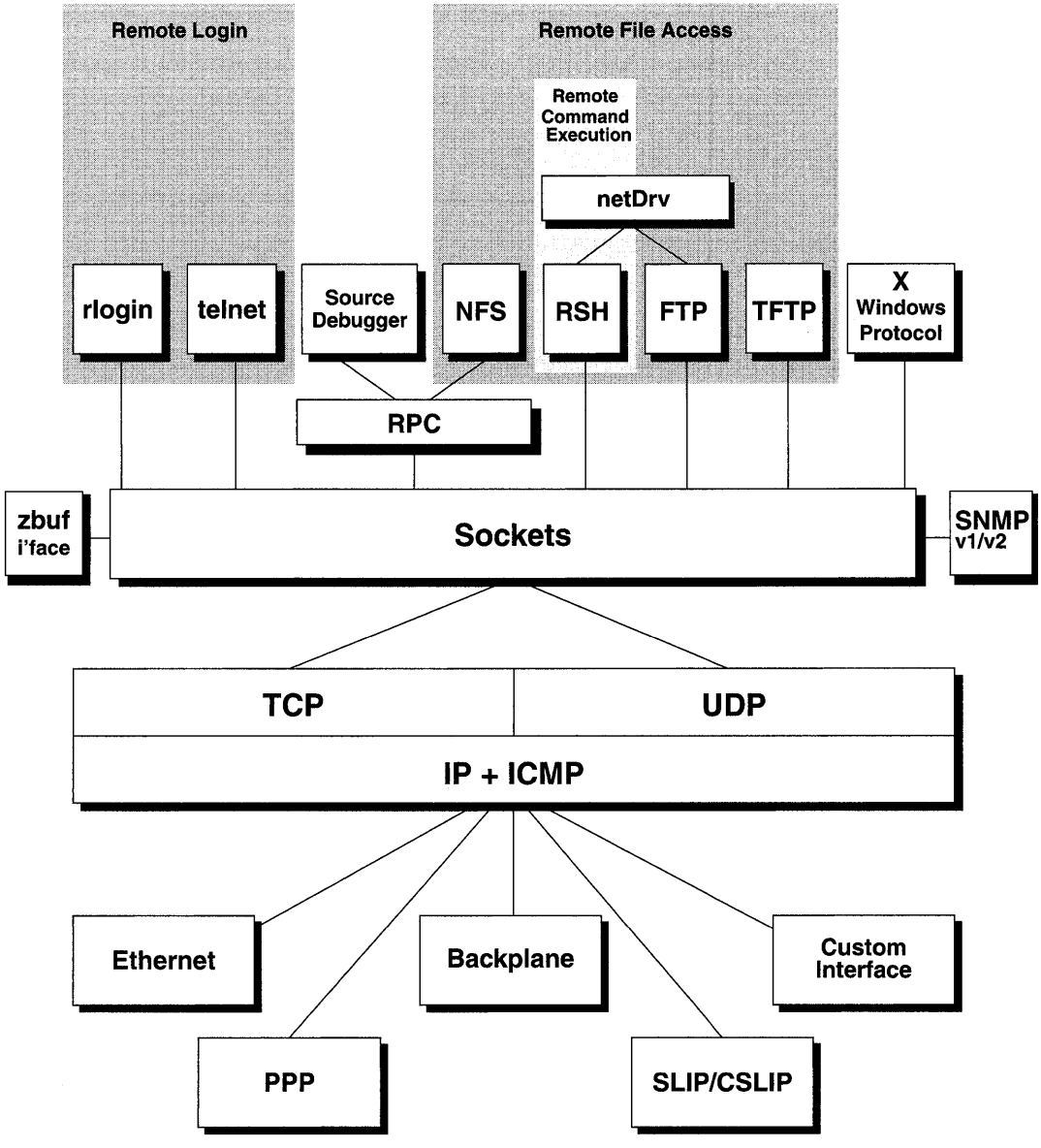
5.2.1 Ethernet

Ethernet is one medium among many over which the VxWorks network operates. Ethernet is a local area network specification that is supported by numerous vendors. It is ideal for most VxWorks applications, but there is nothing inherently tied to Ethernet in either the VxWorks or host network systems.

5.2.2 Serial Line Interface Protocol (SLIP and CSLIP)

The VxWorks network can communicate with the host operating system over serial connections using the Serial Line Interface Protocol (SLIP), or using a version of SLIP with compressed headers (CSLIP). Using SLIP or CSLIP as a network

Figure 5-1 **VxWorks Network Components**



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.

5.2.3 Point-to-Point Protocol (PPP)

The Point-to-Point Protocol (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.

5.2.4 Shared-Memory Network

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 Ethernet are also available over the shared-memory network.

5.2.5 TCP/IP Internet Protocols and Addresses

Protocols

On top of the raw Ethernet and backplane transmission mechanisms, VxWorks uses the Internet protocol suite (often referred to as *TCP/IP*) to effect communication across the network. Three main protocols are used:

- **Internet Protocol (IP)** is the base network protocol of the Internet protocol family. With IP, each *host* (computer) in the network has a unique 4-byte Internet address (described in *Internet Addresses*, p.247). IP accepts packets addressed to a particular host and tries to deliver them. If multiple networks

are connected by gateways, IP forwards a packet from gateway to gateway 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.

- **User Datagram Protocol (UDP)** 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, where a 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.
- **Transmission Control Protocol (TCP)** provides reliable, flow-controlled, two-way, process-to-process transmission of data. TCP is a *connection*-based communication mechanism. This means that before data can be exchanged over TCP, the two communicating processes must first establish a connection through a distinct connection phase. Data is then sent and received as a byte stream at both ends. Like UDP, TCP extends the connection address to include a port address in addition to the host Internet address. That is, a connection is established between a particular port in one host and a particular port in another host. TCP *guarantees* that the delivery of data is correct, in the proper order, and without duplication.

The VxWorks network also fully supports the associated Internet Control Message Protocol (ICMP) and the Ethernet Address Resolution Protocol (ARP), as implemented in UNIX BSD 4.3.

Internet Addresses

Each host in an Internet network has a unique Internet address and an associated address mask. An Internet address is 32 bits long, and begins with a Internet address class, followed by a network identifier and host identifier. The address mask is set to a default value according to class if subnets are not used. For more information, see *5.3.9 Using Subnets*, p.297.

There are three classes of Internet addresses to accommodate different network configurations:

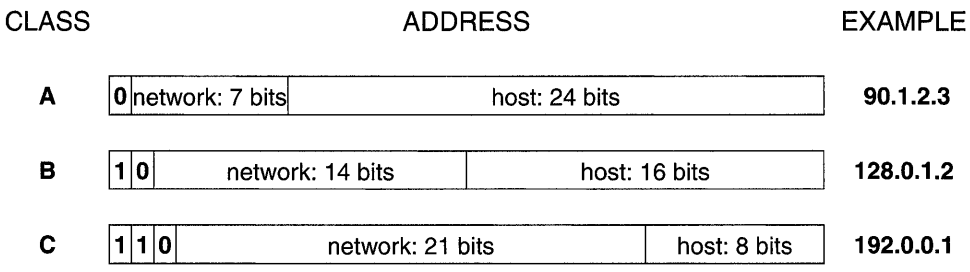
Class A addresses support a small number of networks, each with a large number of hosts.

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

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

The three classes are distinguished by the high-order bits of an Internet address as shown in Figure 5-2.

Figure 5-2 **Internet Address Classes**



By convention, Internet addresses are usually represented as a character string with a dot (.) notation. Dot notation 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 (for example, as an **int**, **long**, **u_long**, or **struct in_addr**). For example, the Internet address 0x5a010203 is 90.1.2.3 in standard dot notation. Each Internet address class has a unique address range determined by the high-order bits and the default address mask (used for masking out the bits used for the network portion of the address) as shown in Table 5-1.

VxWorks includes routines for manipulating Internet addresses. For instance, 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**.

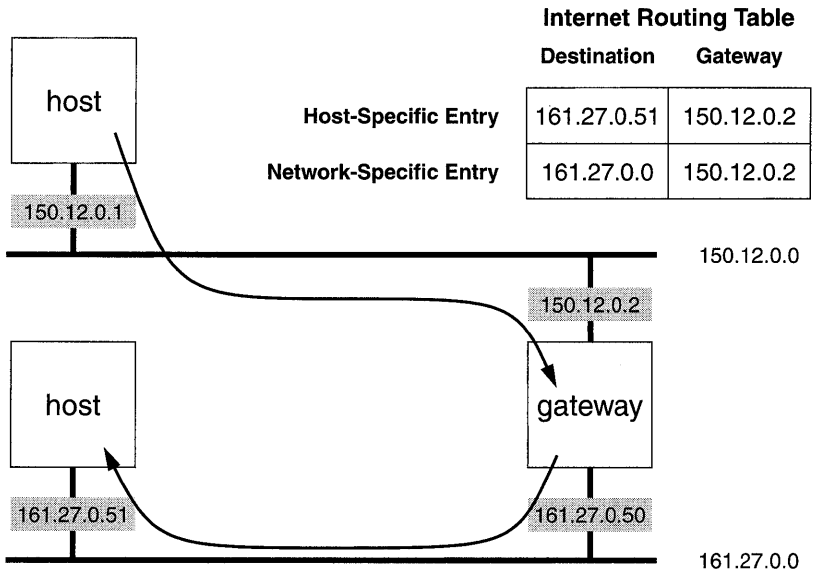
Table 5-1 **Internet Address Ranges**

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

Packet Routing

The IP layer software handles packet routing. For each device connected to the network, internal routing tables contain information about possible destination addresses. These routing tables contain two types of entries: host-specific route entries and network-specific route entries. 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. Figure 5-3 shows an example.

Figure 5-3 **Internet Routing**



The host-specific route entries have precedence over network-specific route entries. The IP layer software first compares the destination address against any host-specific route entries in the table. If there is no match, the IP layer software searches for an appropriate network-specific routing table entry. The appropriate address mask is applied to the destination address to obtain the network identifier. This network identifier is then compared against the network-specific entries in the routing table.

The VxWorks routing table is edited explicitly using ***routeAdd( )*** and ***routeDelete( )***:

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

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

Another routing function, ***routeNetAdd( )***, is equivalent to ***routeAdd( )*** except that it always treats the destination address as a network.

Network Byte Order

Different CPU architectures can be present on a single network. 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. In VxWorks, *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 5-2 convert longs and shorts between host- and network byte order. To minimize the overhead in calling them, macro implementations (which have no effect on architectures where no conversion is needed) are also available, in **h/netinet/in.h**.

Table 5-2 **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.

This example increments **pBuf** four times on little-endian architectures:

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

To avoid macro-expansion side effects, do not apply these macros directly to an expression. Instead, increment separately from the macro call. The following increments **pBuf** only once, whether the architecture is big- or little-endian:

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


5.2.6 Sockets

In VxWorks, the direct interface to the Internet protocol suite is through *sockets*. A socket is an end-point for communications that is *bound* to a UDP or TCP port within the node. There are two types of sockets:

- A process can create a *datagram socket* (which uses UDP) and bind it 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.
- Similarly, a process can create a *stream socket* (which uses TCP) and bind it 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.

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. 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.

VxWorks sockets are UNIX BSD 4.3 compatible. However, VxWorks does not support signal functionality for sockets. 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

Table 5-3 shows the basic socket routines provided by **sockLib**. Table 5-4 lists ***ioctl*( )** functions supported by sockets. Applications must put socket port numbers (the **sin_port** field in a **struct sockaddr**) in network- byte order, with ***htons*( )**; see *Network Byte Order*, p.250.

Table 5-3 **Socket Routines**

Call	Description
socket( )	Create a socket.
bind( )	Bind a name to a socket.
listen( )	Enable connections to a TCP socket.
accept( )	Accept a connection on a TCP socket.
connect( )	Initiate a connection on a socket.
connectWithTimeout( )	Attempt a connection over a socket for a fixed duration.
shutdown( )	Shut down a socket connection.
send( )	Send data to a TCP socket.
sendto( )	Send a message to a UDP socket.
sendmsg( )	Send a message to a UDP socket.
recv( )	Receive data from a TCP socket.
recvfrom( )	Receive a message from a UDP socket.
recvmsg( )	Receive a message from a UDP socket.
setsockopt( )	Set socket options.
getsockopt( )	Get socket options.
getsockname( )	Get the name of a socket.
getpeername( )	Get the name of a connected peer socket.
select( )	Perform synchronous I/O multiplexing on a socket.
read( )	Read from a socket.
write( )	Write to a socket.
ioctl( )	Perform control functions on a socket.
close( )	Close a socket.

Table 5-4 **I/O Control Functions Supported by Sockets**

Function	Description
FIONBIO	Turn non-blocking I/O on/off.
FIONREAD	Report the number of bytes available to read on a socket.
SIOCATMARK	Report whether there is out-of-band data to be read on a socket.

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 acts as though a circuit existed 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 5-5 shows the steps in establishing socket communications with TCP, and the analogy of each step with telephone communications.

Table 5-5 **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 5-1 **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 4.3-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "tcpExample.h"
```

```
/*****************************************************************************
*
* tcpClient - send requests to server over a TCP socket
*
* This routine connects over a TCP socket to a server, and sends a
* user-provided message to the server. Optionally, this routine
* waits for the server's reply message.
*
* This routine may be invoked as follows:
*       -> tcpClient "remoteSystem"
*       Message to send:
*       Hello out there
*       Would you like a reply (Y or N):
*       y
*       value = 0 = 0x0
*       -> MESSAGE FROM SERVER:
*       Server received your message
*
* RETURNS: OK, or ERROR if the message could not be sent to the server.
*/

STATUS tcpClient
    (
    char *              serverName      /* name or IP address of server */
    )
    {
    struct request      myRequest;      /* request to send to server */
    struct sockaddr_in  serverAddr;     /* server's socket address */
    char                replyBuf[REPLY_MSG_SIZE]; /* buffer for reply */
    char                reply;          /* if TRUE, expect reply back */
    int                 sockAddrSize;   /* size of socket address structure */
    int                 sFd;            /* socket file descriptor */
    int                 mlen;           /* length of message */

    /* create client's socket */

    if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
        {
        perror ("socket");
        return (ERROR);
        }

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

    /* build server socket address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_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 4.3-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "taskLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "tcpExample.h"

/* function declarations */

VOID tcpServerWorkTask (int sFd, char * address, u_short port);

/*******************************************************************************
*
* tcpServer - accept and process requests over a TCP socket
*
* This routine creates a TCP socket, and accepts connections over the socket
* from clients. Each client connection is handled by spawning a separate
* task to handle client requests.
*
* This routine may be invoked as follows:
*       -> sp tcpServer
*       task spawned: id = 0x3a6f1c, name = t1
*       value = 3829532 = 0x3a6f1c
*       -> MESSAGE FROM CLIENT (Internet Address 150.12.0.10, port 1027):
*       Hello out there
*
* RETURNS: Never, or ERROR if a resources could not be allocated.
*/

STATUS tcpServer (void)
    {
    struct sockaddr_in  serverAddr;     /* server's socket address */
    struct sockaddr_in  clientAddr;     /* client's socket address */
    int                 sockAddrSize;   /* size of socket address structure */
    int                 sFd;            /* socket file descriptor */
    int                 newFd;          /* socket descriptor from accept */
    int                 ix = 0;         /* counter for work task names */
    char                workName[16];   /* name of work task */
```

```
    /* set up the local address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_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, 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 */
    }
```

Datagram Sockets (UDP)

The User Datagram Protocol (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*.

As TCP is analogous to telephone communications, UDP is analogous to sending mail. Each UDP packet is like a letter. Each packet carries the address of both the destination and the sender. Like the mail, UDP is unreliable: packets that are lost or out-of-sequence are not reported.

Example 5-2 **Datagram Sockets (UDP)**

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.

```
/* udpExample.h - header used by both UDP server and client examples */

#define SERVER_PORT_NUM         5002    /* server's port number for bind() */
#define REQUEST_MSG_SIZE        1024    /* max size of request message */

/* structure used for client's request */

struct request
    {
    int  display;                       /* TRUE = display message */
    char message[REQUEST_MSG_SIZE];     /* message buffer */
    };
```

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

/*
DESCRIPTION
This file contains the client-side of the VxWorks UDP example code.
The example code demonstrates the useage of several BSD 4.3-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.sin_family = AF_INET;
    serverAddr.sin_port = htons (SERVER_PORT_NUM);

    if (((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
        ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR))
        {
        perror ("unknown server name");
        close (sFd);
        return (ERROR);
        }

    /* build request, prompting user for message */

    printf ("Message to send: \n");
    mlen = read (STD_IN, myRequest.message, REQUEST_MSG_SIZE);
    myRequest.message[mlen - 1] = '\0';

    printf ("Would you like the server to display your message (Y or N): \n");
    read (STD_IN, &display, 1);
    switch (display)
```

```
        {
        case 'y':
        case 'Y': myRequest.display = TRUE;
            break;
        default: myRequest.display = FALSE;
            break;
        }

    /* send request to server */

    if (sendto (sFd, (caddr_t) &myRequest, sizeof (myRequest), 0,
        (struct sockaddr *) &serverAddr, sockAddrSize) == ERROR)
        {
        perror ("sendto");
        close (sFd);
        return (ERROR);
        }

    close (sFd);
    return (OK);
    }
```

```
/* udpServer.c - UDP server example */

/*
DESCRIPTION
This file contains the server-side of the VxWorks UDP example code.
The example code demonstrates the useage of several BSD 4.3-style
socket routine calls.
*/

/* includes */

#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "stdioLib.h"
#include "strLib.h"
#include "ioLib.h"
#include "fioLib.h"
#include "udpExample.h"

/*******************************************************************
*
* udpServer - read from UDP socket and display client's message if requested
*
* Example of VxWorks UDP server:
*       -> sp udpServer
*       task spawned: id = 0x3a1f6c, name = t2
*       value = 3809132 = 0x3a1f6c
*       -> MESSAGE FROM CLIENT (Internet Address 150.12.0.11, port 1028):
*       Greetings from UDP client
*
* RETURNS: Never, or ERROR if a resources could not be allocated.
*/
```

```
STATUS udpServer (void)
    {
    struct sockaddr_in  serverAddr;     /* server's socket address */
    struct sockaddr_in  clientAddr;     /* client's socket address */
    struct request      clientRequest;  /* request/Message from client */
    int                 sockAddrSize;   /* size of socket address structure */
    int                 sFd;            /* socket file descriptor */
    char                inetAddr[INET_ADDR_LEN];
                                        /* buffer for client's inet addr */

    /* set up the local address */

    sockAddrSize = sizeof (struct sockaddr_in);
    bzero ((char *) &serverAddr, sockAddrSize);
    serverAddr.sin_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);
            }
        }
    }
```

5.2.7 The Zbuf Socket Interface

VxWorks includes an alternative set of socket calls based on a data abstraction called a *zbuf*, which permits sharing data buffers (or portions of data buffers) between separate software modules. The *zbuf socket interface* allows applications to read and write UNIX BSD 4.3 sockets without copying data between application 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 routines to use the zbuf interface. Applications that use the zbuf interface are thus less portable.

WARNING: The send socket buffer size must exceed that of any zbufs sent over the socket.

To link in (and initialize) the zbuf socket interface, define **INCLUDE_ZBUF_SOCK** in **configAll.h**.

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: The speedups based on using zbufs depend on allowing different modules to share the same buffers. To work, your application must neither modify nor free the data buffer while network software is still using it. Instead of freeing a buffer explicitly, 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. *Manipulating the Zbuf Data Structure*, p.265 describes the routines to create and manage zbufs, and *Zbuf Socket Calls*, p.273 introduces the remaining zbuf-specific socket routines. See also the reference entries for **zbufLib** and **zbufSockLib**.

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 5-4 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 5-4 **Zbuf Addressing Relative to First Segment (NULL)**

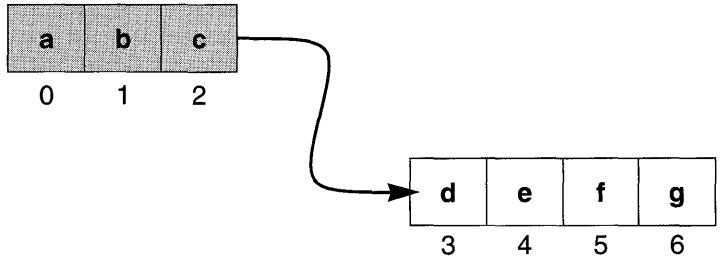
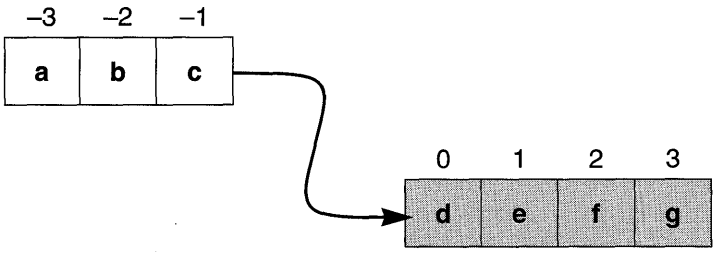


Figure 5-5 shows the same zbuf, but labelled with offsets relative to the second segment. This is the most efficient addressing scheme to refer to bytes d, e, f, or g in the figure.

Figure 5-5 **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

Table 5-6 **Zbuf Creation and Deletion Routines**

Call	Description
zbufCreate( )	Create an empty zbuf.
zbufDelete( )	Delete a zbuf and free any associated segments.

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.

Getting Data In and Out of Zbufs

Table 5-7 **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.

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).

Operations on Zbufs

The routines listed in Table 5-8 perform several fundamental operations on zbufs.

Table 5-8 **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 5-9 give your applications access to the underlying segments in a zbuf.

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

Table 5-9 **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.

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 5-3.

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 5-3 **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.

- 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 propagating references to it.

Zbuf Socket Calls

The zbuf socket calls listed in Table 5-10 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 5-10 **Zbuf Socket Library Routines**

Call	Description
zbufSockLibInit( )	Initialize socket libraries (called automatically with **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 recoding 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 5-4 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 5-1 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 5-3 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 5-1.
2. Include the header file **zbufSockLib.h**.
3. Insert the *zbufDisplay( )* routine from Example 5-3.
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);
}
```

NOTE: 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.

5.2.8 Remote Procedure Calls

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 utilities help generate the client interface routines and the server skeleton.

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 **unsupported/rpc4.0/doc** and **unsupported/rpc4.0/man**), and the reference entry for **rpcLib**.

5.2.9 Remote File Access

Files on a remote machine can be accessed from a VxWorks target transparently with a remote file transfer protocol.

Transparent remote file access allows files on remote systems to be accessed as if they were local. 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 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 with any of three different protocols:

- **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 by 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**.

An alternative remote file transfer protocol that is not transparent is the **Trivial File Transfer Protocol (TFTP)**. The VxWorks implementation provides both client and server functions, and is typically used only for booting. See the reference entries for **tftpLib** and **tftpdLib**.

5.2.10 Remote Command Execution

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**.

5.2.11 Simple Network Management Protocol (WindNet SNMPv1/v2c Option)

WindNet SNMPv1/v2c 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.

With this component, a VxWorks target can become an SNMP agent, allowing the target to be managed and configured remotely with SNMP. WindNet SNMPv1/v2c supports the Management Information Base-II (MIB-II) definitions.

WindNet SNMPv1/v2c is extensible. In addition to the base functionality, you can make extensions to the SNMP agent's MIB to include information that is specific to your application and environment.

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

5.3 Configuring the Network

Before the VxWorks network can be used, both the VxWorks and host development systems must be configured properly. There are two main concerns in configuring the network: establishing system names and addresses and establishing appropriate access permissions for each system.

On Windows, your networking software must be compatible with the Microsoft Windows Sockets (Winsock 1.1) specification. Consult your Windows and networking software documentation for specific information on configuring your host system.

On UNIX, most of the configuration procedures consist of setting up various network "database" files and the system startup files. In VxWorks, most of the configuration information necessary for access to a single host is contained in the boot line. Further initialization can either be added to **src/config/usrNetwork.c**, handled by application code, or done interactively from the Tornado shell.

The network configuration procedures for VxWorks and a UNIX host are discussed in this section and summarized in Table 5-11.

5.3.1 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. Because a system can be connected to several networks (or can even have several connections to the same network), it can 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 by *ifAddrSet( )*. (See the reference entry for **ifLib**.) To associate the Internet address 150.12.0.2 with the interface ln0, enter:

```
ifAddrSet ("ln0", "150.12.0.2");
```

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.

5.3.2 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, **/etc/hosts** contains the map between system names and Internet addresses. Each line consists of an Internet address and the computer name(s) at that address:

```
150.12.0.2    vx1
```

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

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

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

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.[1] In the following example, 150.12.0.1 can be accessed with the names **host**, **myHost**, and **widget**.

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

1. This routine is not built in to the Tornado shell. To use it from the Tornado shell, you must define **INCLUDE_NET_SHOW** in your VxWorks configuration; see *8. Configuration*.

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

```
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
```

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

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

5.3.3 Transparent Remote File Access

As mentioned previously, VxWorks can use any of three different underlying protocols to provide transparent remote file access: remote shell (RSH), the Internet File Transfer Protocol (FTP), or the Network File System (NFS).[2]

The VxWorks I/O driver **netDrv** implements remote file access using either of the first two 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 gains considerable efficiency, both in terms of memory utilization and throughput. However, initial setup is somewhat more cumbersome than the other protocols.

The following sections describe the implementation and configuration of these protocols.

2. 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.

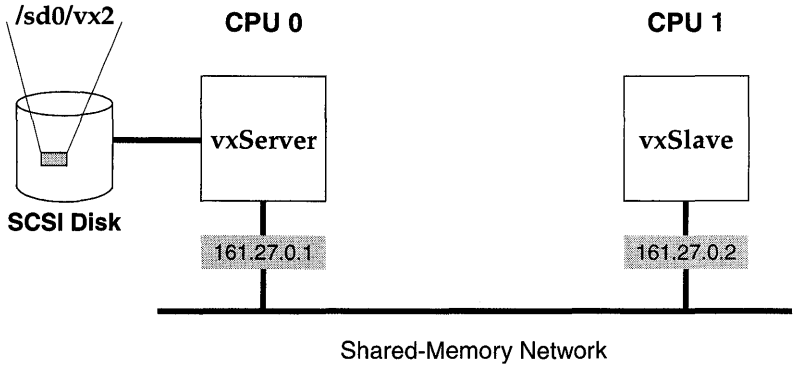
Transparent Remote File Access with RSH and FTP

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 (see Figure 5-6). 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.

Figure 5-6 **FTP Boot Example**



In the following example, a slave on the shared-memory network boots from the master CPU's local SCSI disk. (For more information on shared-memory networks, see *5.4 Shared-Memory Networks*, p.301.) Note that although VxWorks requires that

the **ftp password** field not be blank, the password itself is ignored. 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)                             : jane
ftp password (pw) (blank=use rsh)    : ignored
```

The FTP server daemon is initialized on the VxWorks server by default when **INCLUDE_FTP_SERVER** is defined in **configAll.h**. See the reference entry for **ftpdLib**.

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.

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/fred/myfile** refers to the file **/usr/fred/myfile** on the **mars** system. This file can be read and/or written exactly like a local file. The following Tornado shell command opens that file for I/O access:

```
-> fd = open ("mars:/usr/fred/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.

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 *fred*, and if FTP is used, the password is *flintstone*:

```
-> iam "fred", "flintstone"
```

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

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.

Transparent Remote File Access with 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 routines to manage exporting file systems.

nfsdLib NFS server library. Provides routines to manage requests from remote NFS clients.

The VxWorks NFS library routines are implemented using RPC. See the library reference entries and *5.2.8 Remote Procedure Calls*, p.278.

Allowing Remote File Access with NFS

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.

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.

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/fred/myfile** refers to the file **/usr/fred/myfile** on the host **mars** as if it were local to the VxWorks system.

If **INCLUDE_NFS_MOUNT_ALL** is defined in the VxWorks configuration file **configAll.h**, VxWorks mounts all exported NFS file systems. 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 **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 that the user belongs to.

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 in the header file **configAll.h** by defining the values of **NFS_USER_ID** (default user ID is 2001) and **NFS_GROUP_ID** (default group ID is 100) respectively. The NFS authentication parameters are set to these values at system startup. If NFS file access is unsuccessful, make sure that **NFS_USER_ID** and **NFS_GROUP_ID** are correct.

Allowing Remote Access to VxWorks Files through NFS

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 in your VxWorks configuration, define the constant **INCLUDE_NFS_SERVER** in **configAll.h**. If you wish, you can run a VxWorks system with only NFS server support (and no client support) by including **INCLUDE_NFS_SERVER** but not **INCLUDE_NFS** in **configAll.h**.

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 *4.2.4 Volume Configuration*, p.199 in this manual). 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; to use a RAM disk, see *RAM Disk Drivers*, p.140. Your BSP can also support other suitable device drivers; see your BSP 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 *4.2.6 Using an Already Initialized Disk*, p.204 in this manual), initialize it as follows:

```
dosFsDevInitOptionsSet (DOS_OPT_EXPORT);
dosFsDevInit ("/export", pBlkDev, NULL);
```

Otherwise, specify a pointer to a **DOS_VOL_CONFIG** structure as the third argument to *dosFsDevInit( )* (see the **dosFsLib** reference entry).

⚠ **NOTE:** 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.

⚠ **NOTE:** 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 *4.2 MS-DOS-Compatible File System: dosFs*, p.191 in this manual.

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 routines and arrange to call them when needed. See the reference entries for ***nfsdInit*( )** and ***mountdInit*( )** for information on authorization hooks.

5.3.4 Remote File Transfer Using 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. Typically the TFTP client side is used at boot time to download the VxWorks from the boot host to the target. The VxWorks TFTP server can be used to boot an X-Terminal from VxWorks or 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.

Host TFTP Server

The TFTP server is typically started by the Internet daemon on the host. For added security, some hosts (for example, Sun hosts) start the TFTP server with the secure (**-s**) option enabled by default. If this option is specified, the server roots all TFTP requests in the directory specified (for example, **/tftpboot**) to restrict access to the host.

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.

VxWorks TFTP Server

The TFTP server daemon is initialized by default when **INCLUDE_TFTP_SERVER** is defined in **configAll.h**. See the reference entry for **tftpdLib**.

VxWorks TFTP Client

Include the VxWorks TFTP client side by defining **INCLUDE_TFTP_CLIENT** in **configAll.h**. 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**.

5.3.5 Remote Login from VxWorks to the Host: *rlogin*( )

You can log in to a host system from a VxWorks terminal using *rlogin*( ).

For a Windows host system, VxWorks's ability to remotely log in depends on your version of Windows and the networking software you are using. See that documentation for details.

For 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 *Allowing Remote File Access with RSH*, p.284.

5.3.6 Adding Gateways to a Network

The Internet protocols allow hosts on different but connected networks to communicate. If a machine on one network sends a packet to a machine on another network, then a *gateway* is sought that can forward the message from the sender's network to the destination network. If a system has interfaces to more than one network, it can be a gateway between those networks. One of the primary functions of IP (the lower-level protocol of TCP/IP) is to perform this routing and forwarding among interconnected networks.

Many systems have a routing daemon (**routed**) that exchanges routing information with other systems to determine network connectivity. VxWorks, however, has no routing daemon, and must instead be told explicitly about any gateways it requires. Similarly, if a VxWorks system is a gateway, other systems must be told about it explicitly, because VxWorks does not broadcast routing information.

Adding a Route on Windows

Again, this procedure varies with your version of Windows and your networking software package. See the documentation for your system for details.

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 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.

You can also 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 did 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

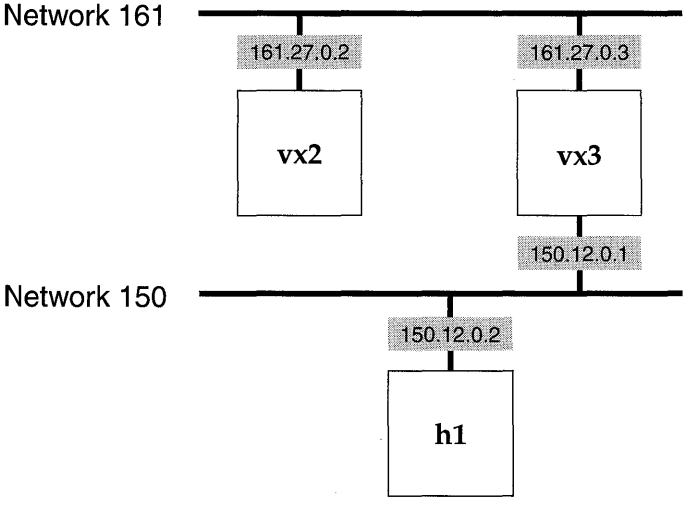
To add gateways to the VxWorks network routing tables, use ***routeAdd*( )**:

```
routeAdd ("destinationAddr", "gatewayAddr")
```

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

For example, consider two VxWorks machines **vx2** and **vx3** (shown in Figure 5-7), 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.

Figure 5-7 **Routing Example**



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");
```

You can confirm that a route is in the routing table with the *routeShow( )* routine.[3] Other routing functions are available in the library **routeLib**.

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.

5.3.7 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 whether another machine is responding to network requests.

The following example shows *ping( )* output for an address that cannot be reached:

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

If the first argument does not have the form of a numeric Internet address, *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
```

3. This routine is not built into the Tornado shell. To use it from the Tornado shell, define **INCLUDE_NET_SHOW** in your VxWorks configuration; see *8. Configuration* in this manual.

```
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 may 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)
    {
    meDownShut();              /* 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.

5.3.8 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 **src/demo/dg/dgTest.c**.

Unfortunately, there is some ambiguity about what 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. However, some older systems use an Internet address with a host part of all zeros.

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.

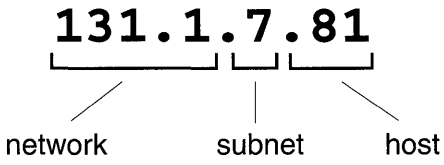
VxWorks normally uses a host part of all ones as the broadcast address. Thus a datagram sent to Internet address 150.255.255.255 (0x5affffff) is broadcast to all systems on network 150. However, to allow compatibility with other systems, VxWorks allows the broadcast address to be reassigned for each network interface by calling the routine ***ifBroadcastSet( )***. For more information, see the reference entry for ***ifBroadcastSet( )***.

5.3.9 Using Subnets

An Internet address consists of a network address portion and a host address portion. As described previously, 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 flexibility in network addressing. Even so, in some environments network addresses are a scarce resource—an organization can be limited to a certain number of network addresses by a higher authority.

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 5-8. Internal to the subnet, the Internet address is interpreted as 131.1.7 for the network portion and 81 for the host portion.

Figure 5-8 **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 5-8, the netmask is 0xffffff00. In VxWorks, use ***ifMaskSet( )*** to specify the net mask for a particular network interface. For more information, see the reference entry for ***ifMaskSet( )***.

Specify a net mask during booting if 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
```

5.3.10 Configuration of Mbufs

You can control the number of buffers (*mbufs*) that can be assigned to the Internet software by modifying the structures **mbufConfig** and **clusterConfig** in **usrNetwork.c** (in **src/config**). These structures allow you to specify the size and location of a memory pool from which the network buffers are allocated. The following structure is used to configure mbufs or mbuf clusters:

```
typedef struct
    {
    int initialAlloc;
    int incrementAlloc;
    int maxAlloc;
    int memPartition;
    int memPartitionSize;
    } MBUF_CONFIG;
```

The fields in this structure are:

initialAlloc — the number of mbufs or clusters to allocate at boot time.

incrementAlloc — the number of mbufs or clusters that are allocated, each time an "out of buffers" condition exits. After these buffers are allocated, they remain permanently in the mbufs pool.

maxAlloc — the maximum number of mbufs or clusters that can be allocated.

memPartition — the default is the system memory pool, which is indicated by passing the value **NULL**. If an address is specified, the system attempts to create a memory partition using this address. If the partition cannot be created, the root task suspends itself and prints an error message on the console.

memPartitionSize
ignored when **memPartition** is NULL. If **memPartition** is not NULL, **memPartitionSize** specifies the size of the memory pool to be used for mbufs or clusters. It must be large enough to allocate the number of buffers specified in **initialAlloc**. If this field is not large enough to accommodate the number of buffers specified in **initialAlloc**, the root task suspends itself and prints an error message on the console.

Changes to the configuration of mbufs or clusters reflects the network traffic requirements of your system. If your network needs are small and your application performs a lot of memory allocation, you can decrease the default values to recover the additional memory. If your network needs are larger, the default values can be increased to help avoid lost packets or increase network performance.

NOTE: Perform the configuration of mbufs and mbuf clusters only after some data about the behavior of the system is collected and the desired behavior determined. The defines for the default configuration are in the file **h/net/mbuf.h**.

Table 5-11 **Network Procedures Summary**

Function	On UNIX	On VxWorks

Associate Internet addresses with network interfaces.	Use **ifconfig** in **/etc/rc.local**:
	`ifconfig ln0 150.12.0.1`
	or:
	`ifconfig ln0 host`
		Call *ifAddrSet*( ):
		`-> ifAddrSet "ln0", "150.12.0.2"`

Associate Internet addresses with system names.	Add address-name pairs to **/etc/hosts**:
	`150.12.0.1 host`
	or:
	`150.12.0.2 vw1 sonny`
		Call *hostAdd*( ):
		`-> hostAdd "host", "150.12.0.1"`
		`-> hostAdd "vw1", "150.12.0.2"`
		`-> hostAdd "sonny", "150.12.0.2"`

Examine host names.	Look at **/etc/hosts**.	Call *hostShow*( ) if **INCLUDE_NET_SHOW** is defined in your VxWorks configuration.

Transparent remote file access with RSH.	Add remote system names to **/etc/hosts.equiv** or */userhome/***.rhosts**:
	`vw1`
	`vw2`
		Create network devices to remote systems using RSH:
		`-> netDevCreate "host:", "host", 0`
		Set user name using *iam*( ):
		`-> iam "fred", 0`
		Access files with created device name:
		`-> copy < /host:/usr/myfile`

Table 5-11 **Network Procedures Summary** *(Continued)*

Function	On UNIX	On VxWorks

Transparent remote file access with FTP.	No action necessary.	Create network devices to remote systems using FTP:
		`-> netDevCreate "host:", "host", 1`
		Set user name and password using *iam*( ):
		`-> iam "fred", "flintstone"`
		Access files with created device name:
		`-> copy < /host:/usr/fred/myfile`

Transparent remote file access with NFS.	Add the names of mountable file systems and a list of groups that have access to them in **/etc/exports**.	Create NFS device:
		`-> nfsMount "host", "/usr", "/hostUsr"`
		Set NFS authentication with *nfsAuthUnixSet*( ) or *nfsAuthUnixPrompt*( ):
		`-> nfsAuthUnixSet "host", uid, gid, 0`
		Access files with mounted name:
		`-> copy < /hostUsr/fred/myfile`

Exporting dosFs file system with NFS.	Use **mount** (as root):
	`/etc/mount vw:/export /mnt`
		Initialize the file system using **DOS_OPT_EXPORT** (0x8):
		`-> dosFsDevInitOptionsSet (0x8)`
		`-> dosFsDevInit "/export", pBlkDev, 0`
		Register with NFS server for export:
		`-> nfsExport "/export", 0, 0, 0`

Remote login from VxWorks to host with *rlogin*( ).	Add remote system names to **/etc/hosts.equiv** or */userhome/***.rhosts**:
	`vw1`
		Set the user name with *iam*( ):
		`-> iam ("fred")`
		Use *rlogin*( ):
		`-> rlogin "host"`

Add gateways to a network.	Add gateway to **/etc/gateways** and restart **routed**:
	`net 161.27.0.0 gateway \`
	`150.12.0.3 metric 1 passive`
	or use **route**:
	`route add 161.27.0.0 150.12.0.3`
		Call *routeAdd*( ):
		`-> routeAdd "150.12.0.0", "vx3"`
		`-> routeAdd "150.12.0.0", "161.27.0.3"`

Examine routing tables.	Use **netstat**: `% netstat -r`	Call *routeShow*( ) if **INCLUDE_NET_SHOW** is defined in your VxWorks configuration.
Examine network interfaces.	Use **ifconfig**: `% ifconfig ln0`	Call *ifShow*( ) if **INCLUDE_NET_SHOW** is defined in your VxWorks configuration: `-> ifShow ("ln0")`

5.4 Shared-Memory Networks

The VxWorks network subsystem has many layers of protocols. At the bottom layer are the network interface drivers; their job is to transmit and receive packets on the physical network medium. In addition to supplying drivers for traditional network media such as Ethernet, VxWorks also supplies a *shared-memory network driver*, **sm**, which provides communication over a backplane bus.

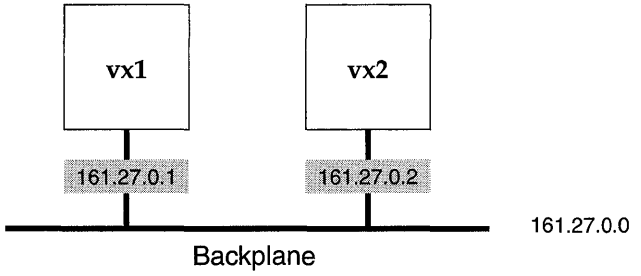
The advantage of a shared-memory network driver compatible with the rest of the network subsystem is that all higher-level protocols are immediately available over the backplane, as they are over Ethernet. Facilities like socket communications, remote login, remote file access, and NFS are all available to and from any processor on the backplane, simultaneously. Use of the network facilities over the backplane is indistinguishable from their use over any other medium.

A multiprocessor backplane bus becomes an Internet network of its own. Each shared-memory network has its own network/subnet number. As usual, each processor on the shared-memory network has a unique Internet address.

⚠ **NOTE:** This is different if you are using proxy ARP; see *5.5 Proxy ARP*, p.316 for additional information.

In the example shown in Figure 5-9, two CPUs are on the 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 5-9 **Shared-Memory Network**



The routing capabilities of the VxWorks Internet protocols allow the processors on the 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, typically an Ethernet network. This makes all levels of network communications available

between any processor on the shared-memory network and any other host or target system on the external network.

Finally, the low-level packet passing mechanism of the shared-memory network driver is also available directly. This allows alternative protocols to be run over the shared-memory network in parallel with the standard ones.

The VxWorks shared-memory network driver uses the following techniques 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 either poll the shared-memory data structures for input packets periodically, or be notified of input packets by interrupts.

The shared-memory network is configured by constants in the header file **config.h** and by parameters specified to the VxWorks boot ROMs. The following sections give the details of the backplane network operation and configuration.

5.4.1 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 on 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 interactively.

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 itself 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 **src/config/usrNetwork.c** is set up to establish processor 0 as the master. 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

In various configurations, the shared-memory pool is located at different locations. In many situations, it is desirable to allocate the shared memory at run-time, rather than fixing its location at the time the system is built.

All processors on the shared-memory network must be able to locate the shared-memory pool, even when its location is not known at the time the system is built. The shared-memory anchor serves as a common point of reference for all processors. The anchor is a small data structure placed at a fixed location when the system is built. This is usually either in low memory of the dual-ported memory of one of the processors, or at some fixed address on the separate memory board.

The anchor contains a pointer to the actual shared-memory pool. This is set up by the master when the shared-memory network is initialized. The anchor's

"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 this offset is the same for all processors.

The backplane anchor address is established in one of two ways: either by parameters in **config.h**, or by boot parameters. For the shared-memory network master, the anchor address is established in the master's configuration header file **config.h** at the time the system image is built. Set the value of **SM_ANCHOR_ADRS**, in **config.h** of the master, to the address of the anchor *as seen by the master.*

For the other processors on the shared-memory network, a default anchor address can be established in the same way, by the setting of **SM_ANCHOR_ADRS** in **config.h**. However, this requires burning boot ROMs with that configuration, because the other processors must boot from the shared-memory network to begin with. For this reason, the anchor address can also be specified in the boot parameters if the shared-memory network is the boot device. This is done by appending the address to the shared-memory network boot device code **sm**, separated by an equal sign (=). Thus the following boot parameter establishes the anchor address at 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

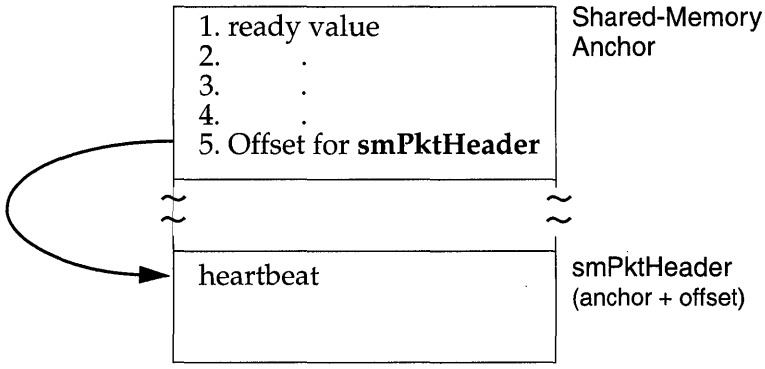
The processors on the shared-memory network cannot communicate over that network until the shared-memory pool initialization is finished. To let the other processors know when the backplane is "alive," the master maintains a *shared-memory heartbeat*. This heartbeat is a counter that is incremented by 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 packet header. The offset of the shared-memory packet header is the fifth 4-byte word in the anchor, as shown in Figure 5-10.

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 *................*
```

Figure 5-10 **Shared-Memory Heartbeat**



The offset to the shared-memory packet header is 0x170. To view the shared-memory packet header, 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. Display this location again to ensure that the heartbeat is alive; if its value has changed, the network is alive.

Shared Memory Location

As mentioned previously, the shared memory can either be put at a fixed location at the time the system is built, or be allocated dynamically at run-time. The location is determined by the value of **SM_MEM_ADRS** in **config.h**. 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 **SM_ANCHOR_ADRS** means 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 determined by the value of **SM_MEM_SIZE** in the header file **config.h**.

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 that can be outstanding 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 **config.h** files delivered with VxWorks contain a conditional compilation that makes it easy to select a pair of typical configurations. The constant **SM_OFF_BOARD** can be defined TRUE to select a typical *off-board* shared-memory pool, or FALSE to select a typical *on-board* shared-memory pool.

A typical *off-board* configuration establishes the backplane anchor and pool to be located at an absolute address of 0x800000 on a separate memory board with a 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 is configured to be ***malloc***'ed from the master's own memory at run time. The size of the pool allocated is set to 64KB.

These configurations are provided as examples; change them to suit your configuration.

Additional configuring may be required to make the shared memory non-cacheable, because the shared-memory pool is accessed by all processors on the backplane. By default, boards with an MMU have the MMU turned on. With the MMU on, memory that is off-board must be made non-cacheable. This is done using the **sysPhysMemDesc[ ]** table in **sysLib.c**. The VME address space used for the shared-memory pool must have a virtual-to-physical mapping in this data structure, as well as mark the memory as non-cacheable (done by default). 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 *7.3 Virtual Memory Configuration*, p.408 in this manual for additional information.

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 support 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 allow this. 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. These techniques can be implemented in the system-dependent library **sysLib.c** for the processor, in the routine ***sysBusTas( )***. The shared-memory network driver calls this routine to effect the mutual exclusion on shared-memory data structures.

NOTE: Define the constant **SM_TAS_TYPE** in **configAll.h** 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**).

5.4.2 Interprocessor Interrupts

Each processor on the backplane has a single *input queue* of packets sent to it 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 periodically. When using interrupts, the processor receives an interrupt from the sending processor when its input queue has packets. Of course, interrupt-driven communication is much more efficient than polling.

Most backplane buses have a limited number of bus-interrupt lines available on the backplane (for example, VMEbus has seven). A processor can use one of these interrupt lines as its input interrupt. However, each processor must have its own interrupt line. Furthermore, not all processor boards are capable of generating bus interrupts. Thus, bus interrupts are difficult to use.

A much better interrupt mechanism is *mailbox interrupts*, also called *location monitors* because they monitor the access to specific memory locations. A mailbox interrupt is 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 processor takes up only a single address on the bus. Most modern processor boards include some kind of mailbox interrupt.

Each processor must tell the other processors what method to use to notify it when its input queue has packets. In the shared-memory data structures, each processor enters its *interrupt type* and up to three parameters about that type. This information is used by the shared-memory network driver of the other processors when sending packets to that processor.

The interrupt type and parameters for each processor are specified in **config.h** by the constants **SM_INT_TYPE** and **SM_INT_ARG***n*. The possible values of **SM_INT_TYPE** and the corresponding parameters are defined in the header file **smNetLib.h**. Table 5-12 summarizes interrupt types and parameters.

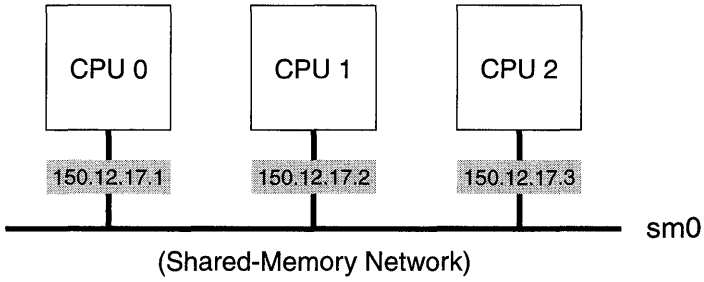
Table 5-12 **Backplane Interrupt Types**

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

5.4.3 Sequential Addressing

Sequential addressing is a method of addressing a target on the network with respect to its location on the backplane. Targets are addressed in sequential ascending order; the master has the lowest address, as shown in Figure 5-11.

Figure 5-11 **Sequential Addressing**



With sequential addressing, a target on the shared-memory network can self-configure its IP address. Only the master must know an IP address (the starting address). All other targets on the network determine their IP address by adding the starting IP address to the local target's processor number.

Sequential addressing provides a more tightly coupled environment for the shared-memory network. Because a target can determine its own Internet address as well as 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, allocate a sequential block of valid IP addresses to a shared-memory network. The master for this network is assigned the lowest address in this block. When the shared-memory network driver is initialized by the master (with ***smNetInit( )***), the starting IP address is passed in as a parameter and is stored in the shared-memory packet header.

Each target sets its interface address with ***ifAddrSet( )***. This routine checks that the address to which the interface is being set is 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 on an active shared-memory network, use ***smNetShow( )***.[4]

4. This routine is not built in to the Tornado shell. To use it from the Tornado shell, define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. Configuration.*

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

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

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

```
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)          : fred
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)          : fred
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 is enabled when **INCLUDE_SM_SEQ_ADDR** is defined in **configAll.h**.

5.4.4 Configuring the Host

For UNIX, configuring the host to support the shared-memory network is done by using the procedures outlined earlier in this chapter for non-shared-memory networks. In particular, a shared-memory network requires that:

- All shared-memory network host names and addresses must be entered in **/etc/hosts**.
- All shared-memory network host names must be entered in **.rhosts** in your home directory or in **/etc/hosts.equiv** (only if you are using RSH).
- A gateway entry must specify 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 *5.5 Proxy ARP*, p.316.)

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.

5.4.5 Example Configuration

This section illustrates the foregoing discussion with an example of a simple shared-memory network. The configuration consists of a single host and two target processors on a single backplane. In addition to the two processors, the backplane also has a separate memory board for the shared-memory pool, and an Ethernet controller board. The additional memory board is not essential, but makes for a configuration that is easier to describe.

The configuration shown in Figure 5-12 has two networks: the Ethernet and the shared-memory network. The Ethernet is assigned network number 150, and the shared-memory network is assigned 161. The host is **h1**, and is assigned the Internet address 150.12.0.1.

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, in fact, on the Ethernet. However, it can communicate with **h1** over the shared-memory network, using **vx1** as a gateway. Of course, gateway use is handled by the Internet protocol and is completely transparent to the user.

The example network address assignments are shown in Table 5-13.

Figure 5-12 **Example Shared-Memory Network**

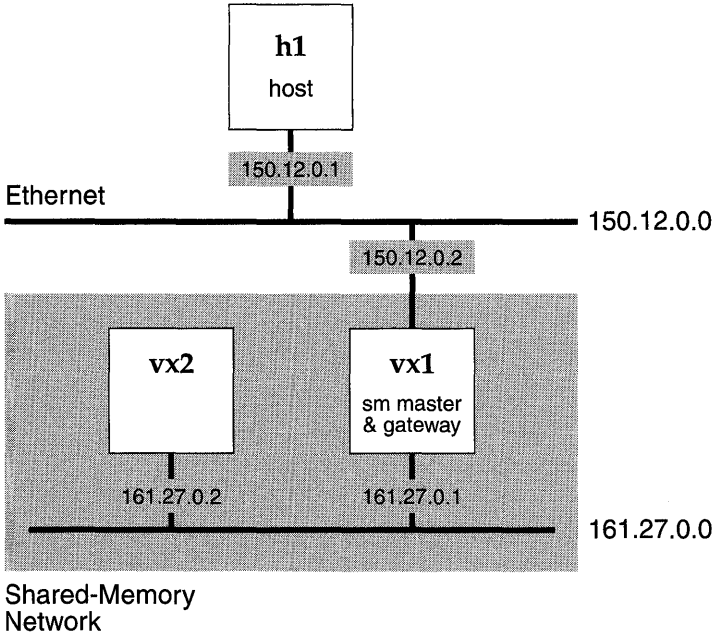


Table 5-13 **Network Address Assignments**

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
```

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 systems are configured in part by the parameters shown in Table 5-14.

Table 5-14 **Parameters in config.h**

Parameter	Value	Comment
SM_ANCHOR_ADRS	`0x800000`	Address of anchor as seen by **vx1**.
SM_MEM_ADRS	`0x800000`	Address of shared-memory pool as seen by **vx1**.
SM_MEM_SIZE	`0x80000`	Size of shared-memory pool, in bytes.
SM_INT_TYPE	SM_INT_MAILBOX_1	Interrupt targets with 1-byte write mailbox.
SM_INT_ARG1	VME_AM_SUP_SHORT_IO	Mailbox in short I/O space.
SM_INT_ARG2	`(0xc000|(sysProcNum * 2))`	Mailbox at: 0xc000 for **vx1** 0xc002 for **vx2**
SM_INT_ARG3	`0`	Write 0 value to mailbox.

