WIND RIVER

VxWorks®

KERNEL PROGRAMMER'S GUIDE

6.6

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

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

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

http://www.windriver.com

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

http://www.windriver.com/support

VxWorks Kernel Programmer's Guide, 6.6

14 Nov 07
Part #: DOC-16075-ND-00

Contents

PART I: CORE TECHNOLOGIES

PART II: MULTIPROCESSING TECHNOLOGIES

PART I
Core Technologies

1
Overview

1.1 Introduction

This guide describes the VxWorks operating system, and how to use VxWorks facilities in the development of real-time systems and applications. The first part, *Core Technologies*, covers the following topics:

- kernel facilities, kernel-based applications, and kernel customization
- boot loader
- multitasking facilities
- POSIX facilities
- memory management
- I/O system
- local file systems
- Network File System (NFS)
- flash file system support with TrueFFS
- error detection and reporting
- target tools, such as the kernel shell, kernel object-module loader, and target symbol table
- C++ development

The second part of this guide describes VxWorks multiprocessor technologies. For an introduction to this material, see *14. Overview of Multiprocessing Technologies*.

NOTE: This book provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the *VxWorks Application Programmer's Guide*.

1.2 Related Documentation Resources

The companion volume to this book, the *VxWorks Application Programmer's Guide*, provides material specific process-based (RTP) applications and process management.

Detailed information about VxWorks libraries and routines is provided in the VxWorks API references. Information specific to target architectures is provided in the VxWorks BSP references and in the *VxWorks Architecture Supplement*.

For information about BSP and driver development, see the *VxWorks BSP Developer's Guide* and the *VxWorks Device Driver Guide*.

The VxWorks networking facilities are documented in the *Wind River Network Stack for VxWorks 6 Programmer's Guide* and the *VxWorks PPP Programmer's Guide.*

For information about migrating applications, BSPs, drivers, and projects from previous versions of VxWorks and the host development environment, see the *VxWorks Migration Guide* and the *Wind River Workbench Migration Guide*.

The Wind River IDE and command-line tools are documented in the *Wind River Workbench User's Guide*, the *VxWorks Command-Line Tools User's Guide*, the Wind River compiler and GNU compiler guides, and the Wind River tools API and command-line references.

1.3 VxWorks Configuration and Build

This document describes VxWorks features; it does not go into detail about the mechanisms by which VxWorks-based systems and applications are configured and built. The tools and procedures used for configuration and build are described in the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

NOTE: In this guide, as well as in the VxWorks API references, VxWorks components and their configuration parameters are identified by the names used in component description files. The names take the form, for example, of **INCLUDE_FOO** and **NUM_FOO_FILES** (for components and parameters, respectively).

You can use these names directly to configure VxWorks using the command-line configuration facilities.

Wind River Workbench displays descriptions of components and parameters, as well as their names, in the **Components** tab of the Kernel Configuration Editor. You can use the **Find** dialog to locate a component or parameter using its name or description. To access the **Find** dialog from the **Components** tab, type CTRL+F, or right-click and select **Find**.

2
Kernel

2.1 Introduction

This chapter provides an overview of the VxWorks kernel architecture and detailed discussions of those features of interest to developers who work directly with kernel facilities. In general, kernel developers can modify and extend the VxWorks kernel in following ways:

- By reconfiguring and rebuilding VxWorks with various standard components to suit the needs of their application development environment, as well as the needs of their deployed products.
- By creating kernel applications that can either be interactively downloaded and run on a VxWorks target system, or configured to execute at boot time and linked with the operating system image.
- By creating custom kernel libraries that can be built into the operating system.
- By creating custom VxWorks components—such as file systems or networking protocols—that can be configured into VxWorks using the operating system configuration utilities.
- By extending the kernel system-call interface with custom APIs that should be accessible to applications running in user space (as real-time process—RTP—applications).
- By creating a custom scheduler for use in place of the traditional VxWorks scheduler or the POSIX thread scheduler.

See *2.4 VxWorks Configuration*, p.14; as well as other chapters throughout this book for information about VxWorks facilities and their use. Chapter *4. Multitasking*, for example, includes discussion of features that are available only in the kernel (such as ISRs and watchdog timers).

Section *2.6 Kernel Applications*, p.51 provides information about creating kernel applications. For information about RTP applications, see the *VxWorks Application Programmer's Guide: Applications and Processes*.

Instructions for creating custom kernel libraries is provided in the *VxWorks Command-Line Tools User's Guide*. Only brief mention of this topic is given in this book in *2.7 Custom Kernel Libraries*, p.67.

See *2.8 Custom VxWorks Components and CDFs*, p.67, *2.9 Custom System Calls*, p.100, and *2.10 Custom Scheduler*, p.118 for information about extending the operating system.

Developers can also write or port drivers and BSPs for VxWorks. These topics are covered by other books in the VxWorks documentation set; see the *VxWorks Device Driver's Guide* and the *VxWorks BSP Developer's Guide*.

2.2 Kernel Architecture

Historically, the VxWorks operating system provided a single memory space with no segregation of the operating system from user applications. All tasks ran in supervisor mode. Although this model afforded performance and flexibility when developing applications, only skilled programming could ensure that kernel facilities and applications coexisted in the same memory space without interfering with one another.[1]

With the release of VxWorks 6.0, the operating system provides support for real-time processes (RTPs) that includes execution of applications in user mode and other features common to operating systems with a clear delineation between kernel and applications. This architecture is often referred to as the *process model*. VxWorks has adopted this model with a design specifically aimed to meet the requirements of determinism and speed that are required for hard real-time systems. (For information about VxWorks processes and developing applications to run in processes, see *VxWorks Application Programmer's Guide: Applications and Processes*.) VxWorks 6.*x* provides full MMU-based protection of both kernel and user space.

At the same time, VxWorks 6.*x* maintains a high level of backward compatibility with VxWorks 5.5. Applications developed for earlier versions of VxWorks, and designed to run in kernel space, can be migrated to VxWorks 6.*x* kernel space with minimal effort (in most cases, merely re-compilation). For more information on this topic, see the *VxWorks Migration Guide*.

Naturally, new applications can be designed for kernel space as well, when other considerations outweigh the advantages of protection that executing applications as processes affords. These considerations might include:

- Size. The overall size of a system is smaller without components that provided for processes and MMU support.
- Speed. Depending on the number of system calls an application might make, or how much I/O it is doing when running as a process in user space, it might be faster running in the kernel.

[1] The VxWorks 5.x optional product VxVMI provides write protection of text segments and the VxWorks exception vector table, as well as an architecture-independent interface to the CPU's memory management unit (MMU). In addition, specialized variants of VxWorks such as VxWorks AE and VxWorks AE653 provide memory protection, but in a manner different from that provided in the current release.

- Kernel-only features. Features such as watchdog timers, ISRs, and VxMP are available only in the kernel. In some cases, however, there are alternatives for process-based applications (POSIX timers, for example).
- Hardware access. If the application requires direct access to hardware, it can only do so from within the kernel.

VxWorks is flexible in terms of both the modularity of its features and its extensibility. The operating system can be configured as a minimal kernel that provides a task scheduler, interrupt handling, dynamic memory management, and little else. Or, it can be configured with components for executing applications as processes, file systems, networking, error detection and reporting, and so on.

The operating system can also be extended by adding custom components or modules to the kernel itself (for example, for new file systems, networking protocols, or drivers). The system call interface can then be extended by adding custom APIs, which makes them available to process-based applications.

2.2.1 Operating System Facilities

VxWorks provides a core set of facilities that are commonly provided by the kernel of a multitasking operating system:

- Startup facilities for system initialization (see *2.3 System Startup*, p.13).
- Clocks and timers (see *4.19 Watchdog Timers*, p.239 and *5.7 POSIX Asynchronous I/O*, p.263).
- Exception and interrupt handling (see *Exception Task*, p.12, *4.6 Task Exception Handling*, p.187, *4.18 Signals*, p.226, and *4.20 Interrupt Service Routines*, p.241).
- Task management (see *4.2 Tasks and Multitasking*, p.160).
- Process management (see the *VxWorks Application Programmer's Guide: Applications and Processes*).
- A system call interface for applications executing in processes (see *VxWorks Application Programmer's Guide: Applications and Processes* and *2.9 Custom System Calls*, p.100).
- Intertask and interprocess communication (see *4.8 Intertask and Interprocess Communication*, p.193.
- Signals (see *4.18 Signals*, p.226).
- Resource reclamation (see *VxWorks Application Programmer's Guide: Applications and Processes*).

- Memory management (see *6. Memory Management*).
- I/O system (see *7. I/O System*).
- File systems (see *8. Local File Systems*).
- NFS (see *9. Network File System: NFS*).

In addition, the VxWorks kernel also provides:

- The WDB target agent, which is required for using the host development tools with VxWorks. It carries out requests transmitted from the tools (by way of the target server) and replies with the results (see *12.6 WDB Target Agent*, p.626).
- Facilities for error detection and reporting (see *11. Error Detection and Reporting*).
- A target-based shell for direct user interaction, with a command interpreter and a C-language interpreter (see *12.2 Kernel Shell*, p.577).
- A specialized facilities for multi-processor intertask communication through shared memory (see *16. Shared-Memory Objects: VxMP*).

For information about basic networking facilities, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

2.2.2 System Tasks

Depending on its configuration, VxWorks includes a variety of system tasks, which are always running. These are described below.

Root Task

The root task **tRootTask** is the first task executed by the kernel. The entry point of the root task is **usrRoot( )**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 completed. For more information **tRootTask** and **usrRoot( )**, see the *VxWorks BSP Developer's Guide*.

Logging Task

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 *7.7 Asynchronous Input/Output*, p.381 and the API reference entry for **logLib**.

Exception Task

The exception task, **tExcTask**, supports the VxWorks exception handling package by performing functions that cannot occur at interrupt level. It is also used for actions that cannot be performed in the current task's context, such as task suicide. 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**.

Network Task

The **tNet0** task is the default network daemon. It handles the task-level (as opposed to interrupt-level) processing required by the VxWorks network. For systems that have been configured with more than one network daemon, the task names are **tNet***n*. The task is primarily used by network drivers. Configure VxWorks with the **INCLUDE_NET_DAEMON** component to spawn the **tNet0** task. For more information on **tNet0**, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

WDB Target Agent Task

The WDB target agent task, **tWdbTask**, is created if the target agent is set to run in task mode. It services requests from the host tools (by way of the target server); for information about this server, see the host development environment documentation. Configure VxWorks with the **INCLUDE_WDB** component to include the target agent. See *12.6 WDB Target Agent*, p.626 for more information about WDB.

Tasks for Optional Components

The following VxWorks system tasks are created if their components are included in the operating system configuration.

tShell*num*

If you have included the kernel shell in the VxWorks configuration, it is spawned as a task. Any routine or task that is invoked from the kernel shell, rather than spawned, runs in the **tShell***num* context.

The task name for a shell on the console is **tShell0**. The kernel shell is re-entrant, and more than one shell task can run at a time (hence the number suffix). In addition, if a user logs in remotely (using **rlogin** or **telnet**) to a VxWorks target, the name reflects that fact as well. For example, **tShellRem1**.

For more information, see *12.2 Kernel Shell*, p.577. Configure VxWorks with the **INCLUDE_SHELL** component to include the kernel shell.

tRlogind

If you have included the kernel 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**_*hexNumber* and **tRlogOutTask**_*hexNumber* (for example, **tRlogInTask_5c4d0**). These tasks exist as long as the remote user is logged on. Configure VxWorks with the **INCLUDE_RLOGIN** component to include the **rlogin** facility.

tTelnetd

If you have included the kernel 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**_*hexNumber* and output task **tTelnetOutTask**_*hexNumber*. These tasks exist as long as the remote user is logged on. Configure VxWorks with the **INCLUDE_TELNET** component to include the telnet facility.

tPortmapd

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

tJobTask

The **tJobTask** executes jobs—that is, function calls—on the behalf of tasks. (The **tExcTask** task executes jobs on the behalf of ISRs.) It runs at priority 0 while waiting for a request, and dynamically adjusts its priority to match that of the task that requests job execution. Configure VxWorks with the **INCLUDE_JOB_TASK** component to include the job facility. For more information see, *4.4.6 Task Deletion and Deletion Safety*, p. 180.

2.3 System Startup

When a VxWorks system is powered on, the boot loader copies an operating system image into memory and directs the CPU to begin executing it. The boot loader is most often located in ROM (although it can also be stored on a disk). The VxWorks image can be stored on a host or network file system, as is usually the case during development—or stored in ROM with the boot loader, as is often the case with production units. The VxWorks boot loader is actually a scaled-down version of VxWorks itself, whose sole purpose is to load a system image and initiate its execution. (See *3. Boot Loader*.) For more information about system startup, see the *VxWorks BSP Developer's Guide: Overview of a BSP*.

2.4 VxWorks Configuration

VxWorks is a flexible, scalable operating system with numerous facilities that can included, excluded, variously configured, and extended with customized technologies, depending on the requirements of your applications and system, and the stage of the development cycle.

VxWorks distributions include default system images for each supported BSP. Each system image is a binary module that can be booted and run on a target system. A system image consists of a set of components linked together into a single non-relocatable object module with no unresolved external references.

The default system images are designed for the development environment. They contain the basic set of components that are necessary to interact with the system using host development tools. In most cases, you will find the supplied system image adequate for initial development (provided the default drivers are appropriate). Using a default VxWorks image, you can interactively download and run kernel applications.

During the development cycle you may want to reconfigure and rebuild VxWorks with components specifically selected to support your applications and development requirements. If, for example, you configure VxWorks with the appropriate components and initialization settings, you can link kernel applications with VxWorks and start them automatically at boot time (see *2.6 Kernel Applications*, p.51). You can also configure VxWorks with support for process-based applications, to store them as part of the system image in ROMFS, and to run them interactively and automatically (see *VxWorks Application Programmer's Guide: Applications and Processes*).

If the VxWorks components provided by Wind River do not provide all the facilities required for your system, you can create custom facilities, such as new file systems and networking protocols, and package them as components (see *2.8 Custom VxWorks Components and CDFs*, p.67), add new system calls for process-based applications (see *2.9 Custom System Calls*, p.100), create your own scheduler (see *2.10 Custom Scheduler*, p.118), and so on.

Finally, for production systems, you will want to reconfigure VxWorks with only those components needed for deployed operation, and to build it as the appropriate type of system image (see *2.4.1 VxWorks Image Types*, p.15). For production systems you will likely want to remove components required for host development support, such as the WDB target agent and debugging components (**INCLUDE_WDB** and **INCLUDE_DEBUG**), as well as to remove any other operating system components not required to support your application. Other considerations include reducing the memory requirements of the system, speeding up boot time, and security issues.

For information about using the Workbench and command-line tools to configure and build VxWorks, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

2.4.1 VxWorks Image Types

Different types of VxWorks system images can be produced for a variety of storage, loading, and execution scenarios. Default versions of the following images are provided in the VxWorks installation. Customized versions with different components can also be created. Note that only one image type requires a boot loader, and that the others are *self-booting*.

The various VxWorks image types, their use, and behavior are:

vxWorks
: This VxWorks image type is intended for use during development and is often referred to as *downloadable*. It is also useful for production systems in which the boot loader and system image are stored on disk. In a development environment, the image is usually stored on the host system (or a server on the network), downloaded to the target system by the boot loader, and loaded into RAM. The symbol table is maintained on the host (in the file **vxWorks.sym**), where it is used by the host development tools. Leaving the symbol table on the host keeps the image size down and reduces boot time. If VxWorks is reconfigured with the **INCLUDE_STANDALONE_SYM_TBL** component, the symbol table is included in the VxWorks image.

vxWorks_rom
: A VxWorks image that is stored in ROM on the target. It copies itself to RAM and then makes the processor switch execution to RAM. Because the image is not compressed, it is larger than the other ROM-based images and therefore

has a slower startup time; but it has a faster execution time than **vxWorks_romResident**.

vxWorks_romCompress
A VxWorks image that is stored in ROM on the target. It is almost entirely compressed, but has small uncompressed portion executed by the processor immediately after power up/reboot. This small portion is responsible for decompressing the compressed section of the ROM image into RAM and for making the processor switch execution to RAM. The compression of the image allows it to be much smaller than other images. However the decompression operation increases the boot time. It takes longer to boot than **vxWorks_rom** but takes up less space than other ROM-based images. The run-time execution is the same speed as **vxWorks_rom**.

vxWorks_romResident
A VxWorks image that is stored in ROM on the target. It copies only the data segment to RAM on startup; the text segment stays in ROM. Thus it is described as being *ROM-resident*. It has the fastest startup time and uses the smallest amount of RAM, but it runs slower than the other image types because the ROM access required for fetching instructions is slower than fetching them from RAM. It is obviously useful for systems with constrained memory resources.

The default VxWorks image files can be found in sub-directories under *installDir***/vxworks-6.***x***/target/proj/***projName*. For example:

/home/moi/myInstallDir/vxworks-6.*x***/target/proj/wrSbc8260_diab/default_rom/vxWorks_rom**

For many production systems it is often necessary to store a kernel application module that is linked with VxWorks in ROM. VxWorks can be configured to execute the application automatically at boot time. The system image can also simply store the application module to allow for its being called by other programs, or for interactive use by end-users (for example, diagnostic programs).

To produce a ROM-based system, you must link the module with VxWorks, and build an image type that is suitable for ROM. See *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64. If you wish to have the application start automatically at boot time, you must also configure VxWorks to do so (see *2.6.10 Configuring VxWorks to Run Applications Automatically*, p.66). Also see *2.6.9 Image Size Considerations*, p.65.

Note that, during development, VxWorks must be configured with the WDB target agent communication interface that is required for the type of connection used between your host and target system (network, serial, and so on). By default, it is configured for an Enhanced Network Driver (END) connection. For more information, see *12.6 WDB Target Agent*, p.626. Also note that before you use the host development tools such as the shell and debugger, you must start a target server that is configured for the same mode of communication.

For information about configuring VxWorks with different operating system facilities (components), see *2.4.3 Device Driver Selection*, p.22.

If you are going to store boot image in flash, and want to user TrueFFS as well, see *10.3.6 Reserving a Region in Flash for a Boot Image*, p.554.

2.4.2 VxWorks Components

A VxWorks component is the basic unit of functionality with which VxWorks can be configured. While some components are autonomous, others may have dependencies on other components, which must be included in the configuration of the operating system for run-time operation. The kernel shell is an example of a component with many dependencies. The symbol table is an example of a component upon which other components depend (the kernel shell and module loader; for more information, see *12. Target Tools*).

The names, descriptions, and configurable features of VxWorks can be displayed with the GUI configuration facilities in Workbench. Workbench provides facilities for configuring VxWorks with selected components, setting component parameters, as well as automated mechanisms for determining dependencies between components during the configuration and build process.

The command-line operating system configuration tool—**vxprj**—uses the naming convention that originated with configuration macros to identify individual operating system components. The convention identifies components with names that begin with **INCLUDE**. For example, **INCLUDE_MSG_Q** is the message queue component. In addition to configuration facilities, the **vxprj** tool provides associated features for listing the components included in a project, and so on.

For information about the Workbench and command-line facilities used for configuring and building VxWorks, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

Textual configuration files identify components with macro names that begin with **INCLUDE**, as well as with user-friendly descriptions. (For information about configuration files, see *2.8.4 Component Description Language*, p.79.)

In this book, components are identified by their macro name. The GUI configuration facilities provide a search facility for finding individual components in the GUI component tree based on the macro name.

Some of the commonly used VxWorks components are described in Table 2-1. Names that end in *XXX* represent families of components, in which the *XXX* is replaced by a suffix for individual component names. For example, **INCLUDE_CPLUS_***XXX* refers to a family of components that includes **INCLUDE_CPLUS_MIN** and others.

Note that Table 2-1 does not include all components provided in the default configuration of VxWorks, and that the VxWorks simulator provides more components by default.