The backplane master, **vx1**, has the following boot parameters:

```
boot device             : ln
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)                : fred
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 you should use, see your BSP documentation.

The other target, **vx2**, has the following boot parameters:

```
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)                : fred
ftp password (pw) (blank=use rsh)†:
flags (f)               : 0
```

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

5.4.6 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.

1. To begin with, 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. Now 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. Next, 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. When VxWorks is up, you can display the state of the shared-memory network with the ***smNetShow( )*** routine,[5] 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 totals to zero.

5. Now power off and add the second processor board. Remember that the second processor must *not* be configured to be 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 in the position of the various boards' memory addresses.

7. With the **d** and **m** boot ROM commands, 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. Using the **d** command on the second processor, look for the shared-memory anchor. The anchor begins with the ready value of 0x8765 (see Figure 5-10). You can also look for the shared-memory heartbeat; see *The Shared-Memory Heartbeat*, p.304.

9. When 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 attaching correctly 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.

---

5. This routine is not built in to the Tornado shell. To use it from the Tornado shell, define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. 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.

5.5 Proxy ARP

Proxy ARP provides transparent network access by using the Address Resolution Protocol (ARP) to make distinct networks appear as one logical network (that is, the networks share the same address space). The proxy ARP scheme implemented in VxWorks provides an alternative to the use of explicit subnets for accessing the shared-memory network. See *5.4 Shared-Memory Networks*, p.301. [6]

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 is 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.

5.5.1 ARP Introduction

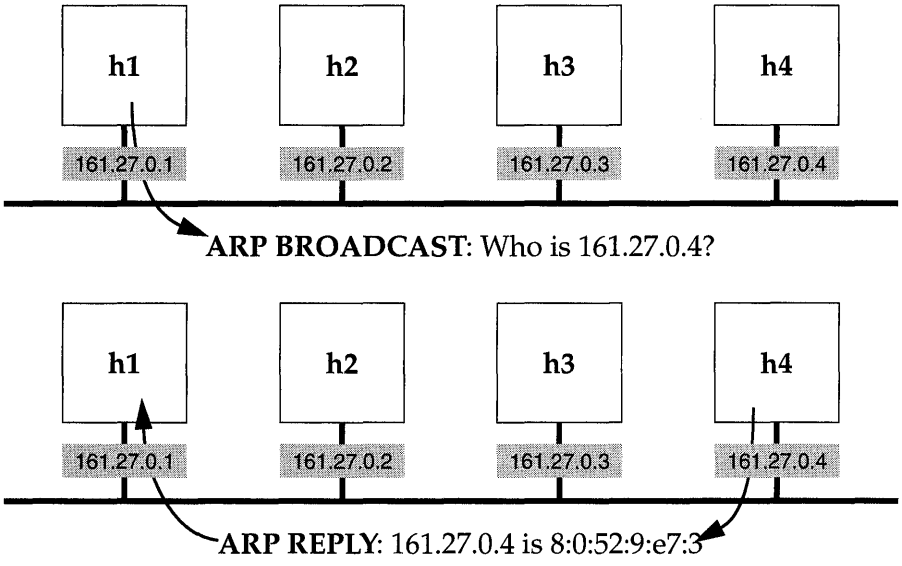
ARP is used to resolve a host's IP address into a hardware address. This is done by broadcasting an ARP request on the physical medium (typically Ethernet). The

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

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 5-13, 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 5-13 **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 5-14, **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:

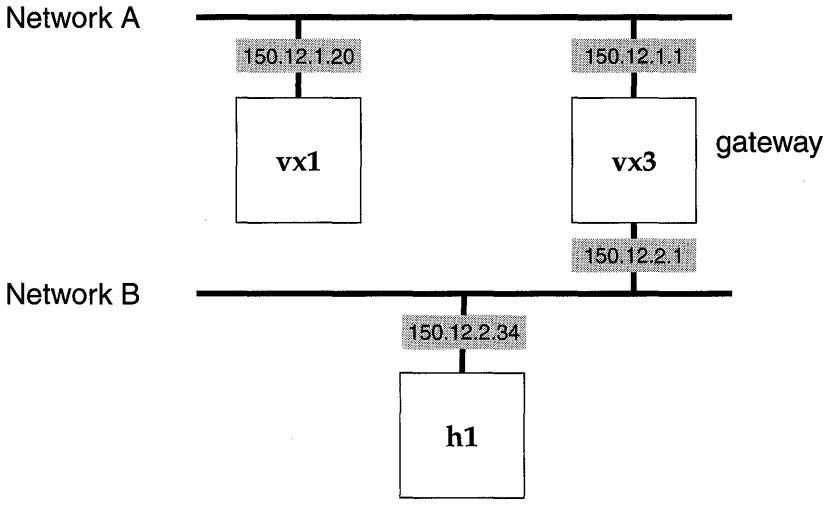
```
node            destination             gateway
vx1             150.12.2.0               150.12.1.1 (network)
```

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 does not know the hardware address for the gateway listed in its routing table, it sends an ARP request for the gateway's hardware address.

Figure 5-14 **Subnets and ARP**

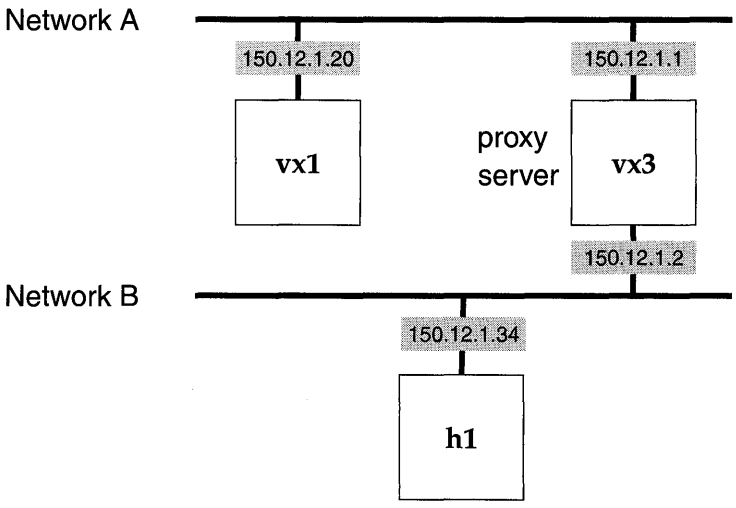


5.5.2 Proxy ARP Overview

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

The example configuration shown in Figure 5-14 looks different when proxy ARP is used. As shown in Figure 5-15, 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 5-15 **Proxy ARP Example**



5.5.3 Routing Issues on the Proxy Server

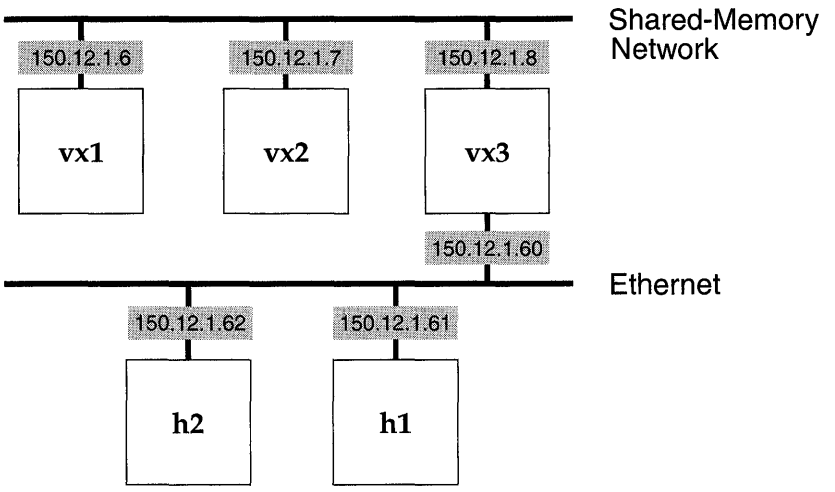
The proxy server provides network transparency by listening to and answering ARP messages, and by manipulating its routing tables. Suppose the proxy server had two interfaces: shared-memory network and Ethernet. Nodes residing on different interfaces can have the same network address if host-specific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

In the example in Figure 5-16, **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 (host)               150.12.1.8
150.12.1.7 (host)               150.12.1.8
150.12.1.0 (network)            150.12.1.60
```

The network on which the proxy server performs host-specific routing (or for which it is acting as a proxy) is referred to as the *proxy network*. The proxy server has a host-specific route 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 5-16, the shared-memory network is the proxy network and the Ethernet is the main network. The routing table of **vx3** has host-specific routes for both **vx1** and **vx2**. To send to nodes **h1** and **h2**, it uses the network route

Figure 5-16 **Proxy Server Example**



(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 host-specific routes 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. Also, it is best to keep the host-specific routes to a minimum, because when resolving routes, the proxy server first searches all host-specific routes, and then all network routes.

5.5.4 Proxy ARP Protocol

ARP Requests for Proxy Clients

If the proxy server receives an ARP request for which the destination is 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. This happens only if the node that generated the ARP request does not reside on the same proxy network as the destination proxy client because if they are on the same network, the destination proxy client answers for itself.

In the example in Figure 5-16, 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 destination of the ARP request is known, the server generates and sends an ARP reply to the source proxy client. If the destination was not resolved previously, 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 5-16, **vx1** sends an ARP request for 150.12.1.62. If **vx3** knows the destination, 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).

5.5.5 Broadcast Datagrams

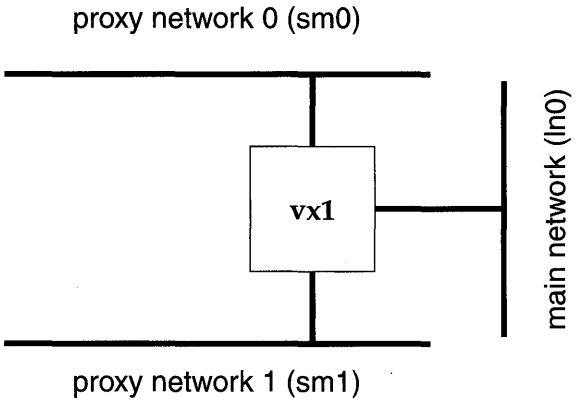
All nodes on a logical network are expected to receive an IP broadcast for that network (for example, 150.12.1.255). Thus, broadcasts must be passed through the proxy server so that nodes on both the proxy network and the main network receive them. Because most broadcast traffic is extraneous, it is 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 must 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 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 5-17, if a datagram comes from **sm1**, it gets forwarded to **ln0** and **sm0**.

Figure 5-17 **Broadcast Datagram Forwarding**



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 5-17, 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.

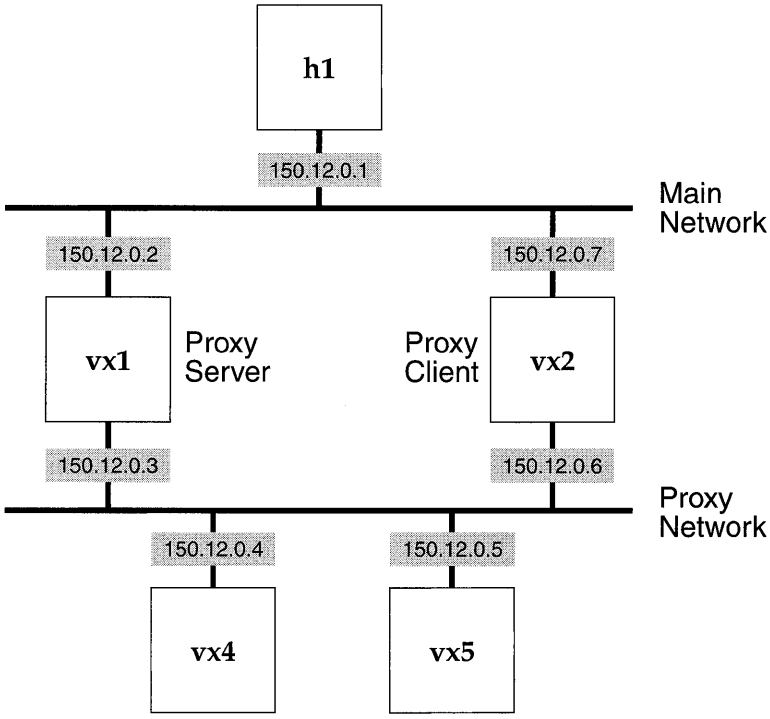
Although forwarding broadcasts between interfaces is potentially dangerous (due to broadcast storms and forwarding loops), the restrictions put on the configuration make these situations unlikely. Even so, forwarding broadcasts between proxy and main interfaces is not recommended. Therefore, forward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

5.5.6 Multi-Homed Proxy Clients

Routing

If a proxy client has an interface to the main network, some additional configuration is required for optimal communications. The proxy client's routing tables must have host-specific routes 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 5-18, **vx1** is the proxy server and **vx2** is a proxy client with an interface on the main network. **vx2** must be configured to have host-specific routes 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**).

Figure 5-18 **Routing Example**



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 (host)             150.12.0.6
150.12.0.5 (host)             150.12.0.6
150.12.0.0 (network)          150.12.0.7
```

Broadcasts

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.)

5.5.7 Single-Tier Support

Proxy ARP works only for a single tier of shared-memory networks. That is, only interfaces directly attached to the proxy server can be proxied. Example configurations that work are shown in Figure 5-19 and Figure 5-21. However, the configuration shown in Figure 5-20 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.

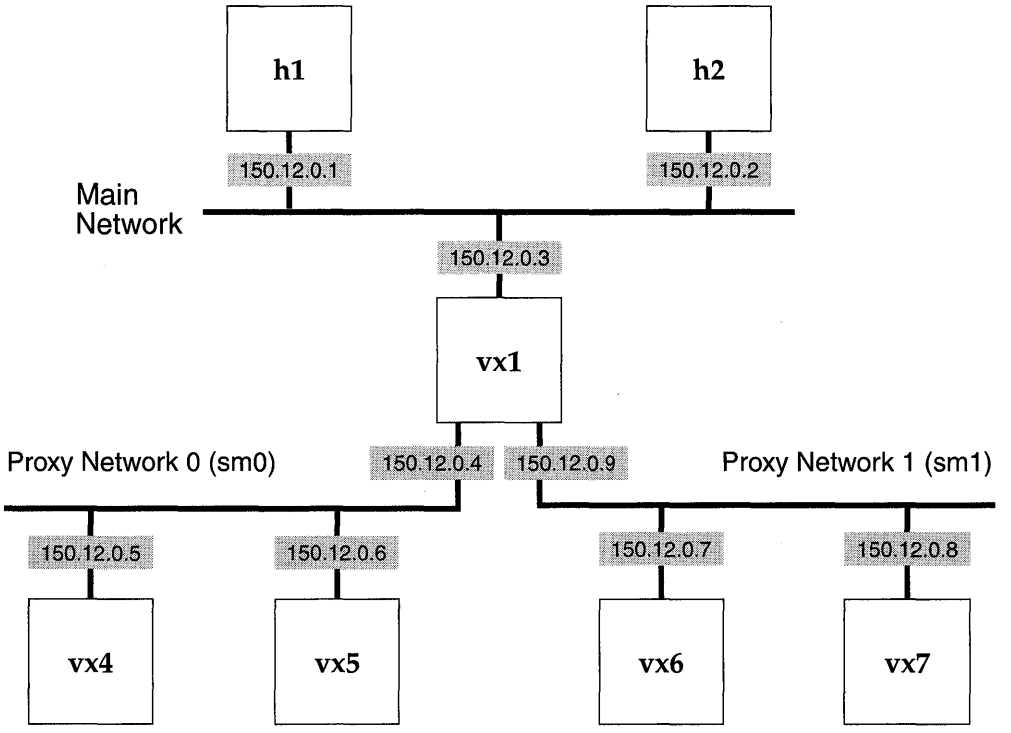
To work, the configuration in Figure 5-20 requires a combination of proxy ARP and IP routing (or standard subnet routing). The modified configuration is shown in Figure 5-22, where Proxy Network 1 has become an IP routing network with a different network address. For **vx6** to send to **h2** in the 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
```

Figure 5-19 **Single-Tier Example Using Proxy ARP with Two Branches**



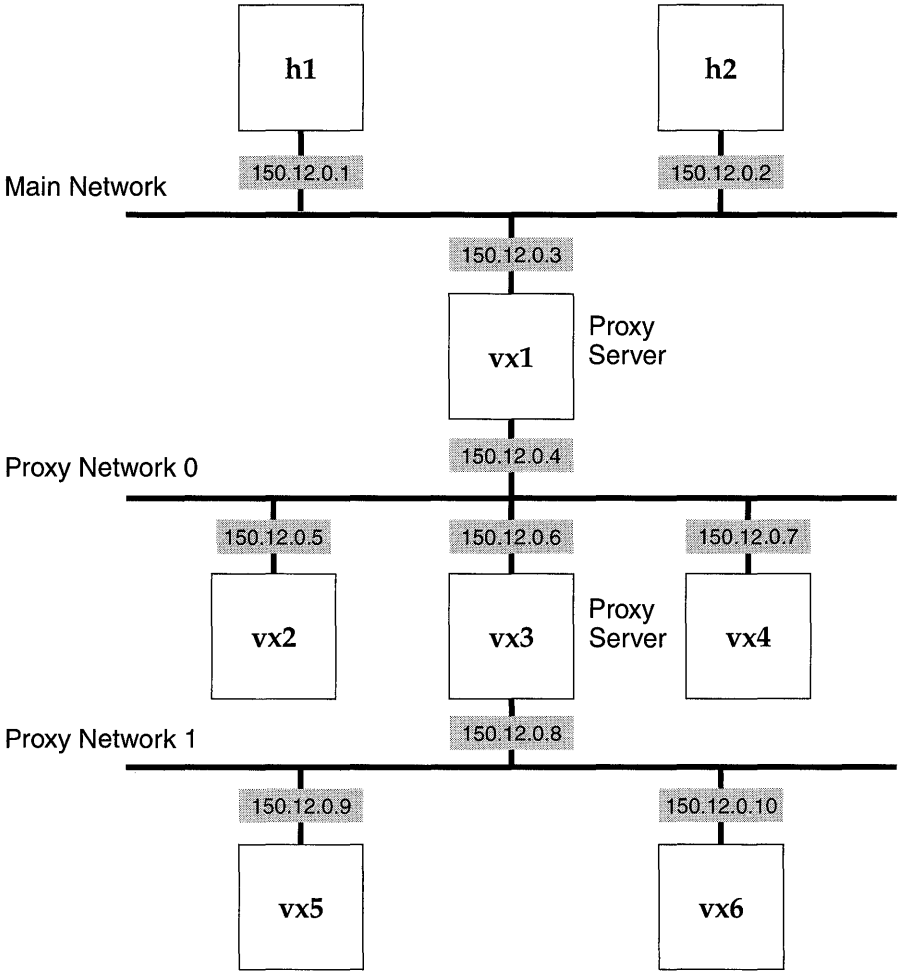
5.5.8 Subnets

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 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, define **INCLUDE_PROXY_SERVER** in **configAll.h** and rebuild VxWorks for the proxy server. If the target is processor zero (the shared-memory network master), the proxy server is enabled using the boot parameters **inet on ethernet (e)** for the main network, and **inet on backplane (b)** for the proxy network. From the example in Figure 5-21, **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
```

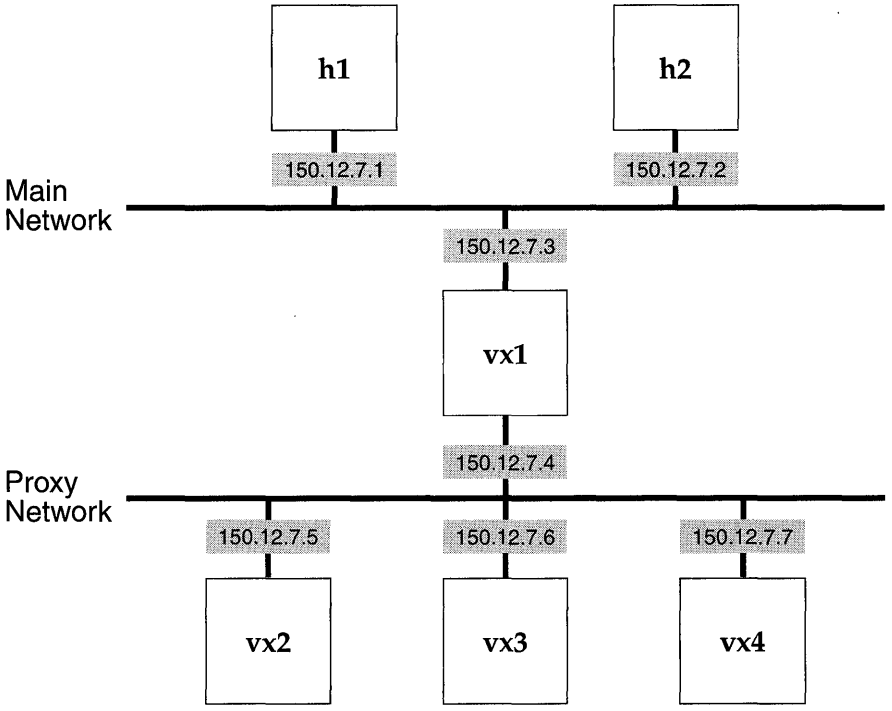
Figure 5-20 **Multi-Tier Configuration that *CANNOT* Be Used with Proxy ARP**



5.5.9 Configuration

The proxy server for the shared-memory network must be the master board. As previously mentioned, the server is configured by defining INCLUDE_PROXY_SERVER in **configAll.h**. If only INCLUDE_PROXY_SERVER is defined, 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.

Figure 5-21 **Another Single-Tier Example Using Proxy ARP**



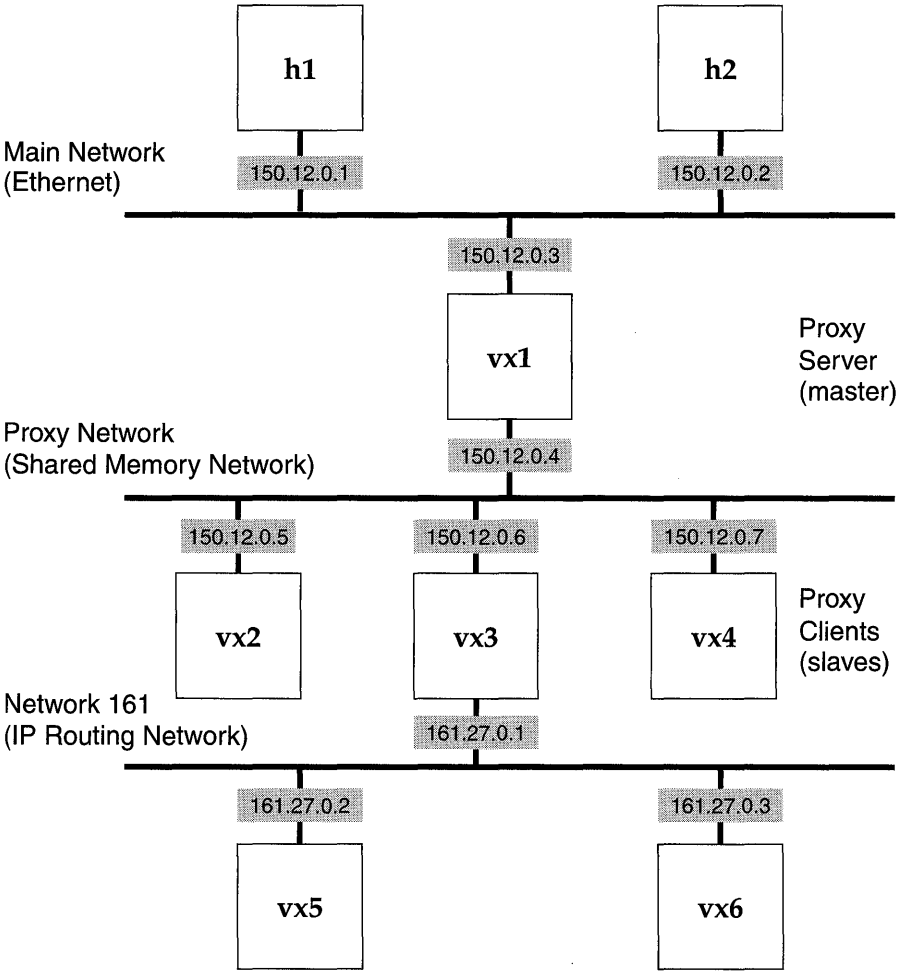
Sequential and Default Addressing

If such control is not required, it is possible to have the proxy server assign the inet addresses to the proxy clients. When **INCLUDE_SM_SEQ_ADDR** is defined, the proxy server assigns incremental inet addresses to the slave boards based on the proxy server's backplane inet address. 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 5-22.)

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 to the proxy server's

Figure 5-22 **Multi-Tier Example Using Proxy ARP and IP Routing**



network interface. The backplane address is assigned automatically by adding 1 (one) to the network interface address. To have the proxy server's backplane address configured by default, sequential addressing must also be used; both **INCLUDE_PROXY_DEFAULT_ADDR** and **INCLUDE_SM_SEQ_ADDR** must be defined in **configAll.h**. 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, assume that both **INCLUDE_PROXY_DEFAULT_ADDR** and **INCLUDE_SM_SEQ_ADDR** are defined: 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 0x100 flag in the boot flags instead of specifying the gateway. For additional information on booting with proxy ARP, see *5.10 Using TFTP, BOOTP, Sequential Addressing, Proxy ARP*, p.369.

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 since they have different **config.h** files.

- **Proxy ARP and Shared Memory Definition in configAll.h**

```
INCLUDE_PING
INCLUDE_SM_NET
INCLUDE_PROXY_SERVER
INCLUDE_SM_SEQ_ADDR             /* required only for default addressing */
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

For information on booting shared memory networks, see *5.4 Shared-Memory Networks*, p.301. After booting **vx1** (the master), use ***smNetShow( )*** to find the shared memory anchor, which is used as the slave boot device (for **vx2**, **vx3**, and **vx4**). 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 5-22:

```
-> 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
```

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

Debugging the Network

Diagnosing Shared Memory Booting Problems

For information on debugging the shared memory network, see *5.4.6 Troubleshooting*, p.314.

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 routine 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 clients.
proxyPortShow( )	This routine displays the ports currently enabled.

5.6 Serial Line Internet Protocol (SLIP and CSLIP)

VxWorks can communicate with the host operating system over serial connections as well as over networks and backplanes. The Serial Line Internet Protocol (SLIP) supports IP layer software with point-to-point configurations such as RS-232 serial connections or long-distance telephone lines. If either end of a SLIP connection has other network interfaces (such as Ethernet) and can forward packets to other machines, a SLIP connection can serve as a gateway between networks.

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).[7]

5.6.1 SLIP Configuration

Configuring your system for SLIP requires both target and host system configuration. See your host development system's manual for information on configuring your host.

WARNING: 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 will receive

7. If your host operating system does not include SLIP or CSLIP facilities, you may be able to use a publicly available implementation. One popular implementation for SunOS 4.1.*x*, the Van Jacobson CSLIP 2.7 release, is provided in **unsupported/cslip-2.7**. This code is publicly available, and is not supported by Wind River Systems; we include it only as a convenience.

packets from the host, but CSLIP packets from the target will not be correctly decoded by the host. Eventually TCP will resend the packets as SLIP packets, at which time the host will receive and acknowledge them. However, the whole process will be very slow. To avoid this, configure the host and target to use the same protocol.

To configure your VxWorks target to use SLIP, define the following in **configAll.h**:

1. To include SLIP, define **INCLUDE_SLIP**. By default this constant is part of the excluded facilities; move it to the INCLUDED SOFTWARE FACILITIES section.
2. To specify the *tty* to be used for the SLIP connection, define **SLIP_TTY**. By default **SLIP_TTY** is set to 1, which sets the serial device to **/tyCo/1**.
3. To specify the baud rate, optionally define **SLIP_BAUDRATE**. If this constant is not defined, SLIP uses the baud rate defined by your serial driver.
4. To specify the use of CSLIP, define either of the following:
   (a) To always use CSLIP to communicate with the host, define **CSLIP_ENABLE**.
   (b) To use plain SLIP unless the VxWorks target receives a CSLIP packet (in which case the target also uses CSLIP), define **CSLIP_ALLOW**.

5.6.2 Booting VxWorks and Accessing Files Using SLIP or CSLIP

When booting using SLIP (or its CSLIP variant), specify the boot device as follows:

```
boot device: sl
```

or:

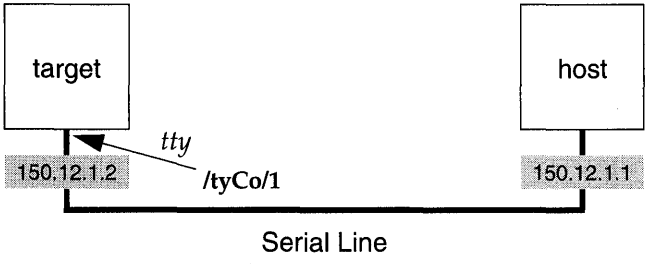
```
sl=device
```

Using the form **sl=***device* allows you to specify the SLIP *tty*, overriding the constant **SLIP_TTY**. The following is a boot example for the configuration shown in Figure 5-23:

```
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 **src/config/usrNetwork.c**. This sets up the SLIP *tty*, and configures the point-to-

Figure 5-23 **SLIP Configuration Example**



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 specified, the SLIP driver assumes that the point-to-point peer address is on the other end of the serial line and enters the appropriate routing entry.

If you do not have a console device:

- Set the constant **CONSOLE_TTY** to **NONE** and define the *tty* port number using the constant **SLIP_TTY** in **config.h.**

```
#define   CONSOLE_TTY   NONE
#define   SLIP_TTY      0       /* use port number 0 for slip */
```

- Specify the boot parameters using the constant **DEFAULT_BOOT_LINE** in **config.h** before making your boot ROMs. For example:

```
#define DEFAULT_BOOT_LINE \
"sl(0,0)phobos:/usr/wind/target/config/ads302/vxWorks h=150.12.1.1
e=150.12.1.2 u=jane"
```

  For the boot device, **sl(0,0)**, the first number is the unit number for the boot device and the second is the processor number. You can determine which unit number was used for the boot device by calling ***ifShow*( )** from the shell.

- If your system has nonvolatile RAM (NVRAM), edit **sysLib.c** and change ***sysNvRamGet*( )** to return **ERROR**. This forces the use of the constant **DEFAULT_BOOT_LINE** instead of using the value stored in NVRAM.

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.

5.7 Point-to-Point Protocol (PPP)

5.7.1 Introduction

PPP for Tornado Features

The following features are supported by PPP:

- **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 the proxy-server peer's connected network to "see" the proxy-client peer without manually adding routing entries.

The Point-to-Point Protocol Compared to SLIP

For many years, networking Internet Protocol (IP) packets over serial lines was almost exclusively accomplished with 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 a smaller amount of 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 networking 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 networking, PPP advantages 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 communication of different network protocols over each link. At present, the only protocols supported by PPP for Tornado 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 serves to automatically ensure 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 of all for using PPP.

5.7.2 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.[8]

Including PPP in VxWorks may cause the loading of the VxWorks system image to fail. 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

8. If your host operating system does not provide PPP facilities, you may be able to use a publicly available implementation. One popular implementation for SunOS 4.1.*x* (and several other hosts) is version **ppp-2.1.2**, which is provided in the **unsupported/ppp-2.1.2** directory. This code is publicly available and is included with the PPP for Tornado only as a convenience. This code is not supported by Wind River Systems.

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.

PPP facilities can be configured into VxWorks by defining the appropriate configuration constants. For general information on configuring VxWorks, see *8. Configuration.*

To include the default PPP configuration in VxWorks, define **INCLUDE_PPP** in the INCLUDED FACILITIES section of **configAll.h**, or define it in **config.h** in your BSP directory.

To include the optional DES cryptographic package for use with the Password Authentication Protocol (PAP), define **INCLUDE_PPP_CRYPT**. It is not included in the standard Tornado Release tape; 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.353).

PPP for Tornado has many optional features (approximately 50 in all) that can be configured in to enable the PPP capabilities listed in *5.7.1 Introduction*, p.334. There are three methods of configuration:

- At compile-time, by setting configuration constants in **configAll.h**. Use this method with ***usrPPPInit( )***. (See *Initializing a PPP Link*, p.343.)
- At run-time, by filling in a PPP options structure. Use this method with ***pppInit( )***. (See *Initializing a PPP Link*, p.343.)
- 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 5-17 on Page 348.

Selecting PPP Options by Using Configuration Constants in configAll.h

The various configuration options offered by PPP for Tornado can be initialized at compile-time by defining a number of configuration constants in **configAll.h**.

First, make sure the **PPP_OPTIONS_STRUCT** constant is defined in **configAll.h** (it is defined by default). Unless **PPP_OPTIONS_STRUCT** is defined, configuration options 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 5-15 lists the principal configuration constants used with PPP for Tornado.

Table 5-15 **PPP Configuration Constants**

Constant	Facility Included
INCLUDE_PPP	Include PPP.
INCLUDE_PPP_CRYPT	Include DES cryptographic package.
PPP_OPTIONS_STRUCT	Enable configuration options set in **configAll.h**.
PPP_TTY	Define default serial interface.
PPP_CONNECT_DELAY	Define initialization delay for link establishment.

Table 5-16 shows the two basic formats used for configuration options in **configAll.h**. The full array of options available with PPP for Tornado appear with their definitions in column 1 of Table 5-17 on page 348. By default, all of these constants are turned off. To enable any **PPP_OPT**_*option* constant, define its value to be 1 (these option constants are boolean values). To set any **PPP_STR**_*optionstring* option, 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"
```

Table 5-16 **PPP Configuration Options in configAll.h**

Configuration Option	Option Included
PPP_OPT_*option*	Specify a PPP configuration option.
PPP_STR_*optionstring*	Specify a PPP configuration option string.

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

configuration file; thus, to realize the changes and enable the configuration options, first recompile VxWorks, then initialize PPP by invoking ***usrPPPInit( )*** manually (see *Initializing a PPP Link*, p.343) or by having it called automatically by the boot code (see *Booting VxWorks Using PPP*, p.345).

Selecting PPP Options by 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.343).

The PPP options structure is typedef'ed to **PPP_OPTIONS**, and its definition is located in **h/netinet/ppp/options.h**, which is included indirectly 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 5-17, page 348. 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 also initializes a PPP interface that uses the target's second serial port (**/tyCo/1**). The local IP address is 90.0.0.1; the IP address of the remote peer is 90.0.0.10. The baud rate is the default rate for the *tty* device. The VJ compression and authentication options have been disabled, and LCP (Link Control Protocol) echo requests have been enabled.

```
PPP_OPTIONS pppOpt;    /* PPP configuration options */

void routine ()
    {
    pppOpt.flags = OPT_PASSIVE_MODE | OPT_NO_PAP | OPT_NO_CHAP |
                   OPT_NO_VJ;
    pppOpt.lcp_echo_interval = "30";
    pppOpt.lcp_echo_failure = "10";

    pppInit (0, "/tyCo/1", "90.0.0.1", "90.0.0.10", 0, &pppOpt, NULL);
    }
```

Setting PPP Options by 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 *Local File Systems*, p.187.

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 **configAll.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 by Using an Options Structure*, p.339. 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
```

5.7.3 The Point-to-Point Protocol (PPP)

The Point-to-Point Protocol (PPP) is comprised of several different protocols that work together with the PPP network interface driver to support a variety of network stacks. PPP for Tornado presently supports only the TCP/IP stack.

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 issued. It is intended that PPP provide a common solution for easy connecting among a variety of hosts, bridges, and routers.

PPP 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.

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 5-24.

Figure 5-24 **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 has been established, PPP provides for an optional authentication. For more information, see RFC 1548 (for more information, see *Requests for Comments (RFC)*, p.360).

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 completed link establishment. IPCP is also responsible for IP address negotiation between peers. For more information, see RFC 1332 (see *Requests for Comments (RFC)*, p.360).

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 establishment of a link. 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 method. 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.353, or refer to RFC 1334 (see *Requests for Comments (RFC)*, p.360).

Challenge-Handshake Authentication Protocol (CHAP)

Challenge-Handshake Authentication Protocol (CHAP) is a more robust authentication protocol offering better security. 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 anytime afterward.

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.354. For further technical details, refer to RFC 1334 (see *Requests for Comments (RFC)*, p.360).

5.7.4 Using PPP

Once configured and initialized, PPP for Tornado 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.347.)

usrPPPInit( )

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( )*** as follows:

- From the VxWorks shell.
- By user application code.

Use either syntax when calling ***usrPPPInit( )***:

```
usrPPPInit ("bootdevice", "local IP address", "remote IP address")
usrPPPInit ("bootdevice", "local host name", "remote host name")
```

You can use host names in ***usrPPPInit( )*** provided the hosts have been previously added to the host database by making calls to ***hostAdd( )*** as follows:

```
hostAdd ("hostname", "host IP address")
```

For example, you can call ***usrPPPInit( )*** in the following way:

```
usrPPPInit ("ppp=/tyCo/1,38400", "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 Using Configuration Constants in configAll.h*, p.337). The ***pppInit( )*** routine can be called multiple times to initialize multiple channels. Note that ***usrPPPInit( )*** is hard-coded to initialize a single channel, PPP unit 0, and that 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.

pppInit( )

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 by Using an Options Structure*, p.339). Unlike ***usrPPPInit( )***, the return value of ***pppInit( )*** does not indicate the status of the PPP link; it merely reports whether the link could be initialized. To check whether the link is actually established, call ***pppInfoGet( )*** and make sure that the state of IPCP is **OPENED**. The following code fragment demonstrates use of this mechanism for PPP unit 2:

```
PPP_INFO    pppInfo;

if ((pppInfoGet (2, &pppInfo) == OK) &&
    (pppInfo.ipcp_fsm.state == OPENED))
    return (OK);                                /* link established */
else
    return (ERROR);                             /* link down */
```

Deleting a PPP Link

There are two ways to delete a PPP link:

- 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 */
```

Booting VxWorks Using PPP

To boot VxWorks using PPP, first configure PPP into the system (see *5.7.2 Configuration*, p.336) 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:

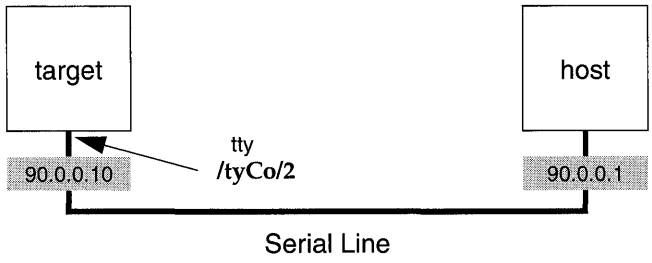
- **boot device: ppp**
- **ppp**=*device*
- **ppp**=*device,baudrate*
- **ppp**,*baudrate*

If using **boot device: ppp**, then the serial channel is set to **PPP_TTY** in **configAll.h** and the baud rate is set to the default baud rate of the channel. Specifying **ppp**=*device* allows you to choose the PPP *tty* (serial channel), overriding the **PPP_TTY** constant. Specifying **ppp**=*device,baudrate* allows you to choose the PPP *tty* (serial channel) and the baud rate of the channel. The default baud rate used by the PPP *tty* (serial channel) can be configured into the system by defining the constant **PPP_BAUDRATE** (in **configAll.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 5-25:

```
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
```

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 configuration options defined at compile-time in **configAll.h** (see *Selecting PPP Options by Using Configuration Constants in configAll.h*, p.337). If a gateway address is specified, the PPP driver adds a routing entry from the gateway address to the host address. If a

Figure 5-25 **PPP Configuration Example**



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.

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 constant **CONSOLE_TTY** to **NONE** and define the *tty* port number using the constant **PPP_TTY** in **configAll.h.**

```
#define   CONSOLE_TTY   NONE
#define   PPP_TTY       0       /* use port number 0 for PPP */
```

2. Specify the boot parameters using **DEFAULT_BOOT_LINE** in **config.h** before making your boot ROMs. Changing any of the default PPP settings requires new boot ROMs. For example:

```
#define DEFAULT_BOOT_LINE \
"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 **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.

PPP Options

Table 5-17 lists all the configuration options supported by PPP for Tornado. Each configuration option is shown in its three forms, corresponding to the configuration methods explained in the following sections:

Column 1: *Selecting PPP Options by Using Configuration Constants in configAll.h*, p.337

Column 2: *Selecting PPP Options by Using an Options Structure*, p.339

Column 3: *Setting PPP Options by Using an Options File*, p.339.

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 **configAll.h** or set by passing the structure **PPP_OPTIONS** to ***pppInit( )***. For example:

- If **PPP_OPT_NO_PAP** is activated in **configAll.h** (negating the use of PAP), a subsequent setting of **require_pap** in **PPP_OPTIONS_FILE** overrides the earlier setting enabling PAP authentication.
- 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 reset to FFFFFF00.

PPP Authentication

PPP for Tornado provides security through two authentication protocols: PAP (see *Password Authentication Protocol (PAP)*, p.342) and CHAP (see *Challenge-Handshake Authentication Protocol (CHAP)*, p.342). This section introduces the use of PPP link-layer authentication (introduced in *Link Control Protocol (LCP)*, p.341), and describes the format of the secrets files.

In VxWorks, the default behavior of PPP is to authenticate itself when requested by a peer but not to require authentication from a peer. If additional security is required, choose PAP or CHAP by turning on 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

Table 5-17 **PPP Configuration Options**

Options			Description
Set in configAll.h	**Set using options structure**	**Set using options file**	
PPP_OPT_NO_ALL	OPT_NO_ALL	no_all	Do not request/allow any options.
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_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_OPT_DEFAULT_ROUTE	OPT_DEFAULT_ROUTE	default_route	When IPCP negotiation is successfully completed. Add a default route to the system routing tables, using the peer as the gateway. This entry is removed when the PPP connection is broken.
PPP_OPT_PROXY_ARP	OPT_PROXY_ARP	proxy_arp	Add an entry to this system's ARP (Address Resolution Protocol) table with IP address of the peer and the Ethernet address of this system.
PPP_OPT_IPCP_ACCEPT_LOCAL	OPT_IPCP_ACCEPT_LOCAL	ipcp_accept_local	Set PPP to accept the 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 peer's idea of its (remote) IP address, even if the remote IP address was specified.
PPP_OPT_NO_IP	OPT_NO_IP	no_ip	Disable IP address negotiation in IPCP.
PPP_OPT_NO_ACC	OPT_NO_ACC	no_acc	Disable address/control compression.
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.

Table 5-17 **PPP Configuration Options** *(Continued)*

Options			Description
Set in configAll.h	**Set using options structure**	**Set using options file**	
PPP_OPT_NO_VJCCOMP	OPT_NO_VJCCOMP	**no_vjccomp**	Disable VJ (Van Jacobson) connection ID compression.
PPP_OPT_NO_VJCCOM	OPT_NO_ASYNCMAP	**no_asyncmap**	Disable async map negotiation.
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_CHAP	OPT_NO_CHAP	**no_chap**	Do not allow CHAP authentication with peer.
PPP_OPT_REQUIRE_PAP	OPT_REQUIRE_PAP	**require_pap**	Require PAP authentication with peer.
PPP_OPT_REQUIRE_CHAP	OPT_REQUIRE_CHAP	**require_chap**	Require CHAP authentication with peer.
PPP_OPT_LOGIN	OPT_LOGIN	**login**	Use the login password database for PAP authentication of peer.
PPP_OPT_DEBUG	OPT_DEBUG	**debug**	Enable PPP daemon debug mode.
PPP_OPT_DRIVER_DEBUG	OPT_DRIVER_DEBUG	**driver_debug**	Enable PPP driver debug mode.
PPP_STR_ASYNCMAP	**char * asyncmap**	**asyncmap** *value*	Set the desired async map 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_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_STR_NETMASK	**char * netmask**	**netmask** *value*	Set the network mask value for negotiation to the specified value.

Table 5-17 **PPP Configuration Options** *(Continued)*

Options			Description
Set in configAll.h	**Set using options structure**	**Set using options file**	
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_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_RESTART	char * lcp_restart	lcp_restart *value*	Set the timeout in seconds for the LCP negotiation 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_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_IPCP_RESTART	char * ipcp_restart	ipcp_restart *value*	Set the timeout in seconds for the IPCP negotiation 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_IIPCP_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.

Table 5-17 **PPP Configuration Options** *(Continued)*

Options			Description
Set in configAll.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_REMOTE_AUTH_NAME	char * remote_auth_name	remote_auth_name *name*	Set the remote name for authentication to the specified name.
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_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_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_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_CHAP_FILE	char * chap_file	chap_file *file*	Get CHAP secrets from the specified file. This option is necessary if either peer requires CHAP authentication.
PPP_STR_CHAP_RESTART	char * chap_restart	chap_restart *value*	Set the timeout in seconds for the CHAP negotiation to the specified value.
PPP_STR_CHAP_INTERVAL	char * chap_interval	chap_interval *value*	Set the interval in seconds for CHAP rechallenge to the specified value.
PPP_STR_MAX_CHALLENGE	char * max_challenge	max_challenge *value*	Set the maximum number of transmissions for CHAP challenge requests to the specified value.

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: 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 5-18 is an example of a secrets file. It could be either a PAP or CHAP secrets file, since their formats are identical.

Table 5-18 **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 5-17, Page 348.)

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 defining **PPP_STR_PAP_FILE** in **configAll.h** with the path name of the PAP secrets file.
- By setting the **pap_file** member of the **PPP_OPTIONS** structure passed to ***pppInit*( )**.
- By adding the following line entry in the options file specified by **PPP_OPTIONS_FILE** in **configAll.h**:

```
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 defining **PPP_OPT_REQUIRE_PAP** as 1 in **configAll.h**.
2. By setting the flag **OPT_REQUIRE_PAP** in the **flags** bitfield of the **PPP_OPTIONS** structure passed to ***pppInit*( )**;
3. By adding the following line entry in the options file specified by **PPP_OPTIONS_FILE** in **configAll.h**.

```
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 defining **PPP_STR_CHAP_FILE** in **configAll.h** with the path name of the CHAP secrets 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 specified by **PPP_OPTIONS_FILE** in **configAll.h**:

```
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 defining **PPP_OPT_REQUIRE_CHAP** to 1 in **configAll.h**.
- By setting the flag **OPT_REQUIRE_CHAP** in the **flags** bitfield of the **PPP_OPTIONS** structure passed to ***pppInit*( )**.

- By adding the following line entry in the options file specified by **PPP_OPTIONS_FILE** in **configAll.h**:

```
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 following code example demonstrates how to hook the example routines, ***connectRoutine( )*** and ***disconnectRoutine( )***, into the PPP connection establishment mechanism and termination mechanism, respectively:

Example 5-5 **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);
    }

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

5.7.5 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 *5.7.2 Configuration*, p.336.
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.343.

The additional PPP link is now ready for use.

PPP Link as a Network Back End for the Target Server on the Host

1. Define the constant **INCLUDE_PPP** in **configAll.h** and make new boot ROMs for the target. For more information, see *5.7.2 Configuration*, p.336.
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 the SUN OS 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.*

The PPP connection is like a network back end except that the connection is 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 on a regular network back end. (See the *Tornado User's Guide.*)

NOTE: 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.*

5.7.6 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 peer should be turned off. (These can always be configured later.)

Problems with PPP generally occur in either of two areas: when establishing links and when pursuing 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, increase the value of **PPP_CONNECT_DELAY** in **configAll.h**. By default, **PPP_CONNECT_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. (Select **PPP_OPT_DEBUG** in **configAll.h**, or use the alternative options in Table 5-17, page 348.) 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.

5.7.7 PPP Reference List

Requests for Comments (RFC)

The following is a list of relevant Requests for Comments (RFC) 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)

RFC 1549 PPP in HDLC Framing

PPP Newsgroup

The **comp.protocols.ppp** USENET newsgroup 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.)

5.8 Network Initialization on Startup

Most of the information that VxWorks uses to set up its network and access its boot host is taken from the boot parameters you supply to the VxWorks boot ROMs with the boot line or the boot menu commands. This section summarizes the network configuration performed automatically by VxWorks, based on these parameters. Most of this configuration is done by ***usrNetInit( )*** in **src/config/usrNetwork.c**. VxWorks startup procedures configure the network based on the following boot parameters:

boot device — The network device to boot from; for example, **ln** for a Lance Ethernet controller. This device is attached and configured automatically with the correct Internet address.

host name — The name of the host to boot from. This need not be the same name used internally by that system. VxWorks adds the host name to the host table and creates a device by that name.

host inet — The Internet address of the host to boot from.

inet on ethernet — The Internet address of this target on the Ethernet, if any. If the target has no Ethernet controller (perhaps because it boots from a backplane network through a gateway), leave this field blank (unless the target is being booted using SLIP). A subnet mask can also be specified as described in *5.3.9 Using Subnets*, p.297.

inet on backplane — The Internet address of this target on the backplane network. This field can be blank if no shared-memory network is required. Again, a subnet mask can be specified as described previously.

gateway inet — The Internet address of the gateway through which to boot, if the host is not on the same network as the target.

file name — The full path name of the VxWorks object module to be booted.

processor number — The backplane processor number of the target CPU. The first CPU must be processor number 0 (zero).

See *5.9.3 The VxWorks Boot Parameters*, p.366 for more boot parameter information.
The preceding parameters configure the following network elements:

Ethernet interface
: If **inet on ethernet** is specified, the Ethernet interface is attached; the Internet address and the optional net mask are set using ***ifAddrSet( )*** and ***ifMaskSet( )***.

Backplane interface
: If **inet on backplane** is specified, the backplane interface is attached; the Internet address and the optional net mask are set using ***ifAddrSet( )*** and ***ifMaskSet( )***.

Host names
: Host name entries are added using ***hostAdd( )*** for the specified boot host and for loop-back ("localhost").

Routing
: If a gateway address is specified, a routing entry is added indicating that the address is a gateway to the network of the specified boot host.

Remote file access device
: This device is created with the name "*boothost*:". If a password is specified, FTP is used; otherwise RSH is used.

Network File System
: If the NFS client is included and **INCLUDE_NFS_MOUNT_ALL** is defined, VxWorks mounts all file systems that are exported by the boot host. You must set the NFS user ID and group ID correctly: either dynamically by calling ***nfsAuthUnixSet( )***, or in the configuration file **config/all/configAll.h**.

Remote login
: **rlogin** is initialized if no password is specified; otherwise **telnet** is initialized.

User name and password
: These are initialized as specified in the boot parameters.

Current working directory
: This is set to the remote file access device called *boothost*:.

5.9 BOOTP (Bootstrap Protocol)

BOOTP is a basic bootstrap protocol implemented over the Internet User Datagram Protocol (UDP). It allows a booting target to configure itself dynamically by obtaining its IP address, the boot file name, and the boot host's IP address over the network, instead of the more traditional method of using the information encoded in the target's non-volatile RAM or ROM. BOOTP retrieves these target parameters. The actual transfer of the boot image is performed by a file transfer program (TFTP, FTP, or RSH). BOOTP and TFTP are commonly used together for network booting. [9]

BOOTP offers centralized management of target boot parameters on the host system. Using BOOTP, the VxWorks target can have the boot parameters specified by the host system, and VxWorks systems can be set up so that configuration on the target is unnecessary; see *5.9.3 The VxWorks Boot Parameters*, p.366.

A BOOTP server must be running or set up (with **inetd**) to run on the boot host, and the boot parameters for the target must be entered into the BOOTP database (**bootptab**). The format of this database is server specific. An example **bootptab** format is described in *5.9.2 The BOOTP Database*, p.364.

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 on the client's IP address if that field is specified; if not, it searches on the client's hardware address.

After 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.

5.9.1 The BOOTP Server

The BOOTP server resides on the UNIX host and is therefore host-specific. Many hosts provide a server as part of the standard operating system. Refer to the manuals for your host for information about the BOOTP server and the structure of the BOOTP database file (**bootptab**).

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 **unsupported/bootp2.1**.

---

9. For the complete BOOTP protocol specification, refer to RFC 951 "Bootstrap Protocol (BOOTP)" and RFC 1048 "BOOTP Vendor Information Extensions."

5.9.2 The BOOTP Database

To register a VxWorks target with the BOOTP server, 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 target **vx244** specifies 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**.

Registering the VxWorks Target

Log onto the boot server and add an entry to the database that corresponds to the target by entering the target address (**ha=**), IP address (**ip=**), and boot file (**bf=**).

To add a target called **vx245**, with Ethernet address 00:00:4B:0B:B3:A8, IP address 150.12.1.245, and boot file **vxBoot/vxWorks**, add the following to the end of the file:

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

The boot file name does not need to be added explicitly, because 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 **vxBoot/vxWorks.vx245**. If the file does not exist, the server looks for **vxBoot/vxWorks**.

Obtaining the Target Ethernet Address

Use the *ifShow*( ) routine[10] to determine the hardware address of a particular VxWorks target. In the following example, the target's Ethernet address is 00:00:4b:0b:b3:a8.

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

10. This routine is not built in to the Tornado shell. To use it from the Tornado shell, you must define **INCLUDE_NET_SHOW** in your VxWorks configuration; see *8. Configuration*.

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

From the VxWorks boot ROMs, obtain the hardware address with the **n** command:

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

5.9.3 The VxWorks Boot Parameters

The **boot device**, **processor number**, and **flags (f)** parameters must be specified in the boot ROMs. The **inet on ethernet (e)**, **file name**, and **host inet (h)** parameters can be obtained with BOOTP. The rest of the parameters can be specified by a default in **configAll.h**. The current defaults in **configAll.h** are:

```
/* Default Boot Parameters */
#define HOST_NAME_DEFAULT       "bootHost"    /* host name */
#define TARGET_NAME_DEFAULT     "vxTarget"    /* target name (tn) */
#define HOST_USER_DEFAULT       "target"      /* user (u) */
#define HOST_PASSWORD_DEFAULT   ""            /* password */
#define SCRIPT_DEFAULT          ""            /* startup script (s) */
#define OTHER_DEFAULT           ""            /* other (o) */
```

Table 5-13 shows where the various boot parameters can be specified.

5.9.4 Booting a VxWorks Target with BOOTP/TFTP

Booting Example

To boot a VxWorks target with BOOTP/TFTP:

1. Copy the VxWorks boot image to the boot directory on the boot host. For a standalone version of VxWorks, enter:

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

Table 5-19 **Specifying Boot Parameters**

Only in boot ROMs	In BOOTP Message	In configAll.h
boot device	inet on ethernet (e)	host name
processor number	file name	target name (tn)
flags (f)	host inet (h)	user (u)
		ftp password (pw)
		startup script (s)
		other (o)

2. Make sure the boot file permissions are accessible by all:

```
% chmod 644 vxWorks.vx245
% ls -l
total 609
drwxrwxrwx 2 root 512 Jul 6 15:58 ./
drwxrwxrwx 3 root 512 Jul 6 14:28 ../
-rw-r--r-- 1 target 519880 Jul 6 19:36 vxWorks.vx245
```

3. If the symbol table is required (for example, if you are using the target shell), copy it to the boot directory on the boot host:

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

NOTE: Although the boot file is retrieved with TFTP (and no authentication is required; see *5.3.4 Remote File Transfer Using TFTP*, p.291), the symbol table is not retrieved with TFTP. If the table is needed, **netDrv** file access is required. (For example, if you are using the target shell, see *9. Target Shell* in this manual.) For **netDrv** file access, either the user/password must be specified in the boot parameters, or a default user/password must be specified in **configAll.h**.

4. Enable BOOTP/TFTP in the VxWorks target boot parameters by specifying 0xc0 in the boot flags (0x40 specifies BOOTP, 0x80 is TFTP).

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

5. Boot the target:

```
[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
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.
Mounting NFS file systems from host bootHost for target vxTarget:
/usr
/home
```

Troubleshooting

If debugging mode is supported, put the BOOTP server in that mode.

No BOOTP Reply

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. Next, the target sends a TFTP request to get the file **/tftpboot/vxBoot/vxWorks.vx245**. 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**. The following commands can be used to do this:

```
% 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
```


5.10 Using TFTP, BOOTP, Sequential Addressing, Proxy ARP

Targets on the shared-memory network can boot with BOOTP only if proxy ARP is enabled (see *5.5.2 Proxy ARP Overview*, p.318). 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 something like:

```
vx230:ht=ethernet:ha=0000530e0018:ip=150.12.1.230:tc=global.dummy
```

The following example is a master processor that uses a combination of BOOTP, TFTP, proxy ARP, sequential addressing, and proxy default addressing for booting:

```
    [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] The BOOTP database

[2] **configAll.h** (defaults)

[3] The definition of **INCLUDE_PROXY_SERVER**, **INCLUDE_SM_SEQ_ADDR**, and **INCLUDE_PROXY_DEFAULT_ADDR** in **configAll.h**. (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.
[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] The definition of **INCLUDE_PROXY_CLIENT** and **INCLUDE_SM_SEQ_ADDR** in **configAll.h**. (Note that the address is equal to the master CPU's backplane address plus the client's processor number.)

[2] The BOOTP database

[3] **configAll.h** (defaults)

6

Shared-Memory Objects

Optional Component VxMP

List of Tables

List of Figures

List of Examples

6.1 Introduction

VxMP is an optional VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running on separate CPUs. For information on how to install VxMP, see the *Wind River Products Installation Guide.*

Shared-memory objects are a class of system objects that can be accessed by tasks running on different processors. They are called *shared-memory* objects because the object's data structures must reside in memory accessible by all processors. Shared-memory objects are an extension of local VxWorks objects. *Local objects* are only available to tasks on a single processor. VxMP supplies three kinds of shared-memory objects:

- shared semaphores (binary and counting)
- shared message queues
- shared-memory partitions (system- and user-created partitions)

Shared-memory objects provide the following advantages:

- A transparent interface that allows shared-memory objects to be manipulated with the same routines that are used for manipulating local objects.
- High-speed inter-processor communication—no unnecessary packet passing is required.
- The shared memory can reside either in dual-ported RAM or on a separate memory board.

The components of VxMP consist of the following: a name database (**smNameLib**), shared semaphores (**semSmLib**), shared message queues (**msgQSmLib**), and a shared-memory allocator (**smMemLib**).

This chapter presents a detailed description of each shared-memory object and internal considerations. It then describes configuration and troubleshooting.

6.2 Using Shared-Memory Objects

VxMP provides a transparent interface that makes it easy to execute code using shared-memory objects on both a multiprocessor system and a single-processor system. After an object is created, tasks can operate on shared objects with the same routines used to operate on their corresponding local objects. For example, shared semaphores, shared message queues, and shared-memory partitions have the same syntax and interface as their local counterparts. Routines such as ***semGive( )***, ***semTake( )***, ***msgQSend( )***, ***msgQReceive( )***, ***memPartAlloc( )***, and ***memPartFree( )*** operate on both local and shared objects. Only the create routines are different. This allows an application to run in either a single-processor or a multiprocessor environment with only minor changes to system configuration, initialization, and object creation.

All shared-memory objects can be used on a single-processor system. This is useful for testing an application before porting it to a multiprocessor configuration. However, for objects that are used only locally, local objects always provide the best performance.

After the shared-memory facilities are initialized (see *6.4 Configuration*, p.396 for initialization differences), all processors are treated alike. Tasks on any CPU can create and use shared-memory objects. No processor has priority over another from a shared-memory object's point of view.[1]

Systems making use of shared memory can include a combination of supported architectures. This enables applications to take advantage of different processor types and still have them communicate. However, on systems where the processors have different byte ordering, you must call the macros **ntohl** and **htonl** to byte-swap the application's shared data (see *Network Byte Order*, p.250 in this manual).

When an object is created, an *object ID* is returned to identify it. For tasks on different CPUs to access shared-memory objects, they must be able to obtain this ID. An object's ID is the same regardless of the CPU. This allows IDs to be passed using shared message queues, data structures in shared memory, or the name database.

Throughout the remainder of this chapter, system objects under discussion refer to shared objects unless otherwise indicated.

1. Do not confuse this type of priority with the CPU priorities associated with VMEbus access.

6.2.1 Name Database

The *name database* allows the association of any value to any name, such as a shared-memory object's ID with a unique name. It can communicate or *advertise* a shared-memory block's address and object type. The name database provides name-to-value and value-to-name translation, allowing objects in the database to be accessed either by name or by value. While other methods exist for advertising an object's ID, the name database is a convenient method for doing this.

Typically the task that creates an object also advertises the object's ID by means of the name database. By adding the new object to the database, the task associates the object's ID with a name. Tasks on other processors can look up the name in the database to get the object's ID. After the task has the ID, it can use it to access the object.

For example, task **t1** on CPU 1 creates an object. The object ID is returned by the creation routine and entered in the name database with the name **myObj**. For task **t2** on CPU 0 to operate on this object, it first finds the ID by looking up the name **myObj** in the name database.

This same technique can be used to advertise a shared-memory address. For example, task **t1** on CPU 0 allocates a chunk of memory and adds the address to the database with the name **mySharedMem**. Task **t2** on CPU 1 can find the address of this shared memory by looking up the address in the name database using **mySharedMem**.

Tasks on different processors can use an agreed-upon name to get a newly created object's value. See Table 6-1 for a list of name service routines. Note that retrieving an ID from the name database need occur only one time for each task, and usually occurs during application initialization.

Table 6-1 **Name Service Routines**

Routine	Functionality
smNameAdd( )	Add a name to the name database.
smNameRemove( )	Remove a name from the name database.
smNameFind( )	Find a shared symbol by name.
smNameFindByValue( )	Find a shared symbol by value.
smNameShow( )	Display the name database to the standard output device if **INCLUDE_SHOW_ROUTINES** is defined.

The name database service routines automatically convert to or from network-byte order; do not call ***htonl( )*** or ***ntohl( )*** explicitly for values from the name database.

The object types listed in Table 6-2 are defined in **smNameLib.h**.

Table 6-2 **Shared-Memory Object Types**

Constant	Hex Value
T_SM_SEM_B	0
T_SM_SEM_C	1
T_SM_MSG_Q	2
T_SM_PART_ID	3
T_SM_BLOCK	4

The following example shows the name database as displayed by ***smNameShow( )*** if **INCLUDE_SHOW_ROUTINES** is defined. The parameter to ***smNameShow( )*** specifies the level of information displayed; in this case, 1 indicates that all information is shown. For additional information on ***smNameShow( )***, see its reference entry.

```
-> smNameShow 1
value = 0 = 0x0
```

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

```
Name in Database Max : 100 Current : 5 Free : 95
Name                    Value           Type
----------------- ------------- -------------
myMemory              0x3835a0       SM_BLOCK
myMemPart             0x3659f9       SM_PART_ID
myBuff                0x383564       SM_BLOCK
mySmSemaphore         0x36431d       SM_SEM_B
myMsgQ                0x365899       SM_MSG_Q
```

6.2.2 Shared Semaphores

Like local semaphores, *shared semaphores* provide synchronization by means of atomic updates of semaphore state information. See 2. *Basic OS* in this manual and the reference entry for **semLib** for a complete discussion of semaphores. Shared semaphores can be given and taken by tasks executing on any CPU with access to the shared memory. They can be used for either synchronization of tasks running on different CPUs or mutual exclusion for shared resources.

To use a shared semaphore, a task creates the semaphore and advertises its ID. This can be done by adding it to the name database. A task on any CPU in the system can use the semaphore by first getting the semaphore ID (for example, from the name database). When it has the ID, it can then take or give the semaphore.

In the case of employing shared semaphores for mutual exclusion, typically there is a system resource that is shared between tasks on different CPUs and the semaphore is used to prevent concurrent access. Any time a task requires exclusive access to the resource, it takes the semaphore. When the task is finished with the resource, it gives the semaphore.

For example, there are two tasks, **t1** on CPU 0 and **t2** on CPU 1. Task **t1** creates the semaphore and advertises the semaphore's ID by adding it to the database and assigning the name **myMutexSem**. Task **t2** looks up the name **myMutexSem** in the database to get the semaphore's ID. Whenever a task wants to access the resource, it first takes the semaphore by using the semaphore ID. When a task is done using the resource, it gives the semaphore.

In the case of employing shared semaphores for synchronization, assume a task on one CPU must notify a task on another CPU that some event has occurred. The task being synchronized pends on the semaphore waiting for the event to occur. When the event occurs, the task doing the synchronizing gives the semaphore.

For example, there are two tasks, **t1** on CPU 0 and **t2** on CPU 1. Both **t1** and **t2** are monitoring robotic arms. The robotic arm that is controlled by **t1** is passing a physical object to the robotic arm controlled by **t2**. Task **t2** moves the arm into position but must then wait until **t1** indicates that it is ready for **t2** to take the object. Task **t1** creates the shared semaphore and advertises the semaphore's ID by adding it to the database and assigning the name **objReadySem**. Task **t2** looks up the name **objReadySem** in the database to get the semaphore's ID. It then takes the semaphore by using the semaphore ID. If the semaphore is unavailable, **t2** pends, waiting for **t1** to indicate that the object is ready for **t2**. When **t1** is ready to transfer control of the object to **t2**, it gives the semaphore, readying **t2** on CPU1.

There are two types of shared semaphores, binary and counting. Shared semaphores have their own create routines and return a **SEM_ID**. Table 6-3 lists the create routines. All other semaphore routines, except ***semDelete( )***, operate transparently on the created shared semaphore.

The use of shared semaphores and local semaphores differs in several ways:

- The shared semaphore queuing order specified when the semaphore is created must be FIFO. Figure 6-1 shows two tasks executing on different CPUs, both trying to take the same semaphore. Task 1 executes first, and is put at the front of the queue because the semaphore is unavailable (empty). Task 2 (executing

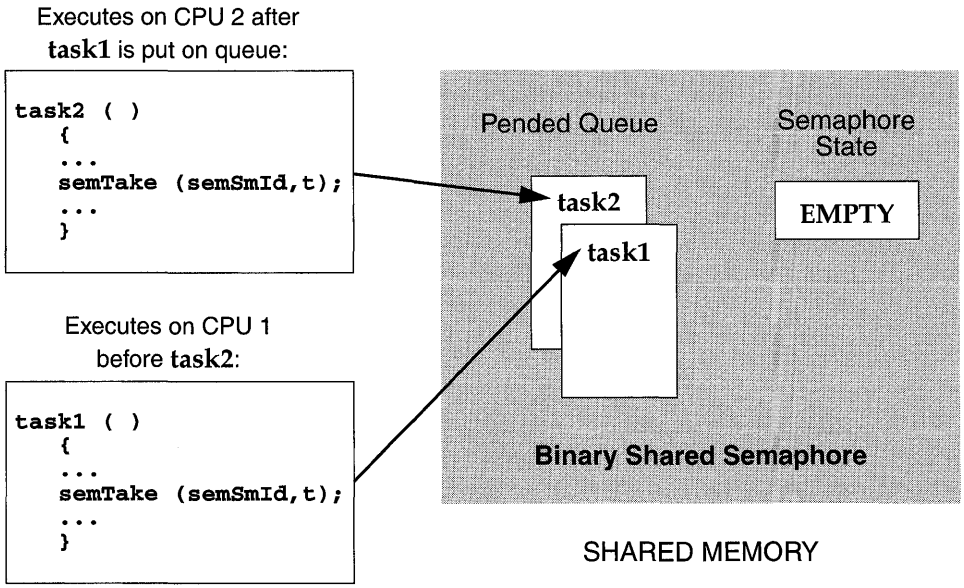
Table 6-3 **Shared Semaphore Create Routines**

Create Routine	Description
semBSmCreate( )	Create a shared binary semaphore.
semCSmCreate( )	Create a shared counting semaphore.

on a different CPU) tries to take the semaphore after task 1's attempt and is put on the queue behind task 1.

- Shared semaphores *cannot* be given from interrupt level.
- Shared semaphores cannot be deleted. Attempts to delete a shared semaphore return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Figure 6-1 **Shared Semaphore Queues**



Use *semInfo( )* to get the shared task control block of tasks pended on a shared semaphore. Use *semShow( )*, if **INCLUDE_SHOW_ROUTINES** is defined, to display the status of the shared semaphore and a list of pended tasks. The following example displays detailed information on the shared semaphore **mySmSemaphoreId** as indicated by the second argument (0 = summary, 1 = details):

```
-> semShow mySmSemaphoreId, 1
value = 0 = 0x0
```

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

```
Semaphore Id     : 0x36431d
Semaphore Type   : SHARED BINARY
Task Queuing     : FIFO
Pended Tasks     : 2
State            : EMPTY
TID             CPU Number      Shared TCB
------------ ------------ --------------
0xd0618             1           0x364204
0x3be924            0           0x36421c
```


Example 6-1 **Shared Semaphores**

The following code example depicts two tasks executing on different CPUs and using shared semaphores. The routine ***semTask1( )*** creates the shared semaphore, initializing the state to full. It adds the semaphore to the name database (to enable the task on the other CPU to access it), takes the semaphore, does some processing, and gives the semaphore. The routine ***semTask2( )*** gets the semaphore ID from the database, takes the semaphore, does some processing, and gives the semaphore.

```
/* semExample.h - shared semaphore example header file */

#define SEM_NAME "mySmSemaphore"

/* semTask1.c - shared semaphore example */

/* This code is executed by a task on CPU #1 */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "taskLib.h"
#include "semExample.h"

/*******************************************************************************
*
* semTask1 - shared semaphore user
*/

STATUS semTask1 (void)
    {
    SEM_ID semSmId;
```

```
    /* create shared semaphore */

    if ((semSmId = semBSmCreate (SEM_Q_FIFO, SEM_FULL)) == NULL)
        return (ERROR);

    /* add object to name database */

    if (smNameAdd (SEM_NAME, semSmId, T_SM_SEM_B) == ERROR)
        return (ERROR);

    /* grab shared semaphore and hold it for awhile */

    semTake (semSmId, WAIT_FOREVER);

    /* normally do something useful */

    printf ("Task1 has the shared semaphore\n");
    taskDelay (sysClkRateGet () * 5);
    printf ("Task1 is releasing the shared semaphore\n");

    /* release shared semaphore */

    semGive (semSmId);

    return (OK);
    }
```

```
/* semTask2.c - shared semaphore example */

/* This code is executed by a task on CPU #2. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "semExample.h"

/*******************************************************************************
*
* semTask2 - shared semaphore user
*/

STATUS semTask2 (void)
    {
    SEM_ID semSmId;
    int    objType;

    /* find object in name database */

    if (smNameFind (SEM_NAME, (void **) &semSmId, &objType, WAIT_FOREVER)
        == ERROR)
        return (ERROR);
```

```
/* take the shared semaphore */

printf ("semTask2 is now going to take the shared semaphore\n");
semTake (semSmId, WAIT_FOREVER);

/* normally do something useful */

printf ("Task2 got the shared semaphore!!\n");

/* release shared semaphore */

semGive (semSmId);

printf ("Task2 has released the shared semaphore\n");

return (OK);
}
```


6.2.3 Shared Message Queues

Shared message queues are FIFO queues used by tasks to send and receive variable-length messages on any of the CPUs that have access to the shared memory. They can be used either to synchronize tasks or to exchange data between tasks running on different CPUs. See *2. Basic OS* in this manual and the reference entry for **msgQLib** for a complete discussion of message queues.

To use a shared message queue, a task creates the message queue and advertises its ID. A task that wants to send or receive a message with this message queue first gets the message queue's ID. It then uses this ID to access the message queue.

For example, consider a typical server/client scenario where a server task **t1** (on CPU 1) reads requests from one message queue and replies to these requests with a different message queue. Task **t1** creates the request queue and advertises its ID by adding it to the name database assigning the name **requestQue**. If task **t2** (on CPU 0) wants to send a request to **t1**, it first gets the message queue ID by looking up the name **requestQue** in the name database. Before sending its first request, task **t2** creates a reply message queue. Instead of adding its ID to the database, it advertises the ID by sending it as part of the request message. When **t1** receives the request from the client, it finds in the message the ID of the queue to use when replying to that client. Task **t1** then sends the reply to the client by using this ID.

To pass messages between tasks on different CPUs, first create the message queue by calling ***msgQSmCreate*( )**. This routine returns a **MSG_Q_ID**. This ID is used for sending and receiving messages on the shared message queue.

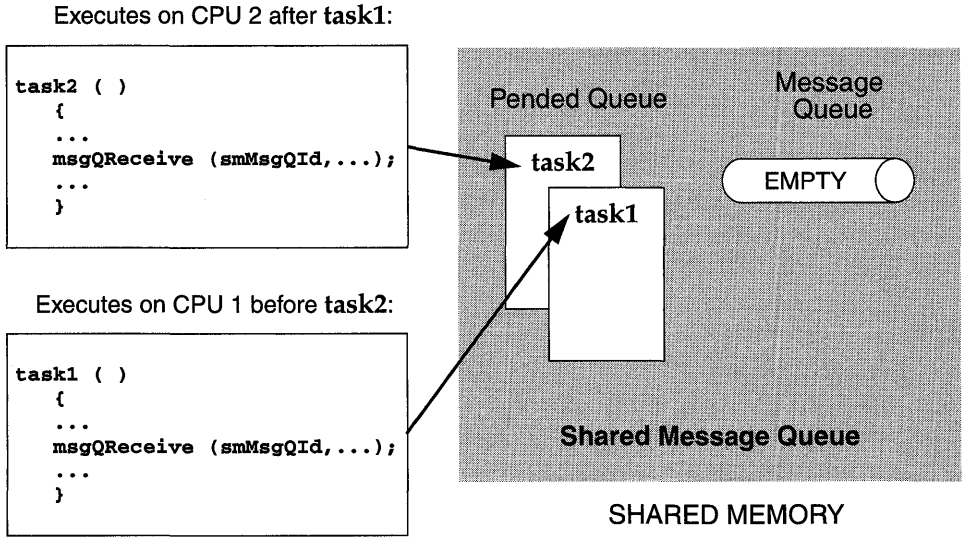
Like their local counterparts, shared message queues can send both urgent or normal priority messages.

The use of shared message queues and local message queues differs in several ways:

- The shared message queue task queueing order specified when a message queue is created must be FIFO. Figure 6-2 shows two tasks executing on different CPUs, both trying to receive a message from the same shared message queue. Task 1 executes first, and is put at the front of the queue because there are no messages in the message queue. Task 2 (executing on a different CPU) tries to receive a message from the message queue after task 1's attempt and is put on the queue behind task 1.
- Messages *cannot* be sent on a shared message queue at interrupt level. (This is true even in **NO_WAIT** mode.)
- Shared message queues cannot be deleted. Attempts to delete a shared message queue return **ERROR** and sets **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

To achieve optimum performance with shared message queues, align send and receive buffers on 4-byte boundaries.

Figure 6-2 **Shared Message Queues**



To display the status of the shared message queue as well as a list of tasks pended on the queue, define **INCLUDE_SHOW_ROUTINES** and call ***msgQShow( )***. The following example displays detailed information on the shared message queue 0x7f8c21 as indicated by the second argument (0 = summary display, 1 = detailed display).

```
-> msgQShow 0x7f8c21, 1
value = 0 = 0x0
```

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

```
Message Queue Id   : 0x7f8c21
Task Queuing       : FIFO
Message Byte Len   : 128
Messages Max       : 10
Messages Queued    : 0
Receivers Blocked  : 1
Send timeouts      : 0
Receive timeouts   : 0
Receivers blocked  :
TID           CPU Number           Shared TCB
---------- -------------------- --------------
0xd0618              1              0x1364204
```

Example 6-2 **Shared Message Queues**

In the following code example, two tasks executing on different CPUs use shared message queues to pass data to each other. The server task creates the request message queue, adds it to the name database, and reads a message from the queue. The client task gets the **smRequestQId** from the name database, creates a reply message queue, bundles the ID of the reply queue as part of the message, and sends the message to the server. The server gets the ID of the reply queue and uses it to send a message back to the client. This technique requires the use of the network byte-order conversion macros ***htonl( )*** and ***ntohl( )***, because the numeric queue ID is passed over the network in a data field.

```
/* msgExample.h - shared message queue example header file */

#define MAX_MSG      (10)
#define MAX_MSG_LEN (100)
#define REQUEST_Q    "requestQue"

typedef struct message
    {
    MSG_Q_ID replyQId;
    char     clientRequest[MAX_MSG_LEN];
    } REQUEST_MSG;
```

```
/* server.c - shared message queue example server */

/* This file contains the code for the message queue server task. */

#include "vxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "stdio.h"
#include "smNameLib.h"
#include "msgExample.h"
#include "netinet/in.h"

#define REPLY_TEXT "Server received your request"

/*******************************************************************************
*
* serverTask - receive and process a request from a shared message queue
*/

STATUS serverTask (void)
    {
    MSG_Q_ID    smRequestQId;  /* request shared message queue */
    REQUEST_MSG request;       /* request text */

    /* create a shared message queue to handle requests */

    if ((smRequestQId = msgQSmCreate (MAX_MSG, sizeof (REQUEST_MSG),
        MSG_Q_FIFO)) == NULL)
        return (ERROR);

    /* add newly created request message queue to name database */

    if (smNameAdd (REQUEST_Q, smRequestQId, T_SM_MSG_Q) == ERROR)
        return (ERROR);

    /* read messages from request queue */

    FOREVER
        {
        if (msgQReceive (smRequestQId, (char *) &request, sizeof (REQUEST_MSG),
            WAIT_FOREVER) == ERROR)
            return (ERROR);

        /* process request - in this case simply print it */

        printf ("Server received the following message:\n%s\n",
            request.clientRequest);

        /* send a reply using ID specified in client's request message */

        if (msgQSend ((MSG_Q_ID) ntohl ((int) request.replyQId),
            REPLY_TEXT, sizeof (REPLY_TEXT),
            WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
```

```
            return (ERROR);
        }
    }
```

```
/* client.c - shared message queue example client */

/* This file contains the code for the message queue client task. */

#include "vxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "msgExample.h"
#include "netinet/in.h"

/*******************************************************************************
*
* clientTask - sends request to server and reads reply
*/

STATUS clientTask
    (
    char * pRequestToServer  /* request to send to the server */
                             /* limited to 100 chars */
    )

    {
    MSG_Q_ID    smRequestQId; /* request message queue */
    MSG_Q_ID  smReplyQId;    /* reply message queue */
    REQUEST_MSG request;       /* request text */
    int         objType;       /* dummy variable for smNameFind */
    char        serverReply[MAX_MSG_LEN]; /*buffer for server's reply */

    /* get request queue ID using its name */

    if (smNameFind (REQUEST_Q, (void **) &smRequestQId, &objType,
        WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* create reply queue, build request and send it to server */

    if ((smReplyQId = msgQSmCreate (MAX_MSG, MAX_MSG_LEN,
        MSG_Q_FIFO)) == NULL)
        return (ERROR);

    request.replyQId = (MSG_Q_ID) htonl ((int) smReplyQId);

    strcpy (request.clientRequest, pRequestToServer);

    if (msgQSend (smRequestQId, (char *) &request, sizeof (REQUEST_MSG),
        WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
```

```
/* read reply and print it */

if (msgQReceive (request.replyQId, serverReply, MAX_MSG_LEN,
    WAIT_FOREVER) == ERROR)
    return (ERROR);

printf ("Client received the following message:\n%s\n", serverReply);

return (OK);
}
```

6.2.4 Shared-Memory Allocator

The *shared-memory allocator* allows tasks on different CPUs to allocate and release variable size chunks of memory that are accessible from all CPUs with access to the shared-memory system. Two sets of routines are provided: low-level routines for manipulating user-created shared-memory partitions, and high-level routines for manipulating a shared-memory partition dedicated to the shared-memory system pool. (This organization is similar to that used by the local-memory manager, **memPartLib**.)

Shared-memory blocks can be allocated from different partitions. Both a shared-memory system partition and user-created partitions are available. User-created partitions can be created and used for allocating data blocks of a particular size. Memory fragmentation is avoided when fixed-sized blocks are allocated from user-created partitions dedicated to a particular block size.

Shared-Memory System Partition

To use the shared-memory system partition, a task allocates a shared-memory block and advertises its address. One way of advertising the ID is to add the address to the name database. The routine used to allocate a block from the shared-memory system partition returns a local address. Before the address is advertised to tasks on other CPUs, this local address must be converted to a global address. Any task that must use the shared memory must first get the address of the memory block and convert the global address to a local address. When the task has the address, it can use the memory.

However, to address issues of mutual exclusion, typically a shared semaphore is used to protect the data in the shared memory. Thus in a more common scenario, the task that creates the shared memory (and adds it to the database) also creates a shared semaphore. The shared semaphore ID is typically advertised by storing it

in a field in the shared data structure residing in the shared-memory block. The first time a task must access the shared data structure, it looks up the address of the memory in the database and gets the semaphore ID from a field in the shared data structure. Whenever a task must access the shared data, it must first take the semaphore. Whenever a task is finished with the shared data, it must give the semaphore.

For example, assume two tasks executing on two different CPUs must share data. Task **t1** executing on CPU 1 allocates a memory block from the shared-memory system partition and converts the local address to a global address. It then adds the global address of the shared data to the name database with the name **mySharedData**. Task **t1** also creates a shared semaphore and stores the ID in the first field of the data structure residing in the shared memory. Task **t2** executing on CPU 2 looks up the name **mySharedData** in the name database to get the address of the shared memory. It then converts this address to a local address. Before accessing the data in the shared memory, **t2** gets the shared semaphore ID from the first field of the data structure residing in the shared-memory block. It then takes the semaphore before using the data and gives the semaphore when it is done using the data.

User-Created Partitions

To make use of user-created shared-memory partitions, a task creates a shared-memory partition and adds it to the name database. Before a task can use the shared-memory partition, it must first look in the name database to get the partition ID. When the task has the partition ID, it can access the memory in the shared-memory partition.

For example, task **t1** creates a shared-memory partition and adds it to the name database using the name **myMemPartition**. Task **t2** executing on another CPU wants to allocate memory from the new partition. Task **t2** first looks up **myMemPartition** in the name database to get the partition ID. It can then allocate memory from it, using the ID.

Using the Shared-Memory System Partition

The shared-memory system partition is analogous to the system partition for local memory. Table 6-4 lists routines for manipulating the shared-memory system partition.

Table 6-4 **Shared-Memory System Partition Routines**

Routine	Functionality
smMemMalloc( )	Allocate a block of shared system memory.
smMemCalloc( )	Allocate a block of shared system memory for an array.
smMemRealloc( )	Resize a block of shared system memory.
smMemFree( )	Free a block of shared system memory.
smMemShow( )	Display usage statistics of the shared-memory system partition on the standard output device if **INCLUDE_SHOW_ROUTINES** is defined.
smMemOptionsSet( )	Set the debugging options for the shared-memory system partition.
smMemAddToPool( )	Add memory to the shared-memory system pool.
smMemFindMax( )	Find the size of the largest free block in the shared-memory system partition.

Routines that return a pointer to allocated memory return a local address (that is, an address suitable for use from the local CPU). To share this memory across processors, this address must be converted to a global address before it is advertised to tasks on other CPUs. Before a task on another CPU uses the memory, it must convert the global address to a local address. Macros and routines are provided to convert between local addresses and global addresses; see the header file **smObjLib.h** and the reference entry for **smObjLib**.

Example 6-3 **Shared-Memory System Partition**

The following code example uses memory from the shared-memory system partition to share data between tasks on different CPUs. The first member of the data structure is a shared semaphore that is used for mutual exclusion. The send task creates and initializes the structure, then the receive task accesses the data and displays it.

```
/* buffProtocol.h - simple buffer exchange protocol header file */

#define BUFFER_SIZE   200           /* shared data buffer size */
#define BUFF_NAME     "myMemory"    /* name of data buffer in database */

typedef struct shared_buff
```

```
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;
```

```
/* buffSend.c - simple buffer exchange protocol send side */

/* This file writes to the shared memory. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*******************************************************************************
*
* buffSend - write to shared semaphore protected buffer
*
*/

STATUS buffSend (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;

    /* grab shared system memory */

    pSharedBuff = (SHARED_BUFF *) smMemMalloc (sizeof (SHARED_BUFF));

    /*
     * Initialize shared buffer structure before adding to database. The
     * protection semaphore is initially unavailable and the receiver blocks.
     */

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        return (ERROR);
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /*
     * Convert address of shared buffer to a global address and add to
     * database.
     */

    if (smNameAdd (BUFF_NAME, (void *) smObjLocalToGlobal (pSharedBuff),
                   T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* put data into shared buffer */

    sprintf (pSharedBuff->buff,"Hello from sender\n");
```

```
    /* allow receiver to read data by giving protection semaphore */

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

```
/* buffReceive.c - simple buffer exchange protocol receive side */

/* This file reads the shared memory. */

#include "vxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*************************************************************************
*
* buffReceive - receive shared semaphore protected buffer
*/

STATUS buffReceive (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;
    int           objType;

    /* get shared buffer address from name database */

    if (smNameFind (BUFF_NAME, (void **) &pSharedBuff,
                    &objType, WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* convert global address of buff to its local value */

    pSharedBuff = (SHARED_BUFF *) smObjGlobalToLocal (pSharedBuff);

    /* convert shared semaphore ID to host (local) byte order */

    mySemSmId = (SEM_ID) ntohl ((int) pSharedBuff->semSmId);

    /* take shared semaphore before reading the data buffer */

    if (semTake (mySemSmId,WAIT_FOREVER) != OK)
        return (ERROR);

    /* read data buffer and print it */

    printf ("Receiver reading from shared memory: %s\n", pSharedBuff->buff);
```

```
    /* give back the data buffer semaphore */

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

Using User-Created Partitions

Shared-memory partitions have a separate create routine, *memPartSmCreate( )*, that returns a **MEM_PART_ID**. After a user-defined shared-memory partition is created, routines in **memPartLib** operate on it transparently. Note that the address of the shared-memory area passed to *memPartSmCreate( )* (or *memPartAddToPool( )*) must be the global address.

Example 6-4 **User-Created Partition**

This example is similar to Example 6-3, which uses the shared-memory system partition. This example creates a user-defined partition and stores the shared data in this new partition. A shared semaphore is used to protect the data.

```
/* memPartExample.h - shared memory partition example header file */

#define CHUNK_SIZE      (2400)
#define MEM_PART_NAME   "myMemPart"
#define PART_BUFF_NAME  "myBuff"
#define BUFFER_SIZE     (40)

typedef struct shared_buff
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;
```

```
/* memPartSend.c - shared memory partition example send side */

/* This file writes to the user-defined shared memory partition. */

#include "vxWorks.h"
#include "memLib.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "smMemLib.h"
```

```
#include "stdio.h"
#include "memPartExample.h"

/*************************************************************************
*
* memPartSend - send shared memory partition buffer
*/

STATUS memPartSend (void)
    {
    char *          pMem;
    PART_ID         smMemPartId;
    SEM_ID          mySemSmId;
    SHARED_BUFF *   pSharedBuff;

    /* allocate shared system memory to use for partition */

    pMem = smMemMalloc (CHUNK_SIZE);

    /* Create user defined partition using the previously allocated
     * block of memory.
     * WARNING: memPartSmCreate uses the global address of a memory
     * pool as first parameter.
     */

    if ((smMemPartId = memPartSmCreate (smObjLocalToGlobal (pMem), CHUNK_SIZE))
          == NULL)
        return (ERROR);

    /* allocate memory from partition */

    pSharedBuff = (SHARED_BUFF *) memPartAlloc ( smMemPartId,
                sizeof (SHARED_BUFF));
    if (pSharedBuff == 0)
        return (ERROR);

    /* initialize structure before adding to database */

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO, SEM_EMPTY)) == NULL)
        return (ERROR);
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /* enter shared partition ID in name database */

    if (smNameAdd (MEM_PART_NAME, (void *) smMemPartId, T_SM_PART_ID) == ERROR)
        return (ERROR);

    /* convert shared buffer address to a global address and add to database */

    if (smNameAdd (PART_BUFF_NAME, (void *) smObjLocalToGlobal(pSharedBuff),
                   T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* send data using shared buffer */

    sprintf (pSharedBuff->buff,"Hello from sender\n");
```

```
    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

```
/* memPartReceive.c - shared memory partition example receive side */

/* This file reads from the user-defined shared memory partition. */

#include "vxWorks.h"
#include "memLib.h"
#include "stdio.h"
#include "semLib.h"
#include "semSmLib.h"
#include "stdio.h"
#include "memPartExample.h"

/*********************************************************************
*
* memPartReceive - receive shared memory partition buffer
*
* execute on CPU 1 - use a shared semaphore to protect shared memory
*
*/

STATUS memPartReceive (void)
    {
    SHARED_BUFF * pBuff;
    SEM_ID        mySemSmId;
    int           objType;

    /* get shared buffer address from name database */

    if (smNameFind (PART_BUFF_NAME, (void **) &pBuff, &objType,
                    WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* convert global address of buffer to its local value */

    pBuff = (SHARED_BUFF *) smObjGlobalToLocal (pBuff);

    /* Grab shared semaphore before using the shared memory */

    mySemSmId = (SEM_ID) ntohl ((int) pBuff->semSmId);
    semTake (mySemSmId,WAIT_FOREVER);
    printf ("Receiver reading from shared memory: %s\n", pBuff->buff);
    semGive (mySemSmId);

    return (OK);
    }
```

Side Effects of Shared-Memory Partition Options

Like their local counterparts, shared-memory partitions (both system- and user-created) can have different options set for error handling; see the reference entries for *memPartOptionsSet( )* and *smMemOptionsSet( )*.

If the **MEM_BLOCK_CHECK** option is used in the following situation, the system can get into a state where the memory partition is no longer available. If a task attempts to free a bad block and a bus error occurs, the task is suspended. Because shared semaphores are used internally for mutual exclusion, the suspended task still has the semaphore, and no other task has access to the memory partition. By default, shared-memory partitions are created without the **MEM_BLOCK_CHECK** option.

6.3 Internal Considerations

6.3.1 System Requirements

The shared-memory region used by shared-memory objects must be visible to all CPUs in the system. Either dual-ported memory on the master CPU (CPU 0) or a separate memory board can be used. The shared-memory objects' anchor must be in the same address space as the shared-memory region. Note that the memory does *not* have to appear at the same address for all CPUs.

NOTE: Boards that make use of VxMP must support hardware test-and-set (indivisible read-modify-write cycle). PowerPC is an exception; see *F. PowerPC*.

All CPUs in the system must support indivisible read-modify-write cycle across the (VME) bus. The indivisible RMW is used by the spin-lock mechanism to gain exclusive access to internal shared data structures; see *6.3.2 Spin-lock Mechanism*, p.395 for details. Because all the boards must support a hardware test-and-set, the constant **SM_TAS_HARD** must be defined in **configAll.h** (this is the default).

CPUs must be notified of any event that affects them. The preferred method is for the CPU initiating the event to interrupt the affected CPU. The use of interrupts is dependent on the capabilities of the hardware. If interrupts cannot be used, a polling scheme can be employed, although this generally results in a significant performance penalty.

The maximum number of CPUs that can use shared-memory objects is 20 (CPUs numbered 0 through 19). The practical maximum is usually a smaller number that depends on the CPU, bus bandwidth, and application.

6.3.2 Spin-lock Mechanism

Internal shared-memory object data structures are protected against concurrent access by a *spin-lock mechanism*. The spin-lock mechanism is a loop where an attempt is made to gain exclusive access to a resource (in this case an internal data structure). An indivisible hardware read-modify-write cycle (hardware test-and-set) is used for this mutual exclusion. If the first attempt to take the lock fails, multiple attempts are made, each with a decreasing random delay between one attempt and the next. The average time it takes between the original attempt to take the lock and the first retry is 70 microseconds on an MC68030 at 20MHz. Comment: It has been suggested to create a table of microprocessor spin-lock times. A table is inappropriate for the reason expressed in the following sentence. (VPG5.3)Operating time for the spin-lock cycle varies greatly because it is affected by the processor cache, access time to shared memory, and bus traffic. If the lock is not obtained after the maximum number of tries specified by **SM_OBJ_MAX_TRIES** (defined in **configAll.h**), **errno** is set to **S_smObjLib_LOCK_TIMEOUT**. If this error occurs, set the maximum number of tries to a higher value. Note that any failure to take a spin-lock prevents proper functioning of shared-memory objects. In most cases, this is due to problems with the shared-memory configuration; see *6.5.2 Troubleshooting Techniques*, p.404.

6.3.3 Interrupt Latency

For the duration of the spin-lock, interrupts are disabled to avoid the possibility of a task being preempted while holding the spin-lock. As a result, the interrupt latency of each processor in the system is increased. However, the interrupt latency added by shared-memory objects is constant for a particular CPU.

6.3.4 Restrictions

Unlike local semaphores and message queues, shared-memory objects cannot be used at interrupt level. No routines that use shared-memory objects can be called from ISRs. An ISR is dedicated to handle time-critical processing associated with an external event; therefore, using shared-memory objects at interrupt time is not

appropriate. On a multiprocessor system, run event-related time-critical processing on the CPU where the time-related interrupt occurred.

Note that shared-memory objects are allocated from dedicated shared-memory pools, and cannot be deleted.

When using shared-memory objects, the maximum number of each object type must be specified in **configAll.h**; see *6.4.3 Initializing the Shared-Memory Objects Package*, p.398. If applications are creating more than the specified maximum number of objects, it is possible to run out of memory. If this happens, the shared object creation routine returns an error and **errno** is set to **S_memLib_NOT_ENOUGH_MEM**. To solve this problem, first increase the maximum number of shared-memory objects of corresponding type (in **configAll.h**); see Table 6-5 for a list of the applicable configuration constants. This decreases the size of the shared-memory system pool because the shared-memory pool uses the remainder of the shared memory. If this is undesirable, increase both the number of the corresponding shared-memory objects (in **configAll.h**) and the size of the overall shared-memory region, **SM_OBJ_MEM_SIZE** (in **config.h**). See *6.4 Configuration*, p.396 for a discussion of the constants used for configuration.

6.3.5 Cache Coherency

When dual-ported memory is used on some boards without MMU or bus snooping mechanisms, the data cache must be disabled for the shared-memory region on the master CPU. If you see the following error message, make sure that the constant **USER_D_CACHE_ENABLE** is **#undef**'ed in **config.h**:

```
usrSmObjInit - cache coherent buffer not available. Giving up.
```

6.4 Configuration

To include shared-memory objects in VxWorks, define **INCLUDE_SM_OBJ** in the configuration file **configAll.h**. Most of the configuration is already done automatically from ***usrSmObjInit( )*** in **usrConfig.c**. However, you may also need to modify some values in **configAll.h** and **config.h** to reflect your configuration; these are described in this section.

6.4.1 Shared-Memory Objects and Shared-Memory Network Driver

Shared-memory objects and the shared-memory network[2] use the same memory region, anchor address, and interrupt mechanism. Configuring the system to use shared-memory objects is similar to configuring the shared-memory network driver. For a more detailed description of configuring and using the shared-memory network, see *5.4 Shared-Memory Networks*, p.301 in this manual. If the default value for the shared-memory anchor address is modified, the anchor must be on a 256-byte boundary.

One of the most important aspects of configuring shared-memory objects is computing the address of the shared-memory anchor. The shared-memory anchor is a location accessible to all CPUs on the system, and is used by both VxMP and the shared-memory network driver. The anchor stores a pointer to the shared-memory header, a pointer to the shared-memory packet header (used by the shared-memory network driver), and a pointer to the shared-memory object header.

The address of the anchor is defined in **config.h** with the constant **SM_ANCHOR_ADRS**. If the processor is booted with the shared-memory network driver, the anchor address is the same value as the boot device (**sm=***anchorAddress*). The shared-memory object initialization code uses the value from the boot line instead of the constant. If the shared-memory network driver is not used, modify the definition of **SM_ANCHOR_ADRS** as appropriate to reflect your system.

Two types of interrupts are supported and defined by **SM_INT_TYPE**: mailbox interrupts and bus interrupts (see *5.4.2 Interprocessor Interrupts*, p.307 in this manual). Mailbox interrupts (**SM_INT_MAILBOX**) are the preferred method, and bus interrupts (**SM_INT_BUS**) are the second choice. If interrupts cannot be used, a polling scheme can be employed (**SM_INT_NONE**), but this is much less efficient.

When a CPU initializes its shared-memory objects, it defines the interrupt type as well as three interrupt arguments. These describe how the CPU is notified of events. These values can be obtained for any attached CPU by calling ***smCpuInfoGet*( )**.

The default interrupt method for a target is defined by **SM_INT_TYPE**, **SM_INT_ARG1**, **SM_INT_ARG2**, and **SM_INT_ARG3** in **config.h**.

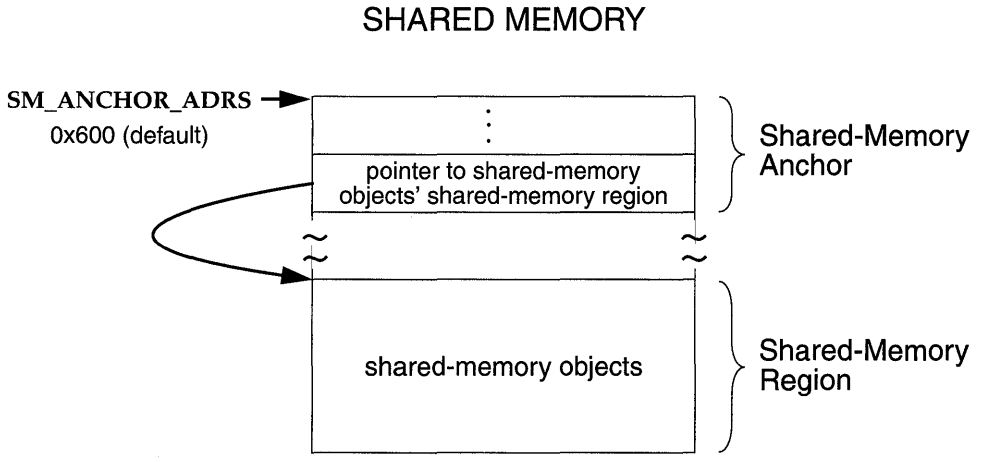
2. Also known as the *backplane network*.

6.4.2 Shared-Memory Region

Shared-memory objects rely on a shared-memory region that is visible to all processors. This region is used to store internal shared-memory object data structures and the shared-memory system partition.

The shared-memory region is usually in dual-ported RAM on the master, but it can also be located on a separate memory card. The shared-memory region address is defined when configuring the system as an offset from the shared-memory anchor address, **SM_ANCHOR_ADRS**, as shown in Figure 6-3.

Figure 6-3 **Shared-Memory Layout**



6.4.3 Initializing the Shared-Memory Objects Package

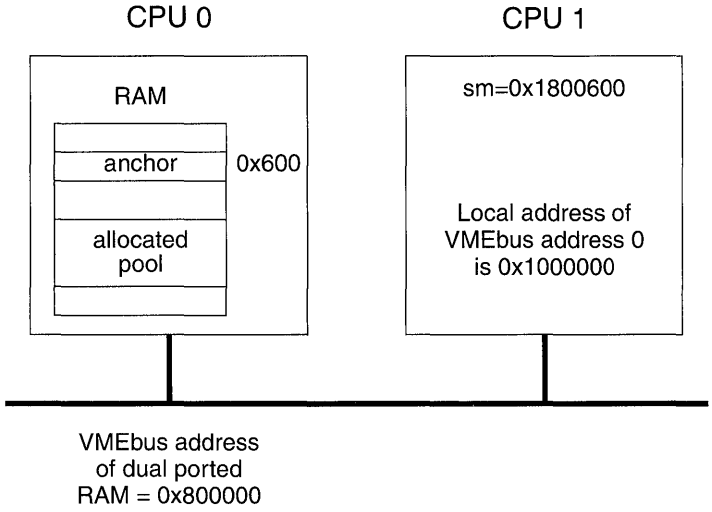
Shared-memory objects are initialized by default in the routine ***usrSmObjInit( )*** in **src/config/usrSmObj.c**. The configuration steps taken for the master CPU differ slightly from those taken for the slaves.

The address for the shared-memory pool must be defined. If the memory on the master CPU is used, it can be malloc'ed at run-time by setting **SM_OBJ_MEM_ADRS** in **config.h** to **NONE**. If the memory is off-board, the value must be calculated (see Figure 6-5).

The example configuration in Figure 6-4 uses the shared memory in the master CPU's dual-ported RAM. In **config.h** for the master, **SM_OFF_BOARD** is FALSE

and **SM_ANCHOR_ADRS** is 0x600. **SM_OBJ_MEM_ADRS** is set to **NONE**, because on-board memory is used; **SM_OBJ_MEM_SIZE** is set to 0x20000. For the slave, the board maps the base of the VME bus to the address 0x1000000. **SM_OFF_BOARD** is TRUE and the anchor address is 0x1800600. This is calculated by taking the VMEbus address (0x800000) and adding it to the anchor address (0x600). Many boards require further address translation, depending on where the board maps VME memory. In this example, the anchor address for the slave is 0x1800600, because the board maps the base of the VME bus to the address 0x1000000.

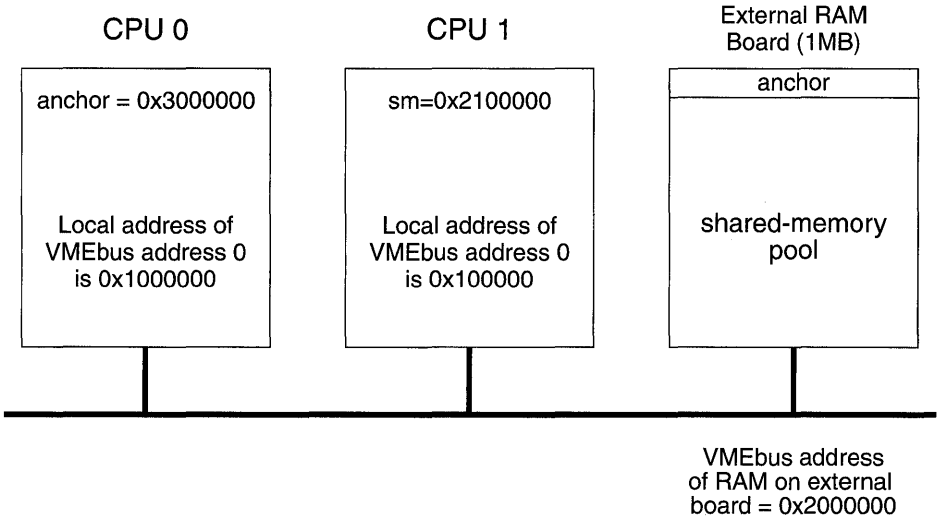
Figure 6-4 **Example Configuration: Dual-Ported Memory**



In the example configuration in Figure 6-5, the shared memory is on a separate memory board. In **config.h** for the master, **SM_OFF_BOARD** is TRUE, **SM_ANCHOR_ADRS** is 0x3000000, **SM_OBJ_MEM_ADRS** is set to **SM_ANCHOR_ADRS**, and **SM_OBJ_MEM_SIZE** is set to 0x100000. For the slave board, **SM_OFF_BOARD** is TRUE and the anchor address is 0x2100000. This is calculated by taking the VMEbus address of the memory board (0x2000000) and adding it to the local VMEbus address (0x100000).

Some additional configuration are sometimes required to make the shared memory non-cacheable, because the shared-memory pool is accessed by all processors on the backplane. By default, boards with an MMU have the MMU turned on. With the MMU on, memory that is off-board must be made non-cacheable. This is done using the data structure **sysPhysMemDesc** in

Figure 6-5 **Example Configuration: an External Memory Board**



sysLib.c. This data structure must contain a virtual-to-physical mapping for the VME address space used for the shared-memory pool, and mark the memory as non-cacheable. (Most BSPs include this mapping by default.) See *7.3 Virtual Memory Configuration*, p.408 in this manual for additional information.

⚠ **NOTE:** For the MC68030, if the MMU is off, data caching must be turned off globally; see the reference entry for **cacheLib**.

When shared-memory objects are initialized, the memory size as well as the maximum number of each object type must be specified. The master processor specifies the size of memory using the constant **SM_OBJ_MEM_SIZE** in **config.h**. Symbolic constants in **configAll.h** are used to set the maximum number of different objects. See Table 6-5 for a list of these constants.

If the size of the objects created exceeds the shared-memory region, an error message is displayed on CPU 0 during initialization. After shared memory is configured for the shared objects, the remainder of shared memory is used for the shared-memory system partition.

If **INCLUDE_SHOW_ROUTINES** is defined, the routine ***smObjShow*( )** displays the current number of used shared-memory objects and other statistics, as follows:

```
-> smObjShow
value = 0 = 0x0
```

Table 6-5 **Configuration Constants for Shared-Memory Objects**

Symbolic Constant	Default Value	Description
SM_OBJ_MAX_TASK	40	Maximum number of tasks using shared-memory objects.
SM_OBJ_MAX_SEM	30	Maximum number of shared semaphores (counting and binary).
SM_OBJ_MAX_NAME	100	Maximum number of names in the name database.
SM_OBJ_MAX_MSG_Q	10	Maximum number of shared message queues.
SM_OBJ_MAX_MEM_PART	4	Maximum number of user-created shared-memory partitions.

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

```
Shared Mem Anchor Local Addr : 0x600
Shared Mem Hdr Local Addr    : 0x363ed0
Attached CPU                 : 2
Max Tries to Take Lock       : 0
Shared Object Type      Current      Maximum      Available
------------------      -------      -------      ---------
Tasks                         1           40             39
Binary Semaphores             3           30             27
Counting Semaphores           0           30             27
Messages Queues               1           10              9
Memory Partitions             1            4              3
Names in Database             5          100             95
```

NOTE: If the master CPU is rebooted, it is necessary to reboot all the slaves. If a slave CPU is to be rebooted, it must not have tasks pended on a shared-memory object.

6.4.4 Configuration Example

The following example shows the configuration for a multiprocessor system with three CPUs. The master is CPU 0, and shared memory is configured from its dual-ported memory. This application has 20 tasks using shared-memory objects, and uses 12 message queues and 20 semaphores. The maximum size of the name database is the default value (100), and only one user-defined memory partition is required. The header file **configAll.h** must reflect this new configuration, as in the following excerpt:

```
#define INCLUDE_SM_OBJ
...
#define SM_OBJ_MAX_TASK         20
#define SM_OBJ_MAX_SEM          20
#define SM_OBJ_MAX_NAME         100
#define SM_OBJ_MAX_MSG_Q        12
#define SM_OBJ_MAX_MEM_PART     1
```

On CPU 0, the shared-memory pool is configured to be on-board. This memory is allocated from the processor's system memory. The following excerpt is taken from CPU 0's **config.h**:

```
#define SM_OFF_BOARD        FALSE
#if SM_OFF_BOARD
...
#else
#define SM_MEM_ADRS         NONE
#define SM_MEM_SIZE         0x10000
#define SM_OBJ_MEM_ADRS     NONE
#define SM_OBJ_MEM_SIZE     0x10000
#endif
```

On CPU 1 and CPU 2, the shared-memory pool is configured to be off-board. The following excerpt is taken from the slaves' **config.h**:

```
#define SM_OFF_BOARD        TRUE
#if SM_OFF_BOARD
#undef SM_ANCHOR_ADRS
#define SM_ANCHOR_ADRS      (char *) 0xfb800000
#define SM_MEM_ADRS         SM_ANCHOR_ADRS
#define SM_MEM_SIZE         0x80000
#define SM_OBJ_MEM_ADRS     (SM_MEM_ADRS + SM_MEM_SIZE)
#define SM_OBJ_MEM_SIZE     0x80000
#else
...
#endif
```

Note that for the slave CPUs, the value of **SM_OBJ_MEM_SIZE** is not used.

6.4.5 Initialization Steps

Initialization is performed by default in *usrSmObjInit( )*, in **src/config/usrSmObj.c**. On the master CPU, the initialization of shared-memory objects consists of the following:

1. Setting up the shared-memory objects header and its pointer in the shared-memory anchor, with *smObjSetup( )*.
2. Initializing shared-memory object parameters for this CPU, with *smObjInit( )*.
3. Attaching the CPU to the shared-memory object facility, with *smObjAttach( )*.

On slave CPUs, only steps 2 and 3 are required.

The routine ***smObjAttach( )*** checks the setup of shared-memory objects. It looks for the *shared-memory heartbeat* to verify that the facility is running. The shared-memory heartbeat is an unsigned integer that is incremented once per second by the master CPU. It indicates to the slaves that shared-memory objects are initialized, and can be used for debugging. The heartbeat is the first field in the shared-memory object header; see *6.5 Troubleshooting*, p.403.

6.5 Troubleshooting

Problems with shared-memory objects can be due to a number of causes. This section discusses the most common problems and a number of troubleshooting tools. Often, you can locate the problem by rechecking your hardware and software configurations.

6.5.1 Configuration Problems

Refer to the following list to confirm that your system is properly configured:

- Be sure to verify that the constant **INCLUDE_SM_OBJ** is defined in **configAll.h** for all processors, or in **config.h** for each processor using VxMP.
- Be sure the anchor address specified is the address seen by the CPU. This can be defined with the constant **SM_ANCHOR_ADRS** in **config.h** or at boot time (**sm=**) if the target is booted with the shared-memory network.
- If there is heavy bus traffic relating to shared-memory objects, bus errors can occur. Avoid this problem by changing the bus arbitration mode or by changing relative CPU priorities on the bus.
- If ***memAddToPool( )***, ***memPartSmCreate( )***, or ***smMemAddToPool( )*** fail, check that any address you are passing to these routines is in fact a global address.

6.5.2 Troubleshooting Techniques

Use the following techniques to troubleshoot any problems you encounter:

- The routine ***smObjTimeoutLogEnable( )*** enables or disables the printing of an error message indicating that the maximum number of attempts to take a spin-lock has been reached. By default, message printing is enabled.
- If **INCLUDE_SHOW_ROUTINES** is defined, the routine ***smObjShow( )*** displays the status of the shared-memory objects facility on the standard output device. It displays the maximum number of tries a task took to get a spin-lock on a particular CPU. A high value can indicate that an application might run into problems due to contention for shared-memory resources.
- The shared-memory heartbeat can be checked to verify that the master CPU has initialized shared-memory objects. The shared-memory heartbeat is in the first 4-byte-word of the shared-memory object header. The offset to the header is in the sixth 4-byte word in the shared-memory anchor. (See *The Shared-Memory Heartbeat*, p.304 in this manual.)

  Thus, if the shared-memory anchor were located at 0x800000:

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

  The offset to the shared-memory object header is 0x170. To view the shared-memory object header display 0x800170:

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

  In the preceding example, the value of the shared-memory heartbeat is 0x50. Display this location again to ensure that the heartbeat is alive; if its value has changed, shared-memory objects are initialized.
- The global variable **smIfVerbose**, when set to 1 (TRUE), causes shared-memory interface error messages to print to the console, along with additional details of shared-memory operations. This variable enables you to get run-time information from the device driver level that would be unavailable at the debugger level. The default setting for **smIfVerbose** is 0 (FALSE). That can be reset programmatically or from the shell.

7

Virtual Memory Interface

Basic Support and Optional Component VxVMI

List of Tables

List of Figures

List of Examples

7.1 Introduction

VxWorks provides two levels of virtual memory support. The basic level is bundled with VxWorks and provides caching on a per-page basis. The full level is unbundled, and requires the optional component, VxVMI. VxVMI provides write protection of text segments and the VxWorks exception vector table, and an architecture-independent interface to the CPU's memory management unit (MMU). For information on how to install VxVMI, see the *Wind River Products Installation Guide*.

This chapter contains the following sections:

- The first describes the basic level of support.
- The second describes configuration, and is applicable to both levels of support.
- The third and fourth parts apply only to the optional component, VxVMI:
  - The third is for general use, discussing the write protection implemented by VxVMI.
  - The fourth describes a set of routines for manipulating the MMU. VxVMI provides low-level routines for interfacing with the MMU in an architecture-independent manner, allowing you to implement your own virtual memory systems.

7.2 Basic Virtual Memory Support

For systems with an MMU, VxWorks allows you to perform DMA and interprocessor communication more efficiently by rendering related buffers noncacheable. This is necessary to ensure that data is not being buffered locally

when other processors or DMA devices are accessing the same memory location. Without the ability to make portions of memory noncacheable, caching must be turned off globally (resulting in performance degradation) or buffers must be flushed/invalidated manually.

Basic virtual memory support is included by defining **INCLUDE_MMU_BASIC** in **configAll.h**; see *7.3 Virtual Memory Configuration*, p.408. It is also possible to allocate noncacheable buffers using ***cacheDmaMalloc( )***; see the reference entry for **cacheLib**.

7.3 Virtual Memory Configuration

The following discussion of configuration applies to both bundled and unbundled virtual memory support.

In **configAll.h**, define the constants in Table 7-1 to reflect your system configuration.

Table 7-1 **MMU Configuration Constants**

Constant	Description
INCLUDE_MMU_BASIC	Basic MMU support without VxVMI option.
INCLUDE_MMU_FULL	Full MMU support with the VxVMI option.
INCLUDE_PROTECT_TEXT	Text segment protection (requires full MMU support).
INCLUDE_PROTECT_VEC_TABLE	Exception vector table protection (requires full MMU support).

The default page size (8KB) is defined by **VM_PAGE_SIZE** in **configAll.h**. For architectures that support different page sizes, redefine **VM_PAGE_SIZE** in **config.h**.

To make memory noncacheable, it must have a virtual-to-physical mapping. The data structure **PHYS_MEM_DESC** in **vmLib.h** defines the parameters used for mapping physical memory. Each board's memory map is defined in **sysLib.c** using **sysPhysMemDesc** (which is declared as an array of **PHYS_MEM_DESC**). In addition to defining the initial state of the memory pages, the **sysPhysMemDesc**

structure defines the virtual addresses used for mapping virtual-to-physical memory. For a discussion of page states, see *Page States*, p.412.

Modify the **sysPhysMemDesc** structure to reflect your system configuration. For example, you may need to add the addresses of interprocessor communication buffers not already included in the structure. Or, you may need to map and make noncacheable the VMEbus addresses of the shared-memory data structures. Most board support packages have a section of VME space defined in **sysPhysMemDesc**; however, this may not include all the space required by your system configuration.

I/O devices and memory not already included in the structure must also be mapped and made noncacheable. In general, off-board memory regions are specified as noncacheable; see *On-Board and Off-Board Options*, p.306.

NOTE: The regions of memory defined in **sysPhysMemDesc** must be page-aligned, and must span complete pages. In other words, the first three fields (virtual address, physical address, and length) of a **PHYS_MEM_DESC** structure must all be even multiples of **VM_PAGE_SIZE**. Specifying elements of **sysPhysMemDesc** that are not page-aligned leads to crashes during VxWorks initialization.

The following example configuration consists of multiple CPUs using the shared-memory network. A separate memory board is used for the shared-memory pool. Because this memory is not already mapped, it must be added to **sysPhysMemDesc** for all the boards on the network. The memory starts at 0x4000000 and must be made noncacheable, as shown in the following code excerpt:

```
/* shared memory */
{
(void *) 0x4000000,          /* virtual address */
(void *) 0x4000000,          /* physical address */
0x20000,               /* length */
/* initial state mask */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |VM_STATE_MASK_CACHEABLE,
/* initial state */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

For MC680*x*0 boards, the virtual address *must* be the same as the physical address. For other boards, the virtual and physical addresses are the same as a matter of convention.

7.4 General Use

This section describes VxVMI's general use and configuration for write-protecting text segments and the exception vector table.

VxVMI uses the MMU to prevent portions of memory from being overwritten. This is done by write-protecting pages of memory. Not all target hardware supports write protection; see the architecture appendices in this manual for further information. For most architectures, the page size is 8KB. An attempt to write to a memory location that is write-protected causes a bus error.

When VxWorks is loaded, all text segments are write-protected; see *7.3 Virtual Memory Configuration*, p.408. The text segments of additional object modules loaded using ***ld*( )** are automatically marked as read-only. When object modules are loaded, memory to be write-protected is allocated in page-size increments. No additional steps are required to write-protect application code.

During system initialization, VxWorks write-protects the exception vector table. The only way to modify the interrupt vector table is to use the routine ***intConnect*( )**, which write-enables the exception vector table for the duration of the call.

To include write-protection, define the following in **configAll.h**:

```
INCLUDE_MMU_FULL
INCLUDE_PROTECT_TEXT
INCLUDE_PROTECT_VEC_TABLE
```

7.5 Using the MMU Programmatically

This section describes the facilities provided for manipulating the MMU programmatically using low-level routines in **vmLib**. You can make data private to a task or code segment, make portions of memory noncacheable, or write-protect portions of memory. The fundamental structure used to implement virtual memory is the *virtual memory context* (VMC).

For a summary of the VxVMI routines, see the reference entry for **vmLib**.

7.5.1 Virtual Memory Contexts

A virtual memory context (**VM_CONTEXT**, defined in **vmLib**) is made up of a translation table and other information used for mapping a virtual address to a physical address. Multiple virtual memory contexts can be created and swapped in and out as desired.

Global Virtual Memory

Some system objects, such as text segments and semaphores, must be accessible to all tasks in the system regardless of which virtual memory context is made current. These objects are made accessible by means of *global virtual memory*. Global virtual memory is created by mapping all the physical memory in the system (the mapping is defined in **sysPhysMemDesc**) to the identical address in the virtual memory space. In the default system configuration, this initially gives a one-to-one relationship between physical memory and global virtual memory; for example, virtual address 0x5000 maps to physical address 0x5000. On some architectures, it is possible to use **sysPhysMemDesc** to set up virtual memory so that the mapping of virtual-to-physical addresses is not one-to-one; see *7.3 Virtual Memory Configuration*, p.408 for additional information.

Global virtual memory is accessible from all virtual memory contexts. Modifications made to the global mapping in one virtual memory context appear in all virtual memory contexts. Before virtual memory contexts are created, add all global memory with ***vmGlobalMap( )***. Global memory that is added after virtual memory contexts are created may not be available to existing contexts.

Initialization

Global virtual memory is initialized by ***vmGlobalMapInit( )*** in ***usrMmuInit( )***, which is called from ***usrRoot( )***. The routine ***usrMmuInit( )*** is in **src/config/usrMmuInit.c**, and creates global virtual memory using **sysPhysMemDesc**. It then creates a default virtual memory context and makes the default context current. Optionally, it also enables the MMU.

Page States

Each virtual memory page (typically 8KB) has a state associated with it. A page can be valid/invalid, writable/nonwritable, or cacheable/noncacheable. See Table 7-2 for the associated constants.

Table 7-2 **State Flags**

Constant	Description
VM_STATE_VALID	Valid translation
VM_STATE_VALID_NOT	Invalid translation
VM_STATE_WRITABLE	Writable memory
VM_STATE_WRITABLE_NOT	Read-only memory
VM_STATE_CACHEABLE	Cacheable memory
VM_STATE_CACHEABLE_NOT	Noncacheable memory

Validity — A valid state indicates the virtual-to-physical translation is true. When the translation tables are initialized, global virtual memory is marked as valid. All other virtual memory is initialized as invalid.

Writability — Pages can be made read-only by setting the state to nonwritable. This is used by VxWorks to write-protect all text segments.

Cacheability — The caching of memory pages can be prevented by setting the state flags to noncacheable. This is useful for memory that is shared between processors (including DMA devices).

Change the state of a page with the routine ***vmStateSet( )***. In addition to specifying the state flags, a state mask must describe which flags are being changed; see Table 7-3. Additional architecture-dependent states are specified in **vmLib.h**.

Table 7-3 **State Masks**

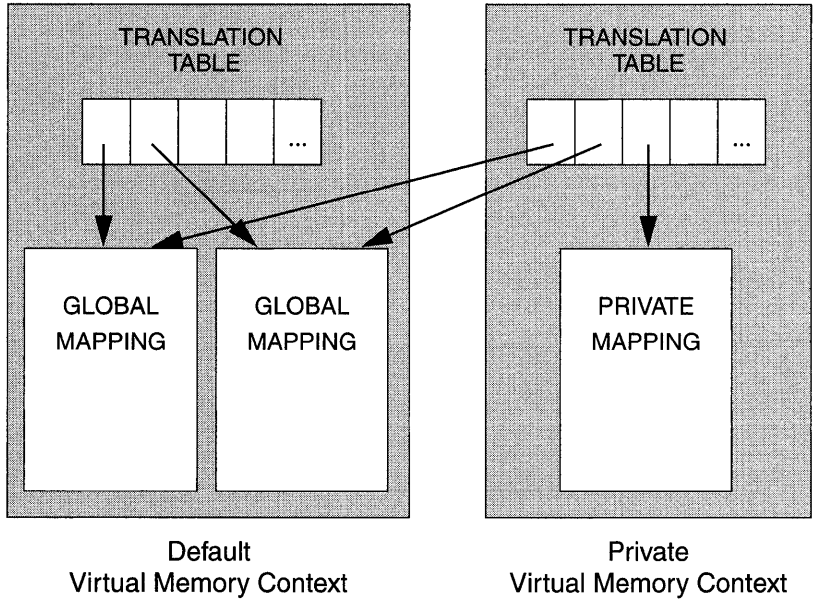
Constant	Description
VM_STATE_MASK_VALID	Modify valid flag
VM_STATE_MASK_WRITABLE	Modify write flag
VM_STATE_MASK_CACHEABLE	Modify cache flag

7.5.2 Private Virtual Memory

Private virtual memory can be created by creating a new virtual memory context. This is useful for protecting data by making it inaccessible to other tasks or by limiting access to specific routines. Virtual memory contexts are not automatically created for tasks, but can be created and swapped in and out in an application-specific manner.

At system initialization, a default context is created. All tasks use this default context. To create private virtual memory, a task must create a new virtual memory context using *vmContextCreate( )*, and make it current. All virtual memory contexts share the global mappings that are created at system initialization; see Figure 7-1. Only the valid virtual memory in the current virtual memory context (including global virtual memory) is accessible. Virtual memory defined in other virtual memory contexts is not accessible. To make another memory context current, use *vmCurrentSet( )*.

Figure 7-1 **Global Mappings of Virtual Memory**



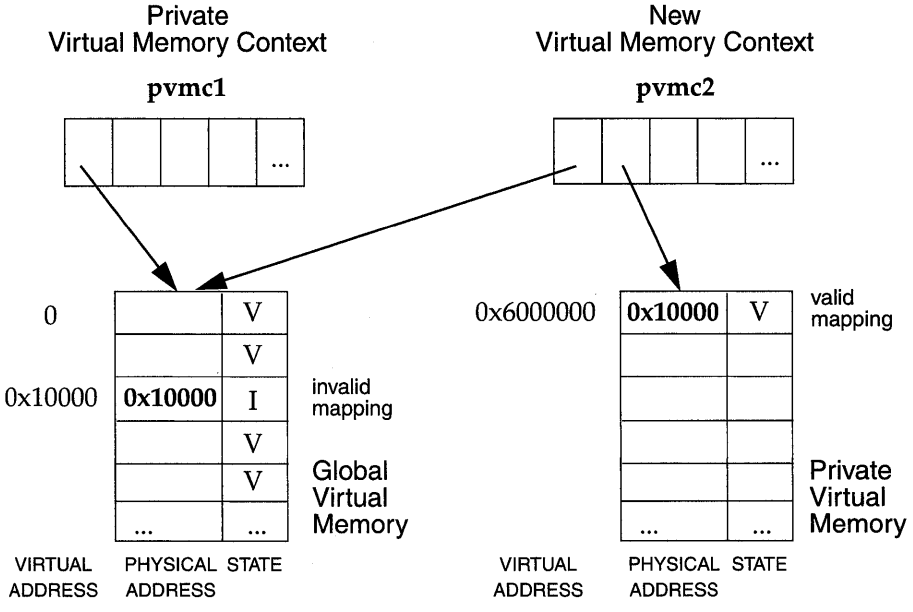
To create a new virtual-to-physical mapping, use *vmMap( )*; both the physical and virtual address must be determined in advance. The physical memory (which must be page aligned) can be obtained using *valloc( )*. The easiest way to

determine the virtual address is to use ***vmGlobalInfoGet( )*** to find a virtual page that is not a global mapping. With this scheme, if multiple mappings are required, a task must keep track of its own private virtual memory pages to guarantee it does not map the same non-global address twice.

When physical pages are mapped into new sections of the virtual space, the physical page is accessible from two different virtual addresses (a condition known as *aliasing*): the newly mapped virtual address and the virtual address equal to the physical address in the global virtual memory. This can cause problems for some architectures, because the cache may hold two different values for the same underlying memory location. To avoid this, invalidate the virtual page (using ***vmStateSet( )***) in the global virtual memory. This also ensures that the data is accessible only when the virtual memory context containing the new mapping is current.

Figure 7-2 depicts two private virtual memory contexts. The new context (**pvmc2**) maps virtual address 0x6000000 to physical address 0x10000. To prevent access to this address from outside of this virtual context (**pvmc1**), the corresponding physical address (0x10000) must be set to invalid. If access to the memory is made using address 0x10000, a bus error occurs because that address is now invalid.

Figure 7-2 **Mapping Private Virtual Memory**



Example 7-1 **Private Virtual Memory Contexts**

In the following code example, private virtual memory contexts are used for allocating memory from a task's private memory partition. The setup routine, ***contextSetup( )***, creates a private virtual memory context that is made current during a context switch. The virtual memory context is stored in the field **spare1** in the task's TCB. Switch hooks are used to save the old context and install the task's private context. Note that the use of switch hooks increases the context switch time. A user-defined memory partition is created using the private virtual memory context. The partition ID is stored in **spare2** in the tasks TCB. Any task wanting a private virtual memory context must call ***contextSetup( )***. A sample task to test the code is included.