Table 2-1 **Key VxWorks Components**

Component	Default	Description
INCLUDE_ANSI_*XXX*	*	Various ANSI C library options
INCLUDE_BOOTLINE_INIT		Parse boot device configuration information
INCLUDE_BOOTP	*	BOOTP support
INCLUDE_CACHE_SUPPORT	*	Cache support
INCLUDE_CPLUS	*	Bundled C++ support
INCLUDE_CPLUS_*XXX*		Various C++ support options
INCLUDE_DEBUG		Kernel shell debugging facilities
INCLUDE_EDR_*XXX*		Error detection and reporting facilities.
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_IO_SYSTEM	*	I/O system and associated interfaces. Directory and path manipulation APIs (for more information, see *7.2 Configuring VxWorks With I/O Facilities*, p.362)

Table 2-1 **Key VxWorks Components** (cont'd)

Component	Default	Description
INCLUDE_LOADER		Target-resident kernel object module loader package
INCLUDE_LOGGING	*	Logging facility
INCLUDE_MEM_MGR_BASIC	*	Core partition memory manager
INCLUDE_MEM_MGR_FULL	*	Full-featured memory manager
INCLUDE_MIB2_*XXX*		Various MIB-2 options
INCLUDE_MMU_BASIC	*	Bundled MMU support
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
INCLUDE_PROTECT_VEC_TABLE		Vector table write protection
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_ROMFS		ROMFS file system
INCLUDE_RTP		Real-time process support.
INCLUDE_SCSI		SCSI support
INCLUDE_SCSI2		SCSI-2 extensions

Table 2-1 **Key VxWorks Components** (cont'd)

Component	Default	Description
INCLUDE_SECURITY		Remote login security package
INCLUDE_SELECT		Select facility
INCLUDE_SEM_BINARY	*	Binary semaphore support
INCLUDE_SEM_COUNTING	*	Counting semaphore support
INCLUDE_SEM_MUTEX	*	Mutual exclusion semaphore support
INCLUDE_SHELL		Kernel (target) shell
INCLUDE_*XXX***_SHOW**		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 buffered 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

Table 2-1 **Key VxWorks Components** (cont'd)

Component	Default	Description
INCLUDE_TIMEX	*	Function execution timer
INCLUDE_TRIGGERING		Function execution timer
INCLUDE_UNLOADER		Target-resident kernel object module unloader package
INCLUDE_VXEVENTS		VxWorks events support.
INCLUDE_WATCHDOGS	*	Watchdog support
INCLUDE_WDB	*	WDB target agent (see *12.6 WDB Target Agent*, p.626)
INCLUDE_WDB_TSFS	*	Target server file system
INCLUDE_WINDVIEW		System Viewer command server (see the *Wind River System Viewer User's Guide*)

By default, VxWorks includes both **libc** and GNU **libgcc**, which are provided with the **INCLUDE_ALL_INTRINSICS** component. If you wish to exclude one or the other library, you can do so by reconfiguring the kernel with either **INCLUDE_DIAB_INTRINSICS** or **INCLUDE_GNU_INTRINSICS**, respectively. Note that these libraries are available in the kernel to enable dynamically downloading and running kernel object modules.

2.4.3 Device Driver Selection

Device drivers are provided as VxWorks components that can be added to or removed from a system using Workbench and **vxprj**. Some drivers are VxBus-compliant, and others (legacy drivers) are not.

NOTE: Only VxBus-compatible drivers can be used with the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

Note that the component names for VxBus drivers do not have the leading **INCLUDE_** element (for example, **DRV_SIO_NS16550**), whereas the names for non-VxBus drivers do (for example, **INCLUDE_ELT_3C509_END**).

For information about the VxBus facility, see the *VxWorks Device Driver Developer's Guide*.

2.4.4 VxWorks Configuration Profiles

In addition to components and component bundles, configuration profiles can be used to configure a VxWorks system. Profiles provide a convenient way of providing a base line of operating system functionality that is different from the default configuration available with the VxWorks product installation. The following profiles are available:

PROFILE_MINIMAL_KERNEL—Minimal VxWorks Kernel Profile
Provides the lowest level of services at which a VxWorks system can operate. It consists of the micro-kernel, and basic CPU and BSP support. This profile is meant to provide a very small VxWorks system that can support multitasking and interrupt management at a very minimum, but semaphores and watchdogs are also supported by default. (See *2.4.5 Small VxWorks Configuration Profiles*, p.24.)

PROFILE_BASIC_KERNEL—Basic VxWorks Kernel Profile
Builds on the minimal kernel profile, adding support for message queues, task hooks, memory allocation and de-allocation, and basic I/O facilities. Applications based on this profile can be more dynamic and feature rich than the minimal kernel. (See *2.4.5 Small VxWorks Configuration Profiles*, p.24.)

PROFILE_BASIC_OS—Basic VxWorks OS Profile

Provides a small operating system on which higher level constructs and facilities can be built. It supports a full I/O system, file descriptors, and related ANSI routines. It also supports task and environment variables, signals, pipes, coprocessor management, and a ROMFS file system. (See *2.4.5 Small VxWorks Configuration Profiles*, p.24.)

PROFILE_COMPATIBLE—VxWorks 5.5 Compatible Profile

Provides the minimal configuration that is compatible with VxWorks 5.5.

PROFILE_DEVELOPMENT—VxWorks Kernel Development Profile

Provides a VxWorks kernel that includes development and debugging components.

PROFILE_ENHANCED_NET—VxWorks Enhanced Network Profile

Adds components appropriate for typical managed network client host devices to the default profile. The primary components added are the DHCP client and DNS resolver, the Telnet server (shell not included), and several command-line-style configuration utilities.

PROFILE_CERT—VxWorks DO-178 Certification Profile

Provides a DO-178B Level A-certifiable API subset of the VxWorks operating system.

PROFILE_BOOTAPP—VxWorks Boot Loader Profile

Provides a VxWorks boot loader. For more information, see *3.7 Customizing and Building Boot Loaders*, p.146.

2.4.5 Small VxWorks Configuration Profiles

VxWorks can be scaled below the size of the default operating system using special configuration profiles (sometimes referred to as *source-scalable* or *scalable profiles*). They provide three levels of operating system functionality, starting with a minimal kernel through a basic operating system. These profiles do not support networking facilities. The small configuration profiles are as follows:

- **PROFILE_MINIMAL_KERNEL**—The minimal kernel profile provides the lowest level of services at which a VxWorks system can operate. It consists of the microkernel, and basic CPU and BSP support. This profile creates a very small VxWorks systems that can support multitasking and interrupt management. It does not provide support for dynamic memory allocation; all objects must be statically instantiated. For more information see *Minimal Kernel Profile*, p.28.

- **PROFILE_BASIC_KERNEL**—The basic kernel profile builds on the minimal kernel profile, adding support for message queues, task hooks, dynamic memory allocation and de-allocation, and a basic I/O system. For more information, see *Basic Kernel Profile*, p.33.

- **PROFILE_BASIC_OS**—The basic OS profile provides a small operating system. It supports a full I/O system, file descriptors, and related ANSI routines. It also supports task and environment variables, signals, pipes, coprocessor management, and the ROMFS file system. This profile is close to a VxWorks 5.5 configuration, but without the network stack and debugging assistance tools. It can also be viewed as a lightweight version of the default VxWorks 6.*x* configuration, but without memory protection, the network stack, System Viewer instrumentation, and debugging assistance support (the WDB target agent). For more information, see *Basic OS Profile*, p.35.

Each of the small VxWorks profiles produce an operating system that is smaller than the current default VxWorks 6.*x* configuration. The smallest configuration can build an image of 100 KB or less. The profiles are not monolithic. Each can be used as a base upon which additional functionality can be added with other VxWorks components, using standard configuration facilities (Workbench and **vxprj**). Custom facilities and applications can also be added, as with VxWorks configurations based on components and other profiles. And, additional measures—such as static instantiation of objects—can be used to keep the system as lean and efficient as possible.

Figure 2-1 illustrates the relationship between the three profiles and the functionality they provide.

NOTE: The small VxWorks profiles are built from source code, so you must install VxWorks source to use them. For this release they can only be built with the Wind River compiler.

NOTE: For this release, the small VxWorks profiles are not available for all BSPs. They are provided for wrSbcPowerQuiccII (for PowerPC) and integrator1136jfs (for ARM).

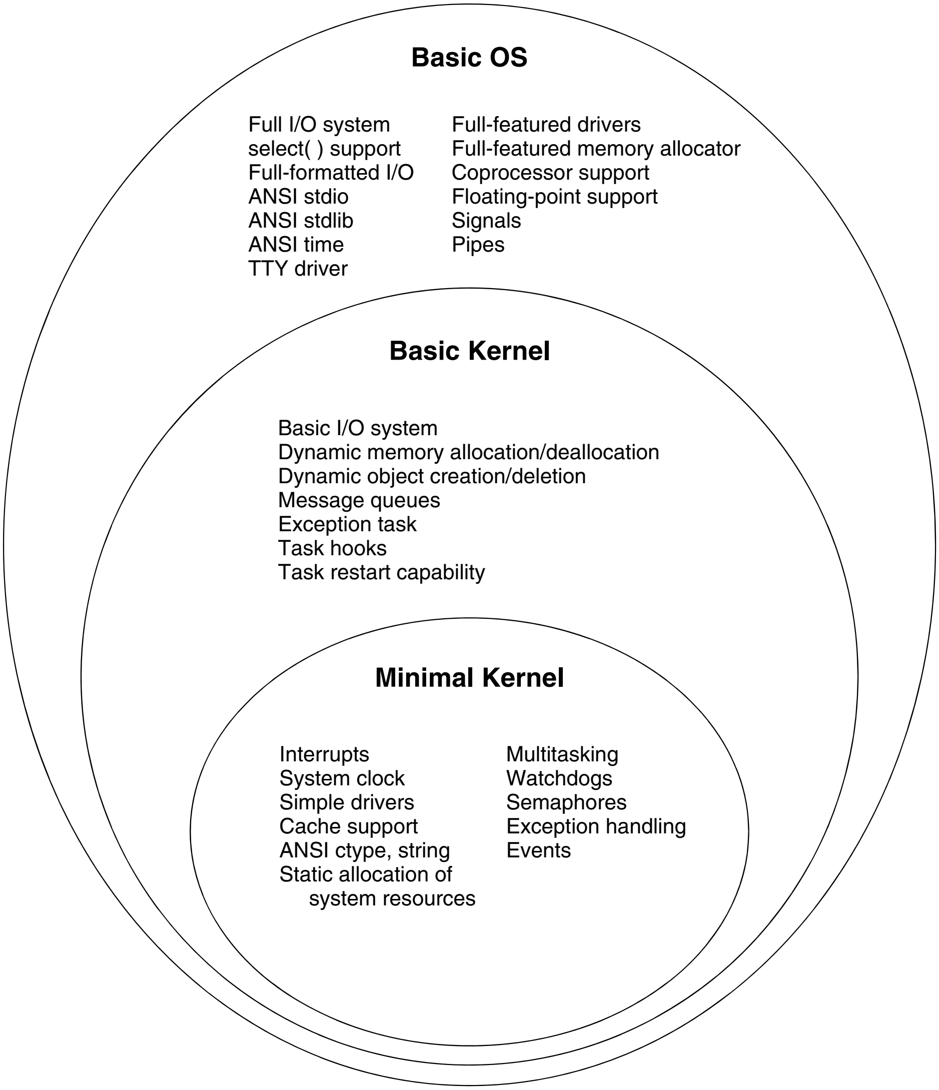
How Small Configuration Profiles Create Smaller Systems

The **PROFILE_MINIMAL_KERNEL**, **PROFILE_BASIC_KERNEL**, and **PROFILE_BASIC_OS** profiles all produce systems that are smaller than those that can be created from standard VxWorks components. They can do so because they are build from conditionally-compiled source code, which selectively eliminates code from individual components that is not strictly required for the level of functionality that each profile provides (as, for example, is done with **taskLib**).

Standard VxWorks components on the other hand, are provided as pre-compiled libraries, which are linked into a system by way of symbolic references. This method is convenient, and provides for quick builds, but it means that components tend to somewhat larger and more general purpose than suits the needs of systems with very tight requirements for size and functionary.

Small configuration profiles allow for building systems that are scaled down to smaller sizes in a simple, predetermined manner. Each provides a subset of the features—and in some cases elements of standard features—that are provided in in the default VxWorks configuration. The small configuration profiles also provide improved system performance, because they only include code that is carefully tailored to the requirements of the system.

Figure 2-1 **Small VxWorks Configuration Profiles**



Configuring and Building Small VxWorks Configurations

The small VxWorks profiles are built from source code, so you must install VxWorks source to use them. You must also create a project using the appropriate configuration profile and select the option for building from source. For the **vxprj** command-line tool, use the **-source** option; for Workbench, select the source build option during project creation.

Each small VxWorks profile has a predefined set of components that belong to it. If you choose to change the default configuration of a profile, you should be aware of the following guidelines and behavior:

- If components are not needed, they can be removed.
- If a component that is used in a higher-level profile is added, the profile is automatically elevated to the next level that uses the component. For example, if you add a message queue component to the **PROFILE_MINIMAL_KERNEL** profile the project is automatically expanded to the **PROFILE_BASIC_KERNEL** profile, with all the components provided by that profile.
- If you add components that are intended to be used with any of the (source) small profiles—such as networking components—the project defaults to using the standard binary components. In effect, the advantage of using a small VxWorks profile is lost.

⚠ **CAUTION:** In order to maintain a small VxWorks configuration profile for a project, you must be careful with adding other components. Adding components that are not part of one of these profiles causes the project to use the default binary components, and produces a larger system with different functionality (even on a per-component basis) than the profile you started with.

Optimization of Small VxWorks Profile Systems

Systems produced with small VxWorks profiles can be further optimized when code is developed using the following methods:

- Static instantiation of kernel objects—which is required for **PROFILE_MINIMAL_KERNEL** applications—can be useful with the other profiles as well. For information in this regard, see *2.6.4 Static Instantiation of Kernel Objects*, p.56.

- Using the **VX_GLOBAL_NO_STACK_FILL** configuration parameter disables stack filling on all tasks in the system (for information about the parameter, see *Filling Task Stacks*, p.175).

Minimal Kernel Profile

The minimal kernel profile (**PROFILE_MINIMAL_KERNEL**) provides the most basic level at which a VxWorks system can operate. This profile consists of the micro-kernel and basic support code for the CPU and BSP.

The profile provides multitasking and interrupt management at a very minimum. Support for semaphores (binary, counting, and mutexes) and watchdogs is also present, but is optional (that is, it can be removed). A subset of the ANSI C library is available in the minimal kernel profile. These include support for **ctype**, string manipulation, and routines converting numbers to strings and vice versa. The minimal kernel profile does not provide an I/O system or mechanisms for dynamic memory allocation.

The profile provides the foundation for small statically-resourced kernel applications, such as small fixed-function controller types of systems that are traditionally implemented using micro-controllers.

Statically-resourced systems are systems in which all application and kernel entities are pre-allocated by the compiler at system build time. There are no memory allocations done at run-time. Therefore minimal kernel systems spend less time initializing themselves, and have no chance of resource unavailability. There is a much higher level of determinism in these systems because resources are guaranteed to be available as soon as the system boots. All the application need do is to initialize the storage appropriately before it is ready for business. Static instantiation of kernel objects are explained in detail in *2.6.4 Static Instantiation of Kernel Objects*, p.56.

Static instantiation of system resources does have its own shortcomings. It is not suitable for systems that are inherently dynamic in nature, that is, ones that have widely differing amounts of loads, or those that must instantiate and destroy objects on demand. Statically-resourced systems therefore are not suitable for all types of applications.

Static Memory Allocation

A unique characteristic of the minimal kernel profile is that it requires static memory allocation; that is, all memory allocation must occur at compile time. There is no support for dynamic memory allocation (that is, **malloc( )**, **free( )**, and related routines are not supported). Both system and application components are expected to declare storage for all objects at compile time (that is, statically). The absence of dynamic allocation in this profile also implies that the kernel cannot dynamically instantiate objects like tasks, semaphores and watchdogs. APIs such as **taskSpawn( )**, **taskDelete( )**, **sem***X***Create( )**, **semDelete( )**, and so on, are not available. However the same kind of objects can be instantiated statically.

Unsupported Facilities

The minimal kernel profile is a very small, limited environment that is suitable only for small, fixed-function systems. Consequently, it is also a very limited programming environment. Significant capabilities that are absent from a minimal kernel, which would otherwise be present in more feature rich configurations of VxWorks. These features are as follows:

- Dynamic memory allocation and de-allocation. The ability to destroy or terminate kernel objects.
- Memory protection (that is, detection of null pointer and other invalid memory accesses).
- Support for task hooks (that is, **taskHookLib**).
- Floating point and other coprocessor support.
- I/O system.
- Signals.
- Processes (RTPs).
- ISR Objects (that is, **isrLib**).
- WDB target agent.
- Networking.
- C++ support.
- System Viewer logging.

Device Drivers for Minimal Kernel Systems

Since there is no I/O system in this profile, there is no support for traditional device access APIs like **open( )**, **close( )**, **read( )**, **write( )**, and so on. Device drivers written for such systems must be standalone programs that manage devices directly. DMA-safe buffers (if needed) must be allocated at compile time. Since there is no support for **malloc( )** or **free( )**, there is correspondingly no support for **cacheDmaMalloc( )** or **cacheDmaFree( )** either.

Formatted Character String Output

The **INCLUDE_FORMATTED_OUT_BASIC** component supplies capability for formatted output with routines like **printf( )**, **sprintf( )**, **snprintf( )**, **printfExc( )**, and so on. In the minimal and basic kernel profiles, these routines are limited to outputting only integer values and strings. The supported formats are **%d**, **%s**, **%c**, **%x**, **%X**, **%u**, and **%o** only. Floating point, 64-bit, or vector type formats are not supported with this profile.

Because there is no I/O system in the minimal kernel profile, there are no file descriptors, and the assumption that **printf( )** output is sent to file descriptor 1 is not true for this profile. The **printf( )** routine works for the formats described above, but its output is sent to the console device through a dedicated function.

Do not attempt to make use of file descriptors, standard output, or standard error with this profile, because they do not operate in the standard manner.

Minimal Kernel Profile Components and Libraries

The components and libraries that make up the VxWorks minimal kernel profile are listed in Table 2-2.

Table 2-2 **Minimal Kernel Profile Components**

Component	Library	Description
INCLUDE_EDR_STUB		Error detection and reporting stub.
INCLUDE_ANSI_CTYPE	**ansiCtype**	ANSI **ctype** routines like **isalpha( )**, **iscntrl( )**, **isdigit( )**, and so on.
INCLUDE_ANSI_BSEARCH		ANSI **bsearch( )** routine
INCLUDE_ANSI_STDLIB_NUMBERS	**ansiStdlib**	ANSI **stdlib** string-number conversion routines.
INCLUDE_ANSI_STRING	**ansiString**	Full set of ANSI string routines.
INCLUDE_ANSI_ABS		ANSI **abs( )** routine.
INCLUDE_ANSI_MEMCHR	**ansiString**	ANSI **memchr( )** routine.
INCLUDE_ANSI_MEMCPY	**ansiString**	ANSI **memcpy( )** routine.
INCLUDE_ANSI_MEMSET	**ansiString**	ANSI **memset( )** routine.
INCLUDE_ANSI_MEMCMP	**ansiString**	ANSI **memcmp( )** routine.
INCLUDE_ANSI_MEMMOVE	**ansiString**	ANSI **memmove( )** routine.
INCLUDE_ANSI_STRCAT	**ansiString**	ANSI **strcat( )** routine.
INCLUDE_ANSI_STRNCAT	**ansiString**	ANSI **strncat( )** routine.

Table 2-2 **Minimal Kernel Profile Components**

Component	Library	Description
INCLUDE_ANSI_STRCMP	**ansiString**	ANSI **strcmp( )** routine.
INCLUDE_ANSI_STRNCMP	**ansiString**	ANSI **strncmp( )** routine.
INCLUDE_ANSI_STRCPY	**ansiString**	ANSI **strcpy( )** routine.
INCLUDE_ANSI_STRNCPY	**ansiString**	ANSI **strncpy( )** routine.
INCLUDE_ANSI_STRLEN	**ansiString**	ANSI **strlen( )** routine.
INCLUDE_REBOOT_HOOKS	**rebootLib**	Support for reboot hooks; that is **rebootHookAdd( )**.
INCLUDE_VXEVENTS	**eventLib, semEvLib, msgQEvLib**	VxWorks events support.
INCLUDE_SEM_BINARY	**semBLib**	Support for binary semaphores.
INCLUDE_SEM_MUTEX	**semMLib**	Support for mutex semaphores.
INCLUDE_SEM_COUNTING	**semCLib**	Support for counting semaphores.
INCLUDE_TASK_UTIL	**taskUtilLib**	Programmatic interface for modifying task information.
INCLUDE_WATCHDOGS	**wdLib**	Support for watchdog timers.
INCLUDE_HOOKS	**hookLib**	Hook routine table support.
INCLUDE_VX_TRADITIONAL_SCHEDULER		VxWorks traditional scheduler (priority-based preemptive scheduling).
INCLUDE_FORMATTED_OUT_BASIC	**fioLib**	Support for **printf( )**, **sprintf( )**, **snprintf( )**, **oprintf( )**, and **printErr( )** only. No support for **scanf( )** and its variants. No support for **vprintf( )** or its variants. No support for floating point or vector formats.

Table 2-2 **Minimal Kernel Profile Components**

Component	Library	Description
INCLUDE_BOOT_LINE_INIT	**bootParseLib**	Parse boot device configuration information

Basic Kernel Profile

The level above the minimal kernel is provided by the basic kernel profile (**PROFILE_BASIC_KERNEL**). The basic kernel profile produces small VxWorks systems that build on the minimal kernel to provide support for moderately complex applications. Systems based on the basic kernel profile are still not much more than a kernel But in addition to the a minimal kernel system, the basic kernel profile offers support for the following facilities:

- Basic I/O system
- Inter-task communication using message queues.
- Support for task hooks.
- Memory allocation and free (using **memPartLib**).
- Ability to dynamically create and delete kernel objects such as tasks, semaphores, watchdogs and message queues (enabled by **memPartLib**).
- Support for ANSI string routine **strdup( )**, which relies on **malloc( )**.

The most notable additions to this profile are support for basic I/O facilities, support for message queues and task hooks, and support for memory allocation and de-allocation. This allows applications based on this profile to be more dynamic and feature-rich than the minimal kernel. What this profile provides, however, is still a kernel and not an operating system. It has a full complement of intertask communications mechanisms and other kernel features, but does not have operating system capabilities such as memory protection, file system support, or higher-level constructs such as pipes and so on.

Device Drivers for Basic Kernel Systems

Like the minimal kernel profile, there is no I/O system present in the basic kernel profile. Hence device drivers for such systems must be standalone programs, managing devices directly. Since **malloc( )** and **free( )** are supported, **cacheDmaMalloc( )** and **cacheDmaFree( )** are available starting with this profile.

Formatted Character String Output

The very same limitations on formatted output apply to the basic kernel profile, as are present for the minimal kernel profile. See *Formatted Character String Output*, p.29.

Basic Kernel Profile Components and Libraries

In addition to the components and libraries provided by the minimal kernel profile (listed in Table 2-2), the basic kernel profile provides those listed in Table 2-3.

Table 2-3 **Basic Kernel Profile Components—in Addition to Minimal Kernel Profile**

Component	Library	Description
INCLUDE_ANSI_STRDUP	**ansiString**	ANSI **strdup( )** routine.
INCLUDE_TASK_CREATE_DELETE	**taskLib**	Support for **taskSpawn( )**, **taskCreate( )**, **taskDelete( )** and **exit( )**.
INCLUDE_TASK_RESTART	**taskLib**	Support for **taskRestart( )**.
INCLUDE_EXC_TASK	**excLib**	Support for **excJobAdd( )**.
INCLUDE_ISR_OBJECTS	**isrLib**	Interrupt service routine objects library.
INCLUDE_MSG_Q	**msgQLib**	Message queue support with **msgQInitialize( )**, **msgQReceive( )**, **msgQSend( )** and so on.
INCLUDE_MSG_Q_CREATE_DELETE	**msgQLib**	Message queue creation and deletion support with **msgQCreate( )** and **msgQDelete( )**.
INCLUDE_MSG_Q_INFO		Support for **msgQInfoGet( )**.
INCLUDE_SEM_DELETE	**semLib**	Support for semaphore deletion with **semDelete( )**.
INCLUDE_SEM_BINARY_CREATE	**semBLib**	Support for **semBCreate( )**.
INCLUDE_SEM_COUNTING_CREATE	**semCLib**	Support for **semCCreate( )**.
INCLUDE_SEM_MUTEX_CREATE	**semMLib**	Support for **semMCreate( )**.
INCLUDE_SEM_INFO		Support for **semInfo( )**.
INCLUDE_TASK_INFO		Support for **taskInfoGet( )**.
INCLUDE_TASK_HOOKS	**taskHookLib**	Support for adding/removing hook routines at task creation, deletion and task switches.
INCLUDE_WATCHDOGS_CREATE_DELETE	**wdLib**	Support for **wdCreate( )** and **wdDelete( )**.

Table 2-3 **Basic Kernel Profile Components—in Addition to Minimal Kernel Profile**

Component	Library	Description
INCLUDE_MEM_MGR_BASIC	**memPartLib**	Memory partition manager; **malloc( )**, **free( ) realloc( )**.
INCLUDE_HASH	**hashLib**	Hash table management library.
INCLUDE_LSTLIB	**lstLib**	Doubly linked list subroutine library.
INCLUDE_RNG_BUF	**rngLib**	Ring buffer management library.
INCLUDE_POOL	**poolLib**	Memory pool management library.
INCLUDE_IO_BASIC		Basic I/O functionality.

Basic OS Profile

The basic OS profile (**PROFILE_BASIC_OS**) builds upon the basic kernel profile to offer a relatively simple real-time operating system. It does not, for example, provide support for networking or real-time processes (RTPs). This configuration is similar to a VxWorks 5.5 configuration, but without the network stack and debugging assistance tools (the WDB target agent). The profile provides an operating system instead of simply a kernel. The new capabilities added in this profile are the following:

- Full I/O system, which includes file system and POSIX support.
- Standard I/O file descriptors and associated API support.
- APIs for directory and path manipulations, and disk utilities.
- Support for **select( )**.
- TTY and pipe driver support.
- Support for logging (**logLib**).
- Support for task and environment variables (**envLib**, **taskVarLib**).
- Support for coprocessor management (**coprocLib**) and floating point.
- Full-featured memory partition manager (**memLib**).
- Full ANSI library support. Adds support for **assert( )**, **setjmp( )** and **longjmp( )**, **stdio**, **stdlib**, and **time** library routines.

Device Drivers for Basic OS Systems

Device drivers for the basic OS can now use additional IO system features, associated capabilities like **select( )**, and so on. File descriptor based I/O and associated APIs are available. Another major addition is coprocessor support, which typically provides support for hardware floating point operations. Vector operations (for some PowerPC BSPs) are also available with the coprocessor support infrastructure (**coprocLib**). More advanced ANSI routines are available, that use the standard I/O system, ANSI time facilities and mathematical routines.

Formatted Character String Output

Full ANSI formatted I/O routines are available starting with the basic OS profile. Formatted output routines like **printf( )**, **sprintf( )**, and so on can handle floating point or vector types if applicable. Formatted input routines such as **scanf( )** and so on are also available. These routines send their output to the standard I/O file descriptors as expected. These capabilities are available with the **INCLUDE_FORMATTED_IO** component.

Basic OS Profile Components and Libraries

In addition to the components and libraries provided by the minimal kernel profile and basic kernel profile (listed in Table 2-2 and Table 2-3), the basic OS profile provides those listed in Table 2-4.

Table 2-4 **Basic OS Profile Components—in Addition to the Minimal and Basic Kernel Profiles**

Component	Library	Description
INCLUDE_ANSI_ASSERT	**ansiAssert**	ANSI **assert( )** routine.
INCLUDE_ANSI_LOCALE	**ansiLocale**	ANSI locale routines **localeconv( )** and **setlocale( )**.
INCLUDE_ANSI_LONGJMP	**ansiSetjmp**	ANSI **setjmp( )** and **longjmp( )** routines.
INCLUDE_ANSI_MATH	**ansiMath**	ANSI math routines.
INCLUDE_ANSI_STDIO	**ansiStdio**	ANSI stdio routines.
INCLUDE_ANSI_STDLIB	**ansiStdlib**	ANSI stdlib routines.
INCLUDE_ANSI_ABORT	**ansiStdlib**	ANSI **abort( )** routine.
INCLUDE_ANSI_TIME	**ansiTime**	ANSI time routines
INCLUDE_ANSI_STRERROR	**ansiStdio**	ANSI **strerror( )** routine.
INCLUDE_POSIX_CLOCKS	**clockLib**	POSIX clock library support for **clock_getres( )**, **clock_setres( )**, **clock_gettime( )** and **clock_settime( )**.
INCLUDE_ENV_VARS	**envLib**	Environment variable library; **getenv( )**, **setenv( )** and so on.
INCLUDE_TASK_VARS	**taskVarLib**	Task variables support library; **taskVarAdd( )**, **taskVarDelete( )**,**taskVarGet( )**,**taskVarSet( )** and so on.
INCLUDE_SIGNALS	**sigLib**	Software signal library; support for **signal( )**, **kill( )** and so on.
INCLUDE_EDR_PM		Error detection and reporting persistent memory region manager.
INCLUDE_TTY_DEV	**ttyLib**	TTY device driver.
INCLUDE_FLOATING_POINT	**floatLib**	Floating point scanning and formatting library.
INCLUDE_FORMATTED_IO	**fioLib**	Full formatted I/O support; **printf( )**, **scanf( )**, and variants.

Table 2-4 **Basic OS Profile Components—in Addition to the Minimal and Basic Kernel Profiles**

Component	Library	Description
INCLUDE_POSIX_FS	**fsPxLib**	POSIX APIs for file systems.
INCLUDE_IO_SYSTEM	**ioLib, iosLib dirLib, pathLib**	I/O system and associated interfaces. Directory and path manipulation API's. For more information, see *7.2 Configuring VxWorks With I/O Facilities*, p.362).
INCLUDE_LOGGING	**logLib**	Message logging support; **logMsg( )**, **logFdSet( )** and so on.
INCLUDE_MEM_MGR_FULL	**memLib**	Support for **calloc( )**, **valloc( )**, **realloc( )** and so on.
INCLUDE_PIPES	**pipeDrv**	Pipe device support.
INCLUDE_TYLIB	**tyLib**	TTY driver support library.
INCLUDE_ROMFS		ROMFS (read-only memory based file system).
INCLUDE_SELECT	**selectLib**	Support for **select( )** and associated API's.
INCLUDE_STDIO	**stdioLib**	Support for **stdioFp( )**.

2.4.6 Customizing VxWorks Code

VxWorks operating system code can itself be customized. This section introduces customization of **usrClock( )**, hardware initialization, and more general features of the operating system.

System Clock Modification

During system initialization at boot time, the system clock ISR—**usrClock( )**—is attached to the system clock timer interrupt. For every system clock interrupt, **usrClock( )** is called to update the system tick counter and to run the scheduler.

You can add application-specific processing to **usrClock( )**. However, you should keep in mind that this is executed in interrupt context, so only limited functions can be safely called. See *4.20.5 Special Limitations of ISRs*, p.245 for a list of routines that can be safely used in interrupt context.

Long power management, if used, allows the processor to sleep for multiple ticks. See *2.5 Power Management*, p.40. The **usrClock( )** routine, and therefore **tickAnnounce( )**, is not called while the processor is sleeping. Instead, **usrClock( )** is called only once, after the processor wakes, if at least one tick has expired. Application code in **usrClock( )** must verify the tick counter each time it is called, to avoid losing time due to setting the processor into sleep mode.

Hardware Initialization Customization

When the application requires custom hardware, or when the application requires custom initialization of the existing hardware, the BSP must be modified to perform the initialization as required. What BSP modifications are required depend on the type of hardware and the type of initialization that must be performed. For information about adding or customizing device drivers, the *VxWorks Device Driver Developer's Guide*, specifically the introductory sections. For information about custom modifications to BSP code, see the *VxWorks BSP Developer's Guide*.

Other Customization

The directory *installDir***/vxworks-6.***x***/target/src/usr** contains the source code for certain portions of VxWorks that you may wish to customize. For example,

usrLib.c is a common place to add target-resident routines that provide application-specific development aids.

If you modify one of these files, an extra step is necessary before rebuilding your VxWorks image: you must replace the modified object code in the appropriate VxWorks archive. The makefile in *installDir***/vxworks-6.***x***/target/src/usr** automates the details. This directory is not specific to a single architecture, however, so you must specify the value of the **CPU** variable with **make** on the command line, as follows:

```
    c:\installDir\vxworks-6.x\target\src\usr> make CPU=cputype TOOL=tool
```

This step recompiles all modified files in the directory, and replaces the corresponding object code in the appropriate architecture-dependent directory. After that, the next time you rebuild VxWorks, the resulting system image includes your modified code.

The following example illustrates replacing **usrLib** with a modified version, rebuilding the archives, and then rebuilding the VxWorks system image. For the sake of conciseness, the **make** output is not shown. The example assumes the **pcPentium** BSP; replace the BSP directory name and **CPU** value as appropriate for your environment.

```
c:\> cd installDir\vxworks-6.1\target\src\usr
c:\installDir\vxworks-6.1\target\src\usr> copy usrLib.c usrLib.c.orig
c:\installDir\vxworks-6.1\target\src\usr> copy develDir\usrLib.c usrLib.c
c:\installDir\vxworks-6.1\target\src\usr> make CPU=PENTIUM TOOL=diab
...
c:\installDir\vxworks-6.1\target\src\usr> cd nstallDir\vxworks-6.1\target\config\pcPentium
c:\installDir\vxworks-6.1\target\config\pcPentium2> make
...
```

2.5 Power Management

Starting with the VxWorks 6.2 release, enhanced power management facilities are provided for the Intel Architecture (IA). Facilities provided in earlier releases for other architectures remain the same. The new facilities will be provided for the other architectures in future releases. See *2.5.1 Power Management for IA Architecture*, p.41 and *2.5.2 Power Management for Other Architectures*, p.49.

2.5.1 Power Management for IA Architecture

VxWorks power management facilities provide for managing the power and performance states of the CPU. These facilities can be used to control CPU power use based on the following:

- CPU utilization
- CPU temperature thresholds
- task and ISR-specific performance states

The VxWorks power management facilities utilize key concepts of the Advanced Configuration and Power Interface (ACPI) Specification, version 3.0. The ACPI specification has not been implemented for VxWorks because of its unsuitability for hard real-time systems and for all the architectures supported by VxWorks. However, the ACPI specification provides useful definitions of power states and power-state transitions, as well as of thermal management, and these definitions have been incorporated into the design of the VxWorks power management facilities.

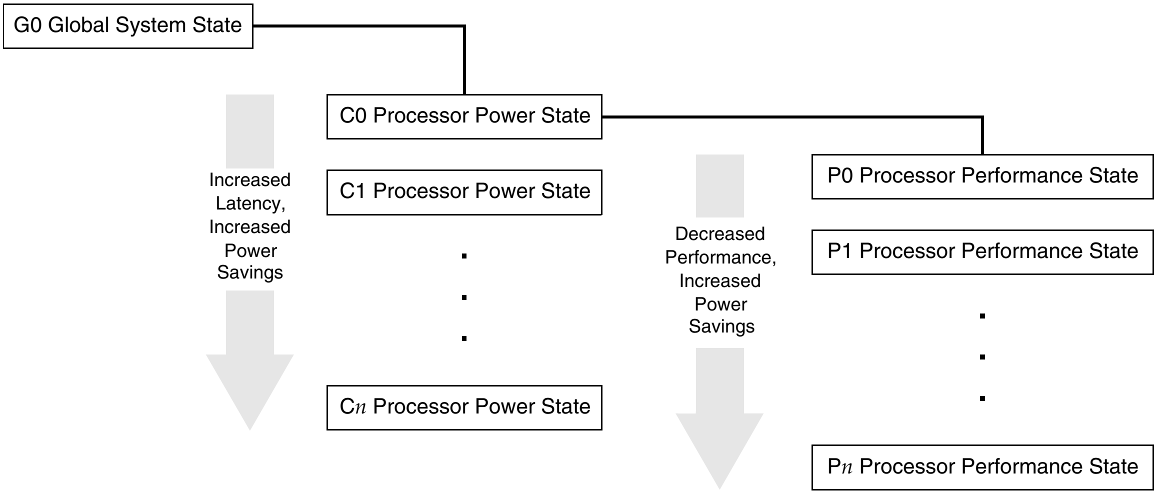
ACPI Processor Power and Performance States

The ACPI 3.0 specification defines processor power states as well as the transitions that take the processor from one state to the other. Essentially it defines the processor power state machine. This aspect of the specification enables the mapping of the power management features of a CPU to one of the defined states, whether the CPU is ACPI compliant or not. For example, ACPI defines that in power state C1, the processor does not execute instructions but the contents of the caches remain valid (bus snooping still takes place). Many CPUs support such a power state, but manufacturers often use different names to identify that state.

In addition to defining processor power states, ACPI defines performance states where the CPU executes instructions, but not necessarily at its maximum throughput. These states correspond to voltage and or frequency scaling capabilities present on some CPUs, and provide a power management scheme with which power can be managed even if the system is not idle.

Figure 2-2 illustrates the ACPI-defined power states that apply to the CPU power management for VxWorks.

Figure 2-2 **Fig 2-2: ACPI Power States**


G0 Global System State
C0 Processor Power State
C1 Processor Power State
C*n* Processor Power State
Increased Latency, Increased Power Savings
P0 Processor Performance State
P1 Processor Performance State
P*n* Processor Performance State
Decreased Performance, Increased Power Savings

The G0 global system state is also known as the working state. ACPI requires that all processor power states reside under the G0 state, which is why other G states are not deemed relevant to this feature. The C0 to C*n* states are processor power states. Key features of these processor power are as follows:

- In the C0 state the processor is fetching and executing instructions
- In the C1 to C*n* states the processor is not executing instructions.
- The higher the power state number, the greater the power saving, but at the cost greater latency in reaction to external events.
- State transitions occur to and from the C0 state. For example, after going from the C0 state to the C1 state the processor must transition back to the C0 state before going to the C2 state. This is because transitions are triggered by software and only the C0 state is allowed to execute instructions.

Under the C0 power state reside the processor performance states. In each of these states the CPU is executing instructions but the performance and power savings in each P-state vary. The higher the performance state number, the greater the power saving, but the slower the rate of instruction execution. Taking the Speedstep technology of the Pentium processor as an example, it defines various voltage-frequency power settings that are mapped to the ACPI-defined P-states. Note that unlike the C-state transitions, P-state transitions can occur between any two states.

See the *VxWorks Architecture Supplement* for information about which states are supported

ACPI Thermal Management

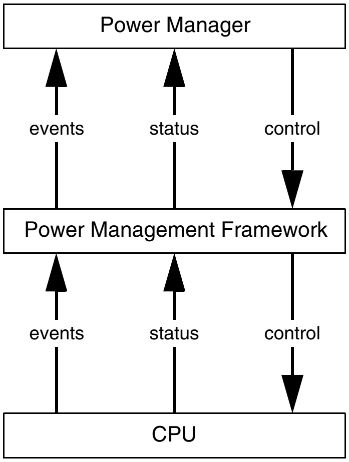
Much like the processor power management concepts, the thermal management concepts defined in ACPI 3.0 are applicable to non ACPI-compliant hardware. Some of these concepts are:

- Differentiation between active cooling and passive cooling. Actively cooling a system consists of activating a cooling device such as a fan. Passive cooling is achieved by reducing the power consumed by a device. In this case *device* includes processors and therefore this is relevant to this feature.
- Critical Shut Down. It is the temperature threshold at which a device or system is shut down so as to protect it from heat induced damages.
- Notification of temperature changes. This allows a power management entity to actively or passively manage the temperature of a system or device without the need to poll the devices to obtain their operating temperature.
- Processor Throttling. This is the link between thermal management and the processor power states. ACPI equations define how to manage the performance states of a processor so as to attempt to keep it inside a temperature zone.

VxWorks Power Management Facilities

The architecture of the VxWorks power management facilities is composed of two basic elements: a power management framework and a power manager. The power management framework is effectively a layer between the power manager and the CPU. It transfers status information from the CPU *to* the power manager, and executes control of the CPU based on instructions *from* the power manager. Figure 2-3 illustrates this relationship.

Figure 2-3 **Fig 2-3: Power Management Architecture**



The power management framework is designed to accommodate two use cases for controlling power consumption: one minimizes the power level of the CPU based on how much work it has to do (and its temperature); the other runs tasks and ISRs at different performance states based on their priority.

Wind River provides the power management framework and a two power managers—only one of which can be used at a time (see *Wind River Power Managers*, p.47).

You can develop your own power manager for either of the two use cases supported by the power management framework (see *Power Management Framework and Use Cases*, p.44). The APIs provided by the power management framework give you the control mechanisms for a power manager (see *Power Management Framework API*, p.48).

Power Management Framework and Use Cases

The VxWorks power management framework is designed to serve two use cases: one bases the control of power consumption on how much work must be done; and the other bases the control of power consumption on task-specific performance states.

One use case involves controlling the power consumed by the CPU based on how much work the CPU has to do. The idea is to keep the power level of the CPU as low as possible while preventing the system from going into overload. That is, prevent running at 100% CPU utilization for a user-defined period of time. It is quite clear that the writers of the ACPI specification had this use case in mind while writing the document.

The second use case is based on the premise that power can be controlled by having tasks and ISRs execute at different performance states (P-states). For example, a task that performs work queued up by interrupts may need to run at the P0 performance state (highest performance) while a maintenance task with no hard deadlines can afford to run at the lowest performance state.

The first use case is more global in nature in that the entire system is running in a certain power state. It is also a scenario in which the power consumption is dynamically adapted to the work that is required of the processor. One of the drawback of the approach however is that it makes it difficult to guarantee deadlines can be met, as a piece of code is not guaranteed to run in the same performance state on every invocation. The second use case provides a finer granularity of control and can be more deterministic, since each task can be set to run in the same performance state at all times. This comes at the price of increased context switching and interrupt handling times.

NOTE: While the power management framework allows for various power management methods to be used, the power manager itself must be designed to ensure that it uses the capacities of the framework in a coherent manner. For example, the framework cannot prevent contention for the CPU if both of the use cases described above are implemented at the same time. For the same reason, only one power manager should be included in the configuration of VxWorks (the two power managers provided by Wind River are mutually exclusive of one another).