```
/* contextExample.h - header file for vm contexts used by switch hooks */

#define NUM_PAGES (3)
```

```
/* context.c - use context switch hooks to make task private context current */

#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
#include "taskLib.h"
#include "taskHookLib.h"
#include "memLib.h"
#include "contextExample.h"

void privContextSwitch (WIND_TCB *pOldTask, WIND_TCB *pNewTask);

/*************************************************************************
*
* initContextSetup - install context switch hook
*
*/

STATUS initContextSetup ( )
    {
    /* Install switch hook */

    if (taskSwitchHookAdd ((FUNCPTR) privContextSwitch) == ERROR)
        return (ERROR);

    return (OK);
    }
```

```
/*****************************************************************************
*
* contextSetup - initialize context and create separate memory partition
*
* Call only once for each task that wants a private context.
*
* This could be made into a create-hook routine if every task on the
* system needs a private context. To use as a create hook, the code for
* installing the new virtual memory context should be replaced by simply
* saving the new context in spare1 of the task's TCB.
*/

STATUS contextSetup (void)
    {
    VM_CONTEXT_ID pNewContext;
    int pageSize;
    int pageBlkSize;
    char * pPhysAddr;
    char * pVirtAddr;
    UINT8 * globalPgBlkArray;
    int newMemSize;
    int index;
    WIND_TCB * pTcb;

    /* create context */

    pNewContext = vmContextCreate();

    /* get page and page block size */

    pageSize = vmPageSizeGet ();
    pageBlkSize = vmPageBlockSizeGet ();
    newMemSize = pageSize * NUM_PAGES;

    /* allocate physical memory that is page aligned */

    if ((pPhysAddr = (char *) valloc (newMemSize)) == NULL)
        return (ERROR);

    /* Select virtual address to map. For this example, since only one page
     * block is used per task, simply use the first address that is not a
     * global mapping. vmGlobalInfoGet( ) returns a boolean array where each
     * element corresponds to a block of virtual memory.
     */

    globalPgBlkArray = vmGlobalInfoGet();
    for (index = 0; globalPgBlkArray[index] == TRUE; index++)
        ;
    pVirtAddr = (char *) (index * pageBlkSize);

    /* map physical memory to new context */

    if (vmMap (pNewContext, pVirtAddr, pPhysAddr, newMemSize) == ERROR)
        {
        free (pPhysAddr);
        return (ERROR);
        }
```

```
    /*
     * Set state in global virtual memory to be invalid - any access to
     * this memory must be done through new context.
     */

    if (vmStateSet(pNewContext, pPhysAddr, newMemSize, VM_STATE_MASK_VALID,
                   VM_STATE_VALID_NOT) == ERROR)
        return (ERROR);

    /* get tasks TCB */

    pTcb = taskTcb (taskIdSelf());

    /* change virtual memory contexts */

    /*
     * Stash the current vm context in the spare TCB field -- the switch
     * hook will install this when this task gets swapped out.
     */

    pTcb->spare1 = (int) vmCurrentGet();

    /* install new tasks context */

    vmCurrentSet (pNewContext);

    /* create new memory partition and store id in task's TCB */

    if ((pTcb->spare2 = (int) memPartCreate (pVirtAddr,newMemSize)) == NULL)
        return (ERROR);

    return (OK);
    }

/*************************************************************************
*
* privContextSwitch - routine to be executed on a context switch
*
* If old task had private context, save it. If new task has private
* context, install it.
*/

void privContextSwitch
    (
    WIND_TCB *pOldTcb,
    WIND_TCB *pNewTcb
    )

    {
    VM_CONTEXT_ID pContext = NULL;

    /* If previous task had private context, save it--reset previous context. */

    if (pOldTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pOldTcb->spare1;
```

```
        pOldTcb->spare1 = (int) vmCurrentGet ();

        /* restore old context */

        vmCurrentSet (pContext);
        }

    /*
     * If next task has private context, map new context and save previous
     * context in task's TCB.
     */

    if (pNewTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pNewTcb->spare1;
        pNewTcb->spare1 = (int) vmCurrentGet();

        /* install new tasks context */

        vmCurrentSet (pContext);
        }
    }
```

```
/* taskExample.h - header file for testing VM contexts used by switch hook */

/* This code is used by the sample task. */

#define MAX (10000000)

typedef struct myStuff {
    int stuff;
    int myStuff;
    } MY_DATA;
```

```
/* testTask.c - task code to test switch hooks */

#include "vxWorks.h"
#include "memLib.h"
#include "taskLib.h"
#include "stdio.h"
#include "vmLib.h"
#include "taskExample.h"

IMPORT char *string = "test\n";

MY_DATA *pMem;
```

```
/*****************************************************************************
*
* testTask - allocate private memory and use it
*
* Loop forever, modifying memory and printing out a global string. Use this
* in conjunction with testing from the shell. Since pMem points to private
* memory, the shell should generate a bus error when it tries to read it.
* For example:
*      -> sp testTask
*      -> d pMem
*/

STATUS testTask (void)
    {
    int val;
    WIND_TCB *myTcb;

    /* install private context */

    if (contextSetup () == ERROR)
        return (ERROR);

    /* get TCB */

    myTcb = taskTcb (taskIdSelf ());

    /* allocate private memory */

    if ((pMem = (MY_DATA *) memPartAlloc((PART_ID) myTcb->spare2,
         sizeof (MY_DATA))) == NULL)
        return (ERROR);

    /*
     * Forever, modify data in private memory and display string in
     * global memory.
     */

    FOREVER
        {
        for (val = 0; val <= MAX; val++)
            {
            /* modify structure */

            pMem->stuff = val;
            pMem->myStuff = val / 2;

            /* make sure can access global virtual memory */

            printf (string);

            taskDelay (sysClkRateGet() * 10);
            }
        }
    return (OK);
    }
```

```
/*****************************************************************************
*
* testVmContextGet - return a task's virtual memory context stored in TCB
*
* Used with vmContextShow()[1] to display a task's virtual memory context.
* For example, from the shell, type:
*   -> tid = sp (testTask)
*   -> vmContextShow (testVmContextGet (tid))
*/

VM_CONTEXT_ID testVmContextGet
    (
    UINT tid
    )
    {
    return ((VM_CONTEXT_ID) ((taskTcb (tid))->spare1));
    }
```

7.5.3 Noncacheable Memory

Architectures that do not support bus snooping must disable the memory caching that is used for interprocessor communication (or by DMA devices). If multiple processors are reading from and writing to a memory location, you must guarantee that when the CPU accesses the data, it is using the most recent value. If caching is used in one or more CPUs in the system, there can be a local copy of the data in one of the CPUs' data caches. In the example in Figure 7-3, a system with multiple CPUs share data, and one CPU on the system (CPU 0) caches the shared data. A task on CPU 0 reads the data [1] and then modifies the value [2]; however, the new value may still be in the cache and not flushed to memory when a task on another CPU (CPU 1) accesses it [3]. Thus the value of the data used by the task on CPU 1 is the old value and does not reflect the modifications done by the task on CPU 0; that value is still in CPU 0's data cache [2].

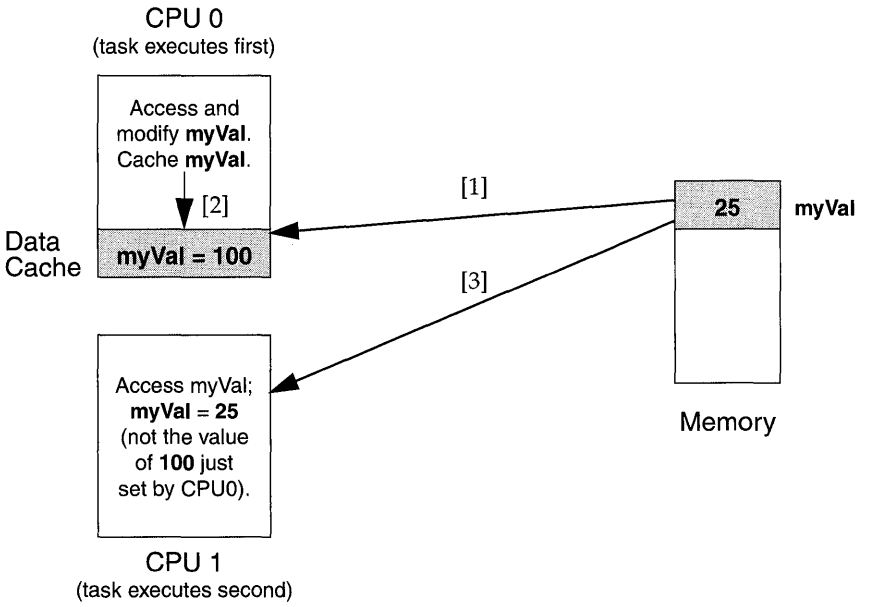
To disable caching on a page basis, use *vmStateSet( )*; for example:

```
vmStateSet (pContext, pSData, len, VM_STATE_MASK_CACHEABLE, VM_STATE_CACHEABLE_NOT)
```

To allocate noncacheable memory, see the reference entry for *cacheDmaMalloc( )*.

1. This routine is *not* built in to the Tornado shell. To use it from the Tornado shell, you must define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. Configuration*. When invoked this routine's output is sent to the standard output device.

Figure 7-3 **Example of Possible Problems with Data Caching**



7.5.4 Nonwritable Memory

Memory can be marked as nonwritable. Sections of memory can be write-protected using *vmStateSet( )* to prevent inadvertent access.

One use of this is to restrict modification of a data object to a particular routine. If a data object is global but read-only, tasks can read the object but not modify it. Any task that must modify this object must call the associated routine. Inside the routine, the data is made writable for the duration of the routine, and on exit, the memory is set to VM_STATE_WRITABLE_NOT.

Example 7-2 **Nonwritable Memory**

In this code example, to modify the data structure pointed to by **pData**, a task must call *dataModify( )*. This routine makes the memory writable, modifies the data, and sets the memory back to nonwritable. If a task tries to read the memory, it is successful; however, if it tries to modify the data outside of *dataModify( )*, a bus error occurs.

```
/* privateCode.h - header file to make data writable from routine only */

#define MAX 1024

typedef struct myData
    {
    char stuff[MAX];
    int moreStuff;
    } MY_DATA;
```

```
/* privateCode.c - uses VM contexts to make data private to a code segment */

#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
#include "privateCode.h"

MY_DATA * pData;
SEM_ID dataSemId;
int pageSize;

/*************************************************************************
*
* initData - allocate memory and make it nonwritable
*
* This routine initializes data and should be called only once.
*
*/

STATUS initData (void)
    {
    pageSize = vmPageSizeGet();

    /* create semaphore to protect data */

    dataSemId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);

    /* allocate memory = to a page */

    pData = (MY_DATA *) valloc (pageSize);

    /* initialize data and make it read-only */

    bzero (pData, pageSize);
    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE_NOT) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
```

```
    /* release semaphore */

    semGive (dataSemId);
    return (OK);
    }

/*************************************************************************
*
* dataModify - modify data
*
* To modify data, tasks must call this routine, passing a pointer to
* the new data.
* To test from the shell use:
*     -> initData
*     -> sp dataModify
*     -> d pData
*     -> bfill (pdata, 1024, 'X')
*/

STATUS dataModify
    (
    MY_DATA * pNewData
    )
    {

    /* take semaphore for exclusive access to data */

    semTake (dataSemId, WAIT_FOREVER);

    /* make memory writable */

    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    /* update data*/

    bcopy (pNewData, pData, sizeof(MY_DATA));

    /* make memory not writable */

    if (vmStateSet (NULL, pData, pageSize, VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE_NOT) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }

    semGive (dataSemId);

    return (OK);
    }
```

7.5.5 Troubleshooting

If **INCLUDE_SHOW_ROUTINES** is defined, you can use *vmContextShow( )* to display a virtual memory context on the standard output device. In the following example, the current virtual memory context is displayed. Virtual addresses between 0x0 and 0x59fff are write protected; 0xff800000 through 0xffbfffff are noncacheable; and 0x2000000 through 0x2005fff are private. All valid entries are listed and marked with a *V+*. Invalid entries are not listed.

```
-> vmContextShow 0
value = 0 = 0x0
```

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

```
VIRTUAL ADDR    BLOCK LENGTH    PHYSICAL ADDR   STATE
0x0             0x5a000         0x0             W- C+ V+ (global)
0x5a000         0x1f3c000       0x5a000         W+ C+ V+ (global)
0x1f9c000       0x2000          0x1f9c000       W+ C+ V+ (global)
0x1f9e000       0x2000          0x1f9e000       W- C+ V+ (global)
0x1fa0000       0x2000          0x1fa0000       W+ C+ V+ (global)
0x1fa2000       0x2000          0x1fa2000       W- C+ V+ (global)
0x1fa4000       0x6000          0x1fa4000       W+ C+ V+ (global)
0x1faa000       0x2000          0x1faa000       W- C+ V+ (global)
0x1fac000       0xa000          0x1fac000       W+ C+ V+ (global)
0x1fb6000       0x2000          0x1fb6000       W- C+ V+ (global)
0x1fb8000       0x36000         0x1fb8000       W+ C+ V+ (global)
0x1fee000       0x2000          0x1fee000       W- C+ V+ (global)
0x1ff0000       0x2000          0x1ff0000       W+ C+ V+ (global)
0x1ff2000       0x2000          0x1ff2000       W- C+ V+ (global)
0x1ff4000       0x2000          0x1ff4000       W+ C+ V+ (global)
0x1ff6000       0x2000          0x1ff6000       W- C+ V+ (global)
0x1ff8000       0x2000          0x1ff8000       W+ C+ V+ (global)
0x1ffa000       0x2000          0x1ffa000       W- C+ V+ (global)
0x1ffc000       0x4000          0x1ffc000       W+ C+ V+ (global)
0x2000000       0x6000          0x1f96000       W+ C+ V+
0xff800000      0x400000        0xff800000      W- C- V+ (global)
0xffe00000      0x20000         0xffe00000      W+ C+ V+ (global)
0xfff00000      0xf0000         0xfff00000      W+ C- V+ (global)
```

7.5.6 Precautions

Memory that is marked as global cannot be remapped using *vmMap( )*. To add to global virtual memory, use *vmGlobalMap( )*. For further information on adding global virtual memory, see *7.5.2 Private Virtual Memory*, p.413.

Performances of MMUs vary across architectures; in fact, some architectures may cause the system to become non-deterministic. For additional information, see the architecture-specific documentation for your hardware.

8 Configuration

List of Tables

List of Figures

8.1 Introduction

The Tornado distribution includes a VxWorks system image for each target shipped. The *system image* is a binary module that can be booted and run on a target system. The system image consists of all desired system object modules linked together into a single non-relocatable object module with no unresolved external references.

In most cases, you will find the supplied system image entirely adequate for initial development. However, later in the cycle you may want to tailor its configuration to reflect your application's requirements.

This chapter describes how to configure the system image, which you accomplish by directly editing configuration files. This chapter covers the following topics:

- The VxWorks board support package (BSP).
- VxWorks configuration files and configuration options and parameters.
- Some of the common alternative configurations of VxWorks.

8.2 The Board Support Package (BSP)

The directory **config/***bspname* contains the *Board Support Package (BSP)*, which consists of files for the particular hardware used to run VxWorks, such as a VME board with serial lines, timers, and other devices. The files include: **Makefile**, **sysLib.c**, **sysSerial.c**, **sysALib.s**, **romInit.s**, *bspname*.**h**, and **config.h**.

In releasing new versions of BSPs for VxWorks 5.3, a new BSP standard has been created, called BSP Version 1.1. The application note *Upgrading a VxWorks BSP for Tornado 1.0* describes how to convert version 1.0 BSPs to version 1.1. The standard is fully described in the *VxWorks BSP Porting Kit*.

The System Library

The file **sysLib.c** provides the board-level interface on which VxWorks and application code can be built in a hardware-independent manner. The functions addressed in this file include:

- Initialization functions
  - initialize the hardware to a known state
  - identify the system
  - initialize drivers, such as SCSI or custom drivers
- Memory/address space functions
  - get the on-board memory size
  - make on-board memory accessible to external bus (optional)
  - map local and bus address spaces
  - enable/disable cache memory
  - set/get nonvolatile RAM (NVRAM)
  - define the board's memory map (optional)
  - virtual-to-physical memory map declarations for processors with MMUs
- Bus interrupt functions
  - enable/disable bus interrupt levels
  - generate bus interrupts
- Clock/timer functions
  - enable/disable timer interrupts
  - set the periodic rate of the timer
- Mailbox/location monitor functions (optional)
  - enable mailbox/location monitor interrupts

The **sysLib** library does not support every feature of every board: some boards may have additional features, others may have fewer, others still may have the same features with a different interface. For example, some boards provide some **sysLib** functions by means of hardware switches, jumpers, or PALs, instead of by software-controllable registers.

The configuration modules **usrConfig.c** and **bootConfig.c** in **config/all** are responsible for invoking this library's routines at the appropriate time. Device drivers can use some of the memory mapping routines and bus functions.

Virtual Memory Mapping

For boards with MMU support, the data structure **sysPhysMemDesc** defines the virtual-to-physical memory map. This table is typically defined in **sysLib.c**, although some BSPs place it in a separate file, **memDesc.c**. It is declared as an array of the data structure **PHYS_MEM_DESC**. No two entries in this descriptor can overlap; each entry must be a unique memory space.

The **sysPhysMemDesc** array should reflect your system configuration, and you may encounter a number of reasons for changing the MMU memory map, for example: the need to change the size of local memory or the size of the VME master access space, or because the address of the VME master access space has been moved. For information on virtual memory mapping, as well as an example of how to modify **sysPhysMemDesc**, see *7.3 Virtual Memory Configuration*, p.408.

⚠ **NOTE:** A bus error can occur if you try to access memory that is not mapped.

The Serial Driver

The file **sysSerial.c** provides board-specific initialization for the on-board serial ports. The actual serial I/O driver is in the **src/drv/sio** directory. The library **ttyDrv** uses the serial I/O driver to provide terminal operations for VxWorks.

BSP Initialization Modules

The following files initialize the BSP:

- The file **romInit.s** contains assembly-level initialization routines.
- The file **sysALib.s** contains initialization and system-specific assembly-level routines.

BSP Documentation

The file **target.nr** in the **config/***bspname* directory is the nroff source of the online man-page entry for target-specific information. (For information on how to view these man pages, see the *Tornado User's Guide: Starting Tornado.*) The **target.nr** file describes the supported board variations, the relevant jumpering, and supported devices. It also includes an ASCII representation of the board layout with an indication of board jumpers (if applicable) and the location of the ROM sockets.

8.3 Configuring VxWorks

The configuration of VxWorks is determined by the configuration header files **config/all/configAll.h** and **config/***bspname***/config.h**. These files are used by the **usrConfig.c**, **bootConfig.c**, and **bootInit.c** modules as they run the initialization routines distributed in the directory **src/config** to configure VxWorks.

The VxWorks distribution includes the configuration files for the default development configuration. You can create your own versions of these files to better suit your particular configurations; this is described in the following subsections. In addition, if you need multiple configurations, environment variables are provided so you can move easily between them.

To rebuild VxWorks for your own configuration, follow the procedures described in the *Tornado User's Guide: Cross-Development*.

Including optional components in your VxWorks image can significantly increase the image size. 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**, be sure to see *8.4.1 Scaling Down VxWorks*, p.447 and *Creating Bootable Applications* in the *Tornado User's Guide: Cross-Development* for information on how to resolve the problem.

8.3.1 The Environment Variables

In a development environment, you may have several different configurations you wish to test, or you may wish to specify different target code in different situations. In order to build VxWorks to these different specifications, you need to modify your environment.

In general, your Tornado environment consists of three parts: the host code (Tornado), the target code, and the configuration files discussed in this section. If you use the default environment, your environment variables are defined as follows:

Host code **$WIND_BASE/host/***hosttype***/bin**

Target code **TGT_DIR = $WIND_BASE/target**

Configuration code
CONFIG_ALL = TGT_DIR/config/all

To use different versions of **usrConfig.c**, **bootConfig.c**, and **bootInit.c**, store them in a different directory and change the value of **CONFIG_ALL**. To use different

target code, point to the alternate directory by changing the value of **TGT_DIR**. You can change the value of **CONFIG_ALL** by changing it either in your makefile or on the command line. The value of **TGT_DIR** must be changed on the command line.

NOTE: Changing **TGT_DIR** will change the default value of **CONFIG_ALL**. If this is not what you want, reset **CONFIG_ALL** as well.

To change **CONFIG_ALL** in your make file, add the following command:

```
CONFIG_ALL = $WIND_BASE/target/config/newDir
```

To change **CONFIG_ALL** on the command line, do the following:

```
% make ... CONFIG_ALL = $WIND_BASE/target/config/newDir
```

To change **TGT_DIR** on the command line, do the following:

```
% make ... TGT_DIR = $ALT_DIR/target
```

8.3.2 The Configuration Header Files

You can control VxWorks's configuration by including or excluding definitions in the global configuration header file **configAll.h** and in the target-specific configuration header file **config.h**. This section describes these files.

The Global Configuration Header File: configAll.h

The **configAll.h** header file, in the directory **config/all**, contains default definitions that apply to all targets, unless redefined in the target-specific header file **config.h**. The following options and parameters are defined in **configAll.h**:

- kernel configuration parameters
- I/O system parameters
- NFS parameters
- selection of optional software modules
- selection of optional device controllers
- cache modes
- maximum number of the different shared memory objects
- device controller I/O addresses, interrupt vectors, and interrupt levels
- miscellaneous addresses and constants

The BSP-specific Configuration Header File: config.h

There is also a BSP-specific header file, **config.h**, in the directory **config/***bspname*. This file contains definitions that apply only to the specific target, and can also redefine default definitions in **configAll.h** that are inappropriate for the particular target. For example, if a target cannot access a device controller at the default I/O address defined in **configAll.h** because of addressing limitations, the address can be redefined in **config.h**.

The **config.h** header file includes definitions for the following parameters:

- default boot parameter string for boot ROMs
- interrupt vectors for system clock and parity errors
- device controller I/O addresses, interrupt vectors, and interrupt levels
- shared memory network parameters
- miscellaneous memory addresses and constants

If any options from **configAll.h** need to be changed for this one BSP, then any previous definition of that option should be undefined and redefined as necessary in **config.h**. Do not change options in **config/all/configAll.h** unless they are to apply to all BSPs at your site.

Selection of Optional Features

VxWorks ships with optional features and device drivers that can be included or omitted from the target system. These are controlled by macros in the configuration header files that cause conditional compilation in the **config/all/usrConfig.c** module.

The distributed versions of the configuration header files **configAll.h** and **config.h** include all the available software options and several network device drivers. You define a macro by moving it from the EXCLUDED FACILITIES section of the header file to the INCLUDED SOFTWARE FACILITIES section. (For a partial listing of the configuration macros, see Table 8-1.) For example, to include the ANSI C **assert** library, make sure the macro **INCLUDE_ANSI_ASSERT** is defined; to include the Network File System (NFS) facility, make sure **INCLUDE_NFS** is defined. Modification or exclusion of particular facilities is discussed in detail in *8.4 Alternative VxWorks Configurations*, p.447.

Macros shown in Table 8-1 that end in *XXX* are not valid macros but represent families of options where the *XXX* is replaced by a suffix declaring a specific routine. For example, **INCLUDE_CPLUS_***XXX* refers to a family of macros that includes **INCLUDE_CPLUS_MIN** and **INCLUDE_CPLUS_BOOCH**.

Table 8-1 **Key VxWorks Options**

Macro	*	Option
INCLUDE_ADA		Ada support
INCLUDE_ANSI_*XXX*	*	Various ANSI C library options
INCLUDE_BOOTP	*	BOOTP support
INCLUDE_CACHE_SUPPORT	*	Cache support
INCLUDE_CPLUS		Bundled C++ support
INCLUDE_CPLUS_*XXX*		Various C++ support options
INCLUDE_CPLUS		Native debugging, for backward-compatible use with target-resident shell
INCLUDE_DEMO		Use simple demo program
INCLUDE_DOSFS		DOS-compatible file system
INCLUDE_FLOATING_POINT	*	Floating-point I/O
INCLUDE_FORMATTED_IO	*	Formatted I/O
INCLUDE_FTP_SERVER	*	FTP server support
INCLUDE_HW_FP		Hardware floating-point support
INCLUDE_INSTRUMENTATION		WindView instrumentation; see the *WindView User's Guide* for details
INCLUDE_IO_SYSTEM	*	I/O system package
INCLUDE_LOADER		Target-resident object module loader package
INCLUDE_LOGGING	*	Logging facility
INCLUDE_MEM_MGR_FULL		Full-featured memory manager
INCLUDE_MIB2_*XXX*		Various MIB-2 options
INCLUDE_MMU_BASIC		Bundled MMU support
INCLUDE_MMU_FULL		Unbundled MMU support (requires VxVMI)
INCLUDE_MSG_Q	*	Message queue support
INCLUDE_NETWORK	*	Network subsystem code
INCLUDE_NFS		Network File System (NFS)
INCLUDE_NFS_SERVER		NFS server
INCLUDE_PIPES	*	Pipe driver
INCLUDE_POSIX_*XXX*		Various POSIX options
INCLUDE_PROTECT_TEXT		Text segment write protection (requires VxVMI)
INCLUDE_PROTECT_VEC_TABLE		Vector table write protection (requires VxVMI)
INCLUDE_PROXY_CLIENT	*	Proxy ARP client support
INCLUDE_PROXY_SERVER		Proxy ARP server support
INCLUDE_RAWFS		Raw file system
INCLUDE_RLOGIN		Remote login with **rlogin**
INCLUDE_RPC		Remote Procedure Calls (RPC)
INCLUDE_RT11FS		RT-11 file system
INCLUDE_SCSI		SCSI support
INCLUDE_SCSI2		SCSI-2 extensions

Table 8-1 **Key VxWorks Options** *(Continued)*

Macro	*	Option
INCLUDE_SCSI_BOOT		Allow booting from a SCSI device
INCLUDE_SECURITY		Remote login security package
INCLUDE_SEM_BINARY	*	Binary semaphore support
INCLUDE_SEM_COUNTING	*	Counting semaphore support
INCLUDE_SEM_MUTEX	*	Mutual exclusion semaphore support
INCLUDE_SHELL		C-expression interpreter (target shell)
INCLUDE_SHOW_ROUTINES		Various system object show facilities
INCLUDE_SIGNALS	*	Software signal facilities
INCLUDE_SM_OBJ		Shared memory object support (requires VxMP)
INCLUDE_SNMPD		SNMP agent
INCLUDE_SPY		Task activity monitor
INCLUDE_STDIO	*	Standard I/O package
INCLUDE_SW_FP		Software Floating point emulation package
INCLUDE_SYM_TBL		Target-resident symbol table support
INCLUDE_TASK_HOOKS	*	Kernel call-out support
INCLUDE_TASK_VARS	*	Task variable support
INCLUDE_TELNET		Remote login with **telnet**
INCLUDE_TFTP_CLIENT		TFTP client support
INCLUDE_TFTP_SERVER		TFTP server support
INCLUDE_TIMEX	*	Function execution timer
INCLUDE_UNLOADER		Target-resident object module unloader package
INCLUDE_WATCHDOGS		Watchdog support
INCLUDE_WDB	*	Target agent
INCLUDE_WINDVIEW		WindView command server; see the *WindView User's Guide* for details
INCLUDE_ZBUF_SOCK		Zbuf socket interface

* Items marked with an asterisk are included in the default configuration. Note that, since this list of options is not complete, not all macros included in the default configuration are listed here. Note also that their inclusion may be overridden in **config.h** for your BSP.

8.3.3 The Configuration Module: usrConfig.c

Use VxWorks configuration header files to configure your VxWorks system to meet your development requirements. Users should not resort to changing the WRS-supplied **usrConfig.c**, or any other module in the directory **config/all**. If an extreme situation requires such a change, we recommend you copy all the files in **config/all**

to another directory, and add a **CONFIG_ALL** macro to your makefile to point the make system to the location of the modified files. For example, add the following to your makefile after the first group of include statements:

```
# ../myAll contains a copy of all the ../all files
    CONFIG_ALL = ../myAll
```

8.3.4 VxWorks Initialization Timeline

This section covers the initialization sequence for VxWorks in a typical development configuration. The steps are described in sequence of execution. This is not the only way VxWorks can be bootstrapped on a particular processor. There are often more efficient or robust techniques unique to a particular processor or hardware; consult your hardware's documentation.

For final production, the sequence can be revisited to include diagnostics or to remove some of the generic operations that are required for booting a development environment, but that are unnecessary for production. This description can provide only an approximate guide to the processor initialization sequence and does not document every exception to this time-line.

The early steps of the initialization sequence are slightly different for ROM-based versions of VxWorks; for infomration, see *8.4.3 Initialization Sequence for ROM-Based VxWorks*, p.452.

For a summary of the initialization time-line, see Table 8-2. The following sections describe the initialization in detail by routine name. For clarity, the sequence is divided into a number of main steps or function calls. The key routines are listed in the headings and are described in chronological order.

The VxWorks Entry Point: *sysInit*( )

The first step in starting a VxWorks system is to load a system image into main memory. This usually occurs as a download from the development host, under the control of the VxWorks boot ROM. Next, the boot ROM transfers control to the VxWorks startup entry point, ***sysInit*( )**. This entry point is configured by **RAM_LOW_ADRS** in the makefile and in **config.h**. The VxWorks memory layout is different for each architecture; for details, see the appendix that describes your architecture.

The entry point, ***sysInit*( )**, is in the system-dependent assembly language module, **sysALib.s**. It locks out all interrupts, invalidates caches if applicable, and initializes processor registers (including the C stack pointer) to default values. It

also disables tracing, clears all pending interrupts, and invokes ***usrInit( )***, a C subroutine in the **usrConfig.c** module. For some targets, ***sysInit( )*** also performs some minimal system-dependent hardware initialization, enough to execute the remaining initialization in ***usrInit( )***. The initial stack pointer, which is used only by ***usrInit( )***, is set to occupy an area below the system image but above the vector table (if any).

The Initial Routine: *usrInit( )*

The ***usrInit( )*** routine (in **usrConfig.c**) saves information about the boot type, handles all the initialization that must be performed before the kernel is actually started, and then starts the kernel execution. It is the first C code to run in VxWorks. It is invoked in supervisor mode with all hardware interrupts locked out.

Many VxWorks facilities cannot be invoked from this routine. Because there is no task context as yet (no TCB and no task stack), facilities that require a task context cannot be invoked. This includes any facility that can cause the caller to be preempted, such as semaphores, or any facility that uses such facilities, such as ***printf( )***. Instead, the ***usrInit( )*** routine does only what is necessary to create an initial task, ***usrRoot( )***. This task then completes the startup.

The initialization in ***usrInit( )*** includes the following:

Cache Initialization

The code at the beginning of ***usrInit( )*** initializes the caches, sets the mode of the caches and puts the caches in a safe state. At the end of ***usrInit( )***, the instruction and data caches are enabled by default.

Zeroing Out the System *bss* Segment

The C and C++ languages specify that all uninitialized variables must have initial values of 0. These uninitialized variables are put together in a segment called *bss*. This segment is not actually loaded during the bootstrap, because it is known to be zeroed out. Because ***usrInit( )*** is the first C code to execute, it clears the section of memory containing *bss* as its very first action. While the VxWorks boot ROMs clear all memory, VxWorks does not assume that the boot ROMs are used.

Initializing Interrupt Vectors

The exception vectors must be set up before enabling interrupts and starting the kernel. First, ***intVecBaseSet( )*** is called to establish the vector table base address.

⚠ **NOTE:** There are exceptions to this in some architectures; see the appendix that describes your architecture for details.

After ***intVecBaseSet( )*** is called, the routine ***excVecInit( )*** initializes all exception vectors to default handlers that safely trap and report exceptions caused by program errors or unexpected hardware interrupts.

Initializing System Hardware to a Quiescent State

System hardware is initialized by calling the system-dependent routine ***sysHwInit( )***. This mainly consists of resetting and disabling hardware devices that can cause interrupts after interrupts are enabled (when the kernel is started). This is important because the VxWorks ISRs (for I/O devices, system clocks, and so on), are not connected to their interrupt vectors until the system initialization is completed in the ***usrRoot( )*** task. However, do not attempt to connect an interrupt handler to an interrupt during the ***sysHwInit( )*** call, because the memory pool is not yet initialized.

Initializing the Kernel

The ***usrInit( )*** routine ends with calls to two kernel initialization routines:

usrKernelInit( ) (defined in **usrKernel.c**)
calls the appropriate initialization routines for each of the specified optional kernel facilities (see Table 8-2 for a list).

kernelInit( ) (part of **kernelLib.c**)
initiates the multitasking environment and never returns. It takes the following parameters:

- The application to be spawned as the "root" task, typically ***usrRoot( )***.
- The stack size.
- The start of usable memory; that is, the memory after the main text, data, and *bss* of the VxWorks image. All memory after this area is added to the system memory pool, which is managed by **memPartLib**. Allocation for dynamic module loading, task control blocks, stacks, and so on, all come out of this region. See *Initializing the Memory Pool*, p.438.
- The top of memory as indicated by ***sysMemTop( )***. If a contiguous block of memory is to be preserved from normal memory allocation, pass ***sysMemTop( )*** less the reserved memory.

- The interrupt stack size. The interrupt stack corresponds to the largest amount of stack space any interrupt-level routine uses, plus a safe margin for the nesting of interrupts.
- The interrupt lock-out level. For architectures that have a *level* concept, it is the maximum level. For architectures that do not have a level concept, it is the mask to disable interrupts. See the appendix that describes your architecture for details.

kernelInit( ) calls ***intLockLevelSet( )***, disables round-robin mode, and creates an interrupt stack if supported by the architecture. It then creates a root stack and TCB from the top of the memory pool, spawns the root task, ***usrRoot( )***, and terminates the ***usrInit( )*** thread of execution. At this time, interrupts are enabled; it is critical that all interrupt sources are disabled and pending interrupts cleared.

Initializing the Memory Pool

VxWorks includes a memory allocation facility, in the module **memPartLib**, that manages a pool of available memory. The ***malloc( )*** routine allows callers to obtain variable-size blocks of memory from the pool. Internally, VxWorks uses ***malloc( )*** for dynamic allocation of memory. In particular, many VxWorks facilities allocate data structures during initialization. Therefore, the memory pool must be initialized before any other VxWorks facilities are initialized.

Note that the Tornado target server manages a portion of target memory to support downloading of object modules and other development functions. See the *Tornado User's Guide: Cross-Development* for more information.

VxWorks makes heavy use of ***malloc( )***, including allocation of space for loaded modules, allocation of stacks for spawned tasks, and allocation of data structures on initialization. You are also encouraged to use ***malloc( )*** to allocate any memory your application requires. Therefore, it is recommended that you assign to the VxWorks memory pool all unused memory, unless you must reserve some fixed absolute memory area for a particular application use.

The memory pool is initialized by ***kernelInit( )***. The parameters to ***kernelInit( )*** specify the start and end address of the initial memory pool. In the default ***usrInit( )*** distributed with VxWorks, the pool is set to start immediately following the end of the booted system, and to contain all the rest of available memory.

The extent of available memory is determined by ***sysMemTop( )***, which is a system-dependent routine that determines the size of available memory. If your system has other noncontiguous memory areas, you can make them available in the general memory pool by later calling ***memAddToPool( )*** in the ***usrRoot( )*** task.

The Initial Task: *usrRoot*( )

When the multitasking kernel starts executing, all VxWorks multitasking facilities are available. Control is transferred to the ***usrRoot*( )** task and the initialization of the system can be completed. For example, ***usrRoot*( )** performs the following:

- initialization of the system clock
- initialization of the I/O system and drivers
- creation of the console devices
- setting of standard in and standard out
- installation of exception handling and logging
- initialization of the pipe driver
- initialization of standard I/O
- creation of file system devices and installation of disk drivers
- initialization of floating-point support
- initialization of performance monitoring facilities
- initialization of the network
- initialization of optional facilities
- initialization of WindView (see the *WindView User's Guide*)
- initialization of target agent
- execution of a user-supplied startup script

To review the complete initialization sequence within ***usrRoot*( )**, see **config/all/usrConfig.c**.

Modify these initializations to suit your configuration. The meaning of each step and the significance of the various parameters are explained in the following sections.

Initialization of the System Clock

The first action in the ***usrRoot*( )** task is to initialize the VxWorks clock. The system clock interrupt vector is connected to the routine ***usrClock*( )** (described in *The System Clock Routine: usrClock( )*, p.444) by calling ***sysClkConnect*( )**. Then, the system clock rate (usually 60Hz) is set by ***sysClkRateSet*( )**. Most boards allow clock rates as low as 30Hz (some even as low as 1Hz), and as high as several thousand Hz. High clock rates (>1000Hz) are not desirable, because they can cause system *thrashing*.[1]

The timer drivers supplied by WRS include a call to ***sysHwInit2*( )** as part of the ***sysClkConnect*( )** routine. Wind River BSPs use ***sysHwInit2*( )** to perform further

1. *Thrashing* occurs when clock interrupts are so frequent that the processor spends too much time servicing the interrupts, and no application code can run.

board initialization that is not completed in ***sysHwInit( )***. For example, an ***intConnect( )*** of ISRs can take place here, because memory can be allocated now that the system is multitasking.

Initialization of the I/O System

If **INCLUDE_IO_SYSTEM** is defined in **configAll.h**, the VxWorks I/O system is initialized by calling the routine ***iosInit( )***. The arguments specify the maximum number of drivers that can be subsequently installed, the maximum number of files that can be open in the system simultaneously, and the desired name of the "null" device that is included in the VxWorks I/O system. This null device is a "bit-bucket" on output and always returns end-of-file for input.

The inclusion or exclusion of **INCLUDE_IO_SYSTEM** also affects whether the console devices are created, and whether standard in, standard out, and standard error are set; see the next two sections for more information.

Creation of the Console Devices

If the driver for the on-board serial ports is included (**INCLUDE_TTY_DEV**), it is installed in the I/O system by calling the driver's initialization routine, typically ***ttyDrv( )***. The actual devices are then created and named by calling the driver's device-creation routine, typically ***ttyDevCreate( )***. The arguments to this routine includes the device name, a serial I/O channel descriptor (from the BSP), and input and output buffer sizes.

The macro **NUM_TTY** specifies the number of *tty* ports (default is 2), **CONSOLE_TTY** specifies which port is the console (default is 0), and **CONSOLE_BAUD_RATE** specifies the bps rate (default is 9600). These macros are specified in **configAll.h**, but can be overridden in **config.h** for boards with a nonstandard number of ports.

PCs can use an alternative console with keyboard input and VGA output; see your PC workstation documentation for details.

Setting of Standard In, Standard Out, and Standard Error

The system-wide standard in, standard out, and standard error assignments are established by opening the console device and calling ***ioGlobalStdSet( )***. These assignments are used throughout VxWorks as the default devices for communicating with the application developer. To make the console device an interactive terminal, call ***ioctl( )*** to set the device options to **OPT_TERMINAL**.

Installation of Exception Handling and Logging

Initialization of the VxWorks exception handling facilities (supplied by the module **excLib**) and logging facilities (supplied by **logLib**) takes place early in the execution of the root task. This facilitates detection of program errors in the root task itself or in the initialization of the various facilities.

The exception handling facilities are initialized by calling ***excInit( )*** when **INCLUDE_EXC_HANDLING** and **INCLUDE_EXC_TASK** are defined. The ***excInit( )*** routine spawns the exception support task, ***excTask( )***. Following this initialization, program errors causing hardware exceptions are safely trapped and reported, and hardware interrupts to uninitialized vectors are reported and dismissed. The VxWorks signal facility, used for task-specific exception handling, is initialized by calling ***sigInit( )*** when **INCLUDE_SIGNALS** is defined.

The logging facilities are initialized by calling ***logInit( )*** when **INCLUDE_LOGGING** is defined. The arguments specify the file descriptor of the device to which logging messages are to be written, and the number of log message buffers to allocate. The logging initialization also includes spawning the logging task, ***logTask( )***.

Initialization of the Pipe Driver

If named pipes are desired, define **INCLUDE_PIPE** in **configAll.h** so that ***pipeDrv( )*** is called automatically to initialize the pipe driver. Tasks can then use pipes to communicate with each other through the standard I/O interface. Pipes must be created with ***pipeDevCreate( )***.

Initialization of Standard I/O

VxWorks includes an optional *standard I/O* package when **INCLUDE_STDIO** is defined.

Creation of File System Devices and Initialization of Device Drivers

Many VxWorks configurations include at least one disk device or RAM disk with a dosFs, rt11Fs, or rawFs file system. First, a disk driver is installed by calling the driver's initialization routine. Next, the driver's device-creation routine defines a device. This call returns a pointer to a **BLK_DEV** structure that describes the device.

The new device can then be initialized and named by calling the file system's device-initialization routine—***dosFsDevInit( )***, ***rt11FsDevInit( )***, or ***rawFsDevInit( )***—when the respective constants **INCLUDE_DOSFS**, **INCLUDE_RT11FS**, and **INCLUDE_RAWFS** are defined. (Before a device can be initialized, the file system module must already be initialized with ***dosFsInit( )***,

rt11FsInit( ), or *rawFsInit( )*.) The arguments to the file system device-initialization routines depend on the particular file system, but typically include the device name, a pointer to the **BLK_DEV** structure created by the driver's device-creation routine, and possibly some file-system-specific configuration parameters.

Initialization of Floating-Point Support

Support for floating-point *I/O* is initialized by calling the routine ***floatInit( )*** when **INCLUDE_FLOATING_POINT** is defined in **configAll.h**. Support for floating-point *coprocessors* is initialized by calling ***mathHardInit( )*** when **INCLUDE_HW_FP** is defined. Support for software floating-point *emulation* is initialized by calling ***mathSoftInit( )*** when **INCLUDE_SW_FP** is defined. See the appropriate architecture appendix for details on your processor's floating-point support.

Inclusion of Performance Monitoring Tools

VxWorks has two built-in performance monitoring tools. A task activity summary is provided by **spyLib**, and a subroutine execution timer is provided by **timexLib**. These facilities are included by defining the macros **INCLUDE_SPY** and **INCLUDE_TIMEX**, respectively, in **configAll.h**.

Initialization of the Network

Before the network can be used, it must be initialized with the routine ***usrNetInit( )***, which is called by ***usrRoot( )*** when the constant **INCLUDE_NET_INIT** is defined in one of the configuration header files. (The source for ***usrNetInit( )*** is in **src/config/usrNetwork.c**.) The routine ***usrNetInit( )*** takes a configuration string as an argument. This configuration string is usually the "boot line" that is specified to the VxWorks boot ROMs to boot the system (see the *Tornado User's Guide: Starting Tornado*). Based on this string, ***usrNetInit( )*** performs the following:

- Initializes network subsystem by calling the routine ***netLibInit( )***.
- Attaches and configures appropriate network drivers.
- Adds gateway routes.
- Initializes the remote file access driver **netDrv**, and adds a remote file access device.
- Initializes the remote login facilities.
- Optionally initializes the Remote Procedure Calls (RPC) facility.
- Optionally initializes the Network File System (NFS) facility.

As noted previously, the inclusion of some of these network facilities is controlled by definitions in **configAll.h**; see Table 8-1 for a list of these constants. The network initialization steps are described in *5. Network*.

Initialization of Optional Products and Other Facilities

Shared memory objects are provided with the optional product VxMP. Before shared memory objects can be used, they must be initialized with the routine ***usrSmObjInit( )*** (in **src/config/usrSmObj.c**), which is called from ***usrRoot( )*** if **INCLUDE_SM_OBJ** is defined.

Basic MMU support is provided if **INCLUDE_MMU_BASIC** is defined. Text protection, vector table protection, and a virtual memory interface are provided with the optional product VxVMI, if **INCLUDE_MMU_FULL** is defined. The MMU is initialized by the routine ***usrMmuInit( )*** in **src/config/usrMmuInit.c**. If the macros **INCLUDE_PROTECT_TEXT** and **INCLUDE_PROTECT_VEC_TABLE** are also defined, text protection and vector table protection are initialized.

The GNU C++ compiler is shipped with Tornado. To initialize C++ support for either the GNU compiler or the optional CenterLine compiler, define either **INCLUDE_CPLUS** or **INCLUDE_CPLUS_MIN**. To include one or more of the Wind Foundation Class libraries, define the appropriate **INCLUDE_CPLUS_***library* macros (listed in Table 8-1).[2]

Initialization of WindView

Kernel instrumentation is provided with the optional product WindView. It is initialized in ***usrRoot( )*** when **INCLUDE_INSTRUMENTATION** is defined in **configAll.h**. Other WindView configuration constants control particular initialization steps; see the *WindView User's Guide: Configuring WindView.*

Initialization of the Target Agent

If **INCLUDE_WDB** is defined, ***wdbConfig( )*** in **src/config/usrWdb.c** is called. This routine initializes the agent's communication interface, then starts the agent. For information on configuring the agent and the agent's initialization sequence, see the *Tornado User's Guide: Getting Started.*

Execution of a Startup Script

The ***usrRoot( )*** routine executes a user-supplied startup script if the target-resident shell is configured into VxWorks, **INCLUDE_STARTUP_SCRIPT** is defined, and the

2. For information on using the GNU C++ compiler and the optional Wind Foundation Classes, see *10. C++ Development* and the *Tornado User's Guide: Cross-Development*.

script's file name is specified at boot time with the startup script parameter (see the *Tornado User's Guide: Starting Tornado*). If the parameter is missing, no startup script is executed.

The System Clock Routine: *usrClock( )*

Finally, the system clock ISR ***usrClock( )*** is attached to the system clock timer interrupt by the ***usrRoot( )*** task described *The Initial Task: usrRoot( )*, p.439. The ***usrClock( )*** routine calls the kernel clock tick routine ***tickAnnounce( )***, which performs OS bookkeeping. You can add application-specific processing to this routine.

Initialization Summary

Table 8-2 shows a summary of the entire VxWorks initialization sequence for typical configurations. For a similar summary applicable to ROM-based VxWorks systems, see *Overall Initialization for ROM-Based VxWorks*, p.453.

Table 8-2 **VxWorks Run-time System Initialization Sequence**

Routine	Activity	File
sysInit( )	(a) lock out interrupts	**sysALib.s**
	(b) invalidate caches, if any	
	(c) initialize system interrupt tables with default stubs (i960 only)	
	(d) initialize system fault tables with default stubs (i960 only)	
	(e) initialize processor registers to known default values	
	(f) disable tracing	
	(g) clear all pending interrupts	
	(h) invoke ***usrInit( )*** specifying boot type	

Table 8-2 **VxWorks Run-time System Initialization Sequence** *(Continued)*

Routine	Activity	File
usrInit( )	(a) zero *bss* (uninitialized data)	**usrConfig.c**
	(b) save **bootType** in **sysStartType**	
	(c) invoke ***excVecInit( )*** to initialize all system and default interrupt vectors	
	(d) invoke ***sysHwInit( )***	
	(e) invoke ***usrKernelInit( )***	
	(f) invoke ***kernelInit( )***	
usrKernelInit( )	The following routines are invoked if their configuration constants are defined.	**usrKernel.c**
	(a) ***classLibInit( )***	
	(b) ***taskLibInit( )***	
	(c) ***taskHookInit( )***	
	(d) ***semBLibInit( )***	
	(e) ***semMLibInit( )***	
	(f) ***semCLibInit( )***	
	(g) ***semOLibInit( )***	
	(h) ***wdLibInit( )***	
	(i) ***msgQLibInit( )***	
	(j) ***qInit( )*** for all system queues	
	(k) ***workQInit( )***	

Table 8-2 **VxWorks Run-time System Initialization Sequence** *(Continued)*

Routine	Activity	File
kernelInit( )	Initialize and start the kernel.	**kernelLib.c**
	(a) invoke ***intLockLevelSet( )***	
	(b) create root stack and TCB from top of memory pool	
	(c) invoke ***taskInit( )*** for ***usrRoot( )***	
	(d) invoke ***taskActivate( )*** for ***usrRoot( )***	
	(e) ***usrRoot( )***	
usrRoot( )	Initialize I/O system, install drivers, and create devices as specified in **configAll.h** and **config.h**.	**usrConfig.c**
	(a) ***sysClkConnect( )***	
	(b) ***sysClkRateSet( )***	
	(c) ***iosInit( )***	
	(d) if (**INCLUDE_TTY_DEV** and **NUM_TTY**) ***ttyDrv( )***, then establish console port, **STD_IN**, **STD_OUT**, **STD_ERR**	
	(e) initialize exception handling with ***excInit( )***, ***logInit( )***, ***sigInit( )***	
	(f) initialize the pipe driver with ***pipeDrv( )***	
	(g) ***stdioInit( )***	
	(h) ***mathSoftInit( )*** or ***mathHardInit( )***	
	(i) ***wdbConfig( )***: configure and initialize target agent	
	(j) run startup script if target-resident shell is configured	

8.4 Alternative VxWorks Configurations

The discussion of the **usrConfig** module in *8.3.3 The Configuration Module: usrConfig.c*, p.434 outlined the default configuration for a development environment. In this configuration, the VxWorks system image contains all of the VxWorks modules that are necessary to allow you to interact with the system through the Tornado host tools.

However, as you approach a final production version of your application, you may want to change the VxWorks configuration in one or more of the following ways:

- Change the configuration of the target agent.
- Decrease the size of VxWorks.
- Run VxWorks from ROM.

The following sections discuss the latter two alternatives to the typical development configuration. For a discussion on reconfiguring the target agent, see the *Tornado User's Guide: Getting Started.*

8.4.1 Scaling Down VxWorks

In a production configuration, it is often desirable to remove some of the VxWorks facilities to reduce the memory requirements of the system, to reduce boot time, or for security purposes.

Optional VxWorks facilities can be omitted by commenting out or using **#undef** to undefine their corresponding control constants in the header files **configAll.h** or **config.h**. For example, logging facilities can be omitted by undefining INCLUDE_LOGGING, and signalling facilities can be omitted by undefining INCLUDE_SIGNALS.

VxWorks is structured to make it easy to exclude facilities you do not need. However, not every BSP will be structured in this way. If you wish to minimize your application, be sure to examine your BSP code and eliminate references to facilities you do not need to include. Otherwise, they will be included even though you undefined them in your VxWorks configuration files.

Excluding Kernel Facilities

The definition of the following constants in **configAll.h** is optional, because referencing any of the corresponding kernel facilities from the application automatically includes the kernel service:

- INCLUDE_SEM_BINARY
- INCLUDE_SEM_MUTEX
- INCLUDE_SEM_COUNTING
- INCLUDE_MSG_Q
- INCLUDE_WATCHDOGS

These configuration constants appear in the default VxWorks configuration to ensure that all kernel facilities are configured into the system, even if not referenced by the application. However, if your goal is to achieve the smallest possible system, exclude these constants; this ensures that the kernel does not include facilities you are not actually using.

There are two other configuration constants that control optional kernel facilities: **INCLUDE_TASK_HOOKS** and **INCLUDE_CONSTANT_RDY_Q**. Define these constants in **configAll.h** if the application requires either kernel callouts (use of task hook routines) or a constant-insertion-time, priority-based ready queue. A ready queue with constant insert time allows the kernel to operate context switches with a fixed overhead regardless of the number of tasks in the system. Otherwise, the worst-case performance degrades linearly with the number of ready tasks in the system. Note that the constant-insert-time ready queue uses 2KB for the data structure; some systems do not have sufficient memory for this. In those cases, the definition of **INCLUDE_CONSTANT_RDY_Q** may be omitted, thus enabling use of a smaller (but less deterministic) ready queue mechanism.

Excluding Network Facilities

In some applications it may be appropriate to eliminate the VxWorks network facilities. For example, in the ROM-based systems or standalone configurations described in the *Tornado User's Guide: Cross-Development*, there may be no need for network facilities.

To exclude the network facilities, be sure the following constants are not defined:

- INCLUDE_NETWORK
- INCLUDE_NET_INIT
- INCLUDE_NET_SYM_TBL
- INCLUDE_NFS
- INCLUDE_RPC
- INCLUDE_RDB

To exclude the Remote Procedure Call library (RPC), undefine **INCLUDE_RPC**.

Option Dependencies

Option dependencies are coded in the file **src/config/usrDepend.c**, so that when a particular option is chosen, everything required is included. This assures you of a working system with minimum effort. Although you can exclude the features that you do not need by undefining them in **config.h** and **configAll.h**, you should be aware that in some cases they may not be excluded because of dependencies.

For example, you cannot use **telnet** without running the network. Therefore, if in your **configAll.h** file, the option **INCLUDE_TELNET** is selected but the option **INCLUDE_NET_INIT** is not, **usrDepend.c** defines **INCLUDE_NET_INIT** for you. Because the network initialization requires the network software, the **userDepend.c** file also defines **INCLUDE_NETWORK**.

Because most of the dependencies are taken care of in **usrDepend.c**, that file is currently included in **usrConfig.c**. This simplifies the build process and the selection of options. However, you can change or add dependencies if you choose.

8.4.2 Executing VxWorks from ROM

You can put VxWorks or a VxWorks-based application into ROM; this is discussed in the *Tornado User's Guide: Cross-Development*. For an example of a ROM-based VxWorks application, see the VxWorks boot ROM program. The file **config/all/bootConfig.c** is the configuration module for the boot ROM, replacing the file **usrConfig.c** provided for the default VxWorks development system.

In such ROM configurations, the *text* and *data* segments of the boot or VxWorks image are first copied into the system RAM, then the boot procedure or VxWorks executes in RAM. On some systems where memory is a scarce resource, it is possible to save space by copying only the data segment to RAM. The text segment remains in ROM and executes from that address space, and thus is termed *ROM resident*. The memory that was to be occupied by the text segment in RAM is now available for an application (up to 300KB for a standalone VxWorks system). Note that ROM-resident VxWorks is not supported on all boards; see your target's man page if you are not sure that your board supports this configuration.

The drawback of a ROM-resident text segment is the limited data widths and lower memory access time of the EPROM, which causes ROM-resident text to execute more slowly than if it was in RAM. This can sometimes be alleviated by using faster EPROM devices or by reconfiguring the standalone system to exclude unnecessary system features.

Aside from program text not being copied to RAM, the ROM-resident versions of the VxWorks boot ROMs and the standalone VxWorks system are identical to the conventional versions. A ROM-resident image is built with an uncompressed version of either the boot ROM or standalone VxWorks system image. VxWorks target makefiles include entries for building these images; see Table 8-3.

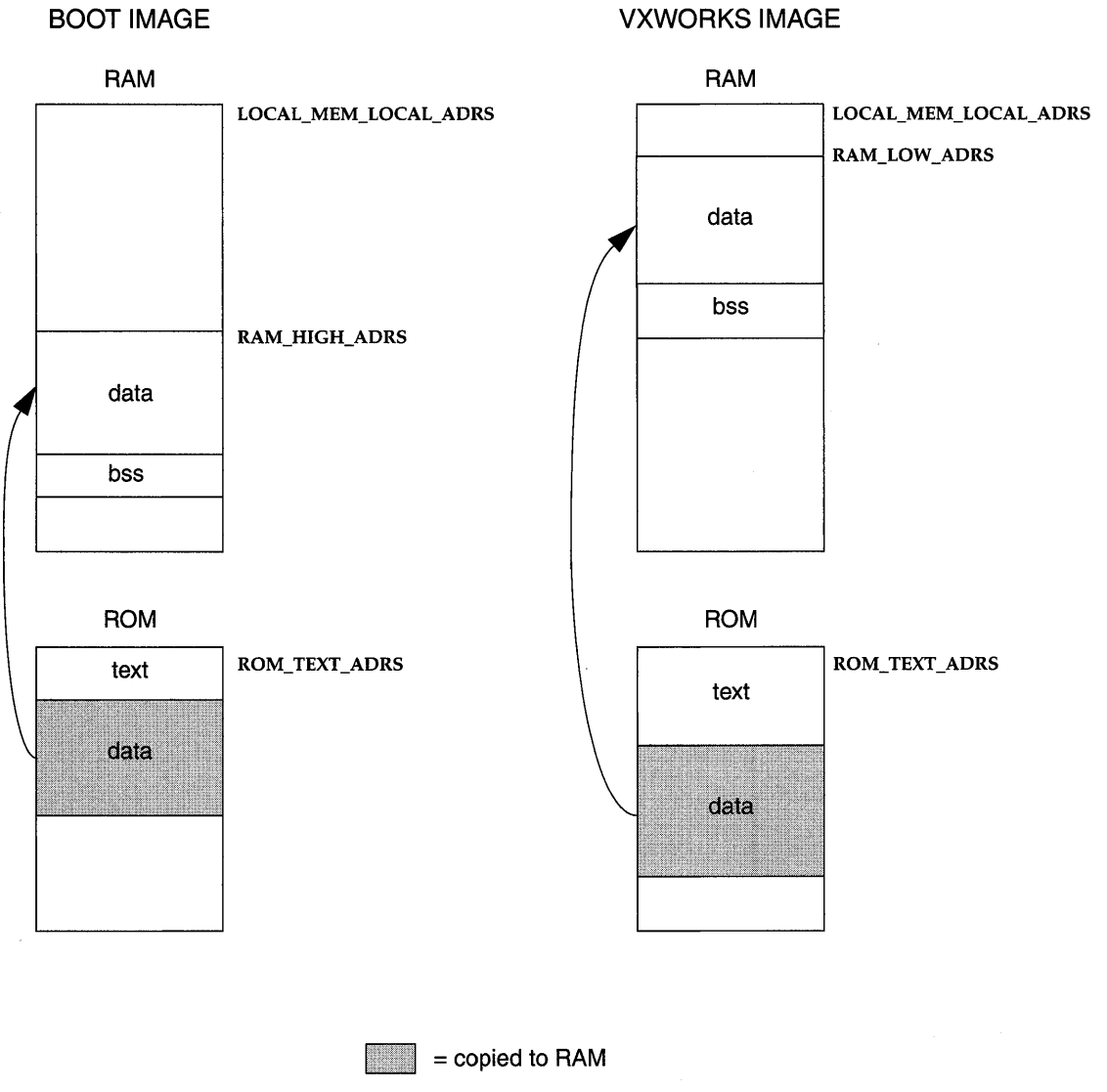
Table 8-3 **Makefile ROM-Resident Images**

Architecture	Image File*	Description
MIPS and PowerPC	**bootrom_res_high**	ROM-resident boot ROM image. The data segment is copied from ROM to RAM at address **RAM_HIGH_ADRS**.
	vxWorks.res_rom_res_low	ROM-resident standalone system image without compression. The data segment is copied from ROM to RAM at address **RAM_LOW_ADRS**.
	vxWorks.res_rom_nosym_res_low	ROM-resident standalone system image without compression or symbol table. Data segment is copied from ROM to RAM at address **RAM_LOW_ADRS**.
All Other Targets	**bootrom_res**	ROM-resident boot ROM image.
	vxWorks.res_rom	ROM-resident standalone system image without compression.
	vxWorks.res_rom_nosym	ROM-resident system image without compression or symbol table. Ideal for the Tornado environment.

* All images have a corresponding file in Motorola S-record or Intel Hex format with the same file name plus the extension **.hex**.

Because of the size of the system image, 512KB of EPROM is recommended for the ROM-resident version of the standalone VxWorks system. More space is probably required if applications are linked with the standalone VxWorks system. For a ROM-resident version of the boot ROM, 256KB of EPROM is recommended. If you use ROMs of a size other than the default, modify the value of **ROM_SIZE** in the target makefile and **config.h**.

Figure 8-1 **ROM-Resident Memory Layout**



A new make target, **vxWorks.res_rom_nosym**, has been created to provide a ROM-resident image without the symbol table. This is intended to be a standard ROM image for use with the Tornado environment where the symbol table resides on the host system. Being ROM-resident, the debug agent and VxWorks are ready almost immediately after power-up or restart.

The data segment of a ROM-resident standalone VxWorks system is loaded at **RAM_LOW_ADRS** (defined in the makefile) to minimize fragmentation. The data segment of ROM-resident boot ROMs is loaded at **RAM_HIGH_ADRS**, so that loading VxWorks does not overwrite the resident boot ROMs. For a CPU board with limited memory (under 1MB of RAM), make sure that **RAM_HIGH_ADRS** is less than **LOCAL_MEM_SIZE** by a margin sufficient to accommodate the data segment. Note that **RAM_HIGH_ADRS** is defined in both the makefile and **config.h**. These definitions *must* agree.

Figure 8-1 shows the memory layout for ROM-resident boot and VxWorks images. The lower portion of the diagram shows the layout for ROM; the upper portion shows the layout for RAM. **LOCAL_MEM_LOCAL_ADRS** is the starting address of RAM. For the boot image, the data segment gets copied into RAM above **RAM_HIGH_ADRS** (after space for *bss* is reserved). For the VxWorks image, the data segment gets copied into RAM above **RAM_LOW_ADRS** (after space for *bss* is reserved). Note that for both images the text segment remains in ROM.

8.4.3 Initialization Sequence for ROM-Based VxWorks

The early steps of system initialization are somewhat different for the ROM-based versions of VxWorks: on most target architectures, the two routines ***romInit*( )** and ***romStart*( )** execute instead of the usual VxWorks entry point, ***sysInit*( )**.

ROM Entry Point: *romInit*( )

At power-up the processor begins executing at ***romInit*( )** (defined in **config/***bspname*/**romInit.s**). The ***romInit*( )** routine disables interrupts, puts the boot type (cold/warm) on the stack, performs hardware-dependent initialization (such as clearing caches or enabling DRAM), and branches to ***romStart*( )**. The stack pointer is initialized to reside below the data section in the case of ROM-resident versions of VxWorks (in RAM versions, the stack pointer instead resides below the text section).

Copying the VxWorks Image: *romStart*( )

Next, the ***romStart*( )** routine (in **config/all/bootInit.c**) loads the VxWorks system image into RAM. If the ROM-resident version of VxWorks is selected, the data

segment is copied from ROM to RAM and memory is cleared. If VxWorks is not ROM resident, all of the text and code segment is copied and decompressed from ROM to RAM, to the location defined by **RAM_HIGH_ADRS** in **Makefile**. If VxWorks is neither ROM resident nor compressed, the entire text and data segment is copied without decompression straight to RAM, to the location defined by **RAM_LOW_ADRS** in **Makefile**.

Overall Initialization for ROM-Based VxWorks

Beyond ***romStart( )***, the initialization sequence for ROM-based VxWorks resembles the normal sequence, continuing with the ***usrInit( )*** call.

Table 8-4 summarizes the complete initialization sequence. For details on the steps after ***romInit( )*** and ***romStart( )***, see *8.3.4 VxWorks Initialization Timeline*, p.435.

Table 8-4 **ROM-Based VxWorks Initialization Sequence**

Routine	Activity	File
1. ***romInit( )***	(a) disable interrupts	**romInit.s**
	(b) save boot type (cold/warm)	
	(c) hardware-dependent initialization	
	(d) branch to ***romStart( )***	
2. ***romStart( )***	(a) copy data segment from ROM to RAM; clear memory	**bootInit.c**
	(b) copy code segment from ROM to RAM, decompressing if necessary	
	(c) invoke ***usrInit( )*** with boot type	
3. ***usrInit( )***	Initial routine.	**usrConfig.c**
4. ***usrKernelInit( )***	Routines invoked if the corresponding configuration constants are defined.	**usrKernel.c**
5. ***kernelInit( )***	Initialize and start the kernel.	**kernelLib.c**
6. ***usrRoot( )***	Initialize I/O system, install drivers, and create devices as configured in **configAll.h** and **config.h**.	**usrConfig.c**
7. Application routine	Application code.	Application source file

9 Target Shell

List of Tables

List of Figures

9.1 Introduction

In the Tornado development system, a full suite of development tools resides and executes on the host machine, thus conserving target memory and resources; see the *Tornado User's Guide* for details. However, a target-resident symbol table and module loader/unloader can be configured into the VxWorks system if necessary, for example, to create a dynamically configured run-time system. In this case, use the target-resident shell for development.

NOTE: If you choose to use the target-resident tools, you must use the target shell. The host tools cannot access the target-resident symbol table; thus symbols defined on the target are not visible to the host.

This chapter briefly describes these target-resident facilities.

9.2 Target-Resident Shell

For the most part, the target-resident shell works the same as the Tornado shell; for details, see the *Tornado User's Guide: Shell*. However, there are some differences, which are described in this section.

9.2.1 Creating the Target Shell

To create the target shell, you must configure it into the VxWorks configuration by defining the **INCLUDE_SHELL** macro (for details, see *8.3 Configuring VxWorks*, p.430). When you do so, ***usrRoot( )*** (in **usrConfig.c**) spawns the target shell task by calling ***shellInit( )***. The first argument to ***shellInit( )*** specifies the target shell's stack

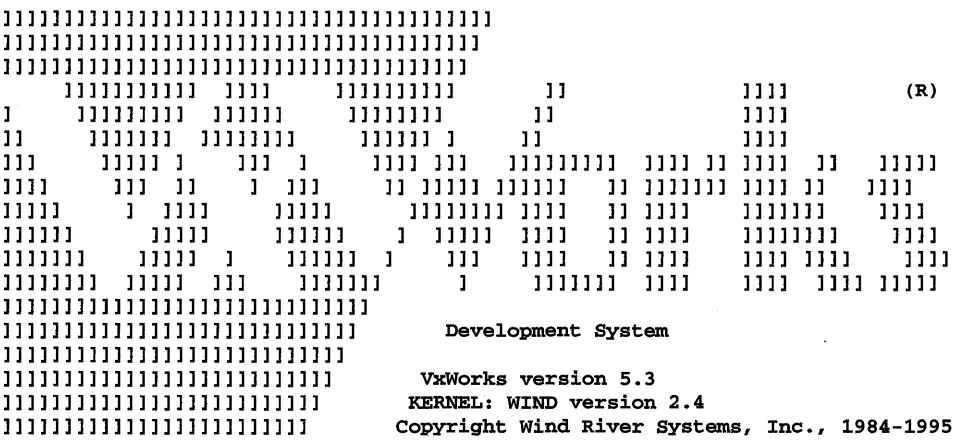
size, which must be large enough to accommodate any routines you call from the target shell. The second argument is a boolean that specifies whether the target shell's input is from an interactive source (TRUE), or a non-interactive source (FALSE) such as a script file. If the source is interactive, then the shell prompts for commands but does not echo them to standard out; the reverse is true if the source is non-interactive.

The shell task (**tShell**) is created with the **VX_UNBREAKABLE** option; therefore, breakpoints cannot be set in this tasks, because a breakpoint in the shell would make it impossible for the user to interact with the system. Any routine or task that is invoked from the target shell, rather than spawned, runs in the **tShell** context.

Only one target shell can run on a VxWorks system at a time; the target shell parser is not reentrant, because it is implemented using the UNIX tool **yacc**.

When the shell is started, the banner displayed in Figure 9-1 appears.

Figure 9-1 **Typical Target Shell Sign-on Banner**



9.2.2 Spawning an Application Instead of the Target Shell

If you undefine **INCLUDE_SHELL** and define **INCLUDE_DEMO** in your VxWorks configuration, then instead of spawning the target shell task, ***usrRoot( )*** spawns the **demo** task. This program serves as an example for initializing bootable applications: it loops forever, prompting for a string and echoing it. If the string is "0" or "1", the demo displays various memory statistics.

To spawn your application instead of the demo program, insert the initialization of your application after the conditional code to start the demo. For example:

```
/* spawn demo if selected */
#if defined(INCLUDE_DEMO)
    taskSpawn ("demo", 20, 0, 2000, (FUNCPTR)usrDemo, 0,0,0,0,0,0,0,0,0,0);
#endif

☛  taskSpawn ("myMod", 100, 0, 20000, (FUNCPTR)myModEntryPt, 0,0,0,0,0,0,0,0,0,0);
```

For more information, see the *Tornado User's Guide: Cross-Development*.

9.2.3 Using the Target Shell

The target shell works almost exactly like the Tornado shell; see the *Tornado User's Guide: Shell* and the **usrLib** reference entry for details. You can also type the following command to display help:

```
-> help
```

The following target shell command lists all the available help routines:

```
-> lkup "Help"
```

The target shell has its own set of terminal-control characters, unlike the Tornado shell, which inherits its setting from the host window from which it was invoked. Table 9-1 lists the target shell's terminal-control characters. The first four of these are defaults that can be mapped to different keys using routines in **tyLib** (see also *Tty Special Characters*, p.133).

The shell line-editing commands are the same as they are for the Tornado shell. For a summary of the commands, see the *Tornado User's Guide: Shell*.

9.2.4 Debugging with the Target Shell

The target shell includes the same debugging utilities as the Tornado shell, if **INCLUDE_DEBUG** is defined in the VxWorks configuration. For details, see the *Tornado User's Guide: Shell* and the reference entry for **dbgLib**.

Table 9-1 **Target Shell Terminal Control Characters**

Command	Description
CTRL+H	Delete a character (backspace).
CTRL+U	Delete an entire line.
CTRL+C	Abort and restart the shell.
CTRL+X	Reboot (trap to the ROM monitor).
CTRL+S	Temporarily suspend output.
CTRL+Q	Resume output.
ESC	Toggle between input mode and edit mode.

In order to use the CrossWind host debugger with the target shell, the RDB daemon must be started by defining **INCLUDE_RDB** in the VxWorks configuration. This starts the **tRdbTask** daemon, which services RPC requests made by remote source-level debuggers.

9.2.5 Aborting the Target Shell

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine may loop excessively, suspend, or wait on a semaphore. This may happen as the result of errors in arguments specified in the invocation, errors in the implementation of the routine itself, or simply oversight as to the consequences of calling the routine at all.

In such cases it is usually possible to abort and restart the target shell task. This is done by pressing the special target-shell abort character on the keyboard, **CTRL+C** by default. This causes the target shell task to restart execution at its original entry point. Note that the abort key can be changed to a character other than **CTRL+C** by calling ***tyAbortSet( )***.

When restarted, the target shell automatically reassigns the system standard input and output streams to the original assignments they had when the target shell was first spawned. Thus any target shell redirections are canceled, and any executing shell scripts are aborted.

The abort facility works only if the following are true:

- ***dbgInit( )*** has been called (see *9.2.4 Debugging with the Target Shell*, p.459).

- ***excTask*( )** is running (see *Installation of Exception Handling and Logging*, p.441).
- The driver for the particular keyboard device supports it (all VxWorks-supplied drivers do).
- The device's abort option is enabled. This is done with an ***ioctl*( )** call, usually in the root task in **usrConfig.c**. For information on enabling the target shell abort character, see *Tty Options*, p.132.

Also, you may occasionally enter an expression that causes the target shell to incur a fatal error such as a bus/address error or a privilege violation. Such errors normally result in the suspension of the offending task, which allows further debugging.

However, when such an error is incurred by the target shell task, VxWorks automatically restarts the target shell, because further debugging is impossible without it. Note that for this reason, as well as to allow the use of breakpoints and single-stepping, it is often useful when debugging to spawn a routine as a task instead of just calling it directly from the target shell.

When the target shell is aborted for any reason, either because of a fatal error or because it is aborted from the terminal, a task trace is displayed automatically. This trace shows where the target shell was executing when it died.

Note that an offending routine can leave portions of the system in a state that may not be cleared when the target shell is aborted. For instance, the target shell might have taken a semaphore, which cannot be given automatically as part of the abort.

9.2.6 Remote Login to the Target Shell

Remote Login From Host: telnet *and* rlogin

When VxWorks is first booted, the target shell's terminal is normally the system console. You can use **telnet** to access the target shell from a host over the network if the constant **INCLUDE_TELNET** is defined in your VxWorks configuration (see *8.3 Configuring VxWorks*, p.430). Defining **INCLUDE_TELNET** creates the **tTelnetd** task. To do so, enter the following command from the host (*targetname* is the name of the target VxWorks system):

```
% telnet "targetname"
```

UNIX host systems also use **rlogin** to provide access to the target shell from the host. Define **INCLUDE_RLOGIN** in your VxWorks configuration to create the

tRlogind task. However, note that VxWorks does not support **telnet** or **rlogin** access from the VxWorks system to the host.

A message is printed on the system console indicating that the target shell is being accessed via **telnet** or **rlogin**, and that it is no longer available from its console.

If the target shell is being accessed remotely, typing at the system console has no effect. The target shell is a single-user system—it allows access either from the system console or from a single remote login session, but not both simultaneously. To prevent someone from remotely logging in while you are at the console, use the routine ***shellLock*( )** as follows:

```
-> shellLock 1
```

To make the target shell available again to remote login, enter the following:

```
-> shellLock 0
```

To end a remote-login target shell session, call ***logout*( )** from the target shell. To end an **rlogin** session, type **TILDE** and **DOT** as the only characters on a line:

```
-> ~.
```

Remote Login Security

You can be prompted to enter a login user name and password when accessing VxWorks remotely:

```
VxWorks login: user_name
Password: password
```

The remote-login security feature is enabled by defining **INCLUDE_SECURITY** in the VxWorks configuration. The default login user name and password provided with the supplied system image is *target* and *password*. You can change the user name and password with the ***loginUserAdd*( )** routine, as follows:

```
-> loginUserAdd "fred", "encrypted_password"
```

To obtain *encrypted_password*, use the tool **vxencrypt** on the host system. This tool prompts you to enter your password, and then displays the encrypted version.

To define a group of login names, include a list of ***loginUserAdd*( )** commands in a startup script and run the script after the system has been booted. Or include the list of ***loginUserAdd*( )** commands to the file **usrConfig.c**, then rebuild VxWorks.

The values for the user name and password apply only to remote login into the VxWorks system. They do not affect network access from VxWorks to a remote system; See *5.3.5 Remote Login from VxWorks to the Host: rlogin( )*, p.292.

The remote-login security feature can be disabled at boot time by specifying the flag bit 0x20 (**SYSFLAG_NO_SECURITY**) in the *flags* parameter on the boot line (see the *Tornado User's Guide: Getting Started*). This feature can also be disabled by undefining **INCLUDE_SECURITY** in the VxWorks configuration.

9.2.7 Summary of Target and Host Shell Differences

For details on the Tornado shell, see the *Tornado User's Guide: Shell.* The following is a summary of the differences between it and the target shell:

- Both shells contain a C interpreter, which allows C-shell and **vi** editing facilities. However, the Tornado shell also provides a Tcl interpreter.
- You can have multiple Tornado shells active for any given target; only one target shell can be active for a target at any one time.
- The Tornado shell allows virtual I/O; the target shell does not.
- The target shell does not have a GNU C++ demangler; it is necessary to use the target tools when C++ demangling is required.
- The Tornado shell is always ready to execute. The target shell, as well as its associated target-resident symbol table and module loader/unloader, must be configured into the VxWorks image by defining constants in **configAll.h** or **config.h** (discussed throughout this chapter).
- Because the target shell is often started from the system console, the standard input and output are directed to the same window. For the Tornado shell, these standard I/O streams are not necessarily directed to the same window as the Tornado shell. For details, see the *Tornado User's Guide: Shell.*
- The Tornado shell can perform many control and information functions entirely on the host without consuming target resources.
- The Tornado shell uses host resources for most functions so that it remains segregated from the target. This means that the Tornado shell can operate on the target from the outside. The target shell, on the other hand, must act on itself. This means that there are limitations to what the target shell can do (for example, while debugging it cannot set breakpoints on itself or on routines it calls). Also, conflicts in priority may occur while using the target shell.
- When the target shell encounters a string literal ("...") in an expression, it allocates space for the string including the null-byte string terminator. The value of the literal is the address of the string in the newly allocated storage. For example, the following expression allocates 12 bytes from the target

memory pool, enters the string in those 12 bytes (including the null terminator), and assigns the address of the string to **x**:

```
-> x = "hello there"
```

The following expression can be used to return those 12 bytes to the target memory pool (see the **memLib** reference entry for information on memory management):

```
-> free (x)
```

Furthermore, even when a string literal is not assigned to a symbol, memory is still permanently allocated for it. For example, the following expression uses 12 bytes of memory that are never freed:

```
-> printf ("hello there")
```

This is because if strings were only temporarily allocated, and a string literal were passed to a routine being spawned as a task, then by the time the task executed and attempted to access the string, the target shell would have already released (and possibly even reused) the temporary storage where the string was held.

After extended development sessions with the target shell, the cumulative memory used for strings may be noticeable. If this becomes a problem, you must reboot your target. Because the Tornado shell has access to a host-controlled target memory pool, this memory leak never occurs.

9.3 Other Target-Resident Facilities

9.3.1 Target Symbol Table, Module Loader, and Module Unloader

To make full use of the target shell's features, you should also define the target symbol table, as well as the target module loader and unloader. Use the following macros in the VxWorks configuration (see *8.3 Configuring VxWorks*, p.430 for configuration information):

- **INCLUDE_SYM_TBL** for target symbol table support, plus one of the following:
  - **INCLUDE_NET_SYM_TBL** to load the symbol table from the network (**vxWorks.sym**; you will also need to separately load **vxWorks**)

  - INCLUDE_STANDALONE_SYM_TBL to build a VxWorks image that includes the target symbol table (**vxWorks.st**)
- INCLUDE_LOADER
- INCLUDE_UNLOADER

If the target symbol table is included, ***usrRoot*( )** runs ***hashLibInit*( )** and ***symLibInit*( )** to initialize the corresponding libraries. The target symbol table is created by calling ***symTblCreate*( )**. For convenience during debugging (see *9.2.4 Debugging with the Target Shell*, p.459), it is most useful to have access to all symbols in the system. On the other hand, a production version of a system can be built that does not require the target symbol table, if (for example) memory resources are constrained.

The ***symTblCreate*( )** call creates an empty target symbol table. VxWorks system facilities are not accessible through the target shell until the symbol definitions for the booted VxWorks system are entered into the target symbol table. This is done by reading the target symbol table from a file called **vxWorks.sym** in the same directory from which **vxWorks** was loaded (**config/***bspname*). This file contains an object module that consists only of a target symbol table section containing the symbol definitions for all the variables and routines in the booted system module. It has zero-length (empty) code, data, and relocation sections. Nonetheless, it is a legitimate object module in the standard object module format.

The symbols in **vxWorks.sym** are entered in the target symbol table by calling ***loadSymTbl*( )** (whose source is in **src/config/usrLoadSym.c**). This routine uses the target-resident module loader to load symbols from **vxWorks.sym** into the target symbol table.

For the most part, the target-resident facilities work the same as their Tornado host counterparts; see the *Tornado User's Guide: Cross-Development*. However, as stated earlier, the target-resident facilities can be useful if you are building dynamically configured applications. For example, with the target-resident loader, you can load from a target disk as well as over the network, with these caveats: If you use the target-resident loader to load a module over the network (as opposed to loading from a target-system disk), the amount of memory required to load an object module depends on what kind of access is available to the remote file system over the network. Loading a file that is mounted over the default network driver requires enough memory to hold two copies of the file simultaneously. First, the entire file is copied to a buffer in local memory when opened; second, the file resides in memory when it is linked to VxWorks. On the other hand, loading an object module from a host file system mounted through NFS only requires enough memory for one copy of the file (plus a small amount of overhead). In any case, however, using the target-resident loader takes away additional memory from

your application—most significantly for the target-resident symbol table required by the target-resident loader.

For information on the target-resident module loader, unloader, and symbol table, see the **loadLib**, **unldLib**, and **symLib** reference entries.

9.3.2 Show Routines

VxWorks includes system information routines which print pertinent system status on the specified object or service; however, they show only a snapshot of the system service at the time of the call and may not reflect the current state of the system. To use these routines, you must define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. Configuration*. When you invoke them, their output is sent to the standard output device. Table 9-2 lists commonly called system show routines.

Table 9-2 **Show Routines**

Call	Description
envShow( )	Display the environment for a given task on *stdout*
memPartShow( )	Show the partition blocks and statistics
memShow( )	System memory show routine
moduleShow( )	Show statistics for all loaded modules
msgQShow( )	Message queue show utility (both POSIX and *wind*)
semShow( )	Semaphore show utility (both POSIX and *wind*)
show( )	Generic object show utility
stdioShow( )	Standard I/O file pointer show utility
taskSwitchHookShow( )	Show the list of task switch routines
taskCreateHookShow( )	Show the list of task create routines
taskDeleteHookShow( )	Show the list of task delete routines
taskShow( )	Display the contents of a task control block
wdShow( )	Watchdog show utility

An alternative method of viewing system information is the Tornado browser, which can be configured to update system information periodically. For information on this tool, see the *Tornado User's Guide: The Tornado Browser.*

VxWorks also includes several network information routines. These routines are initialized by defining **INCLUDE_NET_SHOW** in your VxWorks configuration; see *8. Configuration.* Table 9-3 lists commonly called network show routines.

Table 9-3 **Network Show Routines**

Call	Description
***hostShow*( )**	Display the host table
***ifShow*( )**	Display the attached network interfaces
***routeShow*( )**	Display host and hetwork routing tables

10

C++ Development

Basic Support and the Optional Component Wind Foundation Classes

List of Tables

List of Figures

List of Examples

10.1 Introduction

In the Tornado environment, C++ development support consists of the GNU C++ compilation, run-time support, and the Iostreams class library. In addition, Wind River Systems offers an optional product, the Wind Foundation Classes, providing several class libraries to extend VxWorks functionality.

This chapter discusses basic application development using C++ and provides references to relevant information in other Wind River documentation. In addition, the Iostreams library and the Wind Foundation Classes are documented here.

The Iostreams library provides support for formatted I/O in C++. The C++ language definition (like C) does not include special input and output statements, relying instead on standard library facilities. The Iostreams library provides C++ capabilities analogous to the C functions offered by the *stdio* library. The principal differences are that the Iostreams library gives you enhanced type security and can be extended to support your own class definitions.

The Wind Foundation Classes consist of a group of libraries (some of which are industry standard) that provide a broad range of C++ classes to extend VxWorks functionality in several important ways. They are called *Foundation* classes because they provide basic services which are fundamental to many programming tasks, and which can be used in almost every application domain. For information about how to install the Wind Foundation Classes, see the *Wind River Products Installation Guide*.

The Wind Foundation Classes consist of the following libraries:

- VxWorks Wrapper Class library
- Tools.h++ library from Rogue Wave Software
- Booch Components library from Rogue Wave Software

10.2 C++ Development Under Tornado

Basic C++ support is bundled with the Tornado development environment. VxWorks provides header files containing C++ safe declarations for all routines and the necessary run-time support. The standard Tornado interactive development tools such as the debugger, the shell, and the incremental loader include C++ support.

10.2.1 Tools Support

WindSh

Tornado supports both C and C++ as development languages. WindSh can interpret simple C++ expressions. To exercise C++ facilities that are missing from the C-expression interpreter, you can compile and download routines that encapsulate the special C++ syntax. See the *Tornado User's Guide: Tornado Tools Reference* for WindSh C++ options.

Demangling

When C++ functions are compiled, the class membership (if any) and the type and number of the function's arguments are encoded in the function's linkage name. This is called *name mangling* or *mangling*. The debugging and system information routines in WindSh can print C++ function names in demangled or mangled representations.

The default representation is **gnu**. In addition, **arm** and **none** (no demangling) are available options. To select an alternate mode, modify the Tcl variable **shDemangleStyle**. For instance:

```
-> ?set shDemangleStyle none
```

Overloaded Function Names

When you invoke an overloaded function, WindSh prints the matching functions' signatures and prompts you for the desired function. For more information on how WindSh handles overloaded function names, including an example, see the *Tornado User's Guide: Shell*.

CrossWind

CrossWind supports debugging C++ templates, stepping through constructors, and other facilities for debugging C++ applications. For details, see the *Tornado User's Guide: Tornado Tools Reference* and *Debugging with GDB*.

10.2.2 Programming Issues

Static Constructors

Munching

After compilation, you must *munch* the generated binary to provide VxWorks with the information needed to call static constructors and destructors. Munching is the process of scanning an object module for your application's static objects, and generating data structures that VxWorks can use to call the objects' constructors and destructors. The details are described in the *Tornado User's Guide: Cross-Development*.

Calling Strategy

The default Tornado behavior for handling C++ static constructors in incrementally loaded modules is to call them automatically as a side effect of loading. This means that ***cplusCtors( )*** is called automatically when **INCLUDE_CPLUS** or **INCLUDE_CPLUS_MIN** is defined. It also means that ***cplusDtors( )*** is called automatically as a side effect of unloading. In addition, VxWorks automatically calls static constructors for modules linked with VxWorks when **INCLUDE_CPLUS** or **INCLUDE_CPLUS_MIN** is defined.

To change the default strategy to manual, use ***cplusXtorSet( )***. Under the manual mode, static constructors and destructors are called as a result of invoking ***cplusCtors( )*** and ***cplusDtors( )*** by hand. The manual mode can be used with no argument, to invoke all currently loaded static constructors or destructors or it can be used to call static constructors and destructors explicitly on a module-by-module basis.

Template Instantiation

In general, C++ toolchains that support templates permit templates to be instantiated either at compile time (using explicit instantiation) or at link time

(using implicit instantiation). The GNU compiler supports both methods. However, explicit instantiation is easier to understand and use. With explicit instantiation, it is clear when and where your templates are instantiated, and it is simpler to control the instantiation process, especially in the context of the incremental development methodology supported by Tornado. For more information on compiler support for template instantiation, see the *GNU ToolKit User's Guide*.

To instantiate templates explicitly, your client code (the code that creates the template instantiations) must include header and source files for the templates that you want to instantiate. This is followed by a declaration of a template instantiation. See Example 10-3.

The C++ compiler instantiates the specified templates. The instantiations occur in the module containing the **template class** declaration. Bear this in mind to control the location of your template instantiations, and to avoid redundant or duplicate instantiation of any given template.

Application Size

If application size is an issue, you can link your application with the VxWorks archive located in **lib/lib***cpu***gnuvx.a** rather than using **INCLUDE_TOOLS**. This causes only the necessary modules to be included rather than all Tools.h++ modules.

Header Files

Wind River Systems header files are C++ safe and prototype all VxWorks C API functions to have C linkage (extern "C") when used with C++.

10.2.3 Compiling C++ Applications

For general information on how to compile applications for VxWorks, see the *Tornado User's Guide: Cross-Development*. For more details on the GNU compiler and on the associated tools, see the *GNU ToolKit User's Guide*.

When compiling C++ modules with the GNU compiler, invoke **cc***arch* (just as for C source) on any source file with a C++ suffix (such as **.cpp**). Compiling C++ applications in the VxWorks environment involves the following steps:

1. C++ source code is compiled into object code for a specific target architecture, just as for C applications. In addition, an object containing the data structures from the C++ source is created.
2. The new object module is munched.
3. The munched object is compiled using the C compiler with the **-traditional** flag.
4. The static linker links the compiled data structures to the original object module.

```
% ccarch -fno-builtin -I ${WIND_BASE}/target/h -nostdinc -O2 \
        -mcpu -DCPU=cpu -r foo.cpp bar.cpp baz.cpp -o foobarbaz.o
% nmarch foobarbaz.o | muncharch > __ctordtor.c
% ccarch -traditional -mcpu -c __ctordtor.c
% ldarch -r -o foobarbaz.out __ctordtor.o foobarbaz.o
```


NOTE: If you use a Wind River Systems makefile to build your application, munching is handled by **make**.

WARNING: In the linking step, **-r** is used to specify partial linking. A partially linked file is still relocatable, and is suitable for downloading and linking using the VxWorks module loader. The *GNU ToolKit User's Guide: Using ld* describes a **-Ur** option for resolving references to C++ constructors. That option is for native development, not for cross-development. Do not use **-Ur** with C++ modules for VxWorks.

10.2.4 Configuration Constants

To include C++ support in VxWorks, define one of the following constants in the header file **configAll.h** or **config.h**:

INCLUDE_CPLUS
: Includes all basic C++ run-time support in VxWorks. This enables you to download and run compiled and munched C++ modules. It does not configure any of the Wind Foundation Class libraries into VxWorks.

INCLUDE_CPLUS_MIN
: Includes only the C++ run-time support that is explicitly referenced in the static link of the VxWorks system image.

To include Iostreams, define the following constant in the header file **configAll.h**:

INCLUDE_CPLUS_IOSTREAMS
: Includes the Iostreams class library.

To include one or more of the Wind Foundation Classes, define one or more of the following constants in the header file **configAll.h** or **config.h**:

INCLUDE_CPLUS_VXW
Includes the VxWorks Wrapper Class library.

INCLUDE_CPLUS_TOOLS
Includes Rogue Wave's Tools.h++ class library.

INCLUDE_CPLUS_BOOCH
Includes Rogue Wave's Booch Components class library.

For more information on configuring VxWorks, see *8. Configuration*.

10.3 Iostreams Library

This library is configured into VxWorks with the **INCLUDE_CPLUS_IOSTREAMS** constant; see *10.2.4 Configuration Constants*, p.475. If you use Wind River makefiles, you do not have to worry about munching VxWorks.

The Iostreams library header files reside in the standard VxWorks header file directory, **target/h**. To use this library, include one or more of the header files after the **vxWorks.h** header in the appropriate modules of your application. The most frequently used header file is **iostream.h**, but others are available; see the *AT&T C++ Language System Library Manual* for information.

The standard Iostreams objects (**cin**, **cout**, **cerr**, and **clog**) are global: that is, they are not private to any given task. They are correctly initialized regardless of the number of tasks or modules that reference them, but their member functions do not interlock access when used concurrently by multiple tasks. The responsibility for mutual exclusion rests with the application.

The effect of private standard Iostreams objects can be simulated by creating a new Iostreams object of the same class as the standard Iostreams object (for example, **cin** is an **istream_withassign**), and assigning to it a new **filebuf** object tied to the appropriate file descriptor. The new **filebuf** and Iostreams objects are private to the calling task, ensuring that no other task can accidentally corrupt them.

Consult the *AT&T C++ Language System Library Manual* for general reference information on Iostreams.

10.4 Wind Foundation Classes

The Wind Foundation Classes include three libraries:

- VxWorks Wrapper Class library
- Tools.h++ library from Rogue Wave Software
- Booch Components library from Rogue Wave Software

The VxWorks Wrapper Class library provides a thin C++ interface to several standard VxWorks modules. The Tools.h++ foundation class library from Rogue Wave Software supports a variety of C++ features. The Booch Components library from Rogue Wave provides a collection of domain-independent data structures and algorithms.

NOTE: In order to prevent dependency conflicts between VxWorks libraries and Rogue Wave libraries, all VxWorks libraries, including the VxWorks Wrapper Class Library, should be included before all Rogue Wave libraries, including both the Tools.h++ and Booch Components libraries.

10.4.1 VxWorks Wrapper Class Library

The classes in this library are called *wrapper* classes because each class encapsulates, or *wraps*, the interfaces for some portion of standard VxWorks functionality. Define the **INCLUDE_CPLUS_VXW** constant to configure this library into VxWorks; see *10.2.4 Configuration Constants*, p.475.

The VxWorks Wrapper Class library header files reside in the standard VxWorks header file directory, **target/h**. The classes and their corresponding header files are shown in Table 10-1. To use one of these classes, include the corresponding header file in the appropriate modules of your application.

Table 10-1 **Header Files for VxWorks Wrapper Classes**

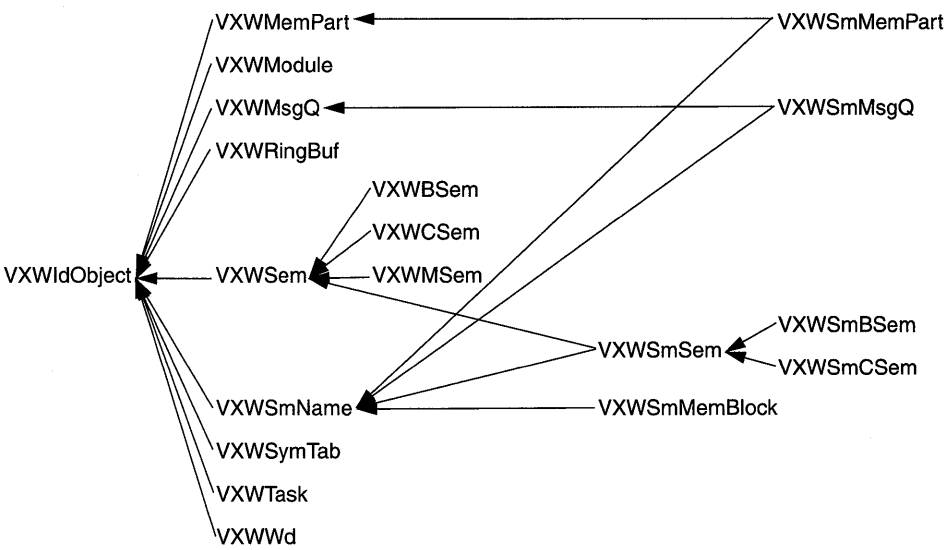
Header File	Description
vxwLoadLib.h	Object module loader and unloader (wraps **loadLib**, **unldLib**, **moduleLib**)
vxwLstLib.h	Linked lists (wraps **lstLib**)
vxwMemPartLib.h	Memory partitions (wraps **memLib**)
vxwMsgQLib.h	Message queues (wraps **msgQLib**)
vxwRngLib.h	Ring buffers (wraps **rngLib**)

Table 10-1 **Header Files for VxWorks Wrapper Classes** *(Continued)*

Header File	Description
vxwSemLib.h	Semaphores (wraps **semLib**)
vxwSmLib.h	Shared memory objects (adds support for shared memory semaphores, message queues, and memory partitions)
vxwSymLib.h	Symbol tables (wraps **symLib**)
vxwTaskLib.h	Tasks (wraps **taskLib**, **envLib**, **errnoLib**, **sigLib**, and **taskVarLib**)
vxwWdLib.h	Watchdog timers (wraps **wdLib**)

The VxWorks Wrapper Classes are designed to provide C++ language bindings to VxWorks modules that are inherently object-oriented, but for which only C bindings have previously been available. Figure 10-1 shows the inheritance relationships for all of the VxWorks Wrapper Classes. The classes are named to correspond with the VxWorks features that they wrap. For example, **VXWMsgQ** is the class of message queues, and provides a C++ interface to **msgQLib**.

Figure 10-1 **Wrapper-Class Inheritance**



NOTE: The classes **VXWError** and **VXWIdObject** are used internally by the VxWorks Wrapper Classes. They are listed in Figure 10-1 for completeness only. These two classes are not intended for direct use by applications.

Example 10-1 **Watchdog Timers**

To illustrate the way in which the wrapper classes provide C++ language bindings for VxWorks objects, the following example exhibits methods in the watchdog timer class, **VXWWd**. See *2.6 Watchdog Timers*, p.99 for general information about watchdog timers.

```
/* Create a watchdog timer and set it to go off in 3 seconds. */

/* includes */

#include "vxWorks.h"
#include "logLib.h"
#include "vxwWdLib.h"

/* defines */

#define  SECONDS (3)

task (void)
    {
    /* Create watchdog */

[1]     VXWWd myWatchDog;

    /* Set timer to go off in SECONDS - printing a message to stdout */

[2]     if (myWatchDog.start (sysClkRateGet( ) * SECONDS, logMsg,
                int ("Watchdog timer just expired\n")) == ERROR)
        return (ERROR);

    while (TIMER_NEEDED)
        {
        /* ... */
        }
[3]     }
```

A notable difference from the C interface is that the wrapper classes allow you to manipulate watchdog timers as objects rather than through an object ID. Line [1] creates and names a watchdog object; C++ automatically calls the **VXWWd** constructor, implicitly invoking the C routine ***wdCreate*( )** to create a watchdog timer.

Line [2] in the example illustrates how to use a method from the wrapper classes. The example invokes the method ***start*( )** for the instance **myWatchDog** of the class **VXWWd** to call the timer. Because this method is invoked on a specific object, the

argument list for the method ***start( )*** does not require an argument to identify which timer to start (unlike ***wdStart( )***, the corresponding C routine).

Finally, because **myWatchDog** is a local object, exiting from the routine ***task( )*** on line [3] automatically calls the destructor for the **VXWWd** watchdog class. This implicit call to the destructor deallocates the watchdog object, and if the timer was still running removes it from the system timer queues. Thus, for objects declared on the stack, it is not necessary to call a routine equivalent to the C routine ***wdDelete( )***. (However, if an object is created dynamically with the operator **new**, you must delete it explicitly with the operator **delete**, once your application no longer needs the object.)

For details of the wrapper classes and on each of the wrapper class functions, see the *VxWorks Reference Manual*.

10.4.2 Tools.h++ Library

Tools.h++ is an industry-standard foundation class library from Rogue Wave Software which supports the following features:

- A complete set of collection classes
- Template based classes
- Persistent store facility
- File classes and file space manager
- B-tree disk retrieval
- Multi-thread safety
- Multi-byte and wide character strings
- Localized string collation
- Parse and format times, dates, and currency in multiple locales
- Support for multiple time zones and daylight savings rules
- Support for localized messages
- Localized I/O streams

This library is configured into VxWorks with the **INCLUDE_CPLUS_TOOLS** constant; see *10.2.4 Configuration Constants*, p.475.

The Tools.h++ library header files reside in the VxWorks header file directory **h/rw**. To use this library, **#include** one or more of these header files after the **#include "vxWorks.h"** statement and after the **#include** statements for all other VxWorks libraries in the appropriate modules of your application. For a list of all the header files and details on this library, see Rogue Wave's *Tools.h++ Introduction and Reference Manual*.

10.4.3 Booch Components Library

The Booch Components library from Rogue Wave provides a collection of domain-independent data structures (such as graphs, queues, rings, and stacks) and algorithms (such as date/time, searching, and sorting). The library represents an application of the Booch object-oriented analysis and design method.

This library is configured into VxWorks with the **INCLUDE_CPLUS_BOOCH** constant; see *10.2.4 Configuration Constants*, p.475.

The Booch Components library header files reside in the VxWorks header file directory, **src/cplus/booch/CppBooch/Include**. To use this library, **#include** one or more of the header files after the **#include "vxWorks.h"** statement and after the **#include** statements for all other VxWorks libraries in the appropriate modules of your application. For a list of all the header files and details on this library, see Rogue Wave's *C++ Booch Components Class Catalog: C++ Class Library for Multithreading and Storage Management*.

Booch Components Source Code

The Booch Components are almost exclusively template-based. The few non-template classes are not used by all client programs. For these reasons, the Booch Components are delivered in source form only, and there is no overall makefile for building the components.

The source code for the Booch Components is located in the VxWorks directory **src/cplus/booch/CppBooch**. Details of the directory structure subordinate to **CppBooch** can be found in the *C++ Booch Components Class Catalog*, which is shipped with Wind Foundation Classes. The most important directory to clients of the Booch Components is **CppBooch/Include**. This directory contains copies of, or links to (depending on your host platform), all of the Booch Components C++ source and header files contained elsewhere below **CppBooch**. Thus, **CppBooch/Include** is normally the only directory that you will need to refer to directly in developing applications that use the Booch Components.

Building Booch Components Applications

Due to its heavy dependence on template classes, building applications that use the Booch Components is slightly more complicated than building C++ applications that do not use templates. For more information, refer to *Template Instantiation*, p.473. For information specific to template use with the GNU

compiler, see *Using GNU CC: Extensions to the C++ Language* in the *GNU ToolKit User's Guide*.

The following section illustrates the complete process of building an application with the Booch Components.

Booch Components Examples

Several examples are included with the Booch Components. These can be found in the directory **src/cplus/booch/CppBooch/Examples**. Source code for the examples is located in the **Tests/Source** subdirectory. The remaining subdirectories contain files documented in the *C++ Booch Components Class Catalog*. The remainder of this section refers only to files in the **Source** subdirectory.

The remainder of this section spells out all the steps to adapt the **BagT** example to VxWorks, running on the mv147 target. The **BagT** example is distributed ready to build, with these adaptations in place for an mv147; it is straightforward to adapt it to other supported BSPs. The **BagT** adaptation also provides a pattern you can imitate to adapt other Booch Components examples to VxWorks. There are three parts to the adaptation. The following sections describe them in detail:

- Run-time adaptation
- Makefile
- Template instantiation

Run-Time Adaptation

The first step is accomplished by simply inserting the following at the start of each source module:

```
#include "vxWorks.h"
```

This configures the other header files, included after **vxWorks.h**, with options specific to VxWorks and to the selected architecture.

Makefile

Example 10-2 shows a makefile that can build the downloadable VxWorks application module **BagT.out**. This makefile is distributed in **CppBooch/Examples/Tests/Source**. You must adapt this makefile if your target has a different CPU than the MC68040 used in the example, or if you are not using the GNU compiler. For supported compilers and architectures, the only adaptation required is to define a different CPU in the first line.

Build the **BagT.out** module by invoking make, specifying **BagT.out** in the make command. Use the following command[1]:

```
% make BagT.out
```

Example 10-2 **Makefile for BagT Example from Booch Components**

```
[1] CPU      = MC68040
    TOOL     = gnu

[2] include $(WIND_BASE)/target/h/make/defs.bsp
    include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)
    include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)
    include $(WIND_BASE)/target/h/make/rules.bsp
[3] BCINC = $(WIND_BASE)/target/src/cplus/booch/CppBooch/Include
[4] EXTRA_INCLUDE = -I$(BCINC) -I$(WIND_BASE)/target/h

[5] BagT.out : Items.o BagT.o

    Items.o : Items.cpp $(BCINC)/BCType.h Items.h

    BagT.o : BagT.cpp \
        $(BCINC)/BCType.h \
        $(BCINC)/BCExcept.h \
        $(BCINC)/BCPool.h \
        $(BCINC)/BCStoreM.h \
        $(BCINC)/BCNodes.h \
        $(BCINC)/BCBound.h \
        $(BCINC)/BCDynami.h \
        $(BCINC)/BCUnboun.h \
        $(BCINC)/BCHashTa.h \
        $(BCINC)/BCBag.h \
        $(BCINC)/BCBagB.h \
        $(BCINC)/BCBagD.h \
        $(BCINC)/BCBagU.h \
        Items.h
```

The following describes the operation of this makefile:

[1] Define values for standard makefile variables CPU and TOOL.

[2] Include standard VxWorks makefile fragments. These files contain default and architecture-specific rules and compiler options for compiling C++ modules.

[3] Define a make variable to represent the directory containing Booch Components source and header files. Files from this directory are included by the application.

1. PC users must first invoke the **torVars.bat** file located in the host **bin** directory. This file configures the Tornado environment variables necessary for invoking tools from the DOS prompt. For more information about **torVars.bat**, see the *Tornado User's Guide: Getting Started*. PC users can also build this example using the Tornado project facility; see the *Tornado User's Guide: Project Facility.*

[4] Define **EXTRA_INCLUDE**, a make variable that is used in the makefile fragments included in [1].

[5] The remaining rules list the header and source file dependencies for the **BagT** example. They define two modules, **Items.o** and **BagT.o** which are used to generate **BagT.out**. The actual rules for compiling and munching these modules are defined in the makefile fragments included in [1]. (For a discussion of the "munching" process, see the *Tornado User's Guide: Cross-Development*.)

Template Instantiation

Example 10-3 shows how to include header and source files for the templates that you are instantiating and how to declare a template instantiation. The sample code is extracted directly from the **BagT** example described above (the complete example is much too long to include here).

This example illustrates the basics of explicit template instantiation. When you compile using the instructions of the previous sections, the C++ compiler automatically instantiates the specified templates. The instantiations occur in the module containing the **template class** declaration. Template instantiations can be distributed among multiple application modules. They can also be gathered together into a single module containing many instantiations, or any of several other alternative organizations, as long as the form of the example is followed: include the source and header files for the templates, then make a **template class** declaration to complete the instantiation.

Example 10-3 **BagT Template Instantiation**

```
// BagT.cpp - This file contains tests for the bag classes.
        ...
[1]  #include "BCBagB.h"
        #include "BCBagB.cpp"
        ...
[2]  template class BC_TBoundedBag<Char, 3U, 100U>;
```

The following details explain the preceding source lines:

[1] Include header and source files that define the template(s) that you want to use. In this case, **BagT.cpp** uses the bounded-bag template class.

[2] Declare the template instantiation that your application needs. In this case, **BagT.cpp** is declaring a bounded bag to contain objects of type **Char**. The other parameters, **3U** and **100U**, define the bag's size and organization.

Appendices

A

Motorola MC680x0

List of Tables

List of Figures

A.1 Introduction

This appendix provides information specific to VxWorks development on Motorola MC680*x*0 targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the MC680*x*0 processors.
- Architecture Considerations: special features and limitations of the MC680*x*0 processors, including a figure showing the VxWorks memory layout for these processors.

For general information on the Tornado development environment's cross-development tools, see the *Tornado User's Guide: Cross-Development*.

A.2 Building Applications

The following sections describe a configuration constant, an environment variable, and compiler options that together specify the information the GNU ToolKit requires to compile correctly for MC680*x*0 targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate architecture-specific features enabled. Define this variable to one of the following values, to match the processor you are using:

- **MC68000**
- **MC68010**
- **MC68020** (used also for MC68030 processors)
- **MC68040**
- **MC68LC040** (used also for MC68EC040 processors)
- **MC68060**
- **CPU32**

For example, to define **CPU** for a MC68040 on the compiler command line, specify the following command-line option when you invoke the compiler:

```
-DCPU=MC68040
```

To provide the same information in a header or source file instead, include the following line in the file:

```
#define CPU MC68040
```

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables.

For developers using UNIX hosts, you must specifically define this variable. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

For developers using Windows hosts, if you are working through the Tornado IDE, the appropriate variable(s) are set automatically. However, before invoking the compiler from a DOS command line, first run the following batch file to set the variable(s):

```
%WIND_BASE%/host/x86-win32/bin/torVars.bat
```

For more information, see the *Tornado User's Guide: Getting Started*.

Compiling C or C++ Modules

The following is an example of a compiler command line for MC680*x*0 cross-development. The file to be compiled in this example has a base name of **applic**.