CPU Utilization Based Power Management

A CPU utilization based power manager is one that uses CPU utilization and CPU temperature to control the power consumption of the CPU. There are really two aspects to this approach. One is to transition the CPU from the C0 power state (executing state) to one of the other C-states (non-executing states) when the VxWorks kernel becomes idle. The other aspect is to control the performance state (P-state) of the CPU so as to keep it inside a specified range of CPU utilization and, optionally, inside a temperature range. In order to support a power manager using this approach, the power management framework has the following features:

- The framework notifies the power manager when the VxWorks kernel goes idle.

- The framework notifies the power manager when the VxWorks kernel comes out of idle state.

- The framework allows the power manager to transition the CPU from the C0 state to any of the non-executing power states: C1, C2, ...C*n*. Note that the transition back to the C0 state occurs when an external event takes place (that is, an interrupt) and therefore this is not a state transition the framework can allow the power manager to perform/control.
- The framework allows the power manager to transition the CPU between performance states (P-states) based on the CPU utilization over a user-defined time interval. This is achieved by the framework keeping track of CPU utilization and reporting that figure to the power manager.
- The framework computes the CPU utilization over two user-specified time intervals. Having two intervals makes it easier for the power manager to implement a quick ramp up, slow ramp down policy through the performance states. The sampling intervals can be modified dynamically.
- The framework notifies the power manager when a CPU-utilization interval has elapsed and provides the CPU utilization figure to the power manager at that time.
- The framework allows the power manager to specify a high and a low temperature threshold for the purpose of being notified whenever the temperature of the CPU crosses either threshold. These thresholds can be modified dynamically. The purpose for these is to allow the power manager to implement a cooling policy such as reducing the CPU performance state to lower power consumption, hence lowering temperature.

The full-featured CPU utilization power manager provided by Wind River is an example of this type of power management. See *Wind River Power Managers*, p.47.

Task Performance-Based Power Management

The per-task performance power manager is based on the premise that power can be controlled by having tasks execute at different performance states (P-states). For example, a task that performs work queued up by interrupts may need to run at the P0 performance state (highest performance) while a maintenance task with no hard deadlines can afford to run at the lowest performance state. In order to support a power manager using this approach, the power management framework has the following features:

- The framework allows a performance state (P-state) to be assigned to each task and allows that state to be set during context switches.
- The framework allows a single performance state to be assigned for all interrupts in the system so that execution of ISRs can be performed in a performance state other than the one of the interrupted task.

Wind River Power Managers

Wind River provides two power managers that implement CPU-utilization-based power management:

- A full-featured CPU-utilization-based power manager. It manages the C-state of the CPU when VxWorks goes idle, as well as managing the P-state of the CPU based on CPU utilization. See *CPU Utilization Based Power Management*, p.45.
- A light version of a CPU-utilization-based power manager, which simply manages the C-state of the CPU when VxWorks goes idle. It implements the same power management algorithm that was provided for VxWorks 6.0 and 6.1; and is included in the default configuration of VxWorks configuration so that the power management behavior of the operating system is the same as in versions prior to 6.2. See *2.5.2 Power Management for Other Architectures*, p.49 for more information about its features.

The components used to configure VxWorks with these power managers are listed in *Configuring VxWorks With Power Management Facilities*, p.49.

Power Management Framework API

Table 2-5 describes the API provided by the power management framework. Power managers use this API to plug into the framework. The routines are available only in the kernel.

Table 2-5 **Power Management Framework API**

Routine	Description
cpuPwrCStateSet( )	Sets the CPU in a specified power state (C-state).
cpuPwrPStateSet( )	Sets the CPU in a specified performance state (P-state).
cpuPwrPStateGet( )	Returns the performance (P-state) state of the CPU.
cpuPwrTaskPStateSet( )	Sets the performance state (P-state) of a task.
cpuPwrTaskPStateGet( )	Gets the performance state (P-state) of a task.
cpuPwrTempThreshSet( )	Sets the CPU temperature thresholds for the CPU (high and low).
cpuPwrTempThreshGet( )	Returns the temperature thresholds for the CPU.
cpuPwrUtilPeriodSet( )	Sets the two time intervals over which CPU utilization is computed.
cpuPwrEventHandlersSet( )	Registers a set of handlers for the various power management events.

For more information about the routines and the power states supported with VxWorks, see the API reference for **cpuPwrLib**.

Also see the *VxWorks Architecture Supplement* for the mappings between the ACPI specification C-states and P-states and the power modes supported by the CPU in question.

Configuring VxWorks With Power Management Facilities

Configure VxWorks with the **INCLUDE_CPU_PWR_MGMT** component for the power management framework.

Use one of the power manager components provided by Wind River:

- **INCLUDE_CPU_UTIL_PWR_MGR**, which is full-featured CPU-utilization-based power manager.
- **INCLUDE_CPU_LIGHT_PWR_MGR**, which is a light version of a CPU-utilization-based power manager.

For full descriptions of these power managers, see *Wind River Power Managers*, p.47.

A custom power manager can be used in place of a Wind River power manager. It can be included as an application module or a custom component. For more information, see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64 and *2.8 Custom VxWorks Components and CDFs*, p.67.

Power Management and System Performance

Performance is a concern for the CPU power management framework because it interfaces to the interrupt handling and scheduling sub-systems in a way that affects the interrupt latency and the task context switch time.

2.5.2 Power Management for Other Architectures

NOTE: For this release, the power management facilities available for architectures other than IA are the same as provided for VxWorks 6.0 and 6.1. These facilities are described in this section.

The features described in *2.5.1 Power Management for IA Architecture*, p.41 will be available for other architectures in future releases.

Power management allows the processor to conserve power by entering a low power state. While in this mode, processor register values and memory contents are retained. This feature is implemented by putting the CPU in a non-executing state while the kernel is idle. This has no impact on the operation of peripheral devices except for the system timer when long sleep mode (described below) is selected. The VxWorks power management facilities provide two modes of operation:

- **Short Sleep Mode**

  In short sleep mode, the CPU is put in a low power state until the next interrupt occurs, including the system clock interrupt. The maximum amount of time the CPU can sleep is therefore the interval in between system clock interrupts. The short sleep mode does not require special BSP support. It is provided at the architecture level. See the *VxWorks Architecture Supplement* to determine if this mode is supported for your CPU.

- **Long Sleep Mode**

  In long sleep mode, the CPU is put in a low power state until the next interrupt occurs, excluding the system clock interrupt. This allows the CPU to sleep until the next system event is scheduled to occur; such as a task that times out on a semaphore take operation or a watchdog that fires. This mode requires BSP support because the system clock interrupt source must be turned off, and a mechanism must schedule the CPU to wake after a specified amount of time.

To provide power management support for your system, configure VxWorks with the **INCLUDE_POWER_MGMT_BSP_SUPPORT** component.

For more information, see the *VxWorks BSP Developer's Guide*. Also see *System Clock Modification*, p.39.

2.6 Kernel Applications

VxWorks applications that execute in the kernel are created as relocatable object modules. They can be referred to most specifically as *kernel-based application modules*, but it is often convenient to refer to them simply as *kernel application modules* or *kernel applications*. In any case, they should not be confused with process-based (RTP) applications.

When a kernel-based application module is built, user code is linked to the required VxWorks libraries, and an ELF binary is produced. Kernel applications use VxWorks facilities by including header files that define operating system interfaces and data structures.

Kernel application modules can be either:

- Downloaded and dynamically linked to the operating system by the object module loader.
- Statically linked to the operating system, making them part of the system image.

Downloading kernel modules is useful for rapid development and debugging, as the operating system image does not need to be rebuilt for each iteration of the application. This method can also be used for diagnostic facilities with production systems. Various development tools, including the debugger and the shell (host or kernel), can be used to download and manage modules. Modules can be downloaded to a target from any host file system for which the kernel has support (NFS, ftp, and so on).

Kernel application modules can also be stored on the target itself in flash or ROM, in the ROMFS file system, or on disk. Once they have been loaded into the target, kernel application modules can be started interactively from the shell or Workbench.

Application modules that are statically linked to the operating system can be run interactively from the shell or Workbench. VxWorks can also be configured to start them automatically at boot time. Static linking and automatic startup are obviously suitable for production systems.

An application that runs in kernel space is not executed as a process; it is simply another set of tasks running in kernel space. The kernel is not protected from any misbehavior that a kernel application might engage in—and the applications are similarly not protected from each other—kernel applications and the kernel run in the same address space in supervisor mode.

Note that VxWorks can also be configured with support for applications that execute in user space as processes. See *VxWorks Application Programmer's Guide: Applications and Processes*.

WARNING: If you wish to port a kernel application to a user-mode application that executes as a real-time process (RTP), you must ensure that it meets the code requirements of an RTP application and is compiled as such. You must also ensure that VxWorks is configured with support for RTPs. For more information, see the *VxWorks Application Programmer's Guide: Applications and Processes*, and the *VxWorks Migration Guide*.

2.6.1 C and C++ Libraries

Both VxWorks native C libraries, and Dinkum C and C++ libraries, are provided for VxWorks application development. As shown in Table 2-6, VxWorks native libraries are used for C kernel application development, and Dinkum libraries are used in all other cases.

Table 2-6 **C and C++ Libraries**

Type of Application	C Language	C++ Language
Kernel-mode application	VxWorks native libraries	Dinkum C++ and embedded (abridged) C++ libraries
User-mode RTP application	Dinkum C libraries	Dinkum C++ and embedded (abridged) C++ libraries

The VxWorks native C libraries provide routines outside the ANSI specification. Note that they provide no support for wide or multi-byte characters.

For more information about these libraries, see the VxWorks and Dinkum API references. For more information about C++ facilities, see *13. C++ Development*.

2.6.2 Application Structure

Kernel application code is similar to common C or C++ applications, with the exception that it does not require a traditional **main( )** routine (unlike a VxWorks process-based application). It simply requires an entry point routine that starts all the tasks required to get the application running.

NOTE: If your kernel application includes a **main( )** routine, do not assume that it will start automatically. Kernel application modules that are downloaded or simply stored in the system image must be started interactively (or be started by another application that is already running). The operating system can also be configured to start applications automatically at boot time (see a*2.6.10 Configuring VxWorks to Run Applications Automatically*, p.66).

The entry-point routine performs any data initialization that is required, and starts all the tasks that the running application uses. For example, a kernel application might have a routine named like **myAppStartUp( )**, which could look something like this:

```
void myAppStartUp (void)
    {
    runFoo();
    tidThis = taskSpawn("tThis", 200, 0, STACK_SIZE,
            (FUNCPTR) thisRoutine,0,0,0,0,0,0,0,0,0,0);
    tidThat = taskSpawn("tThat", 220, 0, STACK_SIZE,
        (FUNCPTR) thatRoutine,0,0,0,0,0,0,0,0,0,0);
    tidAnother = taskSpawn("tAnother", 230, 0, STACK_SIZE,
            (FUNCPTR) anotherRoutine,0,0,0,0,0,0,0,0,0,0);
    return (OK);
    }
```

For information about VxWorks tasks and multitasking, see *4. Multitasking*. For information about working with C++ see *13. C++ Development*.

2.6.3 VxWorks Header Files

Many kernel applications make use of VxWorks operating system facilities or utility libraries. This usually requires that the source module refer to VxWorks header files. The following sections discuss the use of VxWorks header files.

VxWorks header files supply ANSI C function prototype declarations for all global VxWorks routines. VxWorks provides all header files specified by the ANSI X3.159-1989 standard.

VxWorks system header files are in the directory *installDir***/vxworks-6.***x***/target/h** and its subdirectories.

VxWorks Header File: vxWorks.h

The header file **vxWorks.h** *must* be included first by every kernel application module that uses VxWorks facilities. It contains many basic definitions and types that are used extensively by other VxWorks modules. Many other VxWorks header files require these definitions. Include **vxWorks.h** with the following line:

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

Other VxWorks Header Files

Kernel applications can include other VxWorks header files, as needed, to access VxWorks facilities. For example, a module that uses the VxWorks linked-list subroutine library must include the **lstLib.h** file with the following line:

```
#include <lstLib.h>
```

The API reference entry for each library lists all header files necessary to use that library.

ANSI Header Files

All ANSI-specified header files are included in VxWorks. Those that are compiler-independent or more VxWorks-specific are provided in *installDir*/**vxworks-6.***x***/target/h** while a few that are compiler-dependent (for example **stddef.h** and **stdarg.h**) are provided by the compiler installation. Each toolchain knows how to find its own internal headers; no special compile flags are needed.

ANSI C++ Header Files

Each compiler has its own C++ libraries and C++ headers (such as **iostream** and **new**). The C++ headers are located in the compiler installation directory rather than in *installDir*/**vxworks-6.***x***/target/h**. No special flags are required to enable the compilers to find these headers. For more information about C++ development, see *13. C++ Development*.

NOTE: In releases prior to VxWorks 5.5, Wind River recommended the use of the flag -**nostdinc**. This flag *should not* be used with the current release since it prevents the compilers from finding headers such as **stddef.h**.

The -I Compiler Flag

By default, the compiler searches for header files first in the directory of the source module and then in its internal subdirectories. In general, *installDir*/**vxworks-6.***x***/target/h** should always be searched before the compilers' other internal subdirectories; to ensure this, always use the following flag for compiling under VxWorks:

```
-I %WIND_BASE%/target/h %WIND_BASE%/target/h/wrn/coreip
```

Some header files are located in subdirectories. To refer to header files in these subdirectories, be sure to specify the subdirectory name in the include statement, so that the files can be located with a single **-I** specifier. For example:

```
#include <xWorks.h>
#include <sys/stat.h>
```

VxWorks Nested Header Files

Some VxWorks facilities make use of other, lower-level VxWorks facilities. For example, the *tty* management facility uses the ring buffer subroutine library. The *tty* header file **tyLib.h** uses definitions that are supplied by the ring buffer header file **rngLib.h**.

It would be inconvenient to require you to be aware of such include-file interdependencies and ordering. Instead, all VxWorks header files explicitly include all prerequisite header files. Thus, **tyLib.h** itself contains an include of **rngLib.h**. (The one exception is the basic VxWorks header file **vxWorks.h**, which all other header files assume is already included.)

Generally, explicit inclusion of prerequisite header files can pose a problem: a header file could get included more than once and generate fatal compilation errors (because the C preprocessor regards duplicate definitions as potential sources of conflict). However, all VxWorks header files contain conditional compilation statements and definitions that ensure that their text is included only once, no matter how many times they are specified by include statements. Thus, a kernel application module can include just those header files it needs directly, without regard to interdependencies or ordering, and no conflicts will arise.

VxWorks Private Header Files

Some elements of VxWorks are internal details that may change and so should not be referenced in a kernel application. The only supported uses of a module's facilities are through the public definitions in the header file, and through the module's subroutine interfaces. Your adherence ensures that your application code is not affected by internal changes in the implementation of a VxWorks module.

Some header files mark internal details using **HIDDEN** comments:

```
/* HIDDEN */
...
/* END HIDDEN */
```

Internal details are also hidden with *private* header files: files that are stored in the directory *installDir*/**vxworks-6.***x***/target/h/private**. The naming conventions for these files parallel those in *installDir*/**vxworks-6.***x***/target/h** with the library name followed by **P.h**. For example, the private header file for **semLib** is *installDir*/**vxworks-6.***x***/target/h/private/semLibP.h**.

2.6.4 Static Instantiation of Kernel Objects

The VxWorks APIs have a long established convention for the creation and deletion of kernel entities. Objects such as tasks, semaphores, message queues and watchdogs are instantiated using their respective creation APIs (for example, **taskSpawn( )**, **sem***X***Create( )**, and so on) and deleted using their respective delete APIs (for example, **msgQDelete( )**, **wdDelete( )**, and so on.). Object creation is a two-step process: first the memory for the object is allocated from the system, which is then initialized appropriately before the object is considered usable. Object deletion involves invalidation of the object, followed by freeing its memory back to the system. Thus, object creation and deletion are dependent on dynamic memory allocation, usually through the **malloc( )** and **free( )** routines.

Dynamic creation and deletion of objects at run-time is a convenient programming paradigm, though it has certain disadvantages for some real-time critical applications. First, the allocation of memory from the system cannot always be guaranteed. Should the system run out of memory the application cannot create the resources it must have function. The application must then resort to a suitable error recovery process if any exists, or abort in some fashion. Second, dynamic allocation of memory is a relatively slow operation that may potentially block the calling task. This makes dynamic allocation non-deterministic in performance.

Static instantiation of objects is a faster, more deterministic alternative to dynamic creation. In static instantiation, the object is declared as a compile time variable. Thus the compiler allocates storage for the object in the program being compiled. No more allocation need be done. At run-time the objects memory is available immediately at startup for the initialization step. Initialization of pre-allocated memory is much more deterministic and faster than dynamic creation. Such static declaration of objects cannot fail, unless the program itself is too large to fit in the systems memory.

Many applications are suited to exploit static instantiation of objects in varying degrees. Most applications require some resources to be created, that last for the lifetime of the application. These resources are never deleted. In lieu of the latter, objects that last for the lifetime of the application are ideally suited for static (that is, compile time) allocation. To the extent that they are instantiated statically (which we shall see below), the application is that much more fail safe and fast to launch.

See *2.4.5 Small VxWorks Configuration Profiles*, p.24 for information about operating system profiles of particular relevance for static instantiation.

Dynamic Instantiation of an Object

```
struct my_object * pMyObj;
    ...
pMyObj = (struct my_object *) malloc (sizeof (struct my_object));
if (pMyObj != NULL)
    {
    objectInit (pMyOjb);
    return (OK);
    }
else
    {
    /* failure path */
    return (ERROR);
    }
```

Static Instantiation of an Object

```
struct my_object myObj;
...
objectInit (&myOjb);
/* myObj now ready for use */
```

Static instantiation of objects has the following advantages:

- The application logic is made simpler by not having to consider the case when dynamic allocation fails.
- Compile time declaration of objects does not take up space in the executable file or flash memory. If an object is merely declared at compile time but not initialized, it is placed by the compiler in the un-initialized data section (also known as the bss section). Un-initialized data is required by the ANSI C standard to be of value zero. Hence the un-initialized data section (the bss section) of a program does not occupy any space in an executable file or in VxWorks ROM images. Un-initialized data does contribute to the programs run-time footprint in memory, but so does dynamic allocation. The program will not consume any more memory footprint than it did with dynamic allocation of objects.

Using static instantiation whenever possible is more robust, deterministic and fast. Static instantiation of objects is therefore much better suited for real-time applications. On the other hand some applications are inherently dynamic in nature. For these, dynamic creation and deletion is always available.

Static Instantiation of Objects

Kernel objects such as tasks, semaphores, message queues and watchdogs can be instantiated statically using the same principles outlined above.

Normally these objects are created using the appropriate create routines for that type of object, and deleted using the appropriate delete routine. As mentioned before, creation and deletion involve dynamic memory allocation and free respectively.

Static instantiation of objects is a two-step process. First the object to be created is declared, usually at global scope. Next the declared object is initialized using an initialization routine, which makes it ready for use. In contrast, dynamic creation with create routines is a one-step process. Static instantiation of objects is thus a little less convenient, but more deterministic. Users can choose the style that suits their purpose.

NOTE: Static instantiation should only be used for objects that are kernel-resident. It is not meant to be used to create objects in a process (RTP).

The following sections describe an alternative static instantiation method for each of these entities.

Scope Of Static Declarations

The macros declaring kernel objects (that is **VX_BINARY_SEMAPHORE**, **VX_TASK**, and so on) are usually declared as global variables. Since all these kernel objects are used for inter-task communication and synchronization, their IDs are the means by which other tasks use these objects. Hence global objects and global IDs are the common method by which these objects are accessed and used. However it is not always necessary that they be global. An object declaration can also be done at function scope provided the object stays in a valid scope for the duration of its use.

Static Instantiation of Tasks

The **taskSpawn( )** routine has been the standard method for instantiating tasks. This API relies on dynamic allocations. In order to instantiate tasks statically several macros have been provided to emulate the dynamic instantiation capability provided by **taskSpawn( )** and related routines.

The **VX_TASK** macro declares a task object at compilation time. It takes two arguments: the task name and its stack size. When calling **taskSpawn( )** the name may be a **NULL** pointer, but when using the **VX_TASK** macro, a name is mandatory. The stack size must evaluate to a non-zero integer value and must be a compile-time constant.

The **VX_TASK_INSTANTIATE** macro is the static equivalent of the **taskSpawn( )** routine. It initializes and schedules the task, making it run according to its priority. **VX_TASK_INSTANTIATE** evaluates to the task ID of the spawned task if it was successful, or **ERROR** if not.

The following example illustrate spawning tasks statically:

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

VX_TASK(myTask,4096);
int myTaskId;

STATUS initializeFunction (void)
    {
    myTaskId = VX_TASK_INSTANTIATE(myTask, 100, 0, 4096, pEntry, \
                               0,1,2,3,4,5,6,7,8,9);

   if (myTaskId != ERROR)
       return (OK);               /* instantiation succeeded */
   else
       return (ERROR);
    }
```

Sometimes users may prefer to initialize a task, but keep it suspended until needed later. This can be achieved by using the **VX_TASK_INITIALIZE** macro, as illustrated below. Since the task is left suspended, users are responsible for calling **taskActivate( )** in order to run the task.

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

VX_TASK(myTask,4096);
int myTaskId;

STATUS initializeFunction (void)
    {
    myTaskId = VX_TASK_INITIALIZE(myTask, 100, 0, 4096, pEntry, \
                              0,1,2,3,4,5,6,7,8,9);

   if (myTaskId != NULL)
       {
       taskActivate (myTaskId);
       return (OK);
       }
   else
       return (ERROR);
    }
```

It is the programmer's responsibility to pass the same name to **VX_TASK_INSTANTIATE** as was used in the **VX_TASK** declaration, which is **myTask** in this case. The arguments to **VX_TASK_INSTANTIATE** and their meaning are the same as those passed to **taskSpawn( )**. This makes the usage of **VX_TASK_INSTANTIATE** consistent with **taskSpawn( )**. Please note the backslash that continues the argument list on the succeeding line. This backslash character is crucial if the arguments span more than one line. This is to ensure correct macro expansion.

For more information, see the API reference for **taskLib**.

Static Instantiation Of Semaphores

The macros **VX_BINARY_SEMAPHORE**, **VX_COUNTING_SEMAPHORE** and **VX_MUTEX_SEMAPHORE** are used to declare a semaphore of type binary, counting, and mutex respectively. These macros take the semaphore name as an argument. The declared semaphores are initialized by calling routines **semBInitialize( )**, **semCInitialize( )** and **semMInitialize( )** respectively.

The three **sem***X***Initialize( )** routines are the equivalents of their respective **sem***X***Create( )** routines, the only difference being that the semaphore same name used in the associated **VX_***XXX***_SEMAPHORE** be passed to the **sem***X***Initialize( )** routines. The return value from the **sem***X***Initialize( )** routines is a semaphore ID that is then used to perform all operations on the semaphores.

The following example illustrates static instantiation of a binary semaphore:

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

VX_BINARY_SEMAPHORE(mySemB);   /* declare the semaphore */
SEM_ID mySemBId;               /* semaphore ID for further operations */

STATUS initializeFunction (void)
    {
    if ((mySemBId = semBInitialize (mysemB, options, 0)) == NULL)
         return (ERROR);      /* initialization failed */
    else
         return (OK);
    }
```

For more information, see the API reference for **semLib**.

Static Instantiation of Message Queues

The macro **VX_MSG_Q** is used to declare a message queue at compile time. It takes three parameters: the name, the maximum number of messages in the message queue, and the maximum size of each message.

After this declaration, the **msgQInitialize( )** routine is used to initialize the message queue and make it ready for use.

The following example illustrates static instantiation of a message queue:

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

VX_MSG_Q(myMsgQ,100,16);          /* declare the msgQ */
MSG_Q_ID myMsgQId;                /* MsgQ ID to send/receive messages */

STATUS initializeFunction (void)
    {
    if ((myMsgQId = msgQInitialize (myMsgQ, 100, 16, options)) == NULL)
         return (ERROR);      /* initialization failed */
    else
         return (OK);
    }
```

As with other static instantiation macros it is crucial to pass exactly the same name used in the **VX_MSG_Q** declaration to the **msgQInitialize( )** routine, or else compilation errors will result. Also it is crucial to pass exactly the same values for the message size and maximum number of messages as was passed to the **VX_MSG_Q** macro.

For more information, see the API reference for **msgQLib**.

Static Instantiation of Watchdogs

The macro **VX_WDOG** is used to declare a watchdog at compile time. It takes one parameter, the name of the watchdog. After this declaration, the routine **wdInitialize( )** is used to initialize the watchdog and make it ready for use.

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

VX_WDOG(myWdog);    /* declare the watchdog */
WDOG_ID myWdogId;   /* watchdog ID for further operations */

STATUS initializeFunction (void)
    {
    if ((myWdogId = wdInitialize (myWdog)) == NULL)
         return (ERROR);       /* initialization failed */
    else
         return (OK);
    }
```

As with other static instantiation macros it is crucial to pass exactly the same name used in the **VX_WDOG** declaration to the **wdInitialize( )** routine, or else compilation errors will result.

For more information, see the API reference for **wdLib**.

2.6.5 Applications and VxWorks Kernel Component Requirements

VxWorks is a highly configurable operating system. When kernel application modules are built independently of the operating system (see *2.6.6 Building Kernel Application Modules*, p.62), the build process cannot determine if the instance of VxWorks on which the application will eventually run has been configured with all of the components that the application requires (for example, networking and file systems). It is, therefore, useful for application code to check for errors indicating that kernel facilities are not available (that is, check the return values of API calls) and to respond appropriately.

When kernel application modules are linked with the operating system, the build system generates errors with regard to missing components. Both Workbench and the **vxprj** command-line tool also provide a mechanisms for checking dependencies and for reconfiguring VxWorks accordingly.

2.6.6 Building Kernel Application Modules

The VxWorks environment provides simple mechanisms for building kernel application modules, including a useful set of default makefile rules. Both the IDE and command line can be used to build applications. For command-line use, the **wrenv** utility program can be used to open a command shell with the appropriate host environment variables set. See *VxWorks Command-Line Tools User's Guide: Creating a Development Shell with wrenv* and the *VxWorks Command-Line Tools User's Guide*.

Using Makefile Include Files for Kernel Application Modules

You can make use of the VxWorks makefile structure to put together your own application makefiles quickly and efficiently. If you build your application directly in a BSP directory (or in a copy of one), you can use the makefile in that BSP, by specifying variable definitions that include the components of your application.

You can specify values for these variables either from the **make** command line, or from your own makefiles (when you take advantage of the predefined VxWorks **make** include files).

ADDED_CFLAGS
Application-specific compiler options for C programs.

ADDED_C++FLAGS
Application-specific compiler options for C++ programs.

Additional variables can be used to link kernel application modules with the VxWorks image; see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64.

For more information about makefiles, see the *VxWorks Command-Line Tools User's Guide*.

You can also take advantage of the makefile structure if you develop kernel application modules in separate directories. Example 2-1 illustrates the general scheme. Include the makefile headers that specify variables, and list the object modules you want built as dependencies of a target. This simple scheme is usually sufficient, because the makefile variables are carefully designed to fit into the default rules that **make** knows about.[2]

Example 2-1 **Skeleton Makefile for Kernel Applications**

```
# Makefile - makefile for ...
#
# Copyright ...
#
# DESCRIPTION
# This file specifies how to build ...

## It is often convenient to override the following with "make CPU=..."
CPU                = cputype
TOOL               = diab

include $(WIND_BASE)/target/h/make/defs.bsp

## Only redefine make definitions below this point, or your definitions
## will be overwritten by the makefile stubs above.

exe : myApp.o
```

For information about build options, see the *VxWorks Architecture Supplement* for the target architecture in question. For information about using makefiles to build applications, see the *VxWorks Command-Line Tools User's Guide*.

Statically Linking Kernel Application Modules

In general, kernel application modules do not need to be linked before being downloaded to the target. However, when several modules cross reference each other they should be linked to form a single module. With C++ code, this linking should be done before the *munch* step. (See *13.5.2 Munching a C++ Application Module*, p.650.)

[2] However, if you are working with C++, it may be also convenient to copy the **.cpp.out** rule from *installDir*/**vxworks-6.***x***/target/h/make/rules.bsp** into your application's makefile.

The following example is a command to link several modules, using the Wind River linker for the PowerPC family of processors:

```
c:\> dld -o applic.o -r applic1.o applic2.o applic3.o
```

Similarly, this example uses the GNU linker:

```
c:\> ldppc -o applic.o -r applic1.o applic2.o applic3.o
```

In either case, the command creates the object module **applic.o** from the object modules **applic1.o**, **applic2.o**, and **applic3.o**. The **-r** option is required, because the object-module output must be left in relocatable form so that it can be downloaded and linked to the target VxWorks image.

Any VxWorks facilities called by the kernel application modules are reported by the linker as unresolved externals. These are resolved by the loader when the module is loaded into VxWorks memory.

WARNING: Do not link each kernel application module with the VxWorks libraries. Doing this defeats the load-time linking feature of the loader, and wastes space by writing multiple copies of VxWorks system modules on the target.

2.6.7 Downloading Kernel Application Object Modules to a Target

Kernel application object modules can be downloaded from Workbench or from the kernel shell. Once a module has been loaded into target memory, any subroutine in the module can be invoked, tasks can be spawned, debugging facilities employed with the modules, and so on. It is often useful to make use of a startup routine to run the application (see *2.6.2 Application Structure*, p.52).

For information about using the kernel shell and module loader, see *12.2 Kernel Shell*, p.577 and *12.3 Kernel Object-Module Loader*, p.603. For information about using theWorkbench, see the *Wind River Workbench User's Guide*.

2.6.8 Linking Kernel Application Object Modules with VxWorks

In order to produce complete systems that include kernel application modules, the modules must be statically linked with the VxWorks image. The makefile **EXTRA_MODULES** variable can be used to do so. It can be used from the command line as follows:

```
% make EXTRA_MODULES="foo.o"
```

For more information about using makefile variables, see *2.6.6 Building Kernel Application Modules*, p.62.

To include your kernel application modules in the system image using a makefile, identify the names of the application object modules (with the **.o** suffix) with **EXTRA_MODULES**. For example, to link the module **myMod.o** with the operating system, add a line like the following:

```
EXTRA_MODULES = myMod.o
```

Building the system image with the module linked in is the final part of this step. In the project directory, execute the following command:

```
    c:\myVxProj\osProj> make vxWorks
```

For information about how to have kernel applications start automatically at boot time, see *2.6.10 Configuring VxWorks to Run Applications Automatically*, p.66.

2.6.9 Image Size Considerations

The size of the system image is often an important consideration, particularly when kernel application modules are linked with the operating system. This is true whether the image is loaded by a boot loader or is self-booting (see *2.4.1 VxWorks Image Types*, p.15).

⚠ **CAUTION:** For ROM-based images, ensure that **ROM_SIZE** configuration parameter reflects the capacity of the ROMs used.

Boot Loader and Downloadable Image

Generally, VxWorks boot loader code is copied to a start address in RAM above the constant **RAM_HIGH_ADRS**, and the boot loader in turn copies the downloaded system image starting at **RAM_LOW_ADRS**. The values of these constants are architecture dependent, but in any case the system image must not exceed the space between the two. Otherwise the system image will overwrite the boot loader code while downloading, potentially killing the booting process.

To help avoid this, the last command executed when you build a new VxWorks image is **vxsize**, which shows the size of the new executable image and how much space (if any) is left in the area below the space used for boot ROM code:

```
    vxsize 386 -v 00100000  00020000  vxWorks
    vxWorks: 612328(t) + 69456(d) + 34736(b) = 716520 (235720 bytes left)
```

(In this output, **t** stands for text segment, **d** for data segment, and **b** for bss.)

Make sure that **RAM_HIGH_ADRS** is less than **LOCAL_MEM_SIZE**. If your new image is too large, **vxsize** issues a warning. In this case, you can reconfigure the boot loader to copy the boot ROM code to a sufficiently high memory address by increasing the value of **RAM_HIGH_ADRS** in **config.h** and in the BSP's makefile (both values must agree). Then rebuild the boot loader. For more information, see *Persistent Memory Region*, p.150.

Self-Booting Image

For self-booting images, the data segment of a ROM-resident VxWorks system is loaded at **RAM_LOW_ADRS** (defined in the makefile) to minimize fragmentation.

For a CPU board with limited memory (under 1 MB 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 BSP makefile and **config.h** (both values must agree).

2.6.10 Configuring VxWorks to Run Applications Automatically

VxWorks can be configured to start kernel applications automatically at boot time. To do so, perform the following steps:

1. Configure VxWorks with the **INCLUDE_USER_APPL** component.

2. Add a call to the application's entry-point routine in the **usrAppInit( )** routine stub, which is in *installDir***/vxworks-6.***x***/target/proj/***projDir***/usrAppInit.c**.

   Assuming, for example, that the application entry point routine **myAppStartUp( )** starts all the required application tasks, you would add a call to that routine in the **usrAppInit( )** stub as follows:

   ```
   void usrAppInit (void)
       {
   #ifdef USER_APPL_INIT
       USER_APPL_INIT;     /* for backwards compatibility */
   #endif

       myAppStartUp();

       }
   ```

3. Link the kernel-base application object modules with the kernel image (see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64).

2.7 Custom Kernel Libraries

For information about creating custom kernel libraries, see the *VxWorks Command-Line Tools User's Guide*.

2.8 Custom VxWorks Components and CDFs

A VxWorks component is the basic unit of functionality with which VxWorks can be configured. VxWorks kernel components are described in Component Description Files (CDFs), using the Component Description Language (CDL). (For introductory information about the components provided with VxWorks, see *2.4.2 VxWorks Components*, p. 17.)

Wind River recommends using CDFs for developing extensions to the VxWorks operating system. If you wish to develop components that make APIs available to RTP applications (which run in user mode as real-time processes), also see *2.9 Custom System Calls*, p. 100.

NOTE: Functionality can be added to VxWorks in the form of kernel modules that do not have to be defined as VxWorks components (see *2.6 Kernel Applications*, p. 51). However, in order to make use of either Workbench or the **vxprj** command-line configuration facilities, to define dependencies between components, and so on, extensions to the operating system should be developed using CDFs.

A CDF identifies the binary and source code elements that make up the component, its configuration parameters, relationship to other components, and so on. CDFs also define information about how components are displayed in the Workbench kernel configuration facility, and they can be used to group components into predefined sets to facilitate VxWorks configuration. While some components are autonomous, some have dependencies on other components, which must be included in the configuration of the operating system for run-time operation.

Both Workbench and the **vxprj** command-line facility use CDFs for configuring the operating system with selected components, for setting component parameter values, and so on. Workbench also uses information in CDFs to display the names and descriptions of components, and to provide links to online help. For information about Workbench and the command-line facilities used for configuring and building VxWorks, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

The first of the sections that follow provide procedural instructions for creating, installing, and testing a CDF; then additional sections provide reference information about CDL and its constituent elements.

2.8.1 Creating and Modifying Components

This section provides an instructional example of how to create a component or to modify an existing one. The example is generally useful in its illustration of the process, but does not make exhaustive use of CDL objects and properties.

You must follow the CDL conventions when creating a CDF. For detailed information, see the sections referenced below in the instructions, as well as *2.8.13 CDF Template*, p.96.

After creating a CDF, you must place the file in the proper path to ensure that the configuration facility reads the information and includes the component in the hierarchy of components. For information in this regard, see *2.8.2 CDF Precedence and CDF Installation*, p.75 and *Adding New CDFs to the VxWorks Installation Tree*, p.77.

Information about testing a component is provided in *2.8.3 Testing New Components*, p.78.

Defining a Component

This section describes the process of defining your own component. To allow for the greatest flexibility, there is no convention governing the order in which properties describe a component or the sequence in which CDL objects are entered into a component description file. The following steps taken to create the component **INCLUDE_FOO** are a suggested order only; the sequence is not mandatory. Nor is there meant to be any suggestion that you use all of the properties and object classes described.

The naming conventions for CDL are described in *2.8.5 CDF Naming Conventions*, p.80. Note that CDF files must have a **.cdf** suffix.

Step 1: Name and Provide a Synopsis

To create your new component, first name it and provide a synopsis of its utility.

```
Component           INCLUDE_FOO     {
    NAME            foo component
    SYNOPSIS        this is just an example component
    ...
```

In the component **INCLUDE_FOO**, the **NAME** property is **foo component**. The **SYNOPSIS** property instructively informs the user that *this is just an example component*.

NAME and **SYNOPSIS** properties affect user presentation only; they have no bearing on initialization sequence or dependencies.

For information about the properties of the component object (**NAME** and so on), see *2.8.6 Component CDF Object*, p.81.

Step 2: Describe the Code-Related Elements

Next, describe your component's code portions by defining any modules and source configlettes that should be configured in during the build.

If the component **INCLUDE_FOO** has an object module associated with it—and not source code—use the **MODULES** property to specify it. You can specify any number of modules this way. In the following example, **fooLib.o** and **fooShow.o** are listed:

```
    ...
    MODULES         fooLib.o fooShow.o
    HDR_FILES       foo.h
    ARCHIVE         fooLib.a
    CONFIGLETTES    fooConfig.c
    ...
```

The configuration facilities (Workbench and vxprj) automatically analyze object module dependencies in order to calculate component dependencies. (Workbench also offers visibility into component dependencies by graphically presenting component relationships.)

For example, if **fooLib.o** has an unresolved global for **logMsg( )**, an automatic dependency upon the component **INCLUDE_LOGGING** is detected. For components not shipped in object module format, CDL supports the explicit listing of component dependencies.

If the source portion contains a call to **logMsg( )**, the configuration facility does not detect the dependency; instead, an explicit dependency upon **INCLUDE_LOGGING** should be declared using the **REQUIRES** property. See Step 6.

If your module is not located in the standard path, use the **ARCHIVE** property to specify the archive name, for example, **/somePath/fooLib.a**. (For additional instructions concerning the use of **ARCHIVE**, see Step 10.)

NOTE: Developers should create their own archives for custom components. Do not modify Wind River archives.

Use the **HDR_FILES** property to specify any **.h** files associated with the component, like **foo.h**. These files are emitted in **prjConfig.c**, providing the initialization routine with a declaration.

If there is source code that should be compiled as part of the component, put it in a **.c** file and specify the file name in the **CONFIGLETTES** property, for example, **fooConfig.c**. Component parameters should be referenced in the configlette or in the initialization routine; otherwise they have no effect.

For detailed information about the properties of the component object (**MODULES** and so on), see *2.8.6 Component CDF Object*, p.81.

Step 3: Set Up Initialization

If the component must be initialized, use the **INIT_RTN** property of the component object class to specify the initialization routine and its arguments to be called, as in **fooInit(***arg1, arg2***)**. If your component needs more than a single line of initialization, create or add the initialization code to a **.c** file and use the **CONFIGLETTES** property instead. By associating a configlette with an initialization routine, you are securing the configlette's place in the initialization sequence.

```
...
INIT_RTN    fooInit(arg1, arg2);
...
```

If you are not using the **MODULES** property of the component object, use the **LINK_SYMS** property to include your object module from a linked archive. The system generates an unresolved reference to the symbol (**fooRtn1** in this example), causing the linker to resolve it by extracting your module from the archive.

```
...
LINK_SYMS   fooRtn1
...
```

For more information about the **INIT_RTN** property of the component object, see and *2.8.6 Component CDF Object*, p.81.

Step 4: Establish the Initialization Sequence

Initialization order is important. You can control when in the initialization sequence your component is initialized with the **_INIT_ORDER** property. A component (or initialization group) that is bound to an existing initialization group using the **_INIT_ORDER** property is, by default, initialized last within that group. This is typically the desired effect; however, you can override this behavior by explicitly using the **INIT_BEFORE** property.

```
...
_INIT_ORDER    usrRoot
INIT_BEFORE    INCLUDE_USER_APPL
...
```

In the example, **INCLUDE_FOO** is declared a member of the **usrRoot** initialization group. **INIT_BEFORE** has been used for fine control, and **INCLUDE_FOO** is initialized before **INCLUDE_USER_APPL**. For more information about the **_INIT_ORDER** and **INIT_BEFORE** properties of the component object, see and *2.8.6 Component CDF Object*, p.81.

Alternatively, you can create a new initialization group and declare **INCLUDE_FOO** a member; however, you would have to declare the new initialization group a member of an existing initialization group. For information on initialization groups, see *2.8.8 Initialization Group CDF Object*, p.88.

NOTE: **INIT_BEFORE** only affects ordering within the initialization group. Do not reference a component that is not in the initialization group; this has no effect.

NOTE: The **INIT_AFTER** property has no effect in this release.

Step 5: Link Helpful Documentation

Specify related reference entries (in HTML format) with the **HELP** property.

```
...
HELP           fooMan
...
```

By default, a build automatically includes reference entries related to values declared by the **MODULES** and **INIT_RTN** properties. In the case of **INCLUDE_FOO**, in addition to **fooMan.html**, which is specified by **HELP**, the build associates the **fooLib** and **fooShow** libraries and the **fooInit( )** routine.

For more information about the **HELP** property of the component object, see and *2.8.6 Component CDF Object*, p.81.

Step 6: Define Dependencies

Use the **REQUIRES** and **INCLUDE_WHEN** properties to explicitly declare dependencies among components. (See Step 2 to learn how the configuration facility automatically configures object module-related dependencies.)

The configuration facility does not detect implicit dependencies when a component is not shipped in object module format. Likewise, no dependencies are detected when symbols are referenced by pointers at run-time. Both circumstances require you to declare dependencies explicitly.