```
% cc68k -DCPU=MC68040 -I $WIND_BASE/target/h -fno-builtin \
-O -nostdinc -c applic.language_id
```

The options shown in the example have the following meanings:[1]

-DCPU=MC68040
Required; defines the CPU type. If you are using another MC680*x*0 processor, specify the appropriate value (see *Defining the CPU Type*, p.489).

-I $WIND_BASE/target/h
Required; includes VxWorks header files. (Additional **-I** flags may be included to specify other header files.)

-fno-builtin Required; uses library calls even for common library subroutines.

-O Optional; performs standard optimization.

-nostdinc Required; searches only the directory(ies) specified with the **-I** flag (see above) and the current directory for header files. Does not search host-system include files.

-c Required; specifies that the module is to be compiled only, and not linked for execution under the host.

applic.*language_id*
Required; the file(s) to compile. For C compilation, specify a suffix of **.c**. For C++ compilation, specify a suffix of **.cpp**. The output is an unlinked object module in **a.out** format with the suffix **.o**; for the example, the output is **applic.o**.

During C++ compilation, the compiled object module (*applic*.**o**) is *munched*. Munching is the process of scanning an object module for non-local static objects, and generating data structures that VxWorks run-time support can use to call the objects' constructors and destructors. See the *Tornado User's Guide: Cross-Development* for details.

⚠ **NOTE:** Do not use **-msoft-float** on the MC68040 or MC68060. However, do use this flag for floating-point support on the MC68LC040. See *Floating-Point Support*, p.498.

1. For more information on these and other compiler options, see the *GNU ToolKit User's Guide*. WRS supports compiler options used in building WRS software; a list of these options is included in the *Guide*. Other options are not supported, although they are available with the tools as shipped.

A.3 Interface Variations

Because of specific characteristics of the MC68040 or MC68060, certain VxWorks features are not useful on these targets. Conversely, other VxWorks features are particular to one or both of these processors, to exploit specific characteristics.

Note that discussion of the MC68040 also applies to the MC68LC040 unless otherwise noted. The MC68LC040 is a derivative of the MC68040 and differs only in that it has no floating-point unit.

Table A-1 lists such CPU-specific VxWorks interfaces. Section *A.4 Architecture Considerations*, p.492 discusses these interfaces in the context of CPU architecture. For more complete documentation on these routines, see the reference entries.

Table A-1 **VxWorks Interface Variations for MC68040/MC68060**

Routine or Macro Name	CPU	Change	Detailed Discussion
checkStack( )	060	Interrupt stack display meaningless	*MC68060: No Interrupt Stack*, p.494
vxSSEnable( ) *vxSSDisable*( )	060	Only for this architecture	*MC68060 Superscalar Pipeline*, p.494
cacheLock( ) *cacheUnlock*( )	040	Always return **ERROR**	*MC68040 Caches*, p.495
cacheStoreBufEnable( ) *cacheStoreBufDisable*( )	060	Only for this architecture	*MC68060 Caches*, p.496
USER_B_CACHE_ENABLE	060	Architecture-specific configuration	*MC68060 Caches*, p.496
BRANCH_CACHE	060	Architecture-specific cache	*MC68060 Caches*, p.496
VM_STATE...	both	Architecture-specific MMU states	*Memory Management Unit*, p.497

A.4 Architecture Considerations

This section describes the following characteristics of the MC680*x*0 processors (particularly the MC68040 and MC68060) that you should keep in mind as you write a VxWorks application:

- MC68060 unimplemented integer instructions

- Double-word integers
- Interrupt stack
- MC68060 superscalar pipeline
- Caches
- Memory Management Unit
- Floating-point support
- Memory layout

Note that discussion of the MC68040 also applies to the MC68LC040 unless otherwise noted. The MC68LC040 is a derivative of the MC68040 and differs only in that it has no floating-point unit.

For comprehensive documentation of Motorola architectures, see the appropriate Motorola microprocessor user's manual.

The names of macros specific to these architectures, and specialized terms in the remainder of this section, match the terms used by the Motorola manuals.

MC68060 Unimplemented Integer Instructions

Neither the 64-bit divide and multiply instructions, nor the **movep**, **cmp2**, **chk2**, **cas**, and **cas2** instructions are implemented on the MC68060 processor. To eliminate these restrictions, VxWorks integrates the software emulation provided in the Motorola MC68060 software package, version B1. This package contains an exception handler that allows full emulation of the instructions listed above. VxWorks connects this exception handler to the unimplemented-integer-instruction exception (vector 61).

The Motorola exception handler allows the host operating system to add or to substitute its own routines. VxWorks does not add or substitute any routines; the instruction emulation is the full Motorola implementation.

Double-word Integers: long long

The double-word integer **long long** is not supported, except as an artifact of your particular architecture and compiler. For more information about handling unsupported features, please see the *Customer Support User's Guide*.

Interrupt Stack

VxWorks uses a separate interrupt stack whenever the underlying architecture supports it. All MC680*x*0 processors, except the MC68060, have an interrupt stack.

The MC680*x*0 Interrupt Stack

For all MC680*x*0 processors except the MC68060, VxWorks uses the separate interrupt stack instead of the current task stack when the processor takes an interrupt.

The interrupt stack size is defined by the **ISR_STACK_SIZE** macro in the **configAll.h** file. The default size of the interrupt stack is 0x1000 bytes.

MC68060: No Interrupt Stack

When the MC68060 processor takes an interrupt, VxWorks uses the current supervisor stack. To avoid stack overflow, spawn every task with a stack big enough to hold both the task stack and the interrupt stack.

The routine ***checkStack*( )**, which is built in to the Tornado shell, displays the stack state for each task and also for the interrupt stack. Because this routine is the same for all processors that VxWorks supports, ***checkStack*( )** displays a line for the interrupt stack state. For the MC68060, the values that appear on this line are meaningless.

MC68060 Superscalar Pipeline

The MC68060 implements a superscalar pipeline that allows multiple instructions to be executed in a single machine cycle. This feature can be enabled or disabled by setting or clearing the ESS (Enable SuperScalar) bit of the Processor Configuration Register (PCR). For this architecture, VxWorks provides two routines to enable and disable the superscalar pipeline, declared as follows:

```
void vxSSEnable (void)
void vxSSDisable (void)
```

In the default configuration, VxWorks enables the superscalar pipeline.

Caches

The MC68000 and MC68010 processors do not have caches. The MC68020 has only a 256-byte instruction cache; see the general cache information presented in *Cache Coherency*, p.168.

The MC68040 has 4KB instruction and data caches, and the MC68060 has 8KB instruction and data caches. The following subsections augment the information in *Cache Coherency*, p.168.

MC68040 Caches

The MC68040 processor contains an instruction cache and a data cache. By default, VxWorks uses both caches; that is, both are enabled. To disable the instruction cache, undefine the **USER_I_CACHE_ENABLE** macro in **config/all/configAll.h**; to disable the data cache, undefine **USER_D_CACHE_ENABLE** in **configAll.h**.

These caches can be set to the following modes:

- cacheable writethrough (the default for both caches)
- cacheable copyback
- cache-inhibited serialized
- cache-inhibited not-serialized

Choose the mode by setting the **USER_I_CACHE_MODE** macro or the **USER_D_CACHE_MODE** macro in **configAll.h**. The list of possible values for these macros is defined in **h/cacheLib.h**.

For most boards, the cache capabilities must be used with the MMU to resolve cache coherency problems. In that situation, the page descriptor for each page selects the cache mode. This page descriptor is configured by filling the **sysPhysMemDesc[]** data structure defined in the BSP **config/***bspname***/sysLib.c** file. (For more information about cache coherency, see the **cacheLib** reference entry. See also *7. Virtual Memory Interface* for information on VxWorks MMU support. For MMU information specific to the MC680*x0* family, see *Memory Management Unit*, p.497.)

The MC68040 caches do not support cache locking and unlocking. Thus the ***cacheLock*( )** and ***cacheUnlock*( )** routines have no effect on this target, and always return **ERROR**.

The ***cacheClear*( )** and ***cacheInvalidate*( )** routines are very similar. Their effect depends on the cache:

- With the data cache, ***cacheClear*( )** first pushes dirty data[2] to memory (if the cache line contains any) and then invalidates the cache line, while

cacheInvalidate( ) just invalidates the line (in which case any dirty data contained in this line is lost).

- For the instruction cache, both routines have the same result: they invalidate the cache lines.

MC68060 Caches

VxWorks for the MC68060 processor provides all the cache features of the MC68040, and some additional features.

- **Instruction and Data Cache**

Motorola has introduced a change of terminology with the MC68060: the mode called "cache-inhibited serialized mode" on the MC68040 is called "cache-inhibited precise mode" on the MC68060, and the MC68040's "cache-inhibited not-serialized mode" is replaced by "cache-inhibited imprecise mode" on the MC68060.

To make your code consistent with this change, you can use the macros[3] **CACHE_INH_PRECISE** and **CACHE_INH_IMPRECISE** with VxWorks cache routines when writing specifically for the MC68060, instead of using the MC68040-oriented macro names **CACHE_INH_SERIAL** and **CACHE_INH_NONSERIAL**. (The corresponding macros in each pair have the same definition, however, to make MC68040 object code compatible with the MC68060.)

A four-entry first-in-first-out (FIFO) buffer is implemented on the MC68060. This buffer, used by the cacheable writethrough and cache inhibited imprecise mode, is enabled by default. Two VxWorks routines are available to enable or disable this store buffer. Their names and prototypes are declared as follows:

```
void cacheStoreBufEnable (void)
void cacheStoreBufDisable (void)
```

On the MC68060, the instruction cache and data cache can be locked by software. Thus, on this architecture (unlike for the MC68040), the *cacheLock*( ) and *cacheUnlock*( ) routines are effective.

VxWorks does not support the MC68060 option to use only half of the instruction cache or data cache.

2. *Dirty data* refers to data saved in the cache, not in memory (copyback mode only).
3. Defined in **h/arch/mc68k/cacheMc68kLib.h**.

- **Branch Cache**

  In addition to the instruction cache and the data cache, the MC68060 contains a branch cache that VxWorks supports as an additional cache. Use the name **BRANCH_CACHE** to refer to this cache with the VxWorks cache routines.

  Most routines available for both instruction and data caches are also available for the branch cache. However, the branch cache cannot be locked; thus, the ***cacheLock*( )** and ***cacheUnlock*( )** routines have no effect and always return **ERROR**.

  The branch cache uses only one operating mode and does not require a macro to specify the current mode. In the default configuration, VxWorks enables the branch cache. This option can be removed by disabling the definition of the **USER_B_CACHE_ENABLE** macro in **configAll.h**.

  The branch cache can be invalidated only in its entirety. Trying to invalidate one branch cache line, or, as for the instruction cache, clearing the branch cache, invalidates the whole cache.

  The branch cache is automatically cleared by the hardware as part of any instruction-cache invalidate.

Memory Management Unit

VxWorks provides two levels of virtual memory support: the basic level bundled with VxWorks, and the full level, unbundled, that requires the optional product VxVMI. These two levels are supported by the MC68040 and MC68060 processors; however, the MC68000, MC68010, and MC68020 processors do not have MMUs.

For detailed information on VxWorks's MMU support, see *7. Virtual Memory Interface*. The following subsections augment the information in that chapter.

MC68040 Memory Management Unit

On the MC68040, you can set a specific configuration for each memory page. The entire physical memory is described by the data structure **sysPhysMemDesc[]** defined in the BSP file **sysLib.c**. This data structure is made up of state flags for each page or group of pages. All the state flags defined in Table 7-2 of *7. Virtual Memory Interface* are available for MC68040 virtual memory pages.

NOTE: The **VM_STATE_CACHEABLE** flag listed in Table 7-2 of *7. Virtual Memory Interface* sets the cache to copyback mode for each page or group of pages.

In addition, two other state flags are supported:

- **VM_STATE_CACHEABLE_WRITETHROUGH**
- **VM_STATE_CACHEABLE_NOT_NON_SERIAL**

The first flag sets the page descriptor cache mode field in cacheable writethrough mode, and the second sets it in cache-inhibited non-serialized mode.

For more information on memory page states, state flags, and state masks, see *Page States*, p.412.

MC68060 Memory Management Unit

The MMU on the MC68060 is very similar to the MC68040 MMU, and MC68060 virtual memory management provides the same capabilities as the MC68040 virtual memory; see *MC68040 Memory Management Unit*, p.497 for details.

You can use the page state constant **VM_STATE_CACHEABLE_NOT_IMPRECISE** instead of **VM_STATE_CACHEABLE_NOT_NON_SERIAL**, to match changes in Motorola terminology (see *MC68060 Caches*, p.496). Use this constant (as its name suggests) to set the page descriptor cache mode field to "cache-inhibited imprecise mode." To set the page cache mode to "cache-inhibited precise mode," use **VM_STATE_CACHEABLE_NOT**.

The MC68060 does not use the data cache when searching MMU address tables, because the MC68060 tablewalker unit has a direct interface to the bus controller. Therefore, virtual address translation tables are always placed in writethrough space. (Although VxWorks maps virtual addresses to the identical physical addresses, the MMU address translation tables also record the page protection provided through VxVMI.)

Floating-Point Support

The MC68020 uses an MC68881/MC68882 floating-point coprocessor for hardware floating-point support. The MC68040 and MC68060 CPUs (but not the MC68LC040) include internal floating-point units that provide a significant subset of the MC68881/MC68882 instruction set, in addition to the same control, status, and data register programming model. Basic floating-point arithmetic and manipulation functions are provided, but higher-level transcendental functions (for example, trigonometric, logarithmic, rounding) are not. Floating-point support for the MC68LC040 is provided in software only.

Different subsets of the floating-point math routines in **mathALib** are supported for each processor of the MC680*x*0 family. Table A-2 shows the supported double-precision routines.

There is no hardware support for single-precision floating-point. On the MC68000, MC68010, MC68020, MC68LC040, and CPU32, software support is available for the following single-precision routines:

acosf( )	*asinf( )*	*atanf( )*	*atan2f( )*	*cbrtf( )*
ceilf( )	*cosf( )*	*expf( )*	*fabsf( )*	*floorf( )*
infinityf( )	*logf( )*	*log10f( )*	*log2f( )*	*powf( )*
sinf( )	*sincosf( )*	*sqrtf( )*	*tanf( )*	

On the MC68040 or MC68060, there are no supported single-precision floating-point routines.

Floating-Point Support for MC680x0 CPUs Using MC68881/MC68882

VxWorks provides both hardware and software floating-point, in support of those target configurations that include a floating-point coprocessor as well as those that do not. Use the compiler option **-msoft-float** to generate object code that uses software floating-point, and the compiler option **-m68881** for hardware floating-point.

Floating-Point Support for the MC68040 and MC68060

For the MC68040 and the MC68060 (but not the MC68LC040), VxWorks includes support for MC68881/MC68882 floating-point instructions that are not directly supported by the CPU. This emulation is provided by the Floating-Point Software Package (FPSP) from Motorola, which is integrated into VxWorks.

The FPSP is called by special exception handlers that are invoked when one of the unsupported instructions executes. This allows MC68881/MC68882 instructions to be interpreted, although the exception overhead can be significant. Exception handlers are also provided for other floating-point exceptions (for example, floating-point division by zero, over- and underflow).

The initialization routine ***mathHardInit( )*** installs these exception handlers; this routine is called from **usrConfig.c** when you configure VxWorks for hardware floating-point by defining **INCLUDE_HW_FP** in **config/all/configAll.h**. (It is defined by default.)

To avoid the overhead associated with unimplemented-instruction exceptions, the floating-point libraries in VxWorks call specific routines in the FPSP directly. As a result, application code written in C that uses transcendental functions (for example, the ***sin( )*** or ***log( )*** routines) does not suffer from the exception-handling overhead. No special changes to application source code are necessary. (However, support is provided only for double-precision floating-point operations.)

Table A-2 **Double-Precision Floating-Point Routines Supported for MC680x0 Family**

	MC68000/ MC68010	MC68020/ CPU32	MC68040	MC68LC040	MC68060
acos( )	S	HS	E	S	E
asin( )	S	HS	E	S	E
atan( )	S	HS	E	S	E
atan2( )	S	HS	E	S	E
cbrt( )	S	S		S	
ceil( )	S	HS	E	S	H
cos( )	S	HS	E	S	E
cosh( )	S	HS	E	S	E
exp( )	S	HS	E	S	E
fabs( )	S	HS	E	S	H
floor( )	S	HS	E	S	H
fmod( )		H	E		E
infinity( )	S	HS	E	S	H
irint( )		H	E		H
iround( )		H	E		H
log( )	S	HS	E	S	E
log10( )	S	HS	E	S	E
log2( )	S	HS	E	S	E
pow( )	S	HS	E	S	E
round( )		H	E		H
sin( )	S	HS	E	S	E
sincos( )	S	HS	E	S	E
sinh( )	S	HS	E	S	E
sqrt( )	S	HS	E	S	H
tan( )	S	HS	E	S	E
tanh( )	S	HS	E	S	E
trunc( )		H	E		H

S = software floating-point support
H = hardware floating-point support
E = emulated hardware floating-point support

If you are using the GNU ToolKit C compiler (**cc68k**) distributed by Wind River Systems, compile your code *without* the flag **-msoft-float**.

- **MC68040 Floating-Point Software Package**

  On the MC68040, VxWorks uses version 2.2 of the MC68040 Floating-Point Software Package (FPSP) from Motorola. This library makes full use of the floating-point support provided by the MC68040 hardware, as opposed to pure software emulation. The size of this FPSP is approximately 64KB.

- **MC68060 Floating-Point Software Package**

  As with the MC68040, the MC68060 floating-point unit implements only a subset of the MC68881/MC68882 instruction set. The two subsets are not identical (see *§6.5.1 Unimplemented Floating-Point Instructions* in the *MC68060 Microprocessors User's Manual*); hence the MC68060 has its own FPSP. VxWorks uses version B1 of the MC68060 Floating-Point Software Package from Motorola. The size of this FPSP is approximately 84KB.

Floating-Point Support for the MC68LC040

While the MC68LC040 is a derivative of the MC68040 (implementing the same integer unit and memory management unit), it has no floating-point unit. Applications for the MC68LC040 must use software floating-point emulation. Use the compiler option **-msoft-float** to generate object code that uses software floating-point. Be sure to specify a **CPU** value of **MC68LC040** when building VxWorks (see *Defining the CPU Type*, p.489).

Memory Layout

The VxWorks memory layout is the same for all MC680*x*0 processors, except that the MC68060 has no interrupt stack. Figure A-1 shows memory layout, labeled as follows:

Interrupt Vector Table
Table of exception/interrupt vectors.

SM Anchor Anchor for the shared memory network (if there is shared memory on the board).

Boot Line ASCII string of boot parameters.

Exception Message ASCII string of the fatal exception message.

Initial Stack Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image VxWorks itself (three sections: text, data, bss). The entry point for VxWorks is at the start of this region.

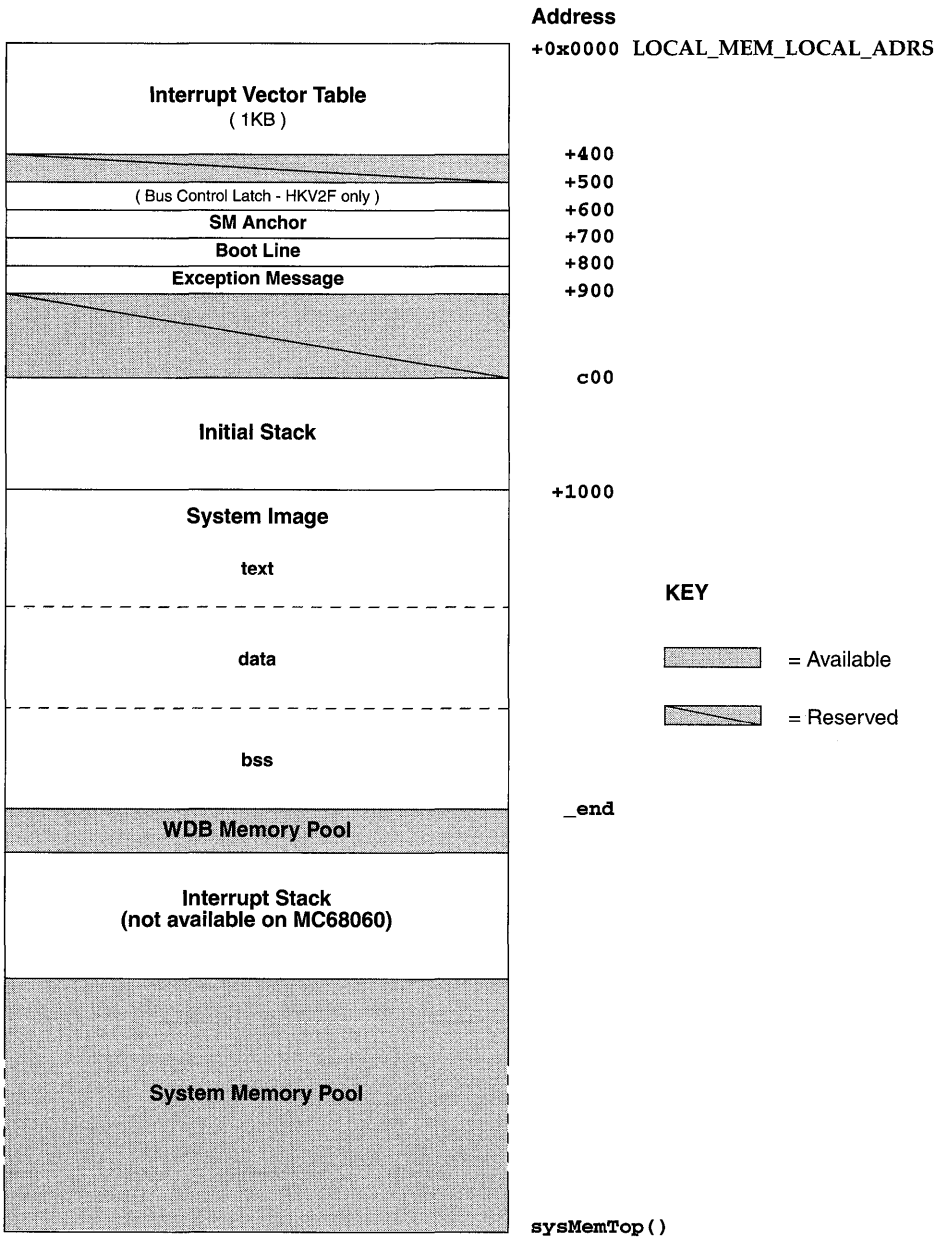
WDB Memory Pool Size depends on the macro **WDB_POOL_SIZE** which defaults to one-sixteenth of the system memory pool. This space is used by the target server to support host-based tools.

Interrupt Stack Stack for interrupt handlers (except MC68060). Size is defined in **configAll.h**. Location depends on system image size.

System Memory Pool
Size depends on size of the system image and (on the all but MC68060) the interrupt stack. The ***sysMemTop( )*** routine returns the end of the free memory pool.

All addresses shown in Figure A-1 are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

Figure A-1 **VxWorks System Memory Layout (MC680*x*0)**



B

Sun SPARC, SPARClite

List of Figures

B.1 Introduction

This appendix provides information specific to VxWorks development on Sun SPARC and SPARClite targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the Sun processors.
- Architecture Considerations: special features and limitations of the Sun processors, including information specific to the SPARClite and a figure showing the VxWorks memory layout for these processors.

For general information on the Tornado development environment's cross-development tools, see the *Tornado User's Guide: Cross-Development*.

B.2 Building Applications

The following sections describe a configuration constant, an environment variable, and compiler options that together specify the information the GNU ToolKit requires to compile correctly for SPARC and SPARClite targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate architecture-specific features enabled. Define this variable to **SPARC** for both the SPARC and SPARClite processors.

For example, to define **CPU** for a SPARC on the compiler command line, specify the following command-line option when you invoke the compiler:

```
-DCPU=SPARC
```

To provide the same information in a header or source file instead, include the following line in the file:

```
#define CPU SPARC
```

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the UNIX environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

For more information, see the *Tornado User's Guide: Getting Started*.

Compiling C or C++ Modules

The following is an example of a compiler command line for SPARClite cross-development. The file to be compiled in this example has a base name of **applic**.

```
% ccsparc -DCPU=SPARC -I $WIND_BASE/target/h -O2 -nostdinc \
-fno-builtin -msparclite -msoft-float -c applic.language_id
```

The options shown in the example have the following meanings:[1]

-DCPU=SPARC Required; defines the CPU type. Use **SPARClite** for SPARClite processors.

-I $WIND_BASE/target/h
Required; includes VxWorks header files. (Additional **-I** flags may be included to specify other header files.)

-O2 Optional; performs level 2 optimization.

1. For more information on these and other compiler options, see the *GNU ToolKit User's Guide*. WRS supports compiler options used in building WRS software; a list of these options is included in the *Guide*. Other options are not supported, although they are available with the tools as shipped.

-nostdinc Required; searches only the directory(ies) specified with the **-I** flag (see above) and the current directory for header files. Does not search host-system include files.

-fno-builtin Required; uses library calls even for common library subroutines.

-msparclite Required for SPARClite; generates SPARClite-specific code.

-msoft-float Optional; generates software floating point library calls, rather than hardware floating point instructions. For more information, see *USS Floating-Point Emulation Library*, p.520,

-c Required; specifies that the module is to be compiled only, and not linked for execution under the host.

applic.*language_id*
Required; the file(s) to compile. For C compilation, specify a suffix of **.c**. For C++ compilation, specify a suffix of **.cpp**. The output is an unlinked object module in **a.out** format with the suffix **.o**; for the example, the output is **applic.o**.

During C++ compilation, the compiled object module (*applic*.**o**) is *munched*. Munching is the process of scanning an object module for non-local static objects, and generating data structures that VxWorks can use to call the objects' constructors and destructors. For details, see the *Tornado User's Guide: Cross-Development*.

B.3 Interface Variations

This section describes particular routines that are specific to SPARC targets in one of the following ways:

- available only for SPARC or SPARClite targets
- parameters specific to SPARC or SPARClite targets
- special restrictions or characteristics on SPARC or SPARClite targets

For complete documentation on these routines, see the reference entries.

Unless otherwise noted, the information in this section applies to both the SPARC and SPARClite. For SPARClite-specific information, see *SPARClite Overview*, p.519.

---

bALib

The following buffer-manipulation routines provided by **bALib** exploit the SPARC LDD and STD instructions.

***bzeroDoubles*( )**	Zeroes out a buffer, 256 bytes at a time.
***bfillDoubles*( )**	Fills a buffer with a specified eight-byte pattern.
***bcopyDoubles*( )**	Copies one buffer to another, eight bytes at a time.

cacheMb930Lib

The library **cacheMb930Lib** contains routines that allow you to initialize, lock, and clear the Fujitsu MB86930 (SPARClite) cache. For more information, see the manual pages and *Instruction and Data Cache Locking*, p.520.

cacheMicroSparcLib

The library **cacheMicroSparcLib** contains routines that allow you to initialize, flush, and clear the MicroSparc I and II caches. For more information, see the manual pages.

dbgLib

If you are using the target shell, note the following architecture-specific information on routines in the **dbgLib**:

- **Optional Parameter for *c*( ) and *s*( )**

  The SPARC versions of ***c*( )** (continue) and ***s*( )** (single-step) can take a second address parameter, *addr1*. With this parameter, you can set **nPC** as well as the **PC**.

  Note that if *addr* is NULL, *addr1* is ignored.

- **Restrictions on *cret*( )**

  In VxWorks for SPARC, ***cret*( )** cannot determine the correct return address. Because the actual return address is determined by code within the routine, only the calling address is known. With C code in general, the calling instruction is a **CALL** and routines return with the following:

  ```
  ret
  restore
  ```

This is the assumption made by ***cret( )*** when it places a breakpoint at the return address of the current subroutine and continues execution. Note that returns other than **%i7 + 8** result in ***cret( )*** setting an incorrect breakpoint value and continuing.

- **Restrictions on *so( )***

  The ***so( )*** routine single-steps a task stopped at a breakpoint, but steps over a subroutine. However, in the SPARC version, if the next instruction is a **CALL** or **JMPL x, %o7**, the routine breaks at the second instruction following the subroutine (that is, the first instruction following the delay slot's instruction). In general, the delay slot loads parameters for the subroutine. This loading can have unintended consequences if the delay slot is also a transfer of control.

- **Trace Routine, *tt( )***

  In general, a task trace works for all non-leaf C-language routines and any assembly language routines that contain the standard prologue and epilogue:

```
save    %sp, -STACK_FRAME_SIZE, %sp
...
ret
restore
```

  Although the ***tt( )*** routine works correctly in general, note the following caveats:

  - Routines written in assembly or other languages, strange entries in routines, or tasks with corrupted stacks, can result in confusing trace information.
  - All parameters are assumed to be 32-bit quantities.
  - The cross-compiler does not handle structures passed as parameters correctly.
  - The current trace-back tag generated by C compilers is limited to 16 parameters; thus, ***tt( )*** does not report the value of parameters above 16. However, this does not mean that your application cannot use routines with more than 16 parameters.
  - If the routine changes the values of its local registers between the time it is called and the time it calls the next level down (or, at the lowest level, the time the task is suspended), ***tt( )*** reports the changed values. It has no way to locate the original values.
  - If the routine changes the values of registers **i0** through **i5** between the time it is called and the time it calls the next level down (or, at the lowest level, the time the task is suspended), ***tt( )*** reports the changed values. It has no way to locate the original values.
  - If you attempt a ***tt( )*** of a routine between the time the routine is called and the time its initial *save* is finished, you can expect strange results.

dbgArchLib

If you are using the target shell, the following architecture-specific show routines are available if **INCLUDE_SHOW_ROUTINES** is defined:

psrShow( ) Displays the symbolic meaning of a specified PSR value on the standard output device.

fsrShow( ) Displays the symbolic meaning of a specified FSR value on the standard output device.

fppArchLib

The SPARC version of **fppArchLib** saves and restores a math coprocessor context appropriate to the SPARC floating-point architecture standard.

intArchLib

- **Parameters for *intLevelSet*( )**

  The SPARC version of ***intLevelSet*( )** takes an argument from 0 to 15.

- **Returns for *intLock*( )**

  The SPARC version of ***intLock*( )** returns an interrupt level.

ioMmuMicroSparcLib

The library **ioMmuMicroSparcLib** contains routines that allow you to initialize and map memory in the microSPARC I/O MMU. For more information, see the manual pages.

mathALib

Because the overall SPARC architecture includes hardware floating-point support, while the SPARClite variant does not, VxWorks includes **mathALib** hardware floating-point support for SPARC and software floating-point support for SPARClite.

- **SPARC**

  On SPARC targets, the following **mathALib** routines are available. Note that these are all double-precision routines; no single-precision routines are supported for SPARC:

  acos( ) *asin( )* *atan( )* *atan2( )* *cbrt( )* *ceil( )* *cos( )*
  cosh( ) *exp( )* *fabs( )* *floor( )* *fmod( )* *irint( )* *iround( )*
  log( ) *log10( )* *pow( )* *round( )* *sin( )* *sinh( )* *sqrt( )*
  tan( ) *tanh( )* *trunc( )*

- **SPARClite**

  On SPARClite targets, the following **mathALib** routines are supported (for information about how to use this support, see *USS Floating-Point Emulation Library*, p.520):

  – Double-precision routines:

  acos( ) *asin( )* *atan( )* *atan2( )* *ceil( )* *cos( )* *cosh( )*
  exp( ) *fabs( )* *floor( )* *fmod( )* *frexp( )* *ldexp( )* *log( )*
  log10( ) *pow( )* *sin( )* *sinh( )* *sqrt( )* *tan( )* *tanh( )*

  – Single-precision routines:

  acosf( ) *asinf( )* *atanf( )* *atan2f( )* *ceilf( )* *cosf( )* *coshf( )*
  expf( ) *fabsf( )* *floorf( )* *fmodf( )* *logf( )* *log10f( )* *modf( )*
  powf( ) *sinf( )* *sinhf( )* *sqrtf( )* *tanf( )* *tanhf( )*

vxALib

The test-and-set primitive *vxTas( )* provides a C-callable interface to the SPARC **ldstub** instruction.

vxLib

The routine *vxMemProbeAsi( )* probes addresses in SPARC ASI space.

B.4 Architecture Considerations

This section describes the following characteristics of the SPARC and SPARClite architectures that you should keep in mind as you write a VxWorks application:

- Reserved registers
- Processor mode
- Vector table initialization
- Double-word Integers
- Interrupt handling
- Floating-point support
- Stack pointer usage
- SPARClite overview
- Memory layout

Reserved Registers

Following the SPARC specification (*Appendix D, Software Considerations*, in *The SPARC Architecture Manual, Version 8* from Sun Microsystems), registers **g5**, **g6**, and **g7** are reserved for VxWorks kernel use. Avoid using these registers in your applications.

Processor Mode

VxWorks for SPARC and SPARClite always runs in Supervisor mode.

Vector Table Initialization

After the VxWorks for SPARC or SPARClite has completed initialization, traps are enabled and the PIL (Processor Interrupt Level) is set to zero. All 15 interrupt levels are active with the coprocessor enables set according to hardware availability and application use.

The TBR (Trap Base Register) points to the active vector table at address 0x1000 in local memory.

Make sure that vectors are not reserved for the processor or the kernel before acquiring them for an application.

Double-word Integers: long long

The double-word integer **long long** is not supported, except as an artifact of your particular architecture and compiler. For more information about handling unsupported features, please see the *Customer Support User's Guide.*

Interrupt Handling

For VxWorks for SPARC and SPARClite, an interrupt stack allows all interrupt processing to be performed on a separate stack. The interrupt stack is implemented in software because the SPARC family does not support such a stack in hardware.

SPARC Interrupts

The SPARC microprocessor allows 15 levels of interrupts. The level is encoded by external hardware on the four interrupt signal lines. The integer unit (CPU) decodes this level and passes control directly to the entry in the vector table at an offset of 0x100 plus the interrupt level times 16 bytes. This corresponds to vectors 16 through 31 (addresses 0x100 to 0x1F0). Each 16-byte entry in the vector table contains up to four instructions. Typically, control passes to an interrupt service routine (ISR) with a call or branch instruction.

The SPARC uses auto-vectored interrupts. The chip does not perform any type of interrupt acknowledge (IACK) cycle. The address in the Trap Base Register (TBR) concatenated with the interrupt level vector displacement allows the SPARC to begin interrupt processing.

The alternative is vectored interrupts. The CPU responds to the interrupt with an IACK cycle so that an interrupt controller chip or individual device can return a value that clears and identifies the source of the interrupt. This is extremely useful for multiple sources of interrupts on a single-interrupt level.

The ability to perform an interrupt acknowledge cycle is a function of the microprocessor (not the software or board-level hardware). However, a target board can synthesize an IACK cycle by accessing an area created in its address space. This is often necessary to clear the interrupt pending bit in an interrupting device. An IACK cycle also differs from a normal read cycle in that the value returned is an interrupt vector. This vector is used to select an offset in the vector table that has the device's ISR connected to that table entry.

VxWorks allows an application to connect ISRs to vectors with the routine ***intConnect( )***. A stub is built dynamically that calls an interrupt entry routine, calls the ISR, and then calls an exit routine. The SPARC, like other RISC processors,

delegates to software the task of building an exception stack frame (ESF) to save volatile information. The kernel builds up two types of exception stack frames: one for interrupts and one for all other exceptions. The code execution sequence following an interrupt is as follows:

- Vector table
- Exception stack frame building
- Overflow exception handling
- Interrupt entry code
- ISR
- Interrupt exit code
- Rescheduling, if the interrupt added work for the kernel (such as a ***semGive*( )**)

Vectored Interrupts

The SPARC kernel was designed to handle vectored interrupts as an option. Because this implementation varies with every target board, the kernel must work with the board support package (BSP). The implementation of vectored interrupts on a processor that does not support them must be done in software.

A table in the BSP allows an IACK for each of the 15 interrupt levels. A NULL (0) entry corresponds to no interrupt acknowledge. If an IACK is required, the table entry corresponds to a routine that performs the necessary operations. Because the SPARC vector table contains 256 entries, a byte-sized vector can select any exception handler.

Note that the microprocessor, the board, and the kernel reserve certain vector table entries. The kernel appends this vector to the TBR and continues execution with the selected ISR. All checking for the IACK condition and performing of the operation is done by the kernel and is transparent. The interrupt connection mechanism is the same, and checking for and clearing the pending interrupt is done before the ISR attached by ***intConnect*( )** is called.

The following shows the structure used on the SPARCengine 1E (also known as a Sun 1E) SPARC board in **config/sun1e/sysLib.c**. It illustrates the use of vectored interrupts for VME, but does not require an IACK cycle for local (on-board) interrupts:

```
extern sysVmeAck();     /* IACK Leaf Functions, code in sysALib */

int (*sysIntAckTable [16])() =
    {
    NULL,               /* Reserved for Kernel                  */
    NULL,               /* Interrupt Level 1 - Software 1       */
    sysVmeAck,          /* Interrupt Level 2 - VME 1            */
    sysVmeAck,          /* Interrupt Level 3 - VME 2            */
    NULL,               /* Interrupt Level 4 - SCSI             */
```

```
    sysVmeAck,          /* Interrupt Level 5 - VME 3            */
    NULL,               /* Interrupt Level 6 - Ethernet         */
    NULL,               /* Interrupt Level 7 - P2 Bus           */
    sysVmeAck,          /* Interrupt Level 8 - VME 4            */
    sysVmeAck,          /* Interrupt Level 9 - VME 5            */
    NULL,               /* Interrupt Level 10 - Timer 0         */
    sysVmeAck,          /* Interrupt Level 11 - VME 6           */
    NULL,               /* Interrupt Level 12 - Serial Ports    */
    NULL,               /* Interrupt Level 13 - Mailbox         */
    NULL,               /* Interrupt Level 14 - Timer 1         */
    NULL                /* Interrupt Level 15 - NMI             */
    };
```

The performance penalty for this added feature is negligible. When vectored interrupts are used, this penalty increases, because an operation is being handled in software that the SPARC microprocessor was not designed to do. There are some restrictions on these vector routines because they are called in a critical section of code. Again, the Sun 1E SPARC board is used as an example. Note that you must use special "leaf" procedures.

The corresponding code for the function table is in **config/sun1e/sysALib.s**:

```
/* IACK Function Call Template
/* Input:      %l5 - return address
/* Volatile:   %l4, %l6 (DO NOT USE OTHER REGISTERS !!!)
/* Return:     %l5 - vector table index */

    .global _sysVmeAck

_sysVmeAck:
    sethi    %hi(SUN_VME_ACK),%l6 /* VMEbus IACK - 0xFFD18001          */
    or       %l6,%lo(SUN_VME_ACK),%l6
    rd       %tbr,%l4             /* Extract interrupt level          */
    and      %l4,0x00F0,%l4
    add      %l4,0x0010,%l4       /* Sun 1E to VME level conversion   */
    srl      %l4,5,%l4            /* Add 1, divide by 2 (no remainder) */
    sll      %l4,1,%l4            /* Multiply VME level by 2          */
    ldub     [%l6 + %l4],%l4      /* VMEbus IACK and get vector       */
    jmpl     %l5,%g0              /* Return address - leaf routine    */
    mov      %l4,%l5              /* Interrupt vector to %l5          */
```

VMEbus Interrupt Handling

SPARC uses fifteen interrupt levels instead of the seven used by VMEbus. The mapping of the seven VMEbus interrupts to the fifteen SPARC levels is board dependent. VMEbus interrupts must be acknowledged.

Floating-Point Support

Floating-Point Contexts

A task can be spawned with floating-point support by setting the **VX_FP_TASK** option. This causes switch hooks to initialize, save, and restore a floating-point context. This option increases the task's context switch time and memory consumption, so only spawn tasks with **VX_FP_TASK** if they must perform floating-point operations.

The floating-point data registers are initialized to NaN (Not-a-Number), which is 0xFFFFFFFF. You can change the FSR's (Floating-point Status Register) value using the global variable **fppFsrDefault**.

Floating-Point Exceptions

The following are SPARC floating-point exceptions (most are deferred):

- FPU Disabled (or not present)
- Unfinished Operation
- Unimplemented Operation
- Sequence Error
- Invalid Operation
- Overflow
- Underflow
- Divide-by-Zero
- Inexact

- **Exception Options**

The application can configure the types of floating-point exceptions that VxWorks handles. The ideal solution is to not generate any floating-point exceptions in the application tasks. However, a more realistic scheme is to mask all exceptions globally (all tasks) in the TEM (Trap Enable Mask) field of the FSR (Floating-point Status Register). Alternatively, this can be done locally (on a per task basis) as tasks are spawned and the FSR is initialized. In addition to global and local masks, individual exceptions (invalid operation, overflow, underflow, divide-by-zero, inexact) can be masked in the TEM. The masked exception continues to accrue (for example, become more inexact, continue to overflow, and so on). The default for VxWorks is to mask only the inexact exception.

- **Exception Handlers**

All floating-point exceptions (if enabled) result in the suspension of the offending task and a message sent through the exception handling task, *excTask*( ). The floating-point unit is flushed so that other tasks can still use the hardware and continue their numeric processing.

- **Deferred Exceptions**

  Floating-point exceptions on the SPARC floating-point units are deferred. When they occur in the FPU, they do not immediately interrupt the CPU (integer unit). Instead they remain pended until they are pushed out of the queue by additional floating-point operations or an FSR access.

  If one of the last floating-point operations causes an unmasked exception before a context switch, saving the task's context flushes out the exception while in the kernel. The exception handler checks for this special case and works its way back to the kernel so that it can continue the context switch. When the task that caused the exception is switched back in, it continues in the exception handler and suspends itself. The relationship between a deferred exception and a context switch cannot be controlled due to its asynchronous nature.

- **Floating-Point Exception Simulation**

  SPARCmon is a product from Sun Microsystems that you can attach to the floating-point exception vectors to handle all exception cases for the SPARC. Any floating-point exceptions must be simulated by software and the queue flushed of all pending operations. This simulation fixes the error that caused the exception whenever possible, or takes some default action (for example, suspends the task).

Stack Pointer Usage

Because the stack pointer can advance without stack memory actually being written or read, it is possible for the stack highwater marker to appear below the current stack pointer. In other words, current stack usage can be greater than the high stack usage. This is an artifact of the SPARC architecture's rolling register windows.

The stack pointer is actually used very little. The local and output registers in each register window perform the bulk of stack operations. The stack is used when argument lists are very long, or if a window overflow exception pushes registers onto the stack.

SPARClite Overview

All information pertaining to the SPARC applies to the SPARClite, with the addition of the architectural enhancements described in the following subsections.

Instruction and Data Cache Locking

The SPARClite allows the global and local locking of the instruction and data caches. The ability to lock instructions and/or data in the caches allows for higher performance and more deterministic systems. The locking must be done in such a way that overall system performance is improved, not degraded. For a better real-time system, call ***cacheMb930LockAuto( )*** to enable instruction and data cache locking. After the caches are locked, they cannot be unlocked or disabled.

To enhance performance, some of the VxWorks kernel data items are locked in the data cache. This uses approximately 128 bytes. The remainder of the data cache is available to the developer. Additional data can be locked in the cache using the BSP.

USS Floating-Point Emulation Library

The SPARClite does not have a floating-point coprocessor; thus, the USS floating-point emulation library is used. Using the **-msparclite** compile flag allows this library to be accessed by your code for floating-point calculations.

Memory Layout

The memory layout of both the SPARC and SPARClite processors is shown in Figure B-1. The memory layout of the microSPARC processor is in Figure B-2. These figures contain the following labels:

SM Anchor Anchor for the shared memory network (if there is shared memory on the board).

Boot Line ASCII string of boot parameters.

Exception Message
ASCII string of the fatal exception message.

Interrupt Vector Table
Table of exception/interrupt vectors.

Initial Stack Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image Entry point for VxWorks.

WDB Memory Pool
Size depends on the macro **WDB_POOL_SIZE** which defaults to one-sixteenth of the system memory pool. This space is used by the target server to support host-based tools.

Interrupt Stack Size defined in **configAll.h**. Location depends on system image size.

System Memory Pool
Size depends on size of system image and interrupt stack. The end of the free memory pool for this board is returned by ***sysMemTop*( )**.

All addresses shown are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

Figure B-1 **VxWorks System Memory Layout (SPARC/SPARClite)**

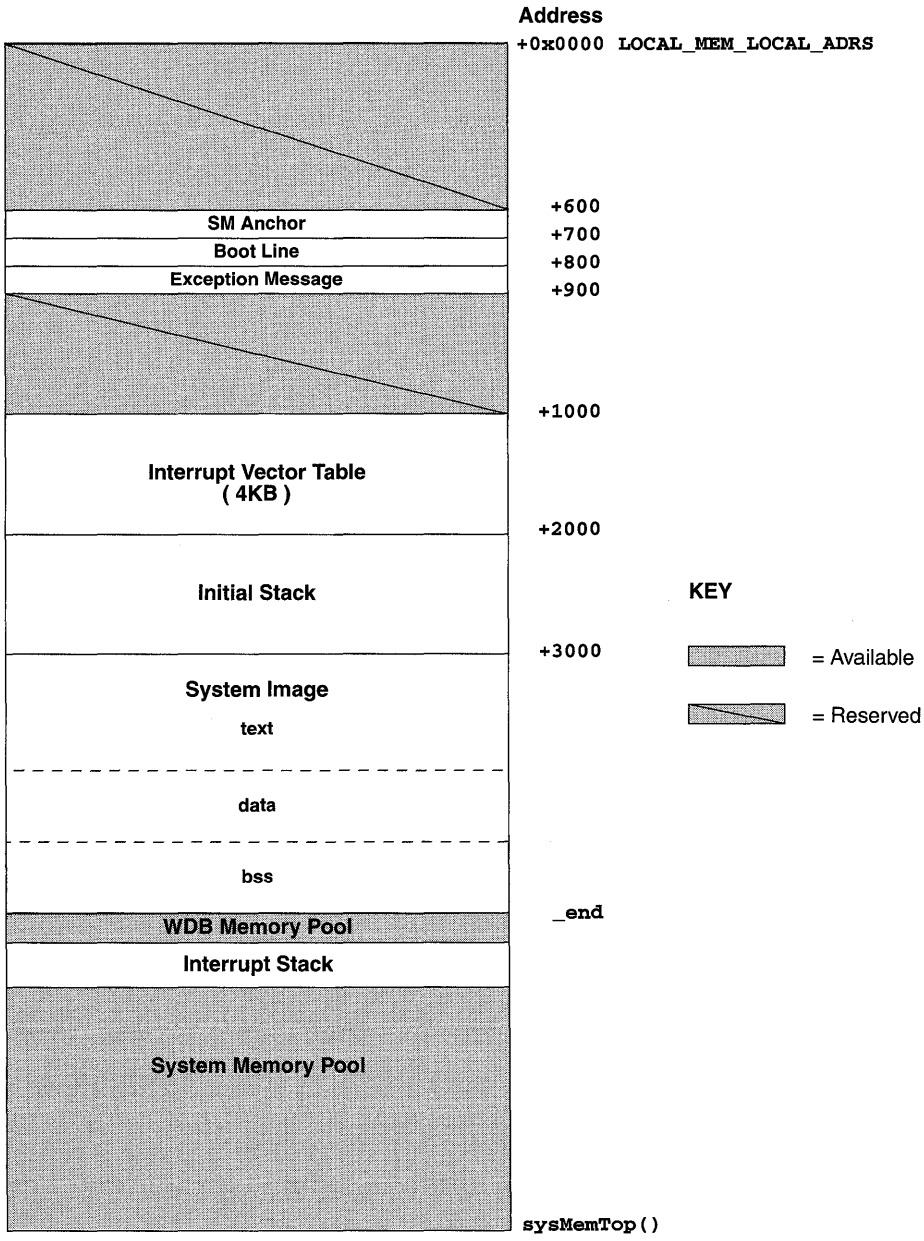
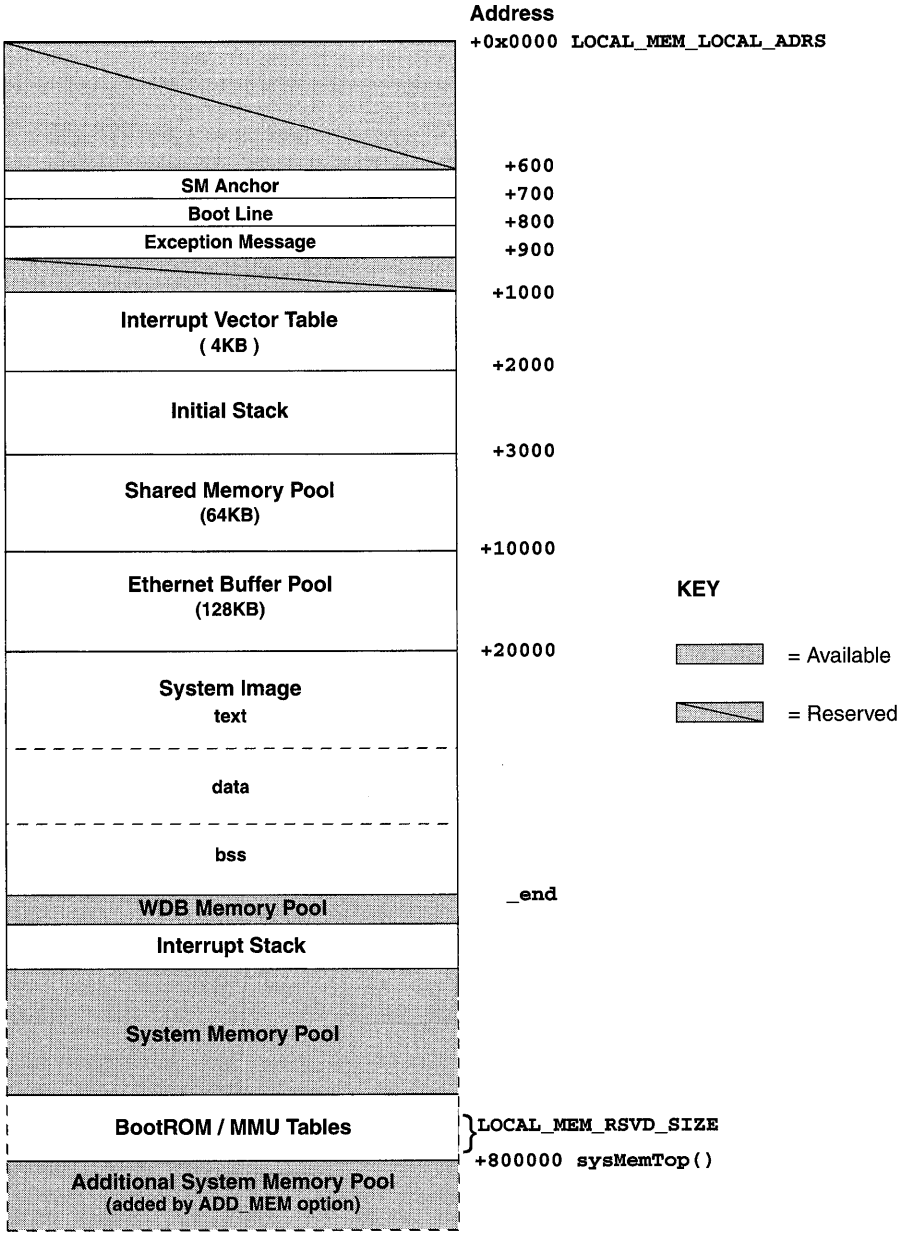


Figure B-2 **VxWorks System Memory Layout (microSPARC I & II)**



C

Intel i960

List of Figures

C.1 Introduction

This appendix provides information specific to VxWorks development on Intel i960CA, JX, KA, and KB targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the i960 processors.
- Architecture Considerations: special features and limitations of the i960 processors.

C.2 Building Applications

The following sections describe a configuration constant, an environment variable, and compiler options that together specify the information the GNU ToolKit requires to compile correctly for i960 targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate architecture-specific features enabled. Define this variable to one of the following values, to match the processor you are using:

- **I960CA**
- **I960JX**
- **I960KA**
- **I960KB**

For example, to define **CPU** for a i960CA on the compiler command line, specify the following command-line option when you invoke the compiler:

```
-DCPU=I960CA
```

To provide the same information in a header or source file instead, include the following line in the file:

```
#define CPU I960CA
```

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables.

For developers using UNIX hosts, you must specifically define this variable. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

For developers using Windows hosts, if you are working through the Tornado IDE, the appropriate variable(s) are set automatically. However, before invoking the compiler from a DOS command line, first run the following batch file to set the variable(s):

```
%WIND_BASE%/host/x86-win32/bin/torVars.bat
```

For more information, see the *Tornado User's Guide: Getting Started.*

Compiling C or C++ Modules

The following is an example of a compiler command line for i960 cross-development. The file to be compiled in this example has a base name of **applic**.

```
% cc960 -fno-builtin -I $WIND_BASE/target/h -O -c -mca\
-mstrict-align -fvolatile -nostdinc -DCPU=I960CA applic.c
```

The options shown in the example have the following meanings:[1]

1. For more information on these and other compiler options, see the *GNU ToolKit User's Guide.* WRS supports compiler options used in building WRS software; a list of these options is included in the *Guide.* Other options are not supported, although they are available with the tools as shipped.

-fno-builtin Required; uses library calls even for common library subroutines.

-I $WIND_BASE/target/h
Required; includes VxWorks header files. (Additional **-I** flags may be included to specify other header files.)

-O Optional; performs standard optimization.

-c Required; specifies that the module is to be compiled only, and not linked for execution under the host.

-mca Required for i960CA and i960JX; specifies the instruction set. For the i960KA and KB, use **-mka** and **-mkb**, respectively.

-mstrict-align Required; do not permit unaligned accesses.

-fvolatile Required; consider all memory references through pointers to be volatile.

-nostdinc Required; searches only the directory(ies) specified with the **-I** flag (see above) and the current directory for header files. Does not search host-system include files.

-DCPU=I960CA Required; defines the CPU type. If you are using an i960 processor other than the CA, specify the appropriate value (see *Defining the CPU Type*, p.527).

applic.*language_id*
Required; the file(s) to compile. For C compilation, specify a suffix of **.c**. For C++ compilation, specify a suffix of **.cpp**. The output is an unlinked object module in **a.out** format with the suffix **.o**; for the example, the output is **applic.o**.

During C++ compilation, the compiled object module (**applic.o**) is *munched*. Munching is the process of scanning an object module for non-local static objects, and generating data structures that VxWorks run-time support can use to call the objects' constructors and destructors. See the *Tornado User's Guide: Cross-Development* for details.

C.3 Interface Variations

This section describes particular routines that are specific to i960 targets in any of the following ways:

- available only on i960 targets
- parameters specific to i960 targets
- special restrictions or characteristics on i960 targets

For complete documentation on these routines, see the reference entries.

Initialization

There are several differences in what ***sysInit( )*** initializes and in the initialization sequence on i960 targets.

Differences in *sysInit( )* Routine

For the i960, the ***sysInit( )*** routine initializes the system interrupt and fault tables with default stubs, in addition to its standard functions.

ROM-Based VxWorks with i960 Targets

As with other target architectures, the routines ***romInit( )*** and ***romStart( )*** execute first. Then initialization continues at the ***sysInit( )*** call, rather than with the ***usrInit( )*** call as for other ROM-based targets.

Data Breakpoint Routine *bh( )*

In addition to being able to break at an instruction with ***b( )***, the i960CA permits breakpoints at a data address using ***bh( )***.

For example, the following command from the VxWorks shell causes a data breakpoint on any access to data address 0xFFFF:

```
-> bh 0xFFFF, 3
```

For more information, see the reference entry for ***bh( )***.

NOTE: The ***bh( )*** routine does not work reliably on instruction fetches; use ***b( )*** to break on instructions.

The delete-breakpoint routines, ***bd( )*** and ***bdall( )***, delete both instruction and data breakpoints. Only two data breakpoints can be present in the system at one time.

Parameter Change for intLevelSet( )

The i960 version of ***intLevelSet( )*** takes an argument from 0 to 31. Level 31 is equivalent to locking all interrupts.

Results Change for memLib

In VxWorks for the i960, the library **memLib** forces both partitions and blocks returned by ***malloc( )*** to be 16-byte aligned.

Math Routines

Mathematics routines using software floating-point emulation are part of the GNU/960 distribution from Cygnus, in the libraries **libm.a**, **libg.a**, and **libgcc.a**. The location of these libraries is described by the variable **LIBS** in **h/make/make.I960*xx*gnu** (where *xx* identifies libraries specific to the CA, JX, KA, or KB variant of the i960 architecture).

The following double-precision floating-point routines are included in the GNU/960 distribution from Cygnus:

acos( )	*asin( )*	*atan( )*	*atan2( )*	*ceil( )*	*cos( )*	*cosh( )*
exp( )	*fabs( )*	*floor( )*	*fmod( )*	*log( )*	*log10( )*	*log2( )*
pow( )	*sin( )*	*sinh( )*	*sqrt( )*	*tan( )*	*tanh( )*	

The following single-precision floating-point routines are also available:

atanf( )	*atan2( )*	*ceilf( )*	*expf( )*
fabsf( )	*floorf( )*	*logf( )*	*log2f( )*
powf( )	*sinf( )*	*sqrtf( )*	*tanf( )*

Adding in Unresolved Routines

Occasions can arise when an application requires **libm.a**, **libg.a**, and **libgcc.a** routines, although the application has *not* been prelinked with the VxWorks image. There are several alternatives for dealing with this situation:

- You can compile and link a set of dummy calls with VxWorks to ensure that the necessary routines are included in the VxWorks image.
- You can explicitly link the appropriate archive with your application module by using **ld960**.
- You can add any unresolved reference symbols to **src/config/mathInit.c** and rebuild VxWorks.

Floating-Point Task Option: VX_FP_TASK

The i960CA, JX, and KA processors contain no floating-point hardware; thus no floating-point context is used. Floating-point emulation is performed in software with the routines provided by the Cygnus libraries (see *Math Routines*, p.531); therefore, the task option **VX_FP_TASK** is not required.

The i960KB has on-board floating-point hardware. The task option **VX_FP_TASK** is required when spawning tasks on the i960KB processor.

C.4 Architecture Considerations

This section describes the following characteristics of the i960 architecture that you should keep in mind as you write a VxWorks application:

- Byte order
- Double-word Integers
- VMEbus interrupt handling
- Memory layout

Byte Order

The i960 architecture uses little-endian byte order. For information about macros and routines to convert byte order (from big-endian to little-endian and vice versa), see *Network Byte Order*, p.250.

The VxWorks loader allows object module headers to be in either big-endian or little-endian byte order. Host utility programs can use the most convenient byte order to process i960 objects. Object file text and data segments must be little endian for i960 processors.

Double-word Integers: long long

The double-word integer **long long** is not supported, except as an artifact of your particular architecture and compiler. For more information about handling unsupported features, please see the *Customer Support User's Guide.*

VMEbus Interrupt Handling

The i960 uses 31 interrupt levels instead of the seven used by VMEbus. The mapping of the seven VMEbus interrupts to the 31 i960 levels is board dependent. VMEbus interrupts must be acknowledged with ***sysBusIntAck( )***. VxWorks does not use the vector submitted by the interrupting device.

For more information, see the file **h/arch/i960/ivI960.h**.

Memory Layout

The figures on the following pages show the layout of a VxWorks system in memory for various target architectures. Areas contain the following labels:

Interrupt Vector Table
Table of exception/interrupt vectors.

SM Anchor Anchor for the shared memory network (if there is shared memory on the board).

Boot Line ASCII string of boot parameters.

Exception Message ASCII string of the fatal exception message.

Initial Stack Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image Entry point for VxWorks.

WDB Memory Pool Size depends on the macro **WDB_POOL_SIZE** which defaults to one-sixteenth of the system memory pool. This space is used by the target server to support host-based tools.

Interrupt Stack Size defined in **configAll.h**. Location depends on system image size.

System Memory Pool
Size depends on size of system image and interrupt stack. The end of the free memory pool for this board is returned by ***sysMemTop( )***.

Figure C-1 shows the memory layout for an i960CA target; Figure C-2 shows the memory layout for an i960JX target; Figure C-3 shows the memory layout for an i960KA or i960KB target.

All addresses shown in these figures are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

Figure C-1 **VxWorks System Memory Layout (i960CA)**

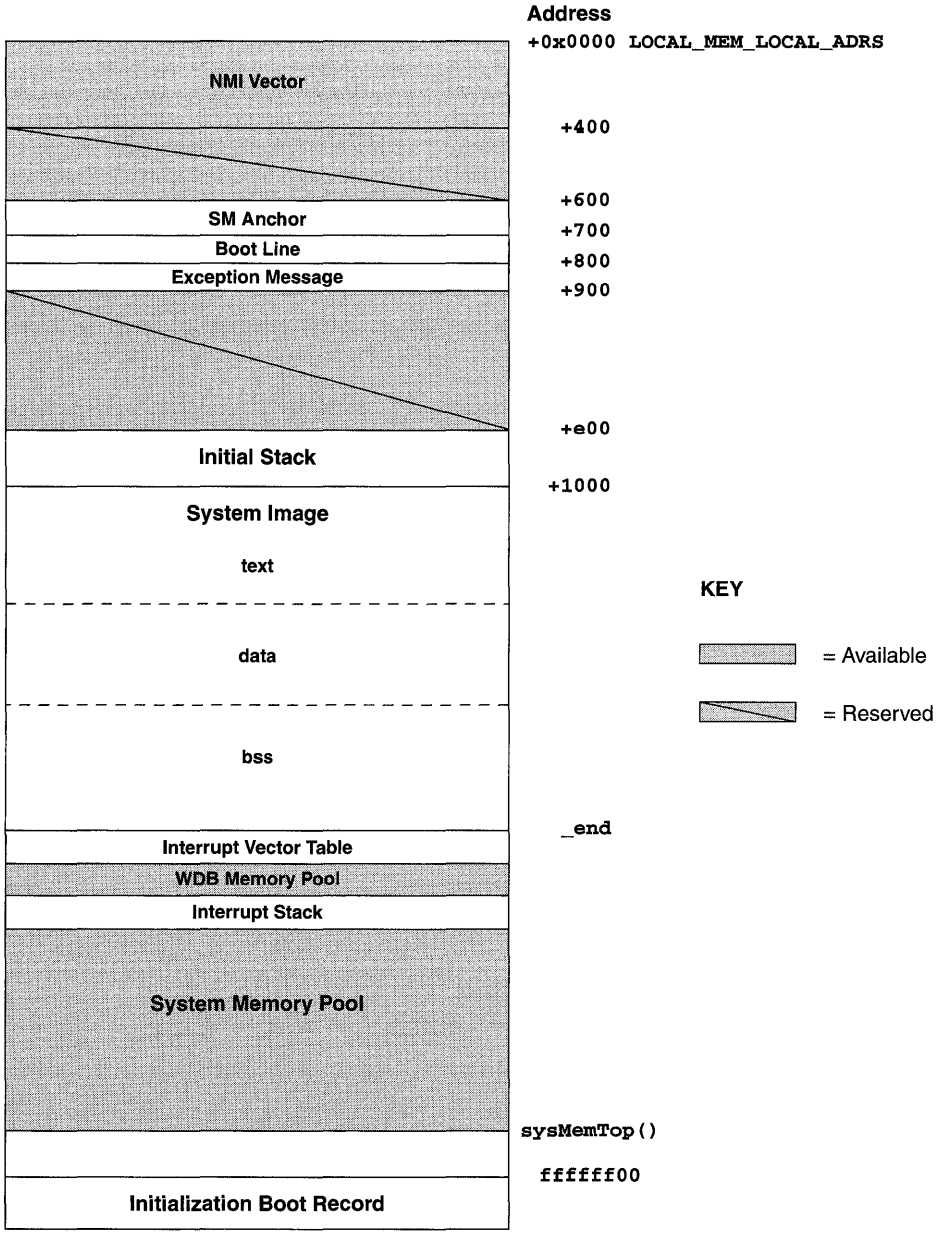


Figure C-2 **VxWorks System Memory Layout (i960JX)**

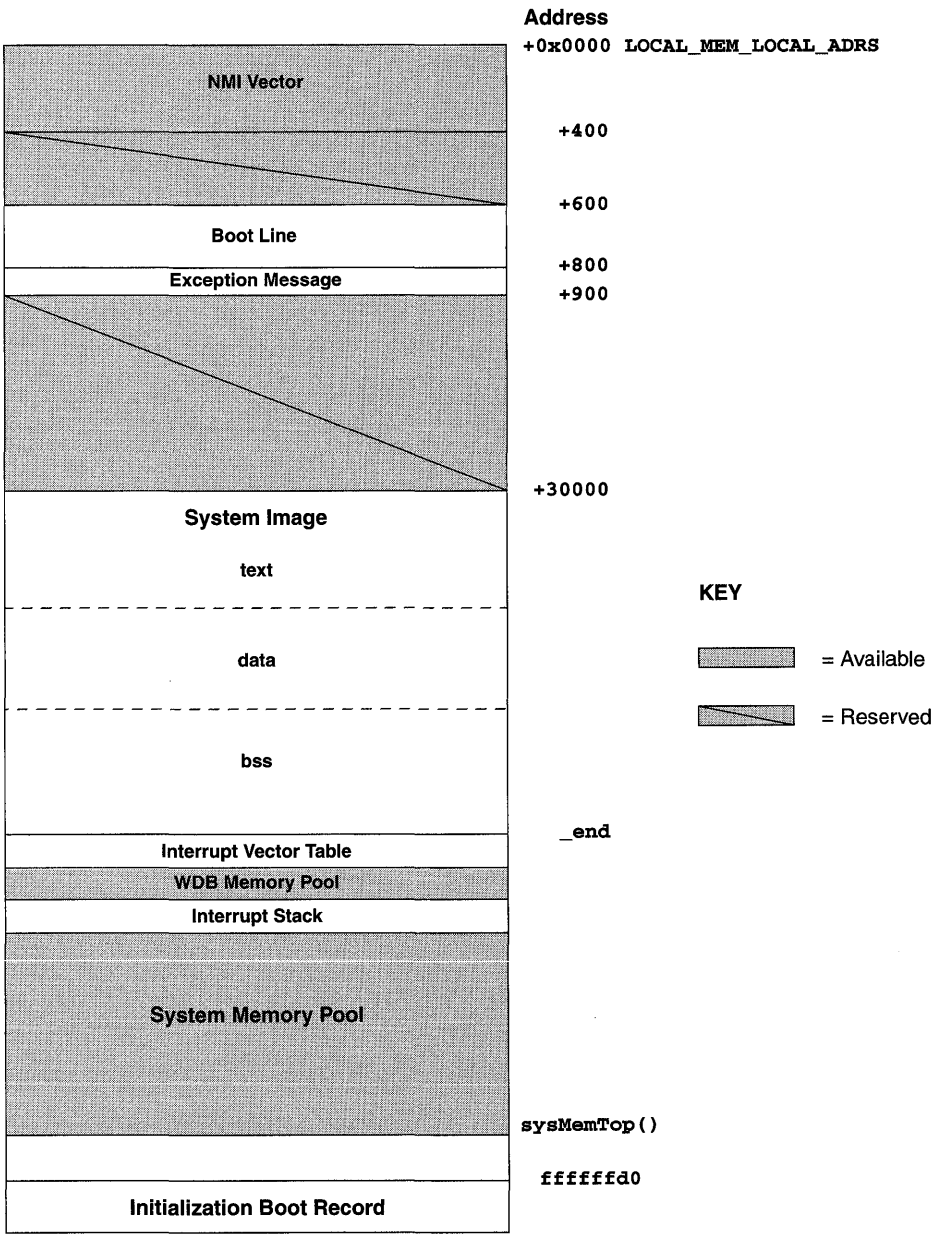
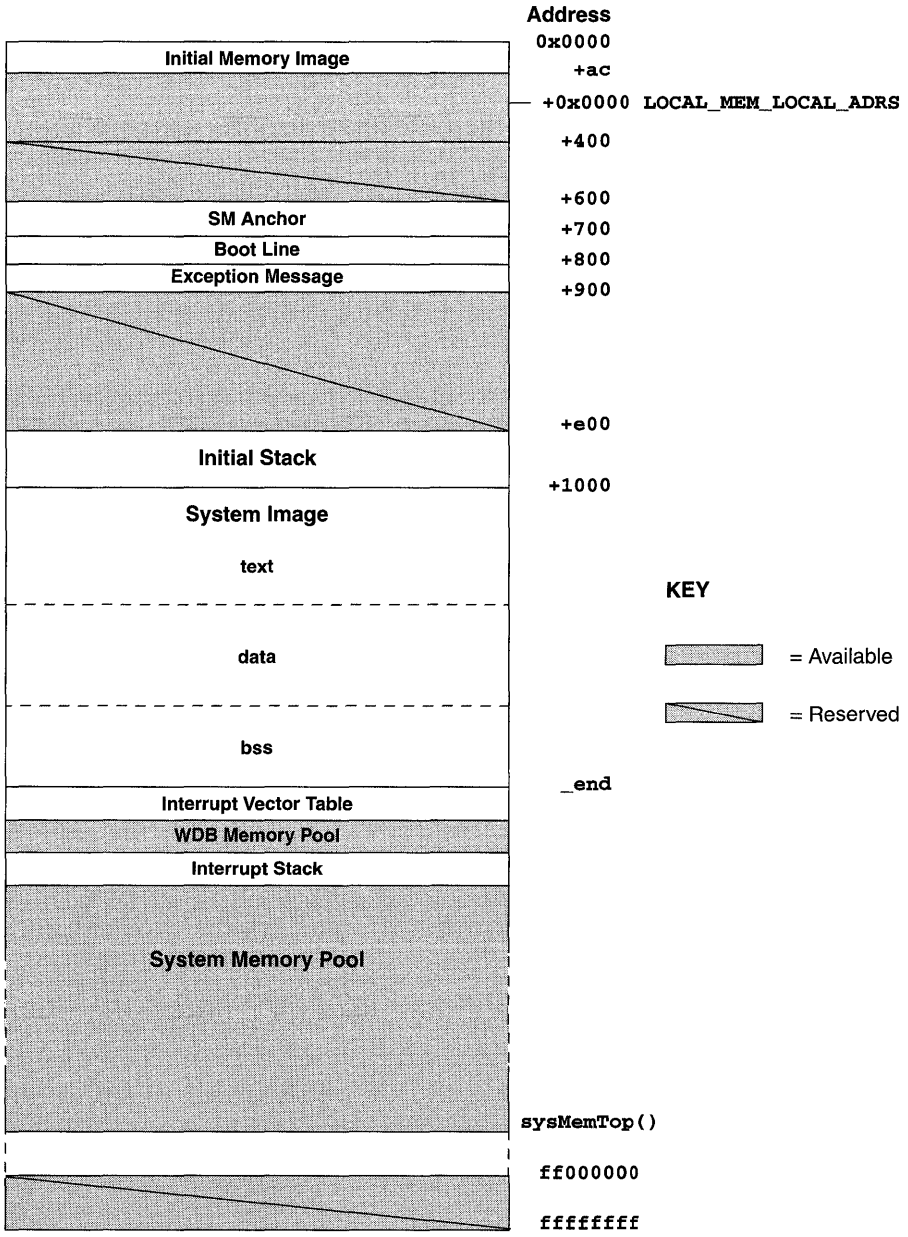


Figure C-3 **VxWorks System Memory Layout (i960KA and i960KB)**



D

Intel x86

List of Tables

List of Figures

D.1 Introduction

This appendix provides information specific to VxWorks development on Intel i386, i486, and Pentium (x86) targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the x86 processors.
- Architecture Considerations: special features and limitations of the x86 processors.
- Board Support Packages: information on specific BSPs and device drivers.

For general information on the Tornado development environment's cross-development tools, see the *Tornado User's Guide: Cross-Development*.

D.2 Building Applications

The following sections describe a configuration constant, an environment variable, and compiler options that together specify the information the GNU ToolKit requires to compile correctly for x86 targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate architecture-specific features enabled. Define this variable to either **I80386** or **I80486**, to match the processor you are using.

For example, to define **CPU** for an i386 on the compiler command line, specify the following command-line option when you invoke the compiler:

```
-DCPU=I80386
```

To provide the same information in a header or source file, include the following line in the file:

```
#define CPU I80386
```

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables.

For developers using UNIX hosts, you must specifically define this variable. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

For developers using Windows hosts, if you are working through the Tornado IDE, the appropriate variable(s) are set automatically. However, before invoking the compiler from a DOS command line, first run the following batch file to set the variable(s):

```
%WIND_BASE%/host/x86-win32/bin/torVars.bat
```

For more information, see the *Tornado User's Guide: Getting Started.*

Compiling C and C++ Modules

The following is an example of a compiler command line for Intel x86 cross-development. The file to be compiled in this example has the base name of **applic**.

```
% cc386 -DCPU=I80386 -I $WIND_BASE/target/h -fno-builtin -O \
-mno-486 -fno-defer-pop -nostdinc -c applic.lang_id
```

The options shown in the example have the following meanings:[1]

1. For more information on these and other compiler options, see the *GNU ToolKit User's Guide.* WRS supports compiler options used in building WRS software; a list of these options is included in the *Guide.* Other options are not supported, although they are available with the tools as shipped.

cc386	Required; use **cc386** for all supported x86 processors.
-DCPU=I80386	Required; defines the CPU type for the i386. If you are using the i486 or Pentium, specify **I80486**.
-I $WIND_BASE/target/h	Required; includes VxWorks header files. (Additional **-I** flags may be included to specify other header files.)
-fno-builtin	Required; uses library calls even for common library routines.
-O	Optional; performs standard optimizations. Note that optimization is not supported for the Pentium.
-mno-486	Required for the i386; generates optimized code for the i386. For the i486, the compiler automatically generates optimized code; no additional flags are required.
-fno-defer-pop	Required; pops the arguments to each subroutine call as soon as that subroutine returns.
-nostdinc	Required; searches only the directory(ies) specified with the **-I** flag (see above) and the current directory for header files. Does not search host-system include files.
-c	Required; specifies that the module is to be compiled only, and not linked for execution under the host.
applic.*lang_id*	Required; the file(s) to compile. For C compilation, specify a suffix of **.c**. For C++ compilation, specify a suffix of **.cpp**. The output is an unlinked object module in **a.out** format with the suffix **.o**; for the example, the output is **applic.o**. During C++ compilation, the compiled object module (**applic.o**) is *munched*. Munching is the process of scanning an object module for non-local static objects, and generating data structures that VxWorks run-time support can use to call the objects' constructors and destructors. For details, see the *Tornado User's Guide: Cross-Development*.

D.3 Interface Variations

This section describes particular features and routines that are specific to x86 targets in any of the following ways:

- available only for x86 targets
- parameters specific to x86 targets
- special restrictions or characteristics on x86 targets

For complete documentation, see the reference entries.

Supported Routines in mathALib

For x86 targets, the following floating-point routines are supported. These routines are also available without a hardware floating-point processor by defining **INCLUDE_SW_FP** in **config.h** or **configAll.h**. For more information about configuring the software floating-point emulation library, see *Software Floating-Point Emulation*, p.552. See **mathALib** and the individual manual entries for descriptions of each routine.

acos( )	*asin( )*	*atan( )*	*atan2( )*	*ceil( )*	*cos( )*
exp( )	*fabs( )*	*floor( )*	*fmod( )*	*infinity( )*	*irint( )*
iround( )	*log( )*	*log10( )*	*log2( )*	*pow( )*	*round( )*
sin( )	*sincos( )*	*sqrt( )*	*tan( )*	*trunc( )*	

Architecture-Specific Global Variables

The file **sysLib.c** contains the global variables shown in Table D-1.

Table D-1 **Architecture-Specific Global Variables**

Global Variable	Value	Description
sysVectorIRQ0	0x20 (default)	A mapping of the base vector for IRQ0.
sysIntIdtType	0x0000fe00 (default) = trap gate 0x0000ee00 = interrupt gate	Used when VxWorks initializes the interrupt vector table. The choice of trap gate vs. interrupt gate affects all interrupts (vectors 0x20 through 0xff).

Table D-1 **Architecture-Specific Global Variables** *(Continued)*

Global Variable	Value	Description
sysGDT[]	0x3ff limit (default)	The Global Descriptor Table has five entries. The first is a null descriptor. The second and third are for task-level routines. The fourth is for interrupt-level routines. The fifth is reserved.
sysProcessor	0 = i386 1 = i486 2 = Pentium	The processor type (set by VxWorks).
sysCoprocessor	0 = no coprocessor 1 = 387 coprocessor 2 = 487 coprocessor	The type of floating-point coprocessor (set by VxWorks).

Architecture-Specific Routines

Register Routines

The following routines read x86 register values, and require one parameter, the task ID:

eax( )	*ebx*( )	*ecx*( )	*edx*( )	*edi*( )
esi( )	*ebp*( )	*esp*( )	*eflags*( )	

Table D-2 shows additional architecture-specific routines. Other architecture-specific routines are described throughout this section.

Breakpoints and the bh( ) Routine

VxWorks for the x86 supports both software and hardware breakpoints. When you set a software breakpoint, VxWorks replaces an instruction with an **int 3** software interrupt instruction. VxWorks restores the original code when the breakpoint is removed. The instruction queue is purged each time VxWorks changes an instruction to a software break instruction.

A hardware breakpoint uses the processor's debug registers to set the breakpoint. The x86 architectures have four breakpoint registers. If you are using the target shell, you can use the ***bh( )*** routine to set hardware breakpoints. The routine is declared as follows:

Table D-2 **Architecture-Specific Routines**

Routine	Function Header	Description
sysInByte( )	`UCHAR sysInByte (int port)`	Read one byte from I/O.
sysInWord( )	`USHORT sysInWord (int port)`	Read one word (two bytes) from I/O.
sysInLong( )	`ULONG sysInLong (int port)`	Read one long word (four bytes) from I/O.
sysOutByte( )	`void sysOutByte (int port, char data)`	Write one byte to I/O.
sysOutWord( )	`void sysOutWord (int port, short data)`	Write one word (two bytes) to I/O.
sysOutLong( )	`void sysOutLong (int port, long data)`	Write one long word (four bytes) to I/O.
sysInWordString( )	`void sysInWordString (int port, short *address, int count)`	Read word string from I/O.
sysInLongString( )	`void sysInLongString (int port, short *address, int count`	Read long string from I/O.
sysOutWordString( )	`void sysOutWordString (int port, short *address, int count)`	Write word string to I/O.
sysOutLongString( )	`void sysOutLongString (int port, short *address, int count)`	Write long string to I/O.
sysDelay( )	`void sysDelay (void)`	Allow enough recovery time for port accesses.
sysIntDisablePIC( )	`STATUS sysIntDisablePIC (int intLevel)`	Disable a Programmable Interrupt Controller (PIC) interrupt level.
sysIntEnablePIC( )	`STATUS sysIntEnablePIC (int intLevel)`	Enable a PIC interrupt level.
sysCpuProbe( )	`UINT sysCpuProbe (void)`	Check for type of CPU (i386, i486, or Pentium).

```
STATUS bh
    (
    INSTR       *addr,      /* where to set breakpoint, or      */
                            /* 0 = display all breakpoints     */
    int         task,       /* task to set breakpoint;          */
                            /* 0 = set all tasks                */
    int         count,      /* number of passes before hit      */
    int         type,       /* breakpoint type; see below       */
    INSTR       *addr0      /* ignored for x86 targets          */
    )
```

The *bh*( ) routine takes the following types in parameter *type*:

BRK_INST	Instruction hardware breakpoint (0x1000)
BRK_DATAW1	Data write 1-byte breakpoint (0x1400)
BRK_DATAW2	Data write 2-byte breakpoint (0x1500)
BRK_DATAW4	Data write 4-byte breakpoint (0x1700)
BRK_DATARW1	Data read-write 1-byte breakpoint (0x1c00)
BRK_DATARW2	Data read-write 2-byte breakpoint (1d00)
BRK_DATARW4	Data read-write 4-byte breakpoint (1f00)

Disassembler: l( )

If you are using the target shell, note that the VxWorks disassembler ***l*( )** does not support 16-bit code compiled for earlier generations of 80*x*86 processors. However, the disassembler does support 32-bit code for both the i386 and i486 processors.

vxMemProbe( )

The ***vxMemProbe*( )** routine, which probes an address for a bus error, is supported on the x86 architectures by trapping both general protection faults and page faults.

D.4 Architecture Considerations

This section describes the following characteristics of the Intel x86 architectures that you should keep in mind as you write a VxWorks application:

- Operating mode, privilege protection, and byte order
- Memory segmentation and the MMU
- Memory considerations for VME
- Interrupts and exceptions
- Context switching
- ISA/EISA bus
- PC104 bus
- PCI bus
- Software floating-point emulation
- VxWorks memory layout

Consult Intel's *Intel486 Microprocessor Family Programmer's Reference Manual* for details on the x86 architectures.

Operating Mode, Privilege Protection, and Byte Order

VxWorks for the x86 runs in the 32-bit protected mode.

No privilege protection is used, thus there are no call gates. The privilege level is always 0, the most privileged level (Supervisor mode).

The x86 byte order is little-endian, but network applications must convert some data to a standard network order, which is big-endian. In particular, in network applications, be sure to convert the port number to network byte order using ***htons*( )**.

See *Network Byte Order*, p.250 for more information about macros and routines to convert byte order (from little-endian to big-endian or vice versa).

Memory Segmentation

The Intel x86 processors support both I/O-mapped devices and memory-mapped devices.

I/O Mapped Devices

For I/O mapped devices, developers may use the following routines from **config/pcx86/sysALib.s**:

sysInByte( )	– input one byte from I/O space
sysOutByte( )	– output one byte to I/O space
sysInWord( )	– input one word from I/O space
sysOutWord( )	– output one word to I/O space
sysInLong( )	– input one long word from I/O space
sysOutLong( )	– output one long word to I/O space
sysInWordString( )	– input a word string from I/O space
sysOutWordString( )	– output a word string to I/O space
sysInLongString( )	– input a long string from I/O space
sysOutLongString( )	– output a long string to I/O space

Memory Mapped Devices

For memory mapped devices, there are two kinds of memory protection provided by VxWorks: the Memory Management Unit and the Global Descriptor Table. Because VxWorks operates at the highest processor privilege level, no "protection rings" exist.

The x86 processors allow you to configure the memory space into valid and invalid areas, even under Supervisor mode. Thus, you receive a page fault only if the processor attempts to access addresses mapped as invalid, or addresses that have not been mapped. Conversely, if the processor attempts to access a nonexistent address space that has been mapped as valid, no page fault occurs.

- **Memory Management Unit (MMU)**

  If **INCLUDE_MMU_BASIC** is defined, then VxWorks enables the MMU with the **mmuPhysDesc[]** table which includes PCI memory mapping information. This is the default.

  If you have other memory mapped devices and if **INCLUDE_MMU_BASIC** is defined in **config.h** (the default), you may need to add your device address space into the MMU table by manually editing the MMU configuration structure **sysPhysMemDesc[]** in **sysLib.c**. For information on editing the **sysPhysMemDesc[]** structure, see *7.3 Virtual Memory Configuration*, p.408. Do not overlap any existing MMU entries, and be sure all entries are page aligned. We recommend that you also maintain a 1:1 correlation between virtual and physical memory, since VxWorks and all tasks use a common address space.

Attempts to access areas not mapped as valid in the MMU result in page faults.

NOTE: The i386 MMU does not have write-protect capability.

- **Global Descriptor Table (GDT)**

  The GDT is defined as the table **sysGDT[]** in **sysALib.s**. The table has five entries: a null entry, an entry for program code, an entry for program data, an entry for ISRs, and a reserved entry. It is initially set so that the available memory range is 0x0-0xffffffff. If **INCLUDE_PCI** is defined, VxWorks does not alter this setting. This memory range is available at run-time with the MMU configuration.

  If **INCLUDE_PCI** is not defined, VxWorks adjusts the GDT using the ***sysMemTop*( )** routine to check the actual memory size during system initialization and set the table so that the available memory range is 0x0-**sysMemTop**. This causes a General Protection Fault to be generated for any memory access outside the memory range 0x0-**sysMemTop**.

Memory Considerations for VME

The global descriptors for x86 targets are configured for a flat 4GB memory space.

If you are running VxWorks for the x86 on a VME board, be aware that addressing nonexistent memory or peripherals does not generate a bus error or fault.

Interrupts and Exceptions

The Interrupt Descriptor Table (IDT) occupies the address range 0x0 to 0x800 (also called the Interrupt Vector Table, see Figure D-1). Vector numbers 0x0 to 0x1f are handled by the default exception handler. Vector numbers 0x20 to 0xff are handled by the default interrupt handler.

By default, vector numbers 0x20 to 0x2f are mapped to IRQ levels 0 to 15. To redefine the base address, edit **sysVectorIRQ0** in **sysLib.c**.

For vector numbers 0x0 to 0x11, no task gates are used, only interrupt gates. By default, vector numbers 0x12 to 0xff are trap gates, but this can be changed by redefining the global variable **sysIntIdtType**.

The difference between an interrupt gate and a trap gate is its effect on the IF flag: using an interrupt gate clears the IF flag, which prevents other interrupts from interfering with the current interrupt handler.

Each vector of the IDT contains the following information:

offset	offset to the interrupt handler
selector	0x0020, fourth descriptor (code) in GDT
descriptor privilege level	3
descriptor present bit	1

The interrupt handler calls ***intEnt( )*** and saves the volatile registers (**eax**, **edx**, and **ecx**). It then calls the ISR, which is usually written in C. Finally, the handler restores the saved registers and calls ***intExit( )***.

There is no designated interrupt stack. The interrupt's stack frame is built on the interrupted task's stack. Thus, each task requires extra stack space for interrupt nesting; the amount of extra space varies, depending on your ISRs and the potential nesting level.

Some device drivers (depending on the manufacturer, the configuration, and so on) generate a stray interrupt on IRQ7, which is used by the parallel driver. The global variable **sysStrayIntCount** (see Table D-3) is incremented each time such an interrupt occurs, and a dummy ISR is connected to handle these interrupts.

The chip generates an exception stack frame in one of two formats, depending on the exception type: (EIP + CS + EFLAGS) or (ERROR + EIP + CS + EFLAGS).

Double-word Integers: long long

The double-word integer **long long** is not supported, except as an artifact of your particular architecture and compiler. For more information about handling unsupported features, please see the *Customer Support User's Guide*.

Context Switching

Hardware multitasking and the TSS descriptor are not used. VxWorks creates a dummy exception stack frame, loads the registers from the TCB, and then starts the task.

ISA/EISA Bus

The optional PC-compatible hardware cards supported in this release (the Ethernet adapter cards and the Blunk Microsystems ROM Card) use the ISA/EISA bus architecture.

PC104 Bus

The PC104 bus is supported and tested with the NE2000-compatible Ethernet card (4i24: Mesa Electronics). Ampro's Ethernet card (Ethernet-II) is also supported.

PCI Bus

The PCI bus is supported and tested with the Intel EtherExpress PRO100B Ethernet card. Several functions to access PCI configuration space are supported. Functions addressed here include:

- Locate the device by deviceID and vendorID.
- Locate the device by classCode.
- Generate the special cycle.
- Access its configuration registers.

Software Floating-Point Emulation

The software floating-point library is supported for the x86 architectures; define **INCLUDE_SW_FP** in **configAll.h** or **config.h** to include the library in your system image. This library emulates each floating point instruction, by using the exception "Device Not Available." For other floating-point support information, see *Supported Routines in mathALib*, p.544.

VxWorks Memory Layout

Two memory layouts for the x86 are shown in the following figures: Figure D-1 illustrates the typical upper memory configuration, while Figure D-2 shows a lower memory option. These figures contain the following labels:

Interrupt Vector Table
: Table of exception/interrupt vectors.

GDT
: Global descriptor table.

: Anchor for the shared memory network (if there is shared memory on the board).

Boot Line
: ASCII string of boot parameters.

Exception Message
: ASCII string of the fatal exception message.

FD DMA Area
: Diskette (floppy device) direct memory access area.

Initial Stack
: Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image
: Entry point for VxWorks.

WDB Memory Pool
: Size depends on the macro **WDB_POOL_SIZE** which defaults to one-sixteenth of the system memory pool. This space is used by the target server to support host-based tools.

Interrupt Stack
: Size defined in **configAll.h**. Location depends on system image size. Note that although an interrupt stack is allocated, it is not used.

System Memory Pool
: Size depends on size of system image and interrupt stack. The end of the free memory pool for this board is returned by ***sysMemTop( )***.

All addresses shown in Figure D-1 are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

In general, the boot image is placed in lower memory and the VxWorks image is placed in upper memory, leaving a gap between lower and upper memory. Some BSPs have additional configurations which must fit within their hardware constraints. For details, see the reference entry for each specific BSP.

Figure D-1 **VxWorks System Memory Layout (x86 Upper Memory)**

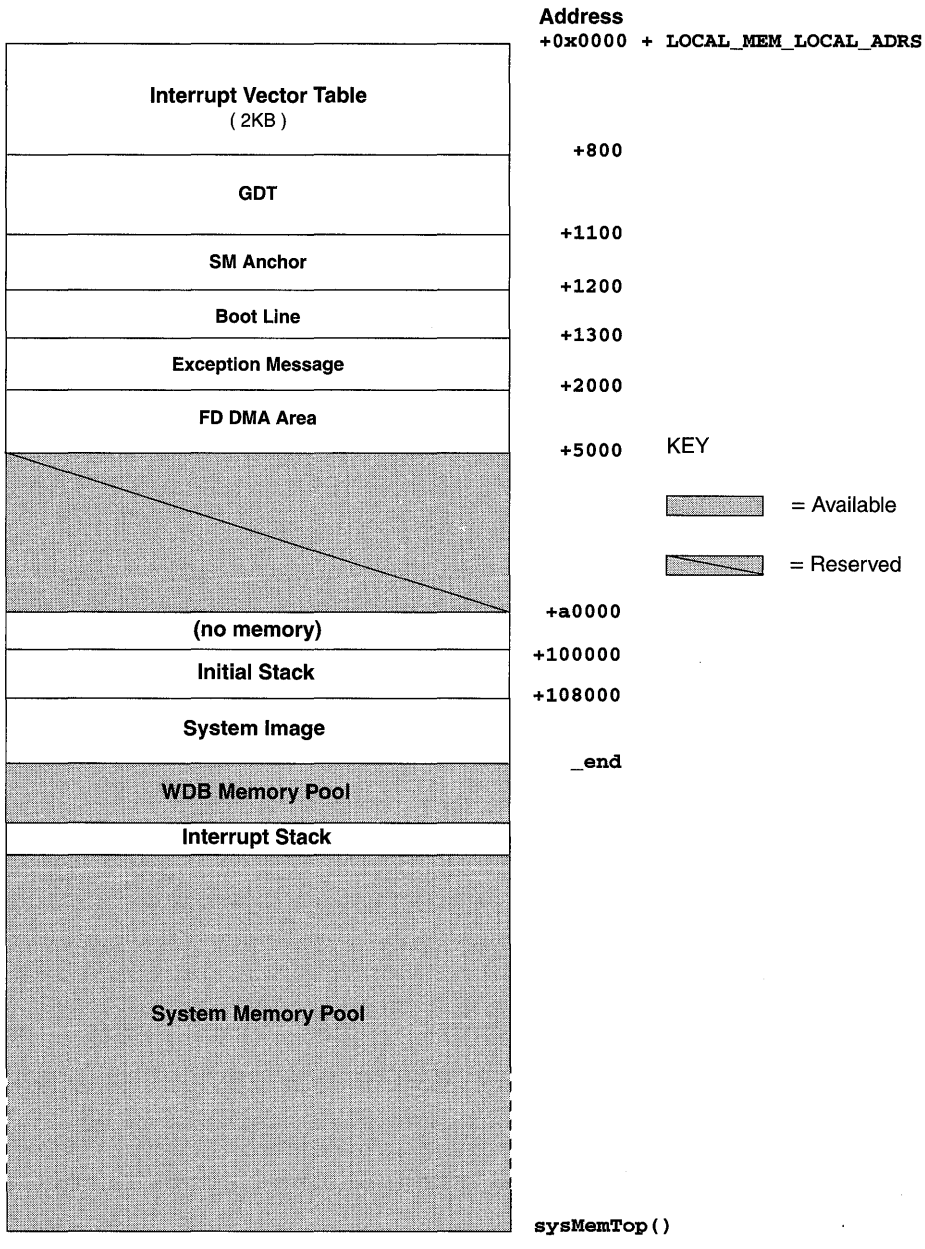
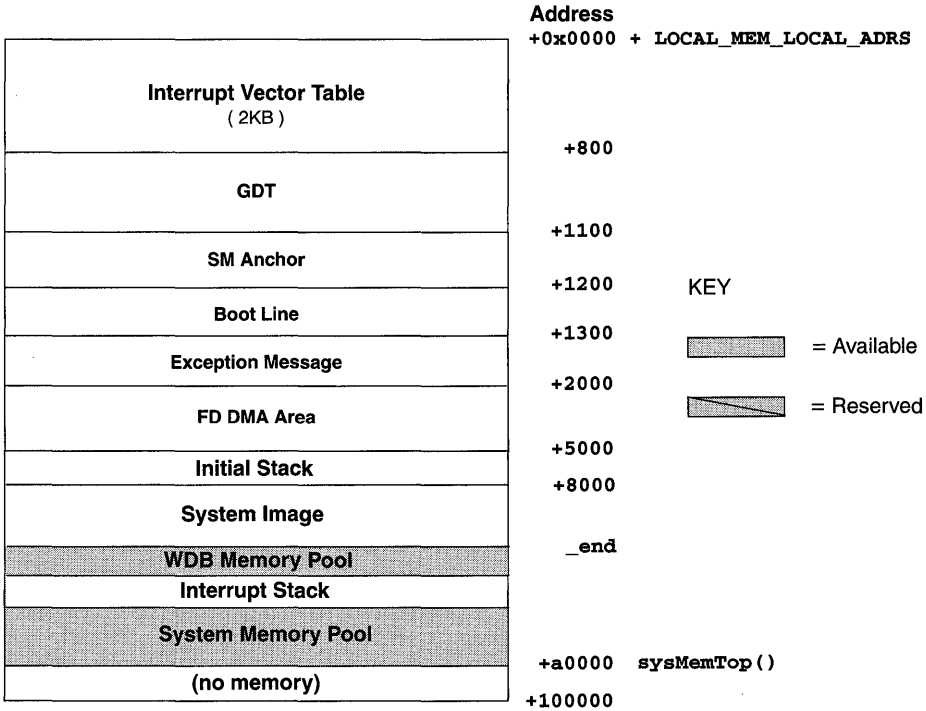


Figure D-2 **VxWorks System Memory Layout (x86 Lower Memory)**



D.5 Board Support Packages

Boot Considerations for pc386, pc486, and epc4

For general information on booting VxWorks, see the *Tornado User's Guide: Getting Started*.

This section describes how to build a boot disk, how to boot VxWorks, and how to mount a DOS file system. Besides the standard tapes, VxWorks for x86 targets also includes the following DOS diskettes (in both 5.25" (1.2MB) and 3.5" (1.44MB) formats):

- The diskette labeled "VxWorks Utility Disk" contains the DOS executables **vxsys.com**, **vxcopy.exe**, **vxload.com**, and **mkboot.bat**.
- The diskette labeled "VxWorks Boot Disk" contains the VxWorks boot sector file, **bootrom.sys**; the minimal boot program, **bootrom_uncmp** (renamed **bootrom.dat**); and standalone VxWorks, **vxWorks.st** (compiled with **BOOTABLE** undefined in **config.h**). These files work for the i386, the i486, or the Pentium.

These utilities help you build new boot disks and are described in the following subsections.

NOTE: These utilities are also included in the Tornado tree at **usr/wind/host/x86-win32/bin**.

Boot Process

Intel x86 targets that are IBM PC compatible have a BIOS ROM. The BIOS ROM is not overwritten by VxWorks. The VxWorks executables that you build on the development host—such as **vxWorks.st_rom** and **bootrom_uncmp**—are loaded by the standard PC bootstrap mechanism[2]; that is, the PC BIOS (basic input/output system) ROM.

In general terms, the PC BIOS boot process works as follows:

1. The BIOS boot process searches the first diskette drive's (**A:**) boot sector for a loader program.

2. You can use a boot ROM if you install the Blunk Microsystems ROM Card 1.0; see *ROM Card and EPROM Support*, p.566.

2. If a loader program was not in drive **A:**'s boot sector, the process searches the first hard disk's (**C:**) boot sector for a loader program.
3. The loader program is loaded into memory and executed.
4. Typically, the loader program loads the operating system from disk and executes it, completing the boot process.

For VxWorks, the process is the same except for step 4. Because the loader program is specific to VxWorks, it loads and executes the file **bootrom.sys**.

To build a VxWorks boot disk (either diskette or hard drive), you must replace the standard loader program with the VxWorks loader program and create the appropriate **bootrom.sys** file. The following subsections describe how to do this from VxWorks and from MS-DOS.

Building a Boot Disk/Diskette from VxWorks

The routine ***mkbootFd( )*** produces a VxWorks boot diskette, and ***mkbootAta( )*** produces a VxWorks boot disk (an IDE or ATA hard disk). Both run on any VxWorks x86 target. They are provided in **config/***bspname*/**mkboot.c**. Use a DOS-formatted disk or diskette.

⚠ NOTE: The ***mkbootFd( )*** routine supports only high-density diskettes.

The ***mkbootFd( )*** and ***mkbootAta( )*** routines write the boot sector so that it contains the VxWorks loader program, and make a boot image named **bootrom.sys**. The boot image can be derived from one of the images listed in *VxWorks Images*, p.565. Before making any version of the image, make sure that **DEFAULT_BOOT_LINE** in **config.h** is set correctly (see the *Tornado User's Guide: Getting Started*), and that the size of the boot image (text+data+bss) is verified to be less than 512KB. It cannot be larger than this, because it is written into lower memory.

During the booting process, the VxWorks loader program reads **bootrom.sys** and then jumps to the entry point of the boot image.

The ***mkbootFd( )*** routine requires the following parameters:

```
STATUS mkbootFd (int drive, int fdType, char *filename)
```

The first two parameters specify the drive number and diskette type, specified as in *Booting VxWorks from a Diskette, an ATA/IDE Disk, or a PC Card*, p.561. The third parameter specifies the file name of the boot image.

The ***mkbootAta( )*** routine requires the following parameters:

```
STATUS mkbootAta (int ctrl, int drive, char *filename)
```

The first two parameters specify the controller number and drive number, specified as in *Booting VxWorks from a Diskette, an ATA/IDE Disk, or a PC Card*, p.561. The third parameter specifies the file name of the boot image.

For example, to create a boot disk for the pc386 BSP, first use the following commands to create the **mkboot.o** object from **mkboot.c**:

```
% cd /usr/wind/target/config/pc386
% make mkboot.o
```

Then, from the Tornado shell, move to the appropriate directory, load **mkboot.o**, and then invoke ***mkbootFd( )*** or ***mkbootAta( )***. Remember to place a formatted, empty diskette in the appropriate drive if you use ***mkbootFd( )***.

In this example, ***mkbootAta( )*** builds a local IDE disk on drive **C:** from **bootrom_uncmp** with the default **ataResources[]** table (see *ATA/IDE Disk Driver*, p.572):

```
-> cd "/usr/wind/target/config/pc386"
-> ld < mkboot.o
-> mkbootAta 0,0,"bootrom_uncmp"
```

Building a Boot Disk/Diskette from MS-DOS

The VxWorks Utility Disk includes several utility programs for creating VxWorks boot disks. These utilities write the VxWorks loader program on a diskette's or a hard disk's boot sector, and then copy the VxWorks executables from the host to the disk in a format suitable for the loader program. The utilities mimic the corresponding MS-DOS utilities, but they must be run under a DOS session of Windows, not "pure" DOS. They are summarized as follows and described in more detail later in this section:

vxsys.com	installs a VxWorks loader program in a disk's boot sector.
vxcopy.exe	copies a VxWorks **a.out** executable to the boot disk in the required format.
vxload.com	loads and executes VxWorks from MS-DOS.
mkboot.bat	an MS-DOS batch file that creates boot disks.

- **Creating a Boot Disk for PC-Compatible Targets**

To create a diskette or a bootable hard disk for PC-compatible targets, follow these steps:

1. On the development host, change to the BSP directory, for example, **config/pc386**. Use **make** to produce the minimal boot program (the target **bootrom_uncmp**) or a bootable VxWorks (the target **vxWorks_boot** or

vxWorks_boot.st).[3] We recommend you copy the resulting file to a legal MS-DOS file name, such as **bootrom.dat**, to simplify the rest of the process.

The commands for this sequence are as follows:

```
% cd wpwr/target/config/pc386
% make bootrom_uncmp
% cp bootrom_uncmp bootrom.dat
```

2. Transfer the executable image to a PC running MS-DOS. In many cases, the PC is networked with the workstation, using PC-NFS or a similar networking package. For example:

```
C:\> copy drive:bootrom.dat c:
```

where *drive* refers to the mounted file system on your PC.

3. Use the **mkboot** utility (or a combination of **vxsys** and **vxcopy**) to create the boot disk. If this boot disk is a diskette, it must be a high-density diskette. The following example shows this step, assuming the diskette is in drive **A**:

```
C:\> mkboot a: bootrom.dat
```

The **mkboot** utility uses **vxsys** to create the VxWorks loader program in the disk's boot sector. **mkboot** then runs **vxcopy** to copy **bootrom.dat** to the boot file **bootrom.sys** on the target disk, excluding the **a.out** header.

4. Check that **bootrom.sys** is contiguous on the boot disk, using the MS-DOS **chkdsk** utility. (The **mkboot** utility runs **chkdsk** automatically.) If **chkdsk** shows that there are non-contiguous blocks, delete all files from the disk and repeat the **vxcopy** operation to ensure that MS-DOS lays down the file contiguously.

The following example shows **chkdsk** output where the boot file is not contiguous (note especially the last line of output):

```
C:\> chkdsk a:bootrom.sys

Volume Serial Number is 2A35-18ED
1457664 bytes total disk space
 895488 bytes in 11 user files
 562176 bytes available on disk

 512 bytes in each allocation unit
 2847 total allocation units on disk
 1098 available allocation units on disk
```

3. Before making either version of the image, make sure that **DEFAULT_BOOT_LINE** in **config.h** is set correctly, and that the size of the boot image (text+data+bss) is less than 512KB. It cannot be larger than this, because it is written into lower memory.

```
    655360 total bytes memory
    602400 bytes free

A:\BOOTROM.SYS Contains 2 non-contiguous blocks
```

5. To test your boot disk, first make sure that the correct drive holds the boot disk (in this example case, drive **A:** holds the boot diskette).

6. Reboot the PC.

   Depending on the configuration of your VxWorks image, if the boot is successful, the VxWorks boot prompt appears either on the VGA console or on the COM1 serial port. You can boot VxWorks by entering @:

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

- **The MS-DOS Boot Utilities in More Detail**

vxsys *drive*:
This command installs a VxWorks loader program in a drive's boot sector. The drive can be either a diskette (drive **A:**), or a hard disk that is searched by the BIOS bootstrap (drive **C:**).[4] The VxWorks loader program searches for the file **bootrom.sys** in the root directory and loads it directly into memory at 0x8000. Execution then jumps to ***romInit( )*** at 0x8000.

After a loader program is installed in the disk's boot sector, you do not need to repeat the **vxsys** operation for new ROM images. Just use **vxcopy** to make a new version of **bootrom.sys**.

---

vxcopy *source_file target_file*
This command copies the VxWorks image file from *source_file* to *target_file*. Normally this copies the **bootrom_uncmp** output to **bootrom.sys** on the boot disk. **vxcopy** strips the 32-byte **a.out** header from *source_file* as it copies.

mkboot *drive*: *source_file*
This command is an MS-DOS batch file that uses **vxsys** to install the VxWorks loader program in the drive's boot sector, and then uses **vxcopy** to transfer *source_file* to *drive*:**bootrom.sys**. It also runs the MS-DOS utility **chkdsk** to check whether **bootrom.sys** is contiguous.

vxload [*image_file*]
This command is used during an MS-DOS session to load and execute the VxWorks image (normally **vxWorks.st** or **bootrom_uncmp**). It can be more convenient or quicker than loading the image via the PC boot cycle. **vxload**

4. For embedded applications, actual disk drives are often replaced by solid state disks. Because there are no moving parts, boot performance and reliability are increased.

takes an optional parameter, the image file name; the default is **vxWorks.st** in the current directory.

⚠ **NOTE:** **vxload** cannot be used to load VxWorks if the MS-DOS session has a protected mode program in use. Typical examples include the MS-DOS RAM disk driver, **vdisk.sys**, and the extended memory manager, **emm386.exe**. To use **vxload**, remove or disable such facilities.

Because **vxload** must read the image file to memory at 0x8000, it checks to see that this memory is not in use by MS-DOS, and generates an error if it is. If you receive such an error, reconfigure your PC to make the space available by loading MS-DOS into high memory and reducing the number of device drivers. Or start **vxload** instead of the MS-DOS command interpreter **command.com**. (If you take this approach, remember to first ensure that you can restore your previous configuration.)

The following is a sample **config.sys** file that shows these suggestions:

```
device=c:\dos\himem.sys
dos=high,umb
shell=c:\vxload.com c:\bootrom.dat
```

The file **bootrom.dat** must have an **a.out** header, unlike the **bootrom.sys** file made by **mkboot**.

Booting VxWorks from a Diskette, an ATA/IDE Disk, or a PC Card

Three boot devices are available in VxWorks for the x86, one for diskettes, one for ATA/IDE hard disks, and one for PCMCIA PC cards. You can also build your own VxWorks boot ROMs using optional hardware; see *ROM Card and EPROM Support*, p.566. Alternatively, as with other VxWorks platforms, you can also boot over an Ethernet (using one of the supported Ethernet cards), or over a SLIP connection.

⚠ **NOTE:** Because standard PC BIOS components do not support initial booting from PCMCIA devices, systems which load VxWorks from PCMCIA devices must use a VxWorks boot disk/diskette. See *Building a Boot Disk/Diskette from VxWorks*, p.557 and *Building a Boot Disk/Diskette from MS-DOS*, p.558.

When booting from a diskette, an ATA/IDE disk, or a PC card, first make sure that the boot device is formatted for an MS-DOS file system. The VxWorks boot program mounts the boot device by automatically calling either ***usrFdConfig( )*** in **src/config/usrFd.c** for diskettes, ***usrAtaConfig( )*** in **src/config/usrAta.c** for ATA/IDE hard disks, or ***usrPcmciaConfig( )*** in **src/config/usrPcmcia.c** for PC cards.

In each case, a *mount point* name is taken from the file name specified as one of the boot parameters. You might choose diskette zero (drive **A:**) to be mounted as **/fd0** (by supplying a boot file name that begins with that string). Similarly, you might choose ATA/IDE hard disk zero (drive **C:**) to be mounted as **/ata0** or you might choose the PC card in socket 0 to be mounted as **/pc0**. In each case, the name of the directory mount point (**fd0**, **ata0**, or **pc0** in these examples) can be any legal file name. (For more information on ***usrFdConfig( )***, ***usrAtaConfig( )***, or ***usrPcmciaConfig( )***, see *Mounting a DOS File System*, p.564.)

Because the PC hardware does not have a standard NVRAM interface, the only way to change default boot parameters is to rebuild the bootstrap code with a new definition for **DEFAULT_BOOT_LINE** in **config.h**. See *Boot Process*, p.556 for instructions on how to rebuild the bootstrap code.

NOTE: To enable rebooting with **CTRL+X**, you must set some of the BSP-specific global variables **sysWarmType**, **sysWarmFdType**, and **sysWarmFdDrive**, **sysWarmAtaCtrl**, and **sysWarmAtaDrive**, depending on which boot device you use. For more information, see *Architecture-Specific Global Variables*, p.544.

- **Booting from Diskette**

To boot from a diskette, specify the boot device as **fd** (for *floppy device*). First, specify the *drive number* on the `boot device:` line of the boot parameters display. Then, specify the *diskette type* (3.5" or 5.25"). The format is as follows:

```
boot device: fd=drive number, diskette type
```

drive number
a digit specifying the diskette drive:

0 = default; the first diskette drive (drive **A:**)

1 = the second diskette drive (drive **B:**)

diskette type
a digit specifying the type of diskette:

0 = default; 3.5" diskette

1 = 5.25" diskette

Thus, to boot from drive **B:** with a 5.25" diskette, enter the following:

```
boot device: fd=1,1
```

The default value of the file-name boot parameter is **/fd0/vxWorks.st**. You can specify another boot image; for example, assume that you have placed your **vxWorks** and **vxWorks.sym** files in the root directory of the 5.25" diskette in drive

A: as the files **A:\vxworks** and **A:\vxworks.sym**, and that the mount point for this drive is **/fd0**. To boot this image, enter the following in the boot parameters display:

```
boot device: fd=0,1
...
file name: /fd0/vxworks
```

- **Booting from ATA/IDE Disk**

To boot from an ATA/IDE disk, specify the boot device as **ata**. First, specify the *controller number* on the boot device line of the boot parameters display. Then, specify the *drive number*. The format is as follows:

```
boot device: ata=controller number, drive number
```

controller number
a digit specifying the controller number:

0 = a controller described in the first entry of the **ataResources** table (in the default configuration, the local IDE disk is the first controller)
1 = a controller described in the second entry of the **ataResources** table (in the default configuration, the ATA PCMCIA PC card is the second controller)

drive number
a digit specifying the hard drive:

0 = the first drive on the controller (drive **C:** or **E:**)
1 = the second drive on the controller (drive **D:** or **F:**)

If your **vxWorks** and **vxWorks.sym** files are in the root directory of your IDE hard disk drive **C:** as the files **C:\vxworks** and **C:\vxworks.sym**, where **C:** is the first IDE disk drive on the system and the mount point for the drive is **/ata0**, then enter the following in the boot parameters display:

```
boot device: ata=0,0
...
file name: /ata0/vxworks
```

- **Booting from PCMCIA PC Card**

To boot from a PCMCIA PC card, specify the boot device as **pcmcia**. Specify the *socket number* on the `boot device:` line of the boot parameters display. The format is as follows:

```
boot device: pcmcia=socket number
```

socket number
a digit specifying the socket number:

0 = the first PCMCIA socket
1 = the second PCMCIA socket

If your **vxWorks** and **vxWorks.sym** files are in the root directory of your ATA or SRAM PCMCIA PC card drive **E:** as the files **E:\vxworks** and **E:\vxworks.sym**, and the mount point for your PC card drive is **/pc0**, then enter the following:

```
boot device: pcmcia=0
...
file name: /pc0/vxworks
```

If you are using an Ethernet PC card, the boot device is the same and the file name is:

```
file name: /usr/wind/target/config/pc386/vxWorks
```

Mounting a DOS File System

You can mount a DOS file system from a diskette, an ATA/IDE disk, or a PC card (SRAM or ATA) to your VxWorks target.

Use the routine ***usrFdConfig( )*** to mount the file system from a diskette. It takes the following parameters:

drive number — the drive that contains the diskette: MS-DOS drive **A:** is 0; drive **B:** is 1.

diskette type — 0 (3.5" 2HD) or 1 (5.25" 2HD).

mount point — from where on the file system to mount, for example, **/fd0/**.

Use the routine ***usrAtaConfig( )*** to mount the file system from an ATA/IDE disk. It takes the following parameters:

controller number — the controller: a controller described in the first entry of the **ataResources[]** table is 0; a controller described in the second entry is 1. In the default configuration, the local IDE disk is 0; the PCMCIA ATA drive is 1.

drive number — the drive: the first drive of the controller is 0; the second drive of the controller is 1. In the default configuration, MS-DOS drive **C:** is 0; drive **D:** is 1.

mount point — from where on the file system to mount, for example, **/ata0/**.

Use *pccardMount*( ) to mount the file system from a PC card (SRAM or ATA). This routine differs from *usrPcmciaConfig*( ) in that *pccardMount*( ) uses the default device. A default device is created by the enabler routine when the PC card is initialized. The default device is removed automatically when the PC card is removed. *pccardMount*( ) takes the following parameters:

socket number the socket that contains the PC card; the first socket is 0.

mount point from where on the file system to mount, for example, /**pc0**/.

Use *pccardMkfs*( ) to initialize a PC card and mount the file system from a PC card (SRAM or ATA). It takes the following parameters:

socket number the socket that contains the PC card; the first socket is 0.

mount point from where on the file system to mount, for example, /**pc0**/.

The *pccardMount*( ) and *pccardMkfs*( ) routines are provided in source form in **src/drv/pcmcia/pccardLib.c**.

DMA Buffer Alignment and cacheLib

If you write your own device drivers that use direct memory access into buffers obtained from **cacheLib**, the buffer must be aligned on a 64KB boundary.

Support for Third-Party BSPs

To support third party pc386 and pc486 BSPs, the global variable **sysCodeSelector** and the routines *sysIntVecSetEnt*( ) and *sysIntVecSetExit*( ) are defined in **sysLib.c**.

VxWorks Images

The executable targets **bootrom_uncmp**, **vxWorks**, and **vxWorks.st** were tested and verified on the pc386, pc486, and epc4 BSPs. The executable target **bootrom_uncmp** uses lower memory (0x0 - 0xa0000), while **vxWorks** and **vxWorks.st** use upper memory (0x100000 - *pcMemSize*). A minimum of 1MB of memory in upper memory is required for **vxWorks** and **vxWorks.st**.

The VxWorks makefile targets listed below are supported in these BSPs. They should be placed on a bootable diskette by **mkboot** (a DOS utility) or by *mkbootFd*( ) or *mkbootAta*( ) (VxWorks utilities). The makefile target

vxWorks_low should be downloaded by the **bootrom_high** bootROM image; for information on all VxWorks makefile targets, see the *Tornado User's Guide: Cross-Development*:

vxWorks_rom	bootable VxWorks:	upper memory
vxWorks_rom_low	bootable VxWorks:	lower memory
vxWorks.st_rom	bootable VxWorks.st (compressed):	upper memory
bootrom	bootROM (compressed):	lower memory
bootrom_uncmp	bootROM:	lower memory
bootrom_high	bootROM (compressed):	upper memory

BSP-Specific Global Variables

The BSP-specific global variables shown in Table D-3 apply to pc386, pc486, and epc4.

Table D-3 **BSP-Specific Global Variables**

Location	Global Variable	Value	Description
sysLib.c	**sysWarmType** **sysWarmFdType** **sysWarmFdDrive** **sysWarmAtaCtrl** **sysWarmAtaDrive**	0 = ROMBIOS 1 (default) = Diskette 2 = ATA	**sysWarmType** controls how **CTRL+X** is processed. If 0, VxWorks asserts SYSRESET line, and **CTRL+X** produces cold start. If 1, VxWorks reads a boot image from the diskette specified by **sysWarmFdType** and **sysWarmFdDrive**, and jumps to the boot image entry point. If 2, VxWorks reads a boot image from the ATA/IDE disk specified by **sysWarmAtaCtrl** and **sysWarmAtaDrive** and jumps to the boot image entry point.
	sysFdBufAddr **sysFdBufSize**	0x2000 0x3000	Address and size of diskette DMA buffer.
	sysStrayIntCount		VxWorks increments this when it catches a stray interrupt on IRQ7.

ROM Card and EPROM Support

A boot EPROM (type 27020 or 27040) is supported with Blunk Microsystems' ROM Card 1.0. For information on booting from these devices, see the Blunk Microsystems documentation.

The following program is provided to support VxWorks with the ROM Card:

config/*bspname*/romcard.s
a loader for code programmed in to the EPROM.

In addition, the following configurations are defined in the makefile to generate Motorola S-record format from **bootrom_uncmp** or from **vxWorks_boot.st**:

romcard_bootrom_512.hex
boot ROM image for 27040 (512 KB)

romcard_bootrom_256.hex
boot ROM image for 27020 (256 KB)

romcard_vxWorks_st_512.hex
bootable VxWorks image for 27040 (512 KB)

Neither the ROM Card nor the EPROM is distributed with VxWorks. To contact Blunk Microsystems, call or FAX 415–960–7190.

Device Drivers

VxWorks for the x86 includes a console driver, network drivers for several kinds of hardware, a diskette driver, an ATA/IDE hard disk driver, and a line printer driver.

NOTE: There is no support for a SCSI device driver in VxWorks 5.3.1.

VGA and Keyboard Drivers

The keyboard and VGA drivers are character-oriented drivers; thus, they are treated as additional serial devices. Because the keyboard deals only with input and the VGA deals only with output, they are integrated into a single driver in the module **src/drv/serial/pcConsole.c**.

To include the console drivers in your configuration, define the **config.h** macro **INCLUDE_PC_CONSOLE**. When this macro is defined, the serial driver automatically initializes the console drivers.

The console drivers do not change any hardware initialization that the BIOS has done. The I/O addresses for the keyboard and the console, and the base address of the on-board VGA memory, are defined in **config/*bspname*/pc.h**.

The macro **PC_KBD_TYPE** in **config/*bspname*/config.h** specifies the type of keyboard. If the keyboard is a portable PC keyboard with 83 keys, define the macro as **PC_XT_83_KBD**.

In the default configuration, **/tyCo/0** is serial device 1 (COM1), **/tyCo/1** is serial device 2 (COM2), and **/tyCo/2** is the console.

You can define the following configuration macros for the console drivers in **pc.h**:

- **INCLUDE_ANSI_ESC_SEQUENCE** supports the ANSI terminal escape sequences. The VGA driver does special processing for recognized escape sequences.
- **COMMAND_8042**, **DATA_8042**, and **STATUS_8042** refer to the I/O base addresses of the various keyboard controller registers.
- **GRAPH_ADAPTER** can be set to either **VGA** or **MONOCHROME**.

Network Drivers

Several network drivers are available, corresponding to an assortment of boards from different manufacturers. For the list of macros to include specific network drivers in your configuration, see *8.3 Configuring VxWorks*, p.430.

For all network drivers, the I/O address, RAM address, RAM size, and interrupt request (IRQ) levels are defined in **config.h** (the I/O address must match the value recorded in the EEPROM). Use the configuration program supplied by the manufacturer to set the I/O address; in some cases you can set IRQ levels with the same configuration program.

You can set the board-specific macro listed in Table D-4 (defined in **config.h**) to specify whether you are using EEPROM, thin coaxial cable (BNC), twisted-pair cable (RJ45), thick coaxial cable (AUI), or some combination (for example, RJ45+AUI and/or RJ45+BNC). The exceptions are the Intel EtherExpress32, which uses EEPROM only, and the Novell/Eagle NE2000, which uses a hardware jumper.

For most network drivers, if **INCLUDE_SHOW_ROUTINES** is defined, a board-specific routine *boardShow*( ) [5] displays statistics collected in the interrupt handler on the standard output device. This routine requires two parameters: *interface unit* and *zap*. For all boards currently supported, *interface unit* is 0; *zap* can be either 0 or 1. If *zap* is 1, all collected statistics are cleared to zero.

Table D-4 shows the software configuration details for each network driver.

5. The prefix *board* is an abbreviation for the corresponding network board. For example, the abbreviation for the 3Com EtherLink III board is *elt*, so the corresponding show routine is *eltShow*( ).

Table D-4 **Network Drivers**

Network Board	IRQ Levels Supported	Ethernet Configuration Macro	Show Routine
SMC Elite 16	2, 3, 4, 5, 7, 9, 10, 11, 15	**CONFIG_ELC**	*elcShow*()*
SMC Elite 16 Ultra	2, 3, 5, 7, 10, 11	**CONFIG_ULTRA**	*ultraShow*()*
Intel EtherExpress†	2, 3, 4, 5, 9, 10, 11	**CONFIG_EEX**	(none)
Intel EtherExpress32†	3, 5, 7, 9, 10, 11, 12, 15	(EEPROM)	(none)
Intel EtherExpress PRO100B	0 - 15	(EEPROM)	(none)
3Com EtherLink III	3, 5, 7, 9, 10, 11, 12, 15	**CONFIG_ELT**	*eltShow*()*
Novell/Eagle NE2000	2, 3, 4, 5, 10, 11, 12, 15	(jumper)	*eneShow*()*
Ampro Ethernet-II	2, 3, 10, 11	**CONFIG_ESMC**	*esmcShow*()*

* These routines are *not* built in to the Tornado shell. To use them from the Tornado shell, you must define **INCLUDE_SHOW_ROUTINES** in your VxWorks configuration; see *8. Configuration*. When you invoke them, their output is sent to the standard output device.

† Auto-detect mode is not supported for these boards.

Certain network boards are also configurable in hardware. Use the jumper settings shown in Table D-5 with the network drivers supplied.

Table D-5 **Network Board Hardware Configuration**

Network Board	Jumpers	Settings
SMC Elite 16	W1 W2	SOFT NONE/SOFT
SMC Elite 16 Ultra	W1	SOFT
Intel EtherExpress	(none)	
Intel EtherExpress32	(none)	
Intel EtherExpress PRO100B	(none)	
3Com EtherLink III	(none)	
Novell/Eagle NE2000	various	follow manufacturer's instructions

Table D-5 **Network Board Hardware Configuration** *(Continued)*

Network Board	Jumpers	Settings
Ampro Ethernet-II	W1	PROM size: 16K or 32K
	W3	No.0: 0x300
	W4	Select IRQ-10,11

Diskette Driver

To include the diskette driver in your configuration, define the macro **INCLUDE_FD** in **config.h** ("fd" stands for *floppy disk*). When **INCLUDE_FD** is defined, the initialization routine ***fdDrv( )*** is called automatically from ***usrRoot( )*** in **config/all/usrConfig.c**. To change the interrupt vector and level used by ***fdDrv( )***, edit the definitions of **FD_INT_VEC** and **FD_INT_LVL** in **config/***bspname***/pc.h**.

The ***fdDevCreate( )*** routine installs a diskette device in VxWorks. You must call ***fdDevCreate( )*** explicitly for each diskette device you wish to install; it is not called automatically. The ***fdDevCreate( )*** routine requires the following parameters:

drive number — the diskette drive that corresponds to this device: MS-DOS drive **A:** is 0; drive **B:** is 1.

diskette type — 0 (3.5" 2HD) or 1 (5.25" 2HD). These numbers are indices to the structure table **fdTypes[]** in **config/***bspname***/sysLib.c**, which is described below.

number of blocks — the size of the device.

offset — the number of blocks to leave unused at the start of a diskette.

As shipped, the **fdTypes[]** table in **sysLib.c** describes two diskette types: the 3.5" 1.44MB 2HD diskette and the 5.25" 1.2MB 2HD diskette. (In particular, there is no entry for low-density diskettes.) To use another type of diskette, add the appropriate disk descriptions to the **fdTypes[]** table, shown below. Note that each entry in the table is a structure. The entry **dataRate** is described in more detail in Table D-6 and the entries **stepRate**, **headUnload**, and **headLoad** are described in Table D-7.