```
...
REQUIRES          INCLUDE_LOGGING
INCLUDE_WHEN      INCLUDE_POSIX_AIO  INCLUDE_POSIX_MQ
...
```

In the example, **REQUIRES** declares that **INCLUDE_LOGGING** must be configured along with **INCLUDE_FOO**. **INCLUDE_WHEN** tells the system that whenever the components **INCLUDE_POSIX_AIO** and **INCLUDE_POSIX_MQ** are included, then **INCLUDE_FOO** must also be included.

For information about properties of the component object (including **REQUIRES** and so on), see *2.8.6 Component CDF Object*, p.81.

NOTE: In general, the configuration facility is designed to increase flexibility when selecting components, that is, to increase scalability; specifying a **REQUIRES** relationship reduces flexibility. Be sure that using **REQUIRES** is the best course of action in your situation before implementing it.

Step 7: List Associated Parameters

In the component object, use the **CFG_PARAMS** property to declare all associated parameters, for example, **FOO_MAX_COUNT**.

```
...
CFG_PARAMS     FOO_MAX_COUNT
...
```

For information about properties of the component object (including **CFG_PARAMS**), see *2.8.6 Component CDF Object*, p.81.

Step 8: Define Parameters

For each parameter declared by **CFG_PARAMS**, create a parameter object to describe it. Provide a name using the **NAME** property.

Use the **TYPE** property to specify the data type, either **int**, **uint**, **bool**, **string**, **exists**, or **untyped**.

Use the **DEFAULT** property to specify a default value for each parameter.

```
    Parameter          FOO_MAX_COUNT    {
        NAME           Foo maximum
        TYPE           uint
        DEFAULT        50
    }
```

For information about the parameter object and its properties, see *2.8.7 Parameter CDF Object*, p.87.

CAUTION: A component is considered mis-configured if it contains a parameter without an assigned value. Be sure to assign default values, unless there is no reasonable default and you want to force the user to set it explicitly. (Other CDF files with higher precedence, may of course be used to set the parameter value; see *CDF File Naming and Precedence*, p.75.)

Step 9: Define Group Membership

A component must be associated with either a folder or selection, otherwise it is not visible in Workbench. Assign a component to a folder because of its logical inclusion in the group defined by the folder, based on similar or shared functionality, for example. By including a component in a folder, you make it possible for the user to load it simultaneously with other components in its group by declaring them as default values for the folder, using the folder object's **DEFAULTS** property.

```
    ...
    _CHILDREN     FOLDER_ROOT
    ...
```

The **_CHILDREN** property declares that **INCLUDE_FOO** is a child component of folder **FOLDER_ROOT**. The prepended underscore ("_") serves to reverse the relationship declared by the property **CHILDREN**, which means that **_CHILDREN** identifies the *parent*. You can also use **_DEFAULTS** in conjunction with **_CHILDREN** to specify a component as a default component of a folder.

If you think a component is becoming too complex, you can divide it into a set of components assigned to a folder or selection object. In the following example, **INCLUDE_FOO** has been specified as part of a selection. You can add or remove the group from your configuration as a unit rather than by its individual components.

For folders, the **DEFAULTS** property specifies the base set of components that are included if the group is configured without any overrides.

For selections, the **DEFAULTS** property specifies the components that are included to satisfy the count (declared by the **COUNT** property), if you provide no alternative values.

In a selection group, the **COUNT** property specifies the minimum and maximum number of included components. If the user exceeds these limits the system flags the selection as mis-configured.

```
Selection        SELECT_FOO    {
    NAME         Foo type
    SYNOPSIS     Select the type of desired FOO support
    COUNT        0-1
    CHILDREN     INCLUDE_FOO_TYPE1 \
                 INCLUDE_FOO_TYPE2 \
                 INCLUDE_FOO_TYPE3
    DEFAULTS     INCLUDE_FOO_TYPE1
}
```

For information about the folder and section objects and its properties, see *2.8.11 Folder CDF Object*, p.92 and *2.8.12 Selection CDF Object*, p.94.

Step 10: Create a Dummy Component

The configuration facility analyzes archives only when they are associated with included components. This creates a chicken and egg problem: in order to know about a particular archive, the configuration facility would need to analyze components before they are actually added. In other words, if you add a component declared with an **ARCHIVE** property, the configuration analysis is done without knowing what the value of **ARCHIVE** is. So, if your component includes an archive with several object modules, you should create a dummy component that is always included, making it possible for the configuration facility to know that a new archive should be read. Call such a component **INSTALL_FOO**. It should contain only **NAME**, **SYNOPSIS**, and **ARCHIVE** properties. The user cannot add other components from the same archive until **INSTALL_FOO** is added.

For information about properties of the component object (including **ARCHIVE**), see *2.8.6 Component CDF Object*, p.81.

⚠ **CAUTION:** Do not alter Wind River-supplied CDFs directly. Use the naming convention to create a file whose higher precedence overrides the default properties of Wind River-supplied components.

Modifying a Component

Do not modify any Wind River CDF.

However, you can *effectively* modify the *properties* of existing components by re-specifying them in another, higher priority CDF file. Third-party CDF files are by convention read last and therefore have the highest priority. Use the naming convention to create a high-precedence CDF file that overrides the default properties of Wind River components.

For detailed information, see *2.8.2 CDF Precedence and CDF Installation*, p.75.

In the following example, the default number of open file descriptors (**NUM_FILES**) in the standard Wind River component **INCLUDE_IO_SYSTEM** has been modified. The normal default value is 50.

```
Parameter NUM_FILES {
    DEFAULT      75
}
```

By adding these example lines of code to a third-party CDF file, by removing and adding the component if it is already in the configuration, and by re-building the project, the value of **NUM_FILES** is changed to 75. The original Wind River CDF file, **00vxWorks.cdf**, is not changed; the default property value is changed because the third-party file has higher precedence. Other property values remain the same unless specifically redefined.

⚠ **CAUTION:** Do not alter the Wind River-supplied source configlette files in *installDir***/vxworks-6.***x***/target/config/comps/src**.

⚠ **CAUTION:** Do not alter Wind River-supplied CDFs directly. Use the naming convention to create a file whose higher precedence overrides the default properties of Wind River-supplied components.

2.8.2 CDF Precedence and CDF Installation

More than one CDF may define a given component and its properties. The precedence of multiple definitions is determined by a numbering scheme used with the CDF naming convention and by the order in which directories containing CDFs are read by the configuration facility.

CDF File Naming and Precedence

Wind River reserves the first 50 numbers, that is **00***fileName***.cdf** through **49***fileName***.cdf**. The remaining numbers, 50 through 99, may be used by third parties for their components.

Higher number files have greater precedence than lower number files (when the remainder of the name is the same); for example, the definition of the **INCLUDE_FOO** component in **66comp_foo.cdf** override those in **55comp_foo.cdf**.

This method of setting precedence allows project, BSP, and CPU architecture-specific overrides of generic components or parameters.

For example, if a BSP provider wanted to change the maximum number of files that can be open (**NUM_FILES**) to 75 from the default value of 50, it can be done in a BSP-specific CDF file with a higher file number that has the following entry in it:

```
Parameter NUM_FILES {
    DEFAULT 75
}
```

CDF Directories and Precedence

The component description files are read at two points by the configuration facility:

- When a project is created.
- After component description files are changed and the project build occurs.

The order in which CDFs are read is significant. If more than one file describes the same property of the same component, the one read last overrides all earlier ones. The intent is to allow component description files to have some precedence level. Files read later have higher precedence than those read earlier.

Precedence is established in two complementary ways:

- CDF files reside in certain directories, and those directories are read in a specified order.
- Within one of these directories, CDFs are read in alphanumeric order.

The configuration facility sources all **.cdf** files in any of the following directories. These directories are read in the order in which they are presented:

1. *installDir***/vxworks-6.***x***/target/config/comps/vxWorks**

   Contains all generic VxWorks components.

2. *installDir***/vxworks-6.***x***/target/config/comps/vxWorks/arch/***arch*

   Contains all architecture-specific VxWorks components (or component overrides).

3. *installDir***/vxworks-6.***x***/target/config/***bspName*

   Contains all BSP-specific components.

4. the project directory

   Contains all other components.

Within a directory, to control alphanumeric order, a two digit number is prepended to each **.cdf** file to determine the order of precedence within a given directory. See *2.8.5 CDF Naming Conventions*, p.80.

Elements of Components in a VxWorks Installation Tree

Wind River delivers the parts of its components in the following locations:

- Source code modules are usually found in the *installDir***/vxworks-6.***x***/target/src** or **target/config** directories.
- Headers are found in *installDir***/vxworks-6.***x***/target/h**; object modules are delivered in *installDir***/vxworks-6.***x***/target/lib/obj***ARCH*.
- Component description files are in *installDir***/vxworks-6.***x***/target/config/comps/vxWorks**.
- Component configlettes (source fragments) are in *installDir***/vxworks-6.***x***/target/config/comps/src**.

Third parties are not limited to this arrangement, and the location of component elements can be fully described in the component description file.

Adding New CDFs to the VxWorks Installation Tree

If you have created a new CDF, you must place it in the appropriate path, based on the nature of the contents and the desired level of precedence. Your choices of paths (as described in *CDF Directories and Precedence*, p.76) are as follows:

- *installDir***/vxworks-6.***x***/target/config/comps/vxWorks** for generic VxWorks component descriptions only
- *installDir***/vxworks-6.***x***/target/config/comps/vxWorks/arch/***arch* for architecture-specific VxWorks component descriptions
- *installDir***/vxworks-6.***x***/target/config/config/***bspName* for board-specific component descriptions
- the project directory for all other files

Wind River recommends that third parties place their component source code and object elements in a directory, such as *installDir***/vxworks-6.***x***/target/config/***vendorName*. The location of the component description file (CDF) depends on where in the system the components should be integrated.

To be able to integrate a new general-purpose VxWorks component into the system, the CDF must be located in *installDir*/**vxworks-6.*x*/target/config/comps/vxWorks**. If it is a BSP-specific component, the file should be located in the BSP directory. If it is specific to a single project, it should be located in the project directory (*installDir*/**vxworks-6.*x*/target/proj/***projectName*).

Be sure to follow the proper naming and numbering conventions, which are described in *2.8.5 CDF Naming Conventions*, p.80.

2.8.3 Testing New Components

There are several tests that can run to verify that components have been written correctly:

- **Check Syntax and Semantics**

  This **vxprj** command provides the most basic test:

  ```
  vxprj component check [projectFile] [component ... ]
  ```

  For example:

  ```
  % vxprj component check MyProject.wpj
  ```

  If no project file is specified, **vxprj** looks for a **.wpj** file in the current directory. If no component is specified, **vxprj** checks every component in the project. This command invokes the **cmpTest** routine, which tests for syntactical and semantic errors

  Based on test output, make any required modifications. Keep running the script until you have removed the errors.

- **Check Component Dependencies**

  You can test for *scalability bugs* in your component by running a second **vxprj** command, which has the following syntax:

  ```
  vxprj component dependencies [projectFile] component [component ... ]
  ```

  For example, the following command displays a list of components required by **INCLUDE_OBJ_LIB**, as well as those that require the component:

  ```
  % vxprj component dependencies INCLUDE_OBJ_LIB
  ```

  If no project file is specified, **vxprj** looks for a **.wpj** file in the current directory.

- **Check the Component Hierarchy in Workbench**

  Verify that selections, folders, and new components you have added are properly included by making a visual check of the Workbench component hierarchy.

Look at how your new elements appear in the folder tree. Check the parameters associated with a component and their parameter default values.

If you have added a folder containing components, and have included that folder in your configuration, the Workbench component hierarchy should display in **boldface** all components listed as defaults for that folder (that is, values for the **DEFAULTS** property).

- **Build and Boot the System**

  Verify that the resulting image builds and boots.

2.8.4 Component Description Language

The component description language (CDL) is used in component description files (CDFs) to describe software components. Each CDF has the suffix **.cdf**. A single file can define more than one component.

The Component Descriptor Language has several classes of objects, each of which has various properties.

Three of the object classes are used to define a software component, its configuration options, and its initialization routine. These objects are discussed in the following sections:

- *2.8.6 Component CDF Object*, p.81
- *2.8.7 Parameter CDF Object*, p.87
- *2.8.8 Initialization Group CDF Object*, p.88

Two classes of objects are used to group components into a single unit for ease of configuration. It is described in the following section:

- *2.8.9 Bundle CDF Object*, p.90
- *2.8.10 Profile CDF Object*, p.91

Two additional classes of objects are used to control the display of components in the Workbench kernel configuration facility. They are described in the following sections:

- *2.8.11 Folder CDF Object*, p.92
- *2.8.12 Selection CDF Object*, p.94

For information about naming conventions, see *2.8.5 CDF Naming Conventions*, p.80.

Based on the operating system components selected by the user, the configuration facilities (Workbench or **vxprj**) create the system configuration files **prjComps.h**, **prjParams.h**, and **prjConfig.c**, which are used in building the specified system. For information about how the configuration and code generation facilities work, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

2.8.5 CDF Naming Conventions

Follow these conventions when creating CDFs, where **FOO** (or **Foo**) is the variable element of the naming convention:

- All bundles names are of the form **BUNDLE_FOO**.
- All component names are of the form **INCLUDE_FOO**.
- All VxBus driver component names are of the form **INCLUDE_***driverType_driverName* (for example, **DRV_SIO_NS16550**).
- All folders names are of the form **FOLDER_FOO**.
- All selection names are of the form **SELECT_FOO**.
- Parameter names should not match the format of any other object type, but are otherwise unrestricted. For example, you can use **FOO_***XXX*, but not **INCLUDE_FOO**.
- All initialization group names should be of the form **initFoo**. However, Wind River initialization groups use the form **usrFoo** for backwards compatibility reasons.
- All component description files have a **.cdf** suffix.
- All **.cdf** file names begin with two decimal digits; for example, **00comp_foo.cdf**. These first two digits control the order in which **.cdf** files are read within a directory. See *2.8.2 CDF Precedence and CDF Installation*, p.75 for more information.

Note that more than one component can be defined in a single CDF.

New component description files should be independent of existing files, with two exceptions:

- New component objects should be bound into an existing folder or selection for GUI display purposes.

- New component object initialization routines must be associated with an existing initialization group.

A new component object can be bound to an existing folder or selection, and to an existing initialization group, without modifying the existing elements. By prepending an underscore ("_") to certain component properties, you can reverse the meaning of the property. For example, if there is already a folder **FOLDER_EXISTING** and an initialization group **initExisting**, you can bind a new component (which is defined in a different file) to it as follows:

```
Component    INCLUDE_FOO
    ...
    _CHILDREN       FOLDER_EXISTING
    _INIT_ORDER     initExisting
}
```

The property **_CHILDREN** has the opposite relationship to **FOLDER_EXISTING** as **CHILDREN**; that is, **_CHILDREN** identifies the *parent*. In other words, it produces the same effect as **FOLDER_EXISTING** having the component **INCLUDE_FOO** on its list of children—without any modifications being made to the CDF containing the **FOLDER_EXISTING** object.

Note that **INIT_BEFORE** can be used **_INIT_ORDER** to define exactly where in the initialization group the component should be placed.

2.8.6 Component CDF Object

Components are the basic units of configurable software. They are the smallest, scalable unit in a system. With either Workbench or the command-line **vxprj** facility, a user can reconfigure VxWorks by including or excluding a component, as well as modifying some of its characteristics. The properties of a component include:

- Identification of the object code (modules) and source code (configlettes) used in the build of a project.
- Identification of configuration parameters, which are typically preprocessor macros used within a component's configuration code.
- Integration information that controls how a component is integrated into an executable *system image* (for example, an initialization routine).

For information about how to group components in to a *bundle* of components, see *2.8.9 Bundle CDF Object*, p.90.

For information about displaying information about components in the Workbench kernel configuration facility, see *2.8.11 Folder CDF Object*, p.92 and *2.8.12 Selection CDF Object*, p.94.

Component Properties

The component object class defines the source and object code associated with a component, much of the integration information, and any associated parameters.

Dependencies among components can be detected and the related components can be automatically added by the configuration facility (GUI or CLI). It does so by analyzing the global symbols in each object module that belongs to the component, and then determining which other components provide the functionality.

As an example, a message logging component could be defined in CDL as follows:

```
Component           INCLUDE_LOGGING {
    NAME            message logging
    SYNOPSIS        Provides logMsg support
    MODULES         logLib.o
    INIT_RTN        logInit (consoleFd, MAX_LOG_MSGS);
    CFG_PARAMS      MAX_LOG_MSGS
    HDR_FILES       logLib.h
}
```

For illustrative purposes, in another configuration system using **#ifdef**/**#endif** statements, the definition of the logging component might look like this:

```
#ifdef INCLUDE_LOGGING
    logInit (consoleFd, MAX_LOG_MSGS);
#endif /* INCLUDE_LOGGING */
```

The component object includes the greatest number of properties. The more commonly used properties are described below.

NAME
The name that appears next to the component icon in the component tree of the Workbench kernel configuration facility.

SYNOPSIS
A brief description of the component's functionality, which is used in Workbench.

The next four properties are all related to the configuring of code in a project:

MODULES

The names of the object files that constitute the component's code (along with any source file configlettes). For example, **semShow.o** is identified in the **MODULES** property of the **INCLUDE_SEM_SHOW** component.

⚠ **CAUTION:** An object file must only be associated with one component, and its name must be globally unique (across the entire system).

CONFIGLETTES

Configlettes are source file fragments used in conjunction with parameters (see *CFG_PARAMS*, p.83 and *2.8.7 Parameter CDF Object*, p.87). They are typically used for parameter switches (for example, to specify a number, a binary setting to true or false, and so on).

Configlette definitions may include the use of macros. For example:

```
CONFIGLETTES    $(MY_PARAM)\myFile.c
```

Where **MY_PARAM** may either be a build macro, or an environment variable.

The default directory for configlettes is *installDir*/**vxworks-6.***x*/**target/config/comps/src**.

HDR_FILES

Header files associated with your configlette code or initialization routine. These are header files that must be included in order for your configlette or initialization routine to compile.

ARCHIVE

The archive file in which to find object modules stored other than in the standard location.

The following property provides configuration information:

CFG_PARAMS

A list of configuration parameters associated with the component, typically a list of preprocessor macros. Each must be described separately by its own parameter object. Also see *CONFIGLETTES*, p.83 and *2.8.7 Parameter CDF Object*, p.87.

The next group of properties control integration of the component into the system image, including initialization and dependency information.

INIT_RTN

A one-line initialization routine. Also see *INIT_BEFORE*, p.84, *_INIT_ORDER*, p.84, and *2.8.8 Initialization Group CDF Object*, p.88.

LINK_SYMS

A list of symbols to look up in order to include components from an archive.

REQUIRES

A list of component(s) that do not otherwise have structural dependencies and must be included if this component is included. List only those components that cannot be detected from **MODULES**; that is, by their associated object files. For example, components with configlettes only or those that reference other components through function pointers. In this latter case, **REQUIRES** can specify a selection.

INCLUDE_WHEN

Sets a dependency to automatically include the specified component(s) when this component is included (that is, it handles nested includes).

INIT_BEFORE

Call the initialization routine of this component before the one specified by this property. This property is effective only in conjunction with **_INIT_ORDER**. Also see *INIT_RTN*, p.83, *_INIT_ORDER*, p.84, and *2.8.8 Initialization Group CDF Object*, p.88.

_INIT_ORDER

The component belongs to the specified initialization group. This property places the specified component at the end of the **INIT_ORDER** property list of the initialization group object. Also see *INIT_RTN*, p.83, *INIT_BEFORE*, p.84, and *2.8.8 Initialization Group CDF Object*, p.88. Like **NAME** and **SYNOPSIS** before them, the following properties also affect user presentation in Workbench:

HELP

List reference pages associated with the component.

_CHILDREN

This component is a child component of the specified folder (or selection). Also see *CHILDREN*, p.93.

_DEFAULTS

This component is a default component of the specified folder (or selection). This property must be used in conjunction with **_CHILDREN**.

Normally, the component would be listed in the container (folder or selection) as **CHILDREN & DEFAULT**. The usage of the complement allows adding components to an existing container without modifying the container's CDF.

Component Template

```
Component          component {  // required for all components

    NAME           name         // readable name (e.g., "foo manager").
                                // should be in all lower case.

    SYNOPSIS       desc         // one-line description

    MODULES        m1 m2 ..     // object modules making up the service.
                                // used to generate dependency
                                // information.
                                // it is important to keep this list
                                // small,since the tool's dependency
                                // engine assumes that the component is
                                // included if *any* of the modules are
                                // dragged in by dependency. It may make
                                // sense to split a large number of
                                // modules into several distinct
                                // components.

    CONFIGLETTES   1 s2 ..      // source files in the component that are
                                // #included in the master configuration
                                // file.
                                // file paths are assumed to be relative
                                // to $(WIND_BASE)/target/config/comps/src

    HDR_FILES      h1 h1 ..     // header files that need to be included
                                // to use this component. Typically
                                // contains prototypes for the
                                // initialization routine.

    CFG_PARAMS     p1 p2 ..     // configuration parameters, typically
                                // macros defined in config[All].h, that
                                // can change the way a component works.
                                // see Parameters, below, for more info.

    INIT_RTN       init(..)     // one-line initialization routine.
                                // if it needs to be more than one line,
                                // put the code in a CONFIGLETTE.

    LINK_SYMS      s1 s2 ..     // reference these symbols in order to drag
                                // in the component from the archive.
                                // this tells the code generator how to
                                // drag in components that don't need to
                                // be initialized.

    REQUIRES       r1 r2 ..     // other components required. Note:
                                // dependencies are automatically calculated
                                // based on a components MODULES and
                                // LINK_SYMS. For example, because nfsLib.o
                                // calls rpcLib.o, the tool is able to
                                // figure out that INCLUDE_NFS requires
                                // INCLUDE_RPC. One only needs to list
                                // requirements that cannot be detected from
```

```
                                    // MODULE dependencies. Typically only needed
                                    // for components that don't have associated
                                    // MODULES (e.g., ones with just
                                    // configlettes).

    HELP            h1 h2 ..        // reference pages associated with the
                                    // component.
                                    // The default is the MODULES and INIT_RTN
                                    // of the component. For example, if the
                                    // component has MODULE fooLib.o, then the
                                    // manual page for fooLib is automatically
                                    // associated with the component (if the
                                    // manual page exists). Similary for the
                                    // components INIT_RTN. If there are any
                                    // other relevant manual pages, they can
                                    // be specified here.

    ARCHIVE         a1              // archive in which to find the MODULES.
                                    // default is lib$(CPU)$(TOOL)vx.a.
                                    // file path is assumed to be relative
                                    // to $(WIND_BASE)/target/lib.
                                    // any archive listed here is
                                    // automatically added to the VxWorks
                                    // link-line when the component is
                                    // included.
                                    // Note: the tool only analyzes archives
                                    // associated with included components.
                                    // This creates a chicken-and-egg problem,
                                    // because the tool analyzes components
                                    // before they are actually added.
                                    // So if you add a component with an ARCHIVE,
                                    // analysis will be done without the ARCHIVE.
                                    // As a work-around, if a separate archive is
                                    // used, create a dummy component that
                                    // lets the tool know that a new archive
                                    // should be read. Such a component
                                    // should be called INSTALL_something.
                                    // It should contain only NAME, SYNOPSIS,
                                    // and ARCHIVE attributes. Only after the
                                    // user adds it can he or she add other
                                    // components from the archive.

    INCLUDE_WHEN    c1 c2 ..        // automatically include this component
                                    // when some other components are included.
                                    // All listed components must be included to
                                    // activate the INCLUDE_WHEN (AND
                                    // relationship). This allows, for example,
                                    // msgQShow to be included whenever msgQ and
                                    // show are included. Similarly, WDB fpp
                                    // support can be included when WDB and fpp
                                    // are included.

    INIT_BEFORE     c1              // if component c1 is present, our init
                                    // routine must be called before c1.
                                    // Only needed for component releases.
    _CHILDREN       fname        // Component is a child of folder or
                                 // selection fname.

_INIT_ORDER         gname       // Component is a member of init group gname
                                 // and is added to the end of the
                                 // initialization sequence by default (see
                                 // INIT_BEFORE).
}
```

WARNING: The **_INIT_ORDER** and **_CHILDREN** elements should refer to elements that have already been defined. If a component is a **_CHILDREN** element of a folder that does not exist, it will not be displayed in the IDE.

2.8.7 Parameter CDF Object

Parameters are one of the primary means of configuring a component. Typically, one or more parameters control a component's features.

Also see *CONFIGLETTES*, p.83 and *CFG_PARAMS*, p.83.

For example, the following parameter defines the maximum number of log messages in the logging queue:

```
Parameter      MAX_LOG_MSGS {
    NAME       max # queued messages
    TYPE       uint
    DEFAULT    50
}
```

This is comparable to the C code macro assignment:

```
#define MAX_LOG_MSGS      50
```

Parameter Properties

The following properties describe a parameter:

NAME
The name that appears in the Workbench.

TYPE
The type of parameter, which can be defined as **int**, **uint**, **bool**, **string**, **exists**, or **untyped**.

When a configuration parameter is defined as type of **exists** the **prjParams.h** file that is automatically generated as part of creating a project will contain either an **#define** or **#undef** statement for the parameter.

For example, the kernel configuration parameter **INCLUDE_CONSTANT_RDY_Q** has a type of **exists**. Thus, if it's value is set to **TRUE** (which is the default), the following will appear in the auto-generated **prjParams.h** file:

```
#define INCLUDE_CONSTANT_RDY_Q
```

If the value is set to **FALSE**, the following will appear:

```
#undef INCLUDE_CONSTANT_RDY_Q
```

DEFAULT

The default value of the parameter.

Parameter Template

```
Parameter       parameter         {
    NAME        name              // readable name (e.g., "max open files")

    SYNOPSIS    desc              // one-line description.

    STORAGE     storage           // MACRO, REGISTRY, ... Default is MACRO.

    TYPE        type              // type of parameter:
                                  // int, uint, bool, string, exists.
                                  // Default is untyped.
                                  // more types will be added later.

    DEFAULT     value             // default value of the parameter.
                                  // default is none - in which case the user
                                  // must define a value to use the component.
                                  // for parameters of type "exists," the
                                  // value is TRUE if the macro should be
                                  // defined, or FALSE if undefined.
}
```

2.8.8 Initialization Group CDF Object

A component must include information that controls how it is integrated into an executable *system image*. This means that an initialization routine must be identified, as well as the point in the initialization sequence that the routine should be called. Initialization groups assemble related components for initialization and, thus, define the system startup sequence. The definition of the initialization group hierarchy is related only to run-time behavior, and has no impact on GUI presentation.

Also see *INIT_RTN*, p.83, *INIT_BEFORE*, p.84, and *_INIT_ORDER*, p.84.

An initialization group is a routine from which components and other initialization groups are called. The code in the routine is determined by the included components and their initialization code fragments. For example:

```
InitGroup           usrIosExtraInit {
    INIT_RTN        usrIosExtraInit();
    SYNOPSIS        extended I/O system
    INIT_ORDER      INCLUDE_EXC_TASK \
                    INCLUDE_LOGGING \
                    INCLUDE_PIPES \
                    ...
}
```

In this example, the components **INCLUDE_EXC_TASK**, **INCLUDE_LOGGING**, and **INCLUDE_PIPES** are part of the initialization group **usrIosExtraInit**. If those components are included, then their initialization routines are called in the order specified as part of this initialization group.

The code in an initialization group is synthesized by the configuration facility into the file **prjConfig.c**, which always part of every BSP project. To see a system's initialization sequence, build the project and examine the generated code in **prjConfig.c**.

Initialization Group Properties

The following properties describe an initialization group:

NAME
The **NAME** property can be used to provide a short name for the initialization group; however, it does not appear anywhere in this release of the GUI.

SYNOPSIS
The **SYNOPSIS** property is used to provide a brief, readable definition of the initialization group.

INIT_RTN
The initialization routine that initializes an associated component(s).

INIT_ORDER
Components and initialization groups belonging to this initialization group listed in the order in which they are to be initialized.

Initialization Group Template

```
InitGroup            group        {
    INIT_RTN         rtn(..)      // initialization routine definition

    INIT_ORDER       i1 i2 ..     // ordered list of init groups and
                                  // components that belong to this init group

    INIT_AFTER       i2 i2 ..     // Only needed for component releases.
}
```

2.8.9 Bundle CDF Object

Bundle object can be used to associate components that are often used together, which facilitates configuration of the operating system with generic sets of facilities.

For example, the network kernel shell (**BUNDLE_NET_SHELL**) includes all the components are required to use the kernel shell with a network symbol table:

```
Bundle BUNDLE_NET_SHELL {
    NAME  network kernel shell
    SYNOPSIS  Kernel shell tool with networking symbol table
    HELP  shell windsh tgtsvr
    COMPONENTS  INCLUDE_SHELL \
        INCLUDE_LOADER \
        INCLUDE_DISK_UTIL \
        INCLUDE_SHOW_ROUTINES \
        INCLUDE_STAT_SYM_TBL \
        INCLUDE_DEBUG \
        INCLUDE_UNLOADER \
        INCLUDE_MEM_SHOW \
        INCLUDE_SYM_TBL_SHOW \
        INCLUDE_CPLUS \
        INCLUDE_NET_SYM_TBL
    _CHILDREN  FOLDER_BUNDLES
}
```

For information about components, see *2.8.6 Component CDF Object*, p.81.

Bundle Properties

NAME
The name of the bundle as it should appear in Workbench.

SYNOPSIS
A description of the bundle's functionality, as it should appear in Workbench.

HELP
HTML help topics that the bundle is related to.

COMPONENTS
A list of components to add to the kernel configuration when this bundle is added.

_CHILDREN
This bundle is a child of the specified folder.

2.8.10 Profile CDF Object

A profile defines a set of components to be used as a starting point to create a project. The profile can only be used at creation time. When used, the profile replace any default configuration. Components defined by the profile are added to the project as if they were added by the user. Therefore, after creation of the project, there is no trace of the profile in the project. A profile, as such, cannot be *removed*. Profiles can be based on other profiles and add to them. There is a special *base profile* defined as **BSP_DEFAULT** that represents the default set of components defined by each BSP. If **BSP_DEFAULT** is not listed in **PROFILES**, then none of the BSP default components will be included in the project and the profile will replace the default configuration.

Profile Properties

NAME
The name of the profile as it should appear in Workbench.

SYNOPSIS
A brief description of the profiles functionality, which is used in Workbench.

PROFILES
A list of base profiles.

COMPONENTS
A list of components to add to the kernel configuration when this profile is added.

Profile Template

```
Profile             profileName {
    NAME            nameForGui
    SYNOPSIS        description
    PROFILES        p1 p2 p3...
    COMPONENTS      comp1 comp2 comp3...
}
```

2.8.11 Folder CDF Object

A folder is used to control the display of components in the Workbench kernel configuration facility.

The Workbench component hierarchy uses folders to group components logically. Instead of presenting all components in a flat list, it presents them hierarchically. For example, top-level folders might organize components under headings such as network, drivers, OS, and so on.

Folder objects provide a directory-type hierarchy for grouping components that are logically related. Folders allows for the graphical presentation of useful information about each component, such as:

- any hierarchical grouping it has with related components
- its associated dependencies on other components
- its configurable properties
- its integration into an existing system (for component releases)

Folder information only affects user presentation. It is not related to initialization group hierarchy, and thereby the startup sequence, which depend solely on a component's initialization group. Folders can contain one or more components, selections, and other folders; they do not have parameter or initialization group objects associated with them.

Folders can contain more than one component. For example, the ANSI functionality contained in a folder might be described as follows:

```
    Folder  FOLDER_ANSI {
            NAME            ANSI C components (libc)
            SYNOPSIS        ANSI libraries
            CHILDREN        INCLUDE_ANSI_ASSERT     \
                            INCLUDE_ANSI_CTYPE      \
                            INCLUDE_ANSI_LOCALE     \
                            INCLUDE_ANSI_MATH       \
                            INCLUDE_ANSI_STDIO      \
                            INCLUDE_ANSI_STDLIB     \
                            INCLUDE_ANSI_STRING     \
                            INCLUDE_ANSI_TIME       \
                            INCLUDE_ANSI_STDIO_EXTRA
            DEFAULTS        INCLUDE_ANSI_ASSERT INCLUDE_ANSI_CTYPE \
                            INCLUDE_ANSI_MATH INCLUDE_ANSI_STDIO \
                            INCLUDE_ANSI_STDLIB INCLUDE_ANSI_STRING \
                            INCLUDE_ANSI_TIME
    }
```

Folders offer great flexibility in grouping components. They allow groups to include more than just a default set of components that are included together. Components can be added and removed from the configuration individually (as defined by the **CHILDREN** property of the folder object).

Folder Properties

The following properties describe a folder:

NAME
The name that appears next to the folder icon in the Workbench component hierarchy.

SYNOPSIS
A brief description of the folder.

CHILDREN
Components, folders, and selections belonging to this folder are called *children*. Also see *_CHILDREN*, p.84.

DEFAULTS
The default component(s) that would be included if the folder were *added* without any overrides. Folder groupings can affect configuration when a folder is added, because components specified by a folder's **DEFAULTS** property are added all at once.

NOTE: Use folder objects only when creating a *new* grouping of components (new or existing). Do not modify existing folders in order to include new components. CDL accommodates that by prepending an underscore to a property name, for example, **_CHILDREN**.

Folder Template

```
Folder          folder        {
    NAME        name         // readable name (e.g., "foo libraries").

    SYNOPSIS    desc         // one-line description

    CHILDREN    i1 i2 ..      // containers and components
                             // that belong to this container.

    DEFAULTS    i1 i2 ..      // default CHILDREN.
                             // if the folder represents a complex
                             // subsystem (such as the WDB agent),
                             // this is used to suggest to the user
                             // which components in the folder are
                             // considered "default." That way the user
                             // can add the whole subsystem at once,
                             // and a reasonable set of subcomponents
                             // will be chosen.
}
```

2.8.12 Selection CDF Object

A selection is used to display alternative options in the Workbench kernel configuration facility.

They are similar to folders, but they are components that implement a common interface, for example, serial drivers, the timestamp mechanism, and the WDB communication interface. These components provide alternatives for the same service, and one or more can be selected for configuration in a single project. The selections CDL class is comparable to **#ifdef**/**#else** constructs in other configuration systems.

Selection information, like that for folders, is for user presentation only. It is not related to initialization group hierarchy, and thereby the startup sequence. Selections contain one or more components only.

Selections behave like folders, except they add a *count* for a range of available components; for example, a selection containing three components might tell the user only one can be configured at a time, or perhaps two of three. Because of the count, selections do not contain folders or other selections; nor do they have parameter or initialization group objects associated with them.

Selection Properties

The following properties describe a selection:

NAME
A readable name, the one that appears next to the selection icon in the Workbench component hierarchy.

SYNOPSIS
A brief description of the selection.

COUNT
Set a minimum and maximum count from available options.

CHILDREN
Components from which to select.

DEFAULTS
The default component(s), depending on the count.

For example, the following example provides for selection from a set of timestamp drivers, for use with the timestamp component:

```
Selection          SELECT_TIMESTAMP {
    NAME           select timestamping
    COUNT          1-1
    CHILDREN       INCLUDE_SYS_TIMESTAMP   \
                   INCLUDE_USER_TIMESTAMP  \
                   INCLUDE_SEQ_TIMESTAMP
    DEFAULTS           INCLUDE_SEQ_TIMESTAMP
}
```

There are three timestamp drivers available, as indicated by the three values for the **CHILDREN** property. The **COUNT** permits a choice of one, that is, a minimum and a maximum of 1.

Selection Template

```
Selection          selection     {
    NAME           name         // readable name (for example , "foo
                                // communication path")

    SYNOPSIS       desc         // one-line description
    COUNT           min-max      // range of allowed subcomponents.
                                // 1-1 means exactly one.
                                // 1- means one or more.

    CHILDREN        i1 i2 ..     // components from which to select

    DEFAULTS        i1 i2 ..     // default CHILDREN.
                                 // this is not used for anything except to
                                 // to suggest to the user which components
                                 // in the selection we consider "default."
}
```

2.8.13 CDF Template

This template includes all objects and properties that can be used in a CDF. Generally, CDFs use only a portion of them.

```
Bundle MY_BUNDLE
    {
    COMPONENTS :
    HELP :
    NAME :
    SYNOPSIS :
    _CHILDREN :
    }

Component MY_COMPONENT
    {
    ARCHIVE :
    CFG_PARAMS :
    CONFIGLETTES :
    ENTRY_POINTS :
    EXCLUDE_WHEN :
    HDR_FILES :
    HELP :
    HIDE_UNLESS :
    INCLUDE_WHEN :
    INIT_AFTER :
    INIT_BEFORE :
    INIT_RTN :
    LINK_DATASYMS :
    LINK_SYMS :
    MODULES :
    NAME :
    PREF_DOMAIN :
    PROJECT :
    PROTOTYPE :
    REQUIRES :
    SHUTDOWN_RTN :
    SYNOPSIS :
    TERM_RTN :
    USES :
    _CHILDREN :
```

```
    _COMPONENTS :
    _DEFAULTS :
    _EXCLUDE_WHEN :
    _HIDE_UNLESS :
    _INCLUDE_WHEN :
    _INIT_AFTER :
    _INIT_BEFORE :
    _INIT_ORDER :
    _LINK_DATASYMS :
    _LINK_SYMS :
    _REQUIRES :
    _USES :
    }

EntryPoint MY_ENTRYPOINT
    {
    NAME :
    PRIVILEGED :
    SYNOPSIS :
    TYPE :
    _ENTRY_POINTS :
    }

EntryPointType MY_ENTRYPOINTTYPE
    {
    SYNOPSIS :
    _TYPE :
    }

Folder MY_FOLDER
    {
    CHILDREN :
    DEFAULTS :
    HELP :
    NAME :
    SYNOPSIS :
    _CHILDREN :
    _DEFAULTS :
    }

InitGroup MY_INITGROUP
    {
    HELP :
    INIT_AFTER :
    INIT_BEFORE :
    INIT_ORDER :
    INIT_RTN :
    NAME :
    PROTOTYPE :
    SHUTDOWN_RTN :
    SYNOPSIS :
    TERM_RTN :
    _INIT_AFTER :
    _INIT_BEFORE :
    _INIT_ORDER :
    }
```

```
Module MY_MODULE
    {
    ENTRY_POINTS :
    NAME :
    SRC_PATH_NAME :
    _MODULES :
    }

Parameter MY_PARAMETER
    {
    DEFAULT :
    HELP :
    NAME :
    STORAGE :
    SYNOPSIS :
    TYPE :
    VALUE :
    _CFG_PARAMS :
    }

Profile MY_PROFILE
    {
    COMPONENTS :
    HELP :
    NAME :
    SYNOPSIS :
    _CHILDREN :
    }

Selection MY_SELECTION
    {
    ARCHIVE :
    CFG_PARAMS :
    CHILDREN :
    CONFIGLETTES :
    COUNT :
    DEFAULTS :
    HDR_FILES :
    HELP :
    HIDE_UNLESS :
    INIT_AFTER :
    INIT_BEFORE :
    INIT_RTN :
    LINK_DATASYMS :
    LINK_SYMS :
    MODULES :
    NAME :
    PROTOTYPE :
    REQUIRES :
    SHUTDOWN_RTN :
    SYNOPSIS :
    USES :
    _CHILDREN :
    _DEFAULTS :
    _HIDE_UNLESS :
```

```
    _INIT_AFTER :
    _INIT_BEFORE :
    _INIT_ORDER :
    _LINK_DATASYMS :
    _LINK_SYMS :
    _REQUIRES :
    _USES :
    }