```
int sectors;        /* number of sectors                          */
int sectorsTrack;   /* sectors per track                          */
int heads;          /* number of heads                            */
int cylinders;      /* number of cylinders                        */
int secSize;        /* 128 << secSize gives bytes per sector      */
char gap1;          /* suggested gap value in read/write cmds     */
                    /* to avoid splice point between data field   */
                    /* and ID field of contiguous sections        */
char gap2;          /* suggested gap values for format-track cmd */
```

```
char dataRate;      /* data transfer rate                          */
char stepRate;      /* stepping rate                               */
char headUnload;    /* head unload time                            */
char headLoad;      /* head load time                              */
char mfm;           /* 1-->MFM (double density),
                       0--> FM (single density)                    */
char sk;            /* if 1, skip bad sectors on read-data cmd     */
char *name;         /* name                                      */
```

The **dataRate** field must have a value ranging from 0 to 3. The bit value controls the data transfer rate by setting the configuration control register in some IBM diskette controllers. The values correspond to transfer rates as shown in Table D-6.

Table D-6 **Diskette Data Transfer Rates**

dataRate	MFM (double density)	FM (single density)
3	1Mbps	invalid
0	500Kbps	250Kbps
1	300Kbps	150Kbps
2	250Kbps	125Kbps

The **stepRate**, **headUnload**, and **headLoad** parameters describe time intervals related to physical operation of the diskette drive. The time intervals are a simple function of the parameter value and of a multiplier corresponding to the data transfer rate, except that 0 has a special meaning for **headUnload** and **headLoad**, as shown in Table D-7.

Table D-7 **Time Interval Parameters in fdTypes[]**

Description	Field	Value	Time (ms) by transfer rate			
			1M	500K	300K	250K
	Transfer rate multiplier (T):		*1*	*2*	*3.33*	*4*
Interval between stepper pulses	**stepRate**	0	8	16	26.7	32
		0–15	(8 – 0.5 × **stepRate**) × *T*			
Interval from end of read or write to head unload	**headUnoad**	0	128	256	426	512
		1–15	8 × **headUnload** × *T*			
Interval from end of head load to start of read or write	**headLoad**	0	128	256	426	512
		1–127	**headLoad** × *T*			

Interleaving is not supported when the driver formats a diskette; the driver always uses a 1:1 interleave. Use the MS-DOS format program to get the recommended DOS interleave factor.

The driver uses memory area 0x2000 to 0x5000 for DMA, for the following reasons:

- The DMA chip has an addressing range of only 24 bits.
- A buffer must fit in one page; that is, a buffer cannot cross the 64KB boundary.

Another routine associated with the diskette driver is ***fdRawio( )***. This routine allows you to read and write directly to the device; thus, the overhead associated with moving data through a file system is eliminated. The ***fdRawio( )*** routine requires the following parameters:

drive number — the diskette drive that corresponds to this device: MS-DOS drive **A:** is 0; drive **B:** is 1.

diskette type — 0 (3.5" 2HD) or 1 (5.25" 2HD). These numbers are indices to the structure table **fdTypes[]** in **config/***bspname***/sysLib.c**.

FD_RAW ptr — pointer to the **FD_RAW[]** structure, where the data that is being read and written is stored; see below.

The following is the definition of the **FD_RAW[]** structure:

```
typedef struct fdRaw
    {
    UINT     cylinder;  /* cylinder (0 -> (cylinders-1))    */
    UINT     head;      /* head (0 -> (heads-1))            */
    UINT     sector;    /* sector (1 -> sectorsTrack)       */
    UINT     *pBuf;     /* ptr to buff (bytesSector*nSecs)  */
    UINT     nSecs;     /* # of sectors (1-> sectorsTrack)  */
    UINT     direction; /* read=0, write=1                  */
    } FD_RAW;
```

ATA/IDE Disk Driver

To include the ATA/IDE disk device driver in your configuration, define the macro **INCLUDE_ATA** in **config.h**. When **INCLUDE_ATA** is defined, the initialization routine ***ataDrv( )*** is called automatically from ***usrRoot( )*** in **usrConfig.c** for the local IDE disk. To change the interrupt vector and level and the configuration type used by ***ataDrv( )***, edit the definitions of the constants **ATA0_INT_VEC**, **ATA0_INT_LVL**, and **ATA0_CONFIG** in **pc.h**. The default configuration is suitable for the i8259 interrupt controller; most PCs use that chip. The ***ataDrv( )*** routine requires the following parameters:

controller number
: the controller: a controller described in the first entry of the **ataResources[]** table is 0; a controller described in the second entry is 1. In the default configuration, the local IDE disk is 0; the PCMCIA ATA drive is 1.

number of drives
: number of drives on the controller: maximum of two drives per controller is supported.

interrupt vector
: interrupt vector

interrupt level
: IRQ level

configuration type
: configuration type

semaphore timeout
: timeout value for the semaphore in the device driver.

watchdog timeout
: timeout value for the watchdog in the device driver.

The ***ataDevCreate( )*** routine installs an ATA/IDE disk device in VxWorks. You must call ***ataDevCreate( )*** explicitly for each local IDE disk device you wish to install; it is not called automatically. The ***ataDevCreate( )*** routine requires the following parameters:

controller number
: the controller: a controller described in the first entry of the **ataResources[]** table is 0; a controller described in the second entry is 1. In the default configuration, the local IDE disk is 0; the PCMCIA ATA drive is 1.

drive number
: the drive: the first drive of the controller is 0; the second drive of the controller is 1. In the default configuration, MS-DOS drive **C:** is 0 on controller 0.

number of blocks
: the size of the device.

offset
: the number of blocks to leave unused at the start of a disk.

If the configuration type specified with ***ataDrv( )*** is 0, the ATA/IDE driver does not initialize drive parameters. This is the right value for most PC hardware, where the ROMBIOS initialization takes care of initializing the ATA/IDE drive. If you have custom hardware and the ATA/IDE drive is not initialized, set the configuration type to 1 to cause the driver to initialize drive parameters.

The drive parameters are the number of sectors per track, the number of heads, and the number of cylinders. The table has two other members used by the driver: the number of bytes per sector, and the precompensation cylinder. For each drive, the information is stored in an **ATA_TYPE** structure, with the following elements:

```
int cylinders;        /* number of cylinders              */
int heads;            /* number of heads                  */
int sectorsTrack;     /* number of sectors per track      */
int bytesSector;      /* number of bytes per sector       */
int precomp;          /* precompensation cylinder         */
```

A structure for each drive is stored in the **ataTypes[]** table in **sysLib.c**. That table has two sets of entries: the first is for drives on controller 0 (the local IDE disk) and the second is for drives on controller 1 (the PCMCIA ATA card). The table is defined as follows:

```
ATA_TYPE ataTypes[ATA_MAX_CTRLS][ATA_MAX_DRIVES] =
    {
    {{761, 8, 39, 512, 0xff},        /* ctrl 0 drive 0 */
     {761, 8, 39, 512, 0xff}},       /* ctrl 0 drive 1 */
    {{761, 8, 39, 512, 0xff},        /* ctrl 1 drive 0 */
     {761, 8, 39, 512, 0xff}},       /* ctrl 1 drive 1 */
    };
```

The *ioctl*( ) function **FIODISKFORMAT** always returns **ERROR** for this driver, because ATA/IDE disks are always preformatted and bad sectors are already mapped.

If **INCLUDE_SHOW_ROUTINES** is defined, the routine *ataShow*( ) displays the table and other drive parameters on the standard output device. This routine requires two parameters: *controller number*, which must be either 0 (local IDE) or 1 (PCMCIA ATA), and *drive number*, which must be either 0 or 1.

Another routine associated with the ATA/IDE disk driver is *ataRawio*( ). This routine allows you to read and write directly to the device; thus, the overhead associated with moving data through a file system is eliminated. The *ataRawio*( ) routine requires the following parameters:

controller number
: the controller: a controller described in the first entry of the **ataResources[]** table is 0; a controller described in the second entry is 1. In the default configuration, the local IDE disk is 0; the PCMCIA ATA drive is 1.

drive number
: the drive: the first drive of the controller is 0; the second drive of the controller is 1. In the default configuration, MS-DOS drive **C:** is 0 on controller 0.

ATA_RAW ptr pointer to the **ATA_RAW** structure, where the data that is being read and written is stored; see below.

The following is the definition of the **ATA_RAW** structure:

```
typedef struct ataRaw
    {
    UINT     cylinder;  /* cylinder (0 -> (cylinders-1))   */
    UINT     head;      /* head (0 -> (heads-1))           */
    UINT     sector;    /* sector (1 -> sectorsTrack)      */
    UINT     *pBuf;     /* ptr to buff (bytesSector*nSecs) */
    UINT     nSecs;     /* #of sectors (1 -> sectorsTrack) */
    UINT     direction; /* read=0, write=1                 */
    } ATA_RAW;
```

The resource table used by *ataDrv( )*, **ataResources[]**, is defined in **sysLib.c** as follows:

```
ATA_RESOURCE ataResources[ATA_MAX_CTRLS] =
    {
    {
    {
    5, 0,
    {ATA0_IO_START0, ATA0_IO_START1}, {ATA0_IO_STOP0, ATA0_IO_STOP1},
     0, 0, 0, 0, 0, 0
     }
    IDE_LOCAL, 1, ATA0_INT_VEC, ATA0_INT_LVL, ATA0_CONFIG,
    ATA_SEM_TIMEOUT, ATA_WDG_TIMEOUT, 0, 0
    },   /* ctrl 0 */
    {
    {
    5, 0,
    {ATA1_IO_START0, ATA1_IO_START1}, {ATA1_IO_STOP0, ATA1_IO_STOP1},
     0, 0, 0, 0, 0, 0
     }
    ATA_PCMCIA, 1, ATA1_INT_VEC, ATA1_INT_LVL, ATA1_CONFIG,
    ATA_SEM_TIMEOUT, ATA_WDG_TIMEOUT, 0, 0
    },   /* ctrl 1 */
    };
```

Each resource in the table is an **ATA_RESOURCE** structure, defined as follows:

```
typedef struct ataResource      /* PCCARD ATA resources */
    {
    PCCARD_RESOURCE resource;  /* must be the first member          */
    int      ctrlType;         /* controller type: IDE_LOCAL        */
                               /* or ATA_PCMCIA                     */
    int      drives;           /* 1,2: number of drives             */
    int      intVector;        /* interrupt vector                  */
    int      intLevel;         /* IRQ level                         */
    int      configType;       /* 0,1: configuration type           */
    int      semTimeout;       /* timeout seconds for sync semaphore */
    int      wdgTimeout;       /* timeout seconds for watch dog     */
    int      sockTwin;         /* socket number for twin card       */
    int      pwrdown;          /* power down mode                   */
    } ATA_RESOURCE;
```

⚠ **NOTE:** This structure applies to both ATA PCMCIA PC cards and local IDE hard disks. For the definition of **PCCARD_RESOURCE**, see *PCMCIA for x86 Release Notes and Supplement*.

Line Printer Driver

This release of VxWorks for the x86 supports write operations to an LPT line printer driver.

To include the line printer driver in your configuration, define the macro **INCLUDE_LPT** in **config.h**. When **INCLUDE_LPT** is defined, the initialization routine ***lptDrv( )*** is called automatically from ***usrRoot( )*** in **usrConfig.c**.

The resource table used by ***lptDrv( )*** is stored in the structure **lptResource[]** in **sysLib.c**. The resources are defined as follows:

```
int   ioBase;          /* IO base address              */
int   intVector;       /* interrupt vector             */
int   intLevel;        /* interrupt level              */
BOOL  autofeed;        /* TRUE if enable autofeed       */
int   busyWait;        /* loop count for BUSY wait      */
int   strobeWait;      /* loop count for STROBE wait    */
int   retryCnt;        /* timeout second for syncSem    */
```

lptDrv( ) takes two arguments. The first argument is the number of channels (0, 1, or 2). The second argument is a pointer to the resource table.

To change ***lptDrv( )***'s interrupt vector or interrupt level, change the value of the appropriate constant (**LPT_INT_VEC** or **LPT_INT_LVL**) in **pc.h**.

Many of the LPT driver's routines are accessible only through the I/O system. However, the following routines are available (see the manual pages for details):

lptDevCreate( ) installs an LPT device into VxWorks. Call ***lptDevCreate( )*** explicitly for each LPT device you wish to install; it is not called automatically. This routine takes the following parameters:

name = device name

channel = physical device channel (0, 1, or 2)

lptAutofeed( ) enables or disables the autofeed feature; takes the parameter *channel* (0, 1, or 2).

lptShow( ) if **INCLUDE_SHOW_ROUTINES** is defined, shows driver statistics; takes the parameter *channel* (0, 1, or 2).

In addition, you can perform the following ***ioctl( )*** functions on the LPT driver:

LPT_GETSTATUS
: gets the value of the status register; takes an integer value where status is stored

LPT_SETCONTROL
: sets the control register; takes a value for the register

E

MIPS R3000, R4000, R4650

List of Figures

E.1 Introduction

This appendix provides information specific to VxWorks development on MIPS targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the MIPS processors.
- Architecture Considerations: special features and limitations of the MIPS processors, including a figure showing the VxWorks memory layout for these processors.

For general information on the Tornado development environment's cross-development tools, see the *Tornado User's Guide: Cross-Development*.

E.2 Building Applications

The following sections describe a configuration constant, environment variables, and compiler options that together specify the information the GNU toolkit requires to compile correctly for the MIPS targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications build with the appropriate architecture-specific features enabled. Define this variable to be **R3000** (for the MIPS R3000 or R3500), **R4000** (for the R4200 or R4600), or **R4650** (for the MIPS R4640 or R4650).

For example, to define **CPU** for an R3500 on the compiler command line, specify the following command-line option when you invoke the compiler:

```
-DCPU=R3000
```

To provide the same information in a header or source file, include the following line in the file:

```
#define CPU R3000
```

All VxWorks makefiles pass along the definition of this variable to the compiler. You can define **CPU** on the **make** command line as follows:

```
% make CPU=R3000 ...
```

You can also set the definition directly in a makefile, with the following line:

```
CPU=R3000
```

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables.

For developers using UNIX hosts, you must specifically define this variable. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

For developers using Windows hosts, if you are working through the Tornado IDE, the appropriate variable(s) are set automatically. However, before invoking the compiler from a DOS command line, first run the following batch file to set the variable(s):

```
%WIND_BASE%/host/x86-win32/bin/torVars.bat
```

For more information, see the *Tornado User's Guide: Getting Started.*

Compiling C or C++ Modules

The following is an example of a compiler command line for R3000 cross-development. The file to be compiled in this example has a base name of **applic**.

```
% ccmips -DCPU=R3000 -I/usr/vw/h -mcpu=r3000 -O2 -funroll-loops \
-nostdinc -G 0 -c applic.c
```

This is an example for the R4000:

```
% ccmips -DCPU=R4000 -I/usr/vw/h -mcpu=r4000 -mips3 -mgp32 \
-mfp32 -O2 -funroll-loops -nostdinc -G 0 -c applic.c
```

The options shown in the examples have the following meanings:[1]

-DCPU=R3000 Required; defines the CPU type for the R3000 or R3500. For the R4200 or R4600, specify **R4000**. For the R4640 or R4650, specify **R4650**.

-I $WIND_BASE/h
Required; gives access to the VxWorks include files. (Additional **-I** flags may be included to specify other header files.)

-mcpu=r3000 Required; tells the compiler to produce code for the R3000 or R3500. For the R4200 or R4600, specify **r4000**. For the R4640 or R4650, specify **r4650**.

-mips3 Required for R4000 targets (R4200 and R4600) and R4650 targets (R4640 and R4650); tells the compiler to issue instructions from level 3 of the MIPS ISA (64-bit instructions). This compiler option does not apply to R3000 or R3500 targets.

-mfp32 Required for R4000 and R4650 targets; tells compiler to issue instructions assuming that fp registers are 32 bits, required for compatibility with **mathALib**.

-mgp32 Required for R4000 and R4650 targets in code which makes calls to varargs functions provided by VxWorks (***printf*( ), *sprintf*( )**, and so forth); tells the compiler to issue instructions assuming that all general-purpose registers are 32 bits.

-msingle-float Required for R4640 and R4650; tells the compiler to assume that the floating-point processor supports only single-precision operations.

-m4650 Required for R4650 targets; sets **-msingle-float** and **-mmad**[2] flags.

-O2 Optional; tells the compiler to use level 2 optimization.

⚠ **NOTE:** To specify optimization for use with GDB, use the **-O0** flag.

1. For more information on these and other compiler options, see the *GNU ToolKit User's Guide*. WRS supports compiler options used in building WRS software; a list of these options is included in the *Guide*. Other options are not supported, although they are available with the tools as shipped.
2. Consult *GNU Toolkit User's Guide*.

-funroll-loops Optional; tells the compiler to use loop unrolling optimization.

-nostdinc Required; searches only the directories specified with the **-I** flag (see above) and the current directory for header files.

-msoft-float Required for software emulation, tells the compiler to issue library callouts for floating point. For more information, see *Floating-Point Support*, p.587.

-G 0 Required; tells the compiler not to use the global pointer. For more information, see *Gprel Addressing*, p.587.

-c Required; specifies that the module is to be compiled only, not linked for execution under the host.

The output is an unlinked object module in ELF format with the suffix **.o**; for the example above, the output would be **applic.o**.

The default for **ccmips** is big-endian (set explicitly with **-EB**) and defines **MIPSEB**. Use **-EL** to compile little-endian and automatically define **MIPSEL**. Users should not define either **MIPSEB** or **MIPSEL**.

E.3 Interface Variations

This section describes particular routines and tools that are specific to MIPS targets in any of the following ways:

- available only on MIPS targets
- parameters specific to MIPS targets
- special restrictions or characteristics on MIPS targets

For complete documentation, see the reference entries for the libraries, subroutines, and tools discussed below.

cacheR3kLib *and* cacheR4kLib

The libraries **cacheR3kLib** and **cacheR4kLib** are specific to the MIPS release. They each contain a routine that initializes the R3000 or R4000 cache library.

dbgLib

In the MIPS release, the routine *tt( )* displays the first four parameters of each subroutine call, as passed in registers **a0** through **a3**. For routines with less than four parameters, ignore the contents of the remaining registers.

For a complete stack trace, use GDB.

intArchLib

In the MIPS release, the routines *intLevelSet( )* and *intVecBaseSet( )* have no effect. For a discussion of the MIPS interrupt architecture, see *Interrupts*, p.588.

mathALib

VxWorks for MIPS supports the same set of **mathALib** functions using either hardware facilities or software emulation.[3]

The following double-precision routines are supported for MIPS architectures:

acos( )	*asin( )*	*atan( )*	*atan2( )*	*ceil( )*	*cos( )*	*cosh( )*
exp( )	*fabs( )*	*floor( )*	*fmod( )*	*log10( )*	*log( )*	*pow( )*
sin( )	*sincos( )*	*sinh( )*	*sqrt( )*	*tan( )*	*tanh( )*	*trunc( )*

The following single-precision routines are supported for MIPS architectures:

acosf( )	*asinf( )*	*atanf( )*	*atan2f( )*	*ceilf( )*	*cosf( )*	*coshf( )*
expf( )	*floorf( )*	*logf( )*	*log2f( )*	*log10f( )*	*sinf( )*	*sinhf( )*
sqrtf( )	*tanf( )*	*tanhf( )*	*truncf( )*			

In addition, the single precision routines *fmodf( )* and *powf( )* are supported for R4650 processors only.

The following math routines are not supported by VxWorks for MIPS:

cbrt( )	*cbrtf( )*	*infinity( )*	*infinityf( )*	*irint( )*	*irintf( )*	*iround( )*
iroundf( )	*log2( )*	*round( )*	*roundf( )*	*sincosf( )*		

3. To use software emulation, compile your application with the **-msoft-float** compiler option as well as defining **INCLUDE_SW_FP**; see *Floating-Point Support*, p.587. Use of these functions on the R4000 requires that your code be compiled with **-mfp32**.

taskArchLib

The routine ***taskSRInit( )*** is specific to the MIPS release. This routine allows you to change the default status register with which a task is spawned. For more information, see *Interrupt Support Routines*, p.589.

MMU Support

MIPS targets do not support memory management units (MMUs). Thus, you do not need to define the constants **INCLUDE_MMU_BASIC** or **INCLUDE_MMU_FULL** in **config.h**, and you do not need to define **sysPhysMemDesc[]** in **sysLib.c**. For more information, see *Virtual Memory Mapping*, p.590.

ELF-specific Tools

The following tools are specific to the ELF format. For more information, see the reference entries for each tool.

elfHex converts an ELF-format object file into Motorola hex records. The syntax is:

```
elfHex [-a adrs] [-l] [-v] [-p PC] [-s SP] file
```

elfToBin extracts text and data segments from an ELF file. The syntax is:

```
elfToBin < inFile > outfile
```

elfToBsd converts ELF object modules to BSD format. The syntax is:

```
elfToBsd < infile_elf > outfile_bsd
```

elfXsyms extracts the symbol table from an ELF file. The syntax is:

```
elfXsyms < objMod > symTbl
```

E.4 Architecture Considerations

This section describes the following characteristics of the MIPS architecture that you should keep in mind as you write a VxWorks application:

- Gprel addressing
- Reserved registers
- Floating-point support
- Interrupts
- Virtual memory mapping
- 64-bit support
- Memory layout

Gprel Addressing

The VxWorks kernel uses *gprel* (**gp**-relative) addressing. However, the VxWorks module loader cannot dynamically load tasks that use gprel addressing.

To keep the loader from returning an error, compile application tasks with the **-G 0** option. This option tells the compiler not to use the global pointer.

Reserved Registers

Registers **k0** and **k1** are reserved for VxWorks kernel use, following standard MIPS usage. The **gp** register is also reserved for the VxWorks kernel, because only the kernel uses gprel addressing, as discussed in above. Avoid using these registers in your applications.

Floating-Point Support

R4650

For the R4650, single precision hardware floating-point support is included by **INCLUDE_HW_FP** (the default). Double precision floating-point support is provided by software emulation when you use **-msoft-float**. (Note that **INCLUDE_SW_FP** is not required with **-msoft-float** for the R4650.)

R3000 and R4000

If your MIPS board includes a floating-point coprocessor (CP1), we recommend you use it for best performance.

However, if this chip is not available, you can use the GNU compiler **-msoft-float** option. This option keeps all floating-point values in integer registers (a pair of them for double-precision) and emulates all floating-point arithmetic.

To use this software emulation support, define **INCLUDE_SW_FP** and undefine **INCLUDE_HW_FP**. Then, in the BSP directory, build VxWorks with the following command:

```
% make [CPU=cpuType] TOOL=sfgnu
```

Building your applications with the **-msoft-float** flag produces library callouts for math routines. If you get unresolved references when downloading your applications, link your applications with the following library:

For R3000 targets:

```
$GCC_EXEC_PREFIX/mips-wrs-vxworks/cygnus-2.7.2-960126/msoft-float/libgcc.a
```

For R4000 targets:

```
$GCC_EXEC_PREFIX/mips-wrs-vxworks/cygnus-2.7.2-960126/mips3/msoft-float/libgcc.a
```

Interrupts

MIPS Interrupts

The MIPS architecture has inputs for six external hardware interrupts and two software interrupts. In cases where the number of hardware interrupts is insufficient, board manufacturers can multiplex several interrupts on one or more interrupt lines.

The MIPS CPU treats exceptions and interrupts in the same way: it branches to a common vector and provides status and cause registers that let system software determine the CPU state. The MIPS CPU does not switch to an interrupt stack or exception stack, nor does it generate an IACK cycle. These functions must be implemented in software or board-level hardware (for example, the VMEbus IACK cycle is a board-level hardware function). VxWorks for MIPS has implemented a single interrupt stack, and uses task stacks for exception conditions.

Because the MIPS CPU does not provide an IACK cycle, your interrupt handler must acknowledge (or clear) the interrupt condition. If the interrupt handler does

not acknowledge the interrupt, VxWorks hangs while trying to process the interrupt condition.

VxWorks for MIPS uses a 256-entry table of vectors. You can attach exception or interrupt handlers to any given vector with the routines ***intConnect( )*** and ***intVecSet( )***. The files **h/arch/mips/ivMips.h** and *bspname*.**h** list the vectors used by VxWorks.

Interrupt Support Routines

Because the MIPS architecture does not use interrupt levels, the ***intLevelSet( )*** routine is not implemented. The six external interrupts and two software interrupts can be masked or enabled by manipulating eight bits in the status register with ***intDisable( )*** and ***intEnable( )***. Be careful to pass correct arguments to these routines, because the MIPS status register controls much more than just interrupt generation.

For interrupt control, the routines ***intLock( )*** and ***intUnlock( )*** are recommended. All interrupts are blocked when calling ***intLock( )***. The routine ***intVecBaseSet( )*** has no meaning on the MIPS; calling it has no effect.

To change the default status register with which all tasks are spawned, use the routine ***taskSRInit( )***. If used, call this routine before ***kernelInit( )*** in ***sysHwInit( )***. ***taskSRInit( )*** is provided in case your BSP must mask interrupts from all tasks. For example, the FPA interrupt must be disabled for all tasks.

VMEbus Interrupt Handling

The processing of VMEbus interrupts is the only case where it is not necessary for an interrupt handler to acknowledge the interrupt condition. If you define the option **VME_VECTORED** as TRUE in **configAll.h** (and rebuild VxWorks), *all* VMEbus interrupts are acknowledged by the low-level exception/interrupt handling code. The VxWorks interrupt vector number corresponds to the VMEbus interrupt vector returned by the VMEbus IACK cycle. With this interrupt handling scheme, VxWorks for MIPS allows multiple VMEbus boards to share the same VMEbus interrupt level without requiring further decoding by a user-attached interrupt handler.

You can still bind to VMEbus interrupts without vectored interrupts enabled, as long as the VMEbus interrupt condition is acknowledged with ***sysBusIntAck( )*** (as defined in **sysLib.c**). In this case, there is no longer a direct correlation with the vector number returned during the VMEbus IACK cycle. The vector number used to attach the interrupt handler corresponds to one of the seven VMEbus interrupt levels as defined in *bspname*.**h.** The mapping of the seven VMEbus interrupts to a single MIPS interrupt is board-dependent.

Vectored interrupts do not change the handling of any interrupt condition except VMEbus interrupts. All the necessary interrupt-acknowledge routines are provided in either **sysLib.c** or **sysALib.s**.

NOTE: Not all boards support VME-vectored interrupts. For more information, see the BSP reference entries.

Virtual Memory Mapping

VxWorks for MIPS operates exclusively in kernel mode and makes use of the **kseg0** and **kseg1** address spaces. A physical addressing range of 512 MB is available. Use of the on-chip *translation lookaside buffer* (TLB) is not supported.

- **kseg0 .** When the most significant three bits of the virtual address are 100, the 2^{29}-byte (512 MB) kernel physical space labeled **kseg0** is the virtual address space selected. References to **kseg0** are not mapped through the TLB; the physical address selected is defined by subtracting 0x8000 0000 from the virtual address. Caches are always enabled for accesses to these addresses.
- **kseg1.** When the most significant three bits of the virtual address are 101, the 2^{29}-byte (512 MB) kernel physical space labeled **kseg1** is the virtual address space selected. References to **kseg1** are not mapped through the TLB; the physical address selected is defined by subtracting 0xa000 0000 from the virtual address. Caches are always disabled for accesses to these addresses; physical memory or memory-mapped I/O device registers are accessed directly.

64-bit Support (R4000 Targets Only)

With VxWorks for MIPS, real-time applications have access to the MIPS R4000 64-bit registers. This lets applications perform 64-bit math for enhanced performance.

To specify 64-bit integers in C, declare them as **long long**. Pointers, integers, and longs are 32-bit quantities in this release of VxWorks.

Memory Layout

The memory layout of the MIPS is shown in Figure E-1. The figure contains the following labels:

Exception Vectors Table of exception/interrupt vectors.

SM Anchor Anchor for the shared memory network (if there is shared memory on the board).

Boot Line ASCII string of boot parameters.

Exception Message ASCII string of the fatal exception message.

Initial Stack Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image Entry point for VxWorks.

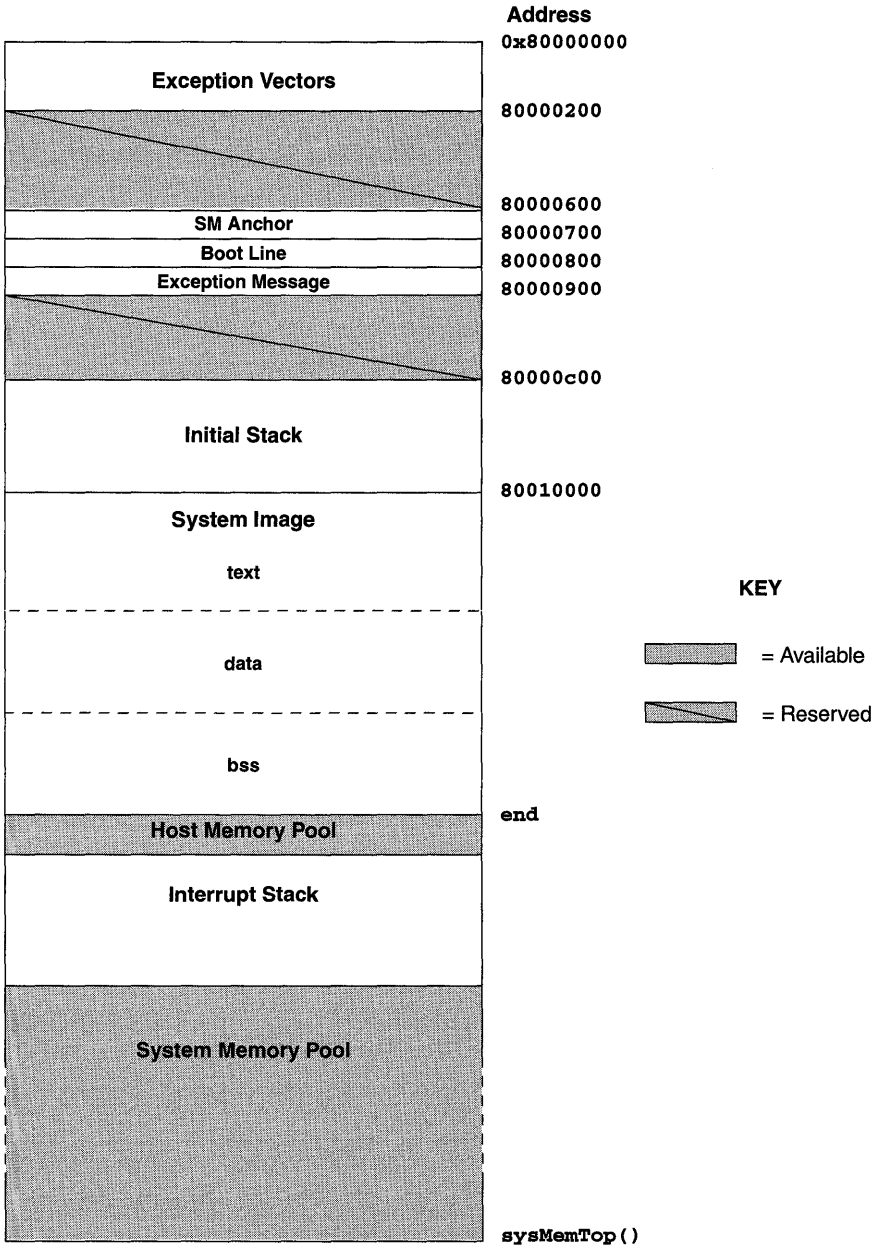
Host Memory Pool Memory allocated by host tools. The size depends on the system image and is defined in **config/all/configAll.h**.

Interrupt Stack Size defined in **configAll.h**. Location depends on system image size.

System Memory Pool Size depends on size of system image and interrupt stack. The end of the free memory pool for this board is returned by ***sysMemTop( )***.

All addresses shown in Figure E-1 depend on the start of memory for a particular target board. The start of memory is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

Figure E-1 **VxWorks System Memory Layout (MIPS)**



F

PowerPC

List of Tables

List of Figures

F.1 Introduction

This appendix provides information specific to VxWorks development on PowerPC targets. It includes the following topics:

- Building Applications: how to compile modules for your target architecture.
- Interface Changes: information on changes or additions to particular VxWorks features to support the PowerPC processors.
- Architecture Considerations: special features and limitations of the PowerPC processors, including a figure showing the VxWorks memory layout for these processors.

For general information on the Tornado development environment's cross-development tools, see the *Tornado User's Guide: Cross-Development*.

F.2 Building Applications

NOTE: The compiler for PowerPC conforms to the Embedded Application Binary Interface (EABI) protocol. Therefore type checking is more rigorous than for other architectures.

The following sections describe a configuration constant, an environment variable, and compiler options that together specify the information the GNU Toolkit requires to compile correctly for PowerPC targets.

Defining the CPU Type

Setting the preprocessor variable **CPU** ensures that VxWorks and your applications are compiled with the appropriate architecture-specific features enabled. This variable should be set to one of the following values, depending on the processor you are using:

- **PPC403**
- **PPC603**
- **PPC604**
- **PPC860**

For example, to specify **CPU** for a PowerPC 603 on the compiler command line, use the following command-line option when you invoke the compiler:

```
-DCPU=PPC603
```

To provide the same information in a header or source file, include the following line in the file:

```
#define CPU PPC603
```

All VxWorks makefiles pass along the definition of this variable to the compiler. You can define **CPU** on the **make** command line as follows:

```
% make CPU=PPC603 ...
```

You can also set the definition directly in a makefile, with the following line:

```
CPU=PPC603
```

NOTE: If you are using a PowerPC 821 processor, define **CPU** to be **PPC860**.

Configuring the GNU ToolKit Environment

Tornado includes the GNU compiler and associated tools. Running the GNU compiler requires that you define the environment variable **GCC_EXEC_PREFIX**. No change is required to the execution path, because the compilation chain is installed in the same **bin** directory as the other Tornado executables.

For developers using UNIX hosts, you must specifically define this variable. For example, if you use the C-shell, add the following to your **.cshrc**:

```
setenv GCC_EXEC_PREFIX $WIND_BASE/host/$WIND_HOST_TYPE/lib/gcc-lib/
```

NOTE: A *trailing slash* is important in the value of **GCC_EXEC_PREFIX** (as shown in the previous examples). If you do not include the slash, compilation fails.

For developers using Windows hosts, if you are working through the Tornado IDE, the appropriate variable(s) are set automatically. However, before invoking the compiler from a DOS command line, first run the following batch file to set the variable(s):

```
%WIND_BASE%/host/x86-win32/bin/torVars.bat
```

For more information, see the *Tornado User's Guide: Getting Started.*

Compiling C and C++ Modules

The following is an example of a compiler command line for PowerPC 603 cross-development. The file to be compiled in this example has a base name of **applic**.

```
% ccppc -O2 -mcpu=603 -I$WIND_BASE/target/h -fno-builtin \
-fno-for-scope -nostdinc -DCPU=PPC603 -D_GNU_TOOL -c applic.language_id
```

The options shown in the example have the following meanings:

-O2 Optional; performs level 2 optimization.

-mcpu=603 Optional for 603 and 604; required for other processors (specify the appropriate processor values: **601**, **403**, **860**, or **821**); instructs the compiler to produce code for the specified PowerPC architecture. The default is 604, which applies to 603 as well.

-I$WIND_BASE/target/h
Required; gives access to the VxWorks include files. (Additional **-I** flags may be included to specify other header files.)

-fno-builtin Required; uses library calls even for common library subroutines.

-fno-for-scope Required; allows the scope of variables declared within a for loop to be outside of the for loop.

-nostdinc Required; searches only the directory or directories specified with the **-I** flag (see above) and the current directory for header files. It does not search host-system include files.

-DCPU=PPC603 Required; defines the CPU type. If you are using another PowerPC processor, specify the appropriate value (see *Defining the CPU Type*, p.596).

-D_GNU_TOOL Required; defines the compilation toolkit used to compile VxWorks or applications.

-c Required; specifies that the module is to be compiled only, and not linked for execution under the host.

applic.*language_id*

Required; specifies the file(s) to compile. For C compilation, specify a suffix of **.c**. For C++ compilations, specifies a suffix of **.cpp**. The output is an unlinked object module in ELF format with the suffix **.o**. For the example above, the output would be **applic.o**.

Compiling Modules for GDB

To compile C modules for debugging in GDB, we recommend using the **-gdwarf** flag to generate DWARF debug information instead of the **-g** flag, which generates STABS information. For example:

```
% ccppc -mcpu=603 -I$WIND_BASE/target/h -fno-builtin -nostdinc \
-DCPU=PPC603 -c -gdwarf test.c
```

where **$WIND_BASE** is the location of your Tornado tree and **-DCPU** specifies the CPU type.

The compiler does not support DWARF debug information for C++. If you are using C++, you must use the **-g** flag:

```
% ccppc -mcpu=603 -I$WIND_BASE/target/h -fno-builtin -nostdinc \
-DCPU=PPC603 -c -g test.cpp
```

Unsupported Features

Prefixed Underscore

In the PowerPC architecture, the compiler does not prefix underscores to symbols. In other words, **symbol** is not equivalent to **_symbol** as it is in other architecture implementations.

Small Data Area

The compiler supports the small data area. However, for this release of Tornado for PowerPC, VxWorks does not support the small data area. Therefore the **-msdata** compiler flag must not be used.

F.3 Interface Changes

This section describes particular routines and tools that are specific to PowerPC targets in any of the following ways:

- available only for PowerPC targets
- parameters specific to PowerPC targets
- special restrictions or characteristics on PowerPC targets

For complete documentation, see the online documentation.

Memory Management Unit

VxWorks provides two levels of virtual memory support: the basic level bundled with VxWorks, and the full level that requires the optional product VxVMI.Check with your sales representative for the availability of VxVMI for PowerPC.

For detailed information on VxWorks MMU support, see *7. Virtual Memory Interface*. The following subsections augment the information in that chapter.

Instruction and Data MMU

The PowerPC MMU introduces a distinction between instruction and data MMU and allows them to be separately enabled or disabled. Two new macros, **USER_I_MMU_ENABLE** and **USER_D_MMU_ENABLE**, are defined in **config/all/configAll.h**. To enable/disable one or both MMUs, define/undefine the corresponding macros in either **configAll.h** or in your BSP's **config.h** file.

60X Memory Mapping

The PowerPC 603 and 604 MMU supports two models for memory mapping. The first, the BAT model, allows mapping of a memory block ranging in size from 128KB to 256MB into a BAT register. The second, the segment model, gives the ability to map the memory in pages of 4KB. Tornado for PowerPC supports both memory models.

- **603/604 Block Address Translation Model**

The size of a BAT register is two words of 32 bits. For the PowerPC 603 and PowerPC 604, eight BAT registers are implemented: four for the instruction MMU and four for the data MMU.

The data structure **sysBatDesc[]**, defined in **sysLib.c**, handles the BAT register configuration. The registers will be set by the initialization software in the MMU library. By default these registers are cleared and set to zero.

All the configuration constants used to fill the **sysBatDesc[]** are defined in **h/arch/ppc/mmu603Lib.h** for both the PowerPC 603 and the PowerPC 604.

- **603/604 Segment Model**

This model specifies the configuration for each memory page. The entire physical memory is described by the data structure **sysPhysMemDesc[]**, defined in **sysLib.c**. This data structure is made up of configuration constants for each page or group of pages. All the configuration constants defined in Table 7-1 of *7. Virtual Memory Interface* are available for PowerPC virtual memory pages.

Use of the **VM_STATE_CACHEABLE** constant listed in Table 7-1 for each page or group of pages, sets the cache to copy-back mode.

In addition to **VM_STATE_CACHEABLE**, the following additional constants are supported:

- **VM_STATE_CACHEABLE_WRITETHROUGH**
- **VM_STATE_MEM_COHERENCY**
- **VM_STATE_MEM_COHERENCY_NOT**
- **VM_STATE_GUARDED**
- **VM_STATE_GUARDED_NOT**

The first constant sets the page descriptor cache mode field in cacheable write-through mode. Cache coherency and guarded modes are controlled by the other constants.

For more information regarding cache modes, refer to *PowerPC Microprocessor Family: The Programming Environments.*

For more information on memory page states, state flags, and state masks, see*7. Virtual Memory Interface.*

The page table size depends on the total memory to be mapped. The larger the memory to be mapped, the bigger the page table will be. The VxWorks implementation of the segment model follows the recommendations given in *PowerPC Microprocessor Family: The Programming Environments.* During MMU library initialization, the total size of the memory to be mapped is computed, allowing dynamic determination of the page table size. The following table shows the correspondence between the total amount of memory to map and the page table size.

Table 10-2 **Page table size**

Total Memory to map	Page table size
8 MB or less	64 KB
16 MB	128 KB
32 MB	256 KB
64 MB	512 KB
128 MB	1 MB
256 MB	2 MB
512 MB	4 MB
1 GB	8 MB
2 GB	16 MB
4 GB	32 MB

ELF-specific Tools

The following tools are specific to the ELF format. For more information, see the reference entries for each tool.

elfHex converts an ELF-format object file into Motorola hex records. The syntax is:

`elfHex` [**`-a`** *adrs*] [**`-l`**] [**`-v`**] [**`-p`** *PC*] [**`-s`** *SP*] *file*

elfToBin extracts text and data segments from an ELF file. The syntax is:

`elfToBin` **`<`** *inFile* **`>`** *outfile*

elfToBsd converts ELF object modules to BSD format. The syntax is:

`elfToBsd` **`<`** *infile_elf* **`>`** *outfile_bsd*

elfXsyms extracts the symbol table from an ELF file. The syntax is:

`elfXsyms` **`<`** *objMod* **`>`** *symTbl*

F.4 Architecture Considerations

This section describes the following characteristics of the PowerPC processors that will affect your VxWorks application:

- supervisor/user mode
- 24-bit addressing
- byte order
- PowerPC register usage
- caches
- memory management unit (MMU)
- floating-point support
- memory layout

For a more comprehensive documentation of PowerPC architectures, see the appropriate Motorola microprocessor user's manual or the IBM user's manual.

Processor Mode

VxWorks always runs in Supervisor mode on processors in the PowerPC family.

24-bit Addressing

The PowerPC architecture limits its relative addressing to 24-bit offsets to conform to the EABI (Embedded Application Binary Interface) standard.

Byte Order

The byte order used by VxWorks for the PowerPC family is big-endian.

PowerPC Register Usage

The PowerPC conventions regarding register usage, stack frame formats, parameter passing between routines, and other factors involving code interoperability, are defined by the ABI (Application Binary Interface) and the EABI (Embedded Application Binary Interface) protocols. The VxWorks implementation for the PowerPC follows these protocols. Table F-1 shows PowerPC register usage in VxWorks.

Table F-1 **PowerPC Registers**

Register Name	Usage
gpr0	Volatile register which may be modified during function linkage.
gpr1	Stack frame pointer, always valid.
gpr2	Second small data area pointer register (_SDA2_BASE_).
gpr3 -gpr4	Volatile registers used for parameter passing and return value.
gpr5-gpr10	Volatile registers used for parameter passing.
gpr11-gpr12	Volatile registers that may be modified during function linkage.
gpr13	Small data area pointer register (_SDA_BASE_).
gpr14-gpr30	Non-volatile registers used for local variables.
gpr31	Used for local variables or "environment pointers."
fpr0	Volatile floating-point register.
fpr1	Volatile floating-point register used for parameter passing and return value.
fpr2-fpr8	Volatile floating-point registers used for parameter passing.
fpr9-fpr13	Volatile floating-point registers.
fpr14-fpr31	Non-volatile floating-point registers used for local variables.

Caches

The following subsections augment the information in *3. I/O System.*

PowerPC processors contain an instruction cache and a data cache. In the default configuration, VxWorks enables both caches. To disable the instruction cache, undefine **USER_I_CACHE_ENABLE** in **config/all/configAll.h.** To disable the data cache, undefine **USER_D_CACHE_ENABLE** in **configAll.h**.

For most boards, the cache capabilities must be used with the MMU to resolve cache coherency problems. The page descriptor for each page selects the cache mode. This page descriptor is configured by filling the data structure **sysPhysMemDesc[]** defined in **sysLib.c**. (For more information about cache coherency, see the reference entry for **cacheLib**. For information about the MMU

and VxWorks virtual memory, see 7. *Virtual Memory Interface*. For MMU information specific to the PowerPC family, see *Memory Management Unit*, p.604.)

The state of both data and instruction caches is controlled by the WIMG[1] information saved either in the BAT (Block Address Translation) registers or in the segment descriptors. Since a default cache state cannot be supplied, each cache may be enabled separately after the corresponding MMU is turned on. For more information on these cache control bits, refer to *PowerPC Microprocessor Family: The Programming Environments*, published jointly by Motorola and IBM.

⚠ **NOTE:** The cache library for the PowerPC 860 is provided with Tornado 1.0.1. Caching may be used with any CPU of revision A.1 or later. However, due to problems with the chip hardware, VxWorks must be run with the cache disabled on any processor earlier than revision A.1.

Memory Management Unit

The PowerPC MMU architecture required some extensions to the standard VxWorks MMU interface. See *Memory Management Unit*, p.599.

Floating-Point Support

PowerPC 403 and 860

The PowerPC 403 and 860 do not support hardware floating-point instructions. However, VxWorks provides a floating-point library that emulates these mathematical functions. All ANSI floating-point functions have been optimized using libraries from U. S. Software.

PowerPC 60X

A subset of the ANSI functions is optimized using libraries from Motorola:

acos( )	*asin*( )	*atan*( )	*atan2*( )
cos( )	*exp*( )	*log*( )	*log10*( )
pow( )	*sin*( )	*sqrt*( )	

1. W: the **WRITETHROUGH** or **COPYBACK** attribute.
   I: the inhibited attribute.
   M: the memory coherency attribute
   G: the guarded attribute

Handling of floating-point exceptions is supported for PowerPC 60X processors. Tasks spawned with the **VX_FP_TASK** option have the following default floating-point exception environment:

- Divide by zero, Overflow, and Underflow exceptions are enabled.
- The rounding mode is "Round to nearest."
- The floating-point exception mode selected is "imprecise and nonrecoverable."

To change the default for a specific task, modify the values of the Machine State Register (MSR) and the Floating Point Status and Control Register (FPSCR) at the beginning of the task code.

- The MSR's FE0 and FE1 bits select the floating-point exception mode.
- The FPSCR's VE, OE, UE, ZE, XE, NI, and RN bits enable or disable the corresponding floating-point exceptions and rounding mode. (See **archPpc.h** for the macros **PPC_FPSCR_VE** and so forth.)

Register values may be accessed by the routines *vxMsrGet( )*, *vxMsrSet( )*, *vxFpscrGet( )*, and *vxFpscrSet( )*.

VxMP Support for Motorola PowerPC Boards

VxMP is an optional VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running on separate CPUs. For complete documentation of the optional component VxMP, see *6. Shared-Memory Objects*.

Normally, boards that make use of VxMP must support hardware test-and-set (*TAS*: atomic read-modify-write cycle). Motorola PowerPC boards do not provide atomic (indivisible) TAS as a hardware function. VxMP for PowerPC provides special software routines which allow these Motorola boards to make use of VxMP.

Boards Affected

The current release of VxMP provides a software implementation of a hardware TAS for PowerPC-based VME boards of the 1300, 1600, and 2600 families manufactured by Motorola. No other PowerPC boards are affected.

NOTE: Some PowerPC board manufacturers, for example Cetia, claim to equip their boards with hardware support for true atomic operations over the VME bus. Such boards do not need the special software written for the Motorola boards.

Implementation

The VxMP product for Motorola PowerPC boards has special software routines which compensate for the lack of atomic TAS operations in the PowerPC and the lack of atomic instruction propagation to and from these boards. This software consists of the routines ***sysBusTas( )*** and ***sysBusTasClear( )***.

The software implementation uses ownership of the VME bus as a semaphore; in other words, no TAS operation can be performed by a task until that task owns the VME bus. When the TAS operation completes, the VME bus is released. This method is similar to the special read-modify-write cycle on the VME bus in which the bus is owned implicitly by the task issuing a TAS instruction. (This is the hardware implementation employed, for example, with a 68K processor.) However, the software implementation comes at a price. Execution is slower because, unlike true atomic instructions, ***sysBusTas( )*** and ***sysBusTasClear( )*** require many clock cycles to complete.

Configuring Hardware TAS

To invoke this feature, define **SM_TAS_TYPE** as **SM_TAS_HARD** in **configAll.h** or in **config.h** for your BSP.

Restrictions for Multi-Board Configurations

Systems using multiple VME boards where at least one board is a Motorola PowerPC board must have a Motorola PowerPC board as the board with a processor ID equal to 0 (the board whose memory is allocated and shared). This is because a TAS operation on local memory by, for example, a 68K processor does not involve VME bus ownership and is, therefore, not atomic as seen from a Motorola PowerPC board.

This restriction does not apply to systems that have globally shared memory boards which are used for shared memory operations. Specifying **SM_OFF_BOARD** as TRUE in **config.h** for the processor with ID of 0 and setting the associated parameters will enable you to assign processor IDs in any configuration.

Memory Layout

The VxWorks memory layout is the same for all PowerPC processors. Figure F-1 shows the memory layout, labeled as follows:

Interrupt Vector Table
Table of exception/interrupt vectors.

SM Anchor Anchor for the shared memory network and VxMP shared memory objects (if there is shared memory on the board).

Boot Line ASCII string of boot parameters.

Exception Message
ASCII string of the fatal exception message.

Initial Stack Initial stack for ***usrInit( )***, until ***usrRoot( )*** gets allocated stack.

System Image VxWorks itself (three sections: text, data, bss). The entry point for VxWorks is at the start of this region, which is BSP dependent. The entry point for each BSP is as follows:

cetCvme604	0x100,000
evb403, ads850	0x10,000
mv1603/4	0x30,000
ultra60X	0x10,000

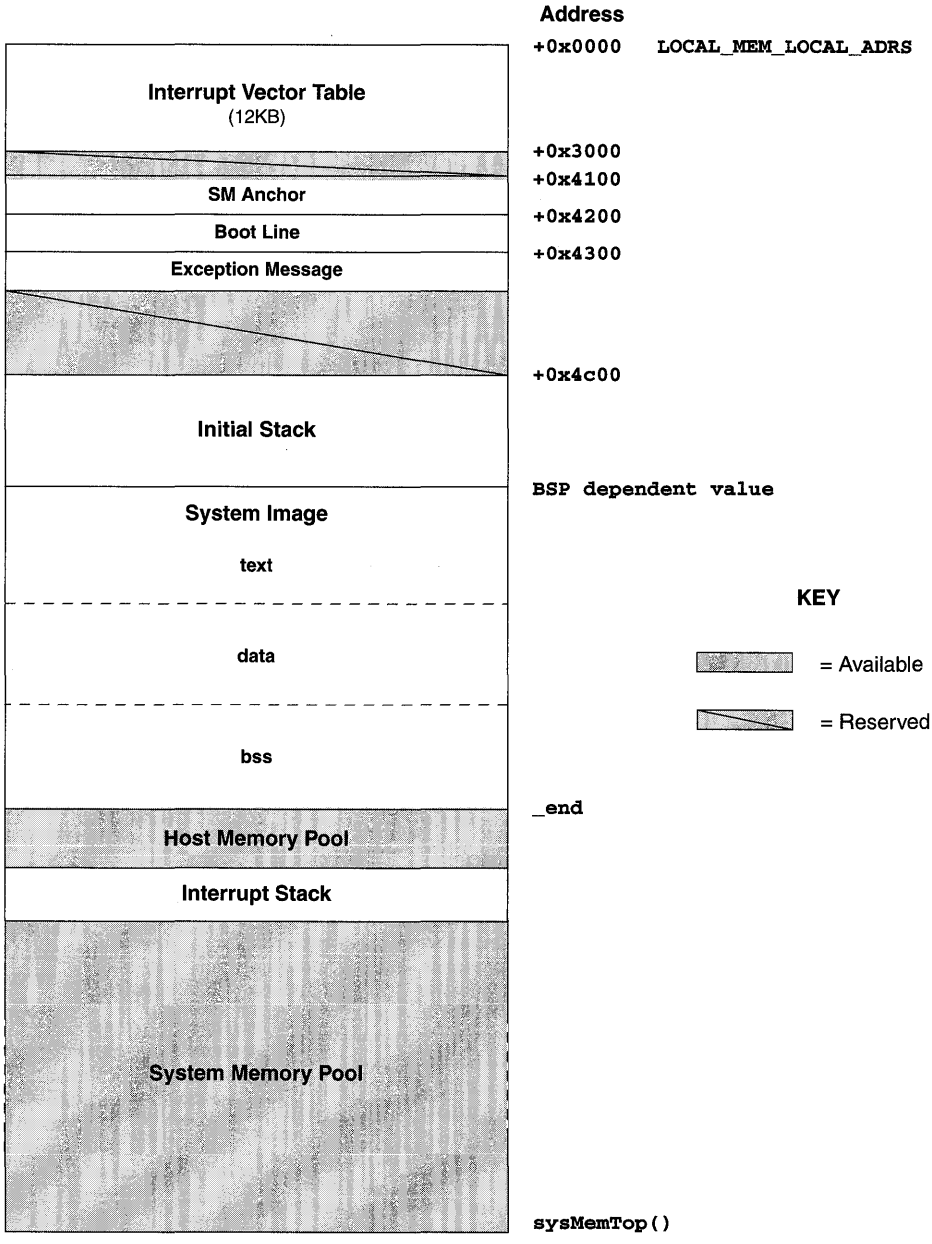
Host Memory Pool
Memory allocated by host tools. The size depends on the system image and is defined in **config/all/configAll.h**.

Interrupt Stack Stack for the interrupt handlers. The size is defined in **configAll.h**. The location depends on the system image and host memory pool sizes.

System Memory Pool
Size depends on the size of the system image. The ***sysMemTop( )*** routine returns the address of the end of the free memory pool.

All addresses shown in Figure F-1 are relative to the start of memory for a particular target board. The start of memory (corresponding to 0x0 in the memory-layout diagram) is defined as **LOCAL_MEM_LOCAL_ADRS** in **config.h** for each target.

Figure F-1 **VxWorks System Memory Layout (PowerPC)**



Index

Numerics

A

NOTE: Index entries of the form "*see also* **bootLib**(1)" refer to the module's reference entry in the *VxWorks Reference Manual* or the equivalent entry in the *Tornado Online Manuals*.

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

IX

T

U

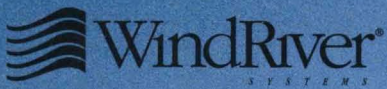
V

W

X

Y

Z



An ISO 9001 Registered Company

Wind River Systems
Corporate Headquarters
1010 Atlantic Avenue
Alameda, CA 94501 USA
1-800/545-WIND toll-free
1-510/748-4100 phone
1-510/749-2010 fax
inquiries@wrs.com
http://www.wrs.com
NASDAQ: WIND

Wind River Systems S.A.R.L.
Southern Europe, Africa
and Middle East
19, Avenue de Norvège
Immeuble OSLO, Bâtiment 3
Z. A. de Courtaboeuf 1
91953 Les Ulis Cédex
France
33-1-60-92-63-00 phone
33-1-60-92-63-15 fax

Wind River Systems
Italia s.r.l.
Centro Direzionale
"Piero della Francesca"
Corso Svizzera 185
10149 Torino
Italy
39-11-771-8058 phone
39-11-748-247 fax
infoItalia@wrsec.fr

Wind River Systems Israel
27-B Hametsuda Street
Industrial Zone
PO Box 11502
Azur 58001
Israel
972-3-559-8144 phone
972-3-559-8244 fax

Wind River Systems GmbH
Central Europe
Freisinger Straße 34
Postfach 1320
D-85737 Ismaning
Germany
49-89-96-24-45-0 phone
49-89-96-24-45-55 fax

Wind River Systems UK Ltd
Western Europe
Unit 5, Ashted Lock Way
Aston Science Park
Birmingham B7 4AZ
United Kingdom
44-121-628-1888 phone
44-121-628-1889 fax

Wind River Systems
Scandinavia
Northern Europe
Turebergs Torg 1
S-191 47 Sollentuna
Sweden
46-8-92-15-80 phone
46-8-92-15-65 fax

Wind River Systems
Japan/Asia-Pacific
Pola Ebisu Bldg. 11F
3-9-19 Higashi
Shibuya-ku
Tokyo 150
Japan
81-03-5467-5900 phone
81-03-5467-5877 fax

Wind River Systems Korea
18F Kyoungam Building
157-27 Samsung-Dong
Kangnam Ku
Seoul 135-090
Korea
82-2-555-7480 phone
82-2-555-5779 fax

DOC-12067-ZD-00