Symbol MY_SYMBOL
    {
    _LINK_DATASYMS :
    _LINK_SYMS :
    }
```

2.9 Custom System Calls

The VxWorks system call interface provides kernel services for applications that are executed as processes in user space. The interface can be easily extended developers who wish to add custom system calls to the operating system to support special needs of their applications. (See the *VxWorks Application Programmer's Guide: Applications and Processes* for information about user-space applications.)

Initially, the developer's main tasks in extending the system call interface are designing the custom system call in accordance with the naming, numbering, and argument guidelines, and then writing the system call handler to support that design. See *2.9.2 System Call Requirements*, p. 101 and *2.9.3 System Call Handler Requirements*, p. 105.

The system call interface can then be extended either statically and dynamically. Static extension involves the use of configuration files and build system facilities to create a VxWorks system image that includes the new system call functionality. Dynamic extension involves using the host or kernel shell, and kernel object module loader, to download a development version of the system call handler to the kernel. See *2.9.4 Adding System Calls*, p. 107 and *2.9.5 Monitoring And Debugging System Calls*, p. 114.

2.9.1 How System Calls Work

System calls are C-callable routines. They are implemented as short pieces of assembly code called system call stubs. The stubs execute a trap instruction, which switches execution mode from user mode to kernel mode. All stubs are identical to each other except for the unique system call number that they pass to the kernel to identify the system call.

In kernel mode, a trap handler copies any system call arguments from the user stack to the kernel stack, and then calls the system call handler.

Each system call handler is given only one argument—the address of its argument array. Handler routines interpret the argument area as a structure whose members are the arguments.

System call handlers may call other routines in the kernel to service the system call request. They must validate the parameters of the system call, and return errors if necessary.

The architecture of the system call dispatcher allows system call handlers to be installed at either compile time or run-time.

2.9.2 System Call Requirements

In order to be able to generate system calls automatically, as well as to ensure proper run-time operation, system calls must adhere strictly to naming, numbering, argument, and return value rules.

System Call Naming Rules

The names of various elements associated with a system call must derive their names from that of the system call itself. It is important to adhere to this convention in order to avoid compilation errors when using the automated mechanisms provided for adding system calls. See Table 2-7.

Table 2-7 **System Call Naming Conventions**

Element	Name Convention
system call	*sysCallName*( )
system call stub	**SYSCALL_STUB_***sysCallName***.s**
system call handler routine	*sysCallName***Sc( )**
system call argument structure	*sysCallName***ScArgs**

The system call name is used by developer in system call definition files. The system call stub is generated automatically from the information in the definition files. The developer must write the system call handler routine, which includes the system call argument structure.

For example, if the name of the system call is **foo( )**, then:

- The system call stub is named **SYSCALL_STUB_foo.s**.

  The stub implements the routine **foo( )** in user mode.

- The system call handler routine for system call **foo** must be named **fooSc( )**.

  Routine **fooSc( )** is called when an application makes a call to **foo( )** in user space. Writing a routine with this name is the kernel developer's responsibility. Unless **fooSc( )** exists, an error will be generated when the kernel is rebuilt.

- If the **foo** system call takes at least one argument, the argument structure for **foo** must be declared as **struct fooScArgs** in the system call handler.

For information about system call handler requirements, see *2.9.3 System Call Handler Requirements*, p. 105. For information about adding system calls to the operating system—both statically and dynamically—see *2.9.4 Adding System Calls*, p. 107.

System Call Numbering Rules

Each system call must have a unique system call number. The system call number is passed by the system call stub to the kernel, which then uses it to identify and execute the appropriate system call handler.

A system call number is a concatenation of two numbers:

- the system call group number
- the routine number within the system call group

The group number is implemented as a ten-bit field, and the routine number as a six-bit field. This allows for up to 1024 system call groups, each with 64 routines in it. The total system-call number-space can therefore accommodate 65,536 system calls.

Six system call groups—numbers 2 through 7—are reserved for customer use. (Customers may request a formal system call group allocation from Wind River.) All other system call groups are reserved for Wind River use.

WARNING: Do not use any system call group numbers other than those reserved for customer use. Doing so may conflict with Wind River or Wind River partner implementations of system calls.

Wind River system call group numbers and system call routine numbers are defined in the **syscallNum.def** file. It should not be modified by customers.

Customer system calls group numbers and system call routine numbers are defined in the **syscallUsrNum.def** file.

The Wind River system call number definition file, and a template for the customer system call definition file are located in *installDir***/vxworks-6.***x***/target/share/h**.

A given system call group is simply a collection of related system calls offering complementary functionality. For example, the VxWorks **SCG_STANDARD** group includes system calls that are commonly found in UNIX-like (POSIX) systems, and the **SCG_VXWORKS** group includes system calls that are unique to VxWorks or that are otherwise dissimilar to UNIX-like system calls.

For information about using the system call definition files to generate system calls, see *Adding System Calls Statically*, p. 107.

System Call Argument Rules

System calls may only take up to eight arguments. Special consideration must be given to 64-bit arguments on 32-bit systems. Floating point and vector-type arguments are not permitted.

Wind River system calls are defined in the **syscallApi.def** file. It should not be modified by customers.

Customer system calls are defined in the **syscallUsrApi.def** file. See *Adding System Calls Statically*, p. 107 for information about editing this file.

Number of Arguments

System calls can take up to a maximum of eight arguments (the maximum that the trap handler can accommodate). Each argument is expected to be one *native-word* in size. The size of a native-word is 32 bits for a 32-bit architecture and 64 bits for 64-bit architectures. For the great majority of system calls (which use 32 bits), therefore, the number of words in the argument list is equal to the number of parameters the routine takes.

In cases where more than eight arguments are required the arguments should be packed into a structure whose address is the parameter to the system call.

64-Bit Argument Issues

64-bit arguments are permitted, but they may only be of the type **long long**. For 32-bit architectures, a 64-bit argument takes up two native-words on the argument list, although it is still only one parameter to the routine.

There are other complications associated with 64-bit arguments to routines. Some architectures require 64-bit arguments to be aligned to either even or odd numbered registers, while some architectures have no restrictions.

It is important for system call developers to take into account the subtleties of 64-bit argument passing on 32-bit systems. The definition of a system call for

VxWorks requires identification of how many words are in the argument list, so that the trap handler can transfer the right amount of data from the user-stack to the kernel-stack. Alignment issues may make this less than straightforward.

Consider for example, the following routine prototypes:

```
int foo (int a, int b, long long c, int d);
int bar (int p, long long q, int r);
```

The ARM and Intel x86 architectures have no alignment constraints for 64-bit arguments, so the size of the argument list for **foo( )** would be five words, while the size of the argument for **bar( )** would be four words.

PowerPC requires **long long** arguments to be aligned on eight-byte boundaries. Parameter **c** to routine **foo( )** is already at an eight-byte offset with respect to the start of the argument list and is hence aligned. So for PowerPC, the argument list size for **foo( )** is five words.

However, in the case of **bar( )** the **long long** argument **q** is at offset four from the first argument, and is therefore not aligned. When passing arguments to bar, the compiler will skip one argument register and place **q** at offset eight so that it is aligned. This alignment pad is ignored by the called routine, though it still occupies space in the argument list. Hence for PowerPC, the argument list for bar is five words long. When describing a system call such as **bar( )**, it is thus advised that the argument list size be set to five for it to work correctly on all architectures.

Consult the architecture ABI documentation for more information. There are only a few routines that take 64-bit arguments.

System Call Return Value Rules

System calls may return only a native word as a return value (that is, integer values or pointers, and so on).

64-bit return values are not permitted directly, though they may be emulated by using private routines. To do so, a system call must have a name prefixed by an underscore, and it must a pointer to the return value as one of the parameters. For example the routine:

```
long long get64BitValue (void)
```

must have a companion routine:

```
void _get64BitValue (long long *pReturnValue)
```

Routine **_get64BitValue( )** is the actual system call that should be defined in the **syscallUsrNum.def** and **syscallUsrApi.def** files. The routine **get64BitValue( )** can then be written as follows:

```
long long get64BitValue (void)
   {
   long long value;

   _get64BitValue (&value);
   return value;
   }
```

(The **get64BitValue( )** routine would be written by the user and placed in a user mode library, and the **_get64BitValue( )** routine would be generated automatically; see *2.9.4 Adding System Calls*, p.107.)

The value -1 (**ERROR**) is the only permitted error return value from a system call. No system call should treat -1 as a valid return value. When a return value of -1 is generated, the operating system transfers the **errno** value correctly across the trap boundary so that the user-mode code has access to it.

If **NULL** must be the error return value, then the system call itself must be implemented by another routine that returns -1 as an error return. The -1 value from the system call can then be translated to NULL by another routine in user mode.

2.9.3 System Call Handler Requirements

System call handlers must adhere to naming conventions, and to organizational requirements for the system call argument structure. They should validate arguments. If an error is encountered, they set **errno** and return **ERROR**.

A system call handler typically calls one or more kernel routines that provide the functionality required. In some cases, the code will call the public kernel API directly; in other cases, it may do otherwise to skip the kernel level validation, and call the underlying functionality directly.

⚠ **CAUTION:** In order to enforce isolation between kernel and user space, not all kernel APIs may be called from a system call handler. In particular, APIs cannot be called if their operation involves passing a user-side task ID or an RTP ID. APIs also cannot be called to create objects in the kernel if those APIs are already directly accessible in user space by way of the standard system call interface. Examples include **taskSpawn( )**, **taskCreate( )**, **msgQCreate( )**, **pthread_create( )**, the various semaphore creation routines, and so on.

System Call Handler Naming Rules

System call handlers must be named in accordance with the system call naming conventions, which means that they must use the same name as the system call, but with an **Sc** appended. For example, the **foo( )** system call must be serviced by the **fooSc( )** system call handler.

All system call handlers take a single parameter, which is a pointer to their argument structure. The argument structure must also be named in accordance with the system call naming conventions, which means that they must use the same name as the system call handler, but with **Args** appended. For example, the argument structure for **fooSc** must be declared as **struct fooScArgs**.

For example, the **write( )** system call is declared as:

```
int write (int fd, char * buf, int nbytes)
```

The system call handler routine for write is therefore named **writeSc( )**, and it is declared as:

```
int writeSc (struct writeScArgs * pArgs)
```

And the argument structure is **writeScArgs**, which is declared as:

```
struct writeScArgs
        {
        int    fd;
        char * buf;
        int    nbytes;
        };
```

See *System Call Naming Rules*, p.101.

System Call Handler Argument Validation

A system call handler should validate all arguments. In particular, it should:

- Bounds-check numerical values.
- Validate any memory addresses to ensure they are accessible within the current memory context (that is memory within the process, and not within the kernel).

See the **scMemValidate( )** API reference entry for information on pointer validation across system calls.

System Call Handler Error Reporting

At the end of the system call, in the case of failure, the system call handler should ensure **errno** is set appropriately, and then return -1 (ERROR). If the return value is -1 (ERROR) the kernel **errno** value is then copied into the calling process' **errno**. If there is no error, simply return a value that will be copied to user mode. If the handlers set their **errno** before returning ERROR, user mode code sees the same **errno** value.

2.9.4 Adding System Calls

System calls can be added both statically and dynamically. This means that they can be either configured and built into the VxWorks operating system image, or they can be added interactively to the operating system while it is running on a target.

Dynamic addition is useful for rapid prototyping and debugging of system calls. Static configuration is useful for more stable development efforts, and production systems.

Adding System Calls Statically

The process of adding system calls statically is based on the use of the **syscallUsrNum.def** and **syscallUsrApi.def** system call definition files.

The files define the system call names and numbers, their prototypes, the system call groups to which they belong, and (optionally) the components with which they should be associated. The **scgen** utility program uses these files—along with comparable files for standard VxWorks system calls—to generate the system call apparatus required to work with the system call handler written by the developer. The **scgen** program is integrated into the build system, and is run automatically when the build system detects that changes have been made to **syscallUsrNum.def** and **syscallUsrApi.def**.

The template files **syscallUsrNum.def.template** and **syscallUsrApi.def.template** are in *installDir***/vxworks-6.***x***/target/share/h**. Make copies of files in the same directory without the **.template** extension, and create the appropriate entries in them, as described below.

After you have written a system call handler, the basic steps required to add a new system call to VxWorks are:

1. If you are creating a new system call group, add an entry for the group to **syscallUsrNum.def**. See *Defining a New System Call Group*, p.109. Remember that only groups 2 through 7 are available to developers; do not use any other group numbers. (Contact Wind River if you need to have a group formally added to VxWorks.)

2. Add an entry to **syscallUsrNum.def** to assign the system call to a system call group, and to associate the system call name with a system call number. See *Defining a New System Call*, p.110.

3. Add an entry to **syscallUsrApi.def** to define the system call name and its arguments. See *Defining a New System Call*, p.110.

4. Write the system call handler routine. See *2.9.3 System Call Handler Requirements*, p.105.

5. Rebuild the kernel-mode and user-mode source code trees *installDir***/vxworks-6.***x***/target/src** and *installDir***/vxworks-6.***x***/target/usr/src**. Use the following command in each directory:

   ```
   make CPU=cpuType TOOL=toolType
   ```

   This command automatically detects changes in **syscallUsrNum.def** and **syscallUsrApi.def**, invokes the **scgen** utility, and then rebuilds the source trees.

What scgen Does

Using the system call definitions in the both Wind River and the customer system call definition files **scgen** generates the following:

1. The files *installDir***/vxworks-6.***x***/target/h/syscall.h** and *installDir***/vxworks-6.***x***/target/usr/h/syscall.h**. The contents of both files are identical. They define all system call numbers and group numbers in the system. These files provide information shared between kernel and user space code.

2. One system call assembly stub file for each system call. The stubs are placed into the appropriate architecture directory under *installDir***/vxworks-6.***x***/target/usr/src/arch** for compilation into **libvx.a** or **libc.so**.

3. A file containing argument structures for all system calls in the system. This file is architecture/ABI specific, and is used by the system call handlers located in the kernel. This file is named **syscallArgs***ArchAbi***.h** under *installDir***/vxworks-6.***x***/target/h/arch/***archName* (for example, *installDir***/vxworks-6.***x***/target/h/arch/ppc/syscallArgsppc.h**).

4. A file containing a pre-initialized system call group table for all system call groups known at compile-time. This file is *installDir***/vxworks-6.***x***/target/h/syscallTbl.h**.

All of this output is then used by the build system automatically; no user intervention is required to build the appropriated system call infrastructure into the system.

The **scgen** utility can also be run from the command line for debugging purposes.

Defining a New System Call Group

If you need to define a new system call group, add it to **syscallUsrNum.def** using the following syntax:

SYSCALL_GROUP **SCG_***sgcGroupName* *groupNum* *componentNames*

Six system call groups—numbers 2 through 7—are reserved for customer use. All other system call groups are reserved for Wind River use. (See *System Call Numbering Rules*, p. 102.) Group names must be unique.

WARNING: Do not use any system call group numbers other than those reserved for customer use. Doing so may conflict with Wind River or Wind River partner implementations of system calls.

Identification of component names is optional, and provides the means of associating a system call group (all its calls) with specific operating system components for inclusion in a VxWorks configuration. It works as follows:

- If a component name is not defined, the system call group is always included in the system.
- If a component is defined, the system call group will either be included in the system or left out of it—depending on the presence or absence of the component. That is, if the component is included in a VxWorks configuration by the user, then the system call group is included automatically. But if the component is not included in the configuration, the group is likewise not included.

The fields must be separated by one or more space characters.

For example, a new group called **SCG_MY_NEW_GROUP** could be defined with the following entry (where *N* is the group number selected for use):

```
SYSCALL_GROUP        SCG_MY_NEW_GROUP    N  INCLUDE_FOO
```

The system calls that are part of the system call group are identified below the **SYSCALL_GROUP** definition line. Up to 64 system calls can be identified within each group. See *Defining a New System Call*, p.110.

Defining a New System Call

To define a new system call, you must create entries in two different files:

- One entry in **syscallUsrNum.def**, which assigns it to a system call group and associates the system call name and number.
- One entry in **syscallUsrApi.def**, which defines the system call name and its arguments.

System Call Definition Syntax

To add a system call to a call group, add an entry to **syscallUsrApi.def** under the appropriate system call group name, using the following syntax:

sysCallNum sysCallName

Note that it is important to add system calls to the end of a system call group; do use numbers that have already been assigned. Reusing an existing number will break binary compatibility with existing binaries; and all existing applications must be recompiled. System call numbers need not be strictly sequential (that is there can be gaps in the series for future use).

To define a system call itself, add an entry to **syscallUsrApi.def** using the following syntax:

sysCallName numArgs [*argType arg1; argType arg2; argType angN;*] \
  CompName **INCLUDE** *headerFileName***.h**

System call definition lines can be split over multiple lines by using the backslash character as a connector.

The name of the system call used in **syscallUsrApi.def** must match the name used in **syscallUsrNum.def**.

When defining the number of arguments, take into consideration any 64-bit arguments and adjust the number accordingly (for issues related to 64-bit arguments, see *System Call Argument Rules*, p.103).

The arguments to the system call are described in the bracket-enclosed list. The opening bracket must be followed by a space; and the closing bracket preceded by one. Each argument must be followed by a semicolon and then at least one space. If the system call does not take any arguments, nothing should be listed—not even the bracket pair.

More than one component name can be listed. If any of the components is included in the operating system configuration, the system call will be included when the system is built. (For information about custom components, see *2.8 Custom VxWorks Components and CDFs*, p.67.)

The following mistakes are commonly made when editing **syscallUsrApi.def** and **syscallUsrNum.def**, and can confuse the **scgen** utility:

- No space after the opening bracket of an argument list.
- No space before the closing bracket of an argument list.
- No backslash at the end of a line (if the argument list continues onto the next line).
- An empty pair of brackets that encloses no arguments at all. This will cause the generated temporary C file to have a compile error.

Bear in mind that there can be no more than 64 routines in any system call group.

If the system call includes the definition of a new type in a header file, the header file must be identified with the **INCLUDE** statement. The **scgen** utility must resolve all types before generating the argument structures, and this is the mechanism by which it is informed of custom definitions.

For examples of how this syntax is used, see *System Call Definition Example*, p.111. Also consult the Wind River system call definitions files (**syscallNum.def** and **syscallApi.def**), but do not modify these files.

System Call Definition Example

Assume that we want to add the custom system call **myNewSyscall( )** to a new system call group **SCG_USGR0** (which is defined in **syscallNum.def**).

First, create **syscallUsrNum.def** file by copying **syscallUsrNum.def.template**. Then edit the file **syscallUsrNum.def**, adding a system call group entry for the appropriate group, and the system call number and name under it. System call groups 2 through 7 are reserved for customer use; do not use any other group numbers.

For example:

```
SYSCALL_GROUP    SCG_USER0    2
1 myNewSyscall
```

Then we must edit **syscallUsrApi.def** to define the system call itself.

The C prototype for **myNewSyscall( )** is:

```
int myNewSyscall (MY_NEW_TYPE a, int b, char *c);
```

The call has three arguments, and a type defined in a custom header file. Assume that we also want to implement the system call conditionally, depending on whether or not the component **INCLUDE_FOO** is configured into the operating system.

The entry in **syscallUsrApi.def** would therefore look like this:

```
INCLUDE <myNewType.h>
myNewSyscall 3 [ MY_NEW_TYPE a; int b; char *c; ] INCLUDE_FOO
```

Adding System Calls Dynamically

You can dynamically extend the system call interface on a target by downloading a kernel object module that includes code for installing system call handlers as well as the system call handler routines themselves. You do not need to modify the system call definition files, to run **scgen**, or to rebuild the kernel.

This approach is useful for rapid prototyping. It would rarely be useful or advisable with a deployed system.

System Call Installation Code

The code required to install your system call handlers in the kernel consists of:

- an initialized table for the system call handler routines
- a call to a system call registration routine

This code should be included in the same module with the system call handlers. You must identify a system call group for the system calls, and it should be a group that is otherwise unused in the target system.

Routine Table

The system call handler routine table is used to register the system call handler routines with the system call infrastructure when the module is downloaded.

For example, if the system handler routines are **testFunc0( )**, **testFunc1( )**, **testFunc2( )**, and **testFunc3( )**,the table should be declared as follows:

```
_WRS_DATA_ALIGN_BYTES(16) SYSCALL_RTN_TBL_ENTRY testScRtnTbl [] =
       {
       {(FUNCPTR) testFunc0,       1, "testFunc0",   0}, /* routine 0 */
       {(FUNCPTR) testFunc1,       2, "testFunc0",   1}, /* routine 1 */
       {(FUNCPTR) testFunc2,       3, "testFunc0",   2}, /* routine 2 */
       {(FUNCPTR) testFunc3,       4, "testFunc0",   3}  /* routine 3 */
       }
```

The **_WRS_DATA_ALIGN_BYTES(16)** directive instructs the compiler/linker to align the table on a 16-byte boundary. This directive is optional, but is likely to improve performance as it increases the chance of locating the table data on a cache line boundary.

Building the Object Module

Build the object module containing the system call handlers and registration code as you would any module. See *2.6.6 Building Kernel Application Modules*, p.62.

Downloading the Module and Registering the System Calls

After you have built the module, download it, register it, and check that registration has been successful:

1. Download it to the target system with the debugger, host shell, or kernel shell. From the shell (using the C interpreter) the module **foo.o** could be loaded as follows:

   ```
   -> ld < foo.o
   ```

2. Register the new handlers with the system call infrastructure before any system calls are routed to your new handlers. This is done by calling **syscallGroupRegister( )**. For example:

   ```
   -> syscallGroupRegister (2, "testGroup", 4, &testScRtnTbl, 0)
   ```

   The first argument is a variable holding the group number (an integer); the second is the group name; the second is the group name; the third is the number of system handler routines, as defined in the table; the fourth is the name of the table; and the last is set to that the registration does not forcibly overwrite an existing entry. (Note that you use the ampersand address operator with the third argument when you execute the call from the shell—which you would not do when executing it from a program.)

   It is important to check the return value from **syscallGroupRegister( )** and print an error message if an error was returned. See the API reference for **syscallGroupRegister( )** for more information.

3. Verify that the group is registered by running **syscallShow( )** from the shell (host or kernel).

The system call infrastructure is now ready to route system calls to the newly installed handlers.

Making System Calls from a Process

The quickest method of testing a new system call is to create and run a simple RTP application

First, calculate the system call numbers for your new system calls. In order to do so, use the **SYSCALL_NUMBER( )** utility macro (defined in **syscall.h**). For example, if you used group number 2 for your test group and the routine number for **testFunc0( )** is 0 (as described above), then the system call number for this routine is the value returned by the following call:

```
SYSCALL_NUMBER (2, 0)
```

The system call number for **testFunc1( )** is the value returned by this call:

```
SYSCALL_NUMBER (2, 1)
```

And so on.

To make the actual system calls, the application calls the **syscall( )** routine. The first eight arguments (all integers) are the arguments passed to your system call, and the ninth argument is the system call number.

For example, to have your user-mode applications to call **testFunc0( )** from process, you should implement **testFunc0( )** like this:

```
int testFunc0
    (
    int arg1,
    int arg2,
    int arg3,
    int arg4,
    int arg5
    )
    {
    return syscall (arg1, arg2, arg3, arg4, arg5, 0, 0, 0,
                    SYSCALL_NUMBER(2,0));
    }
```

Note that you must use nine arguments with **syscall( )**. The last argument is the system call number, and the preceding eight are for the system call arguments. If your routine takes less than eight arguments, you must use zeros as placeholders for the remainder.

2.9.5 Monitoring And Debugging System Calls

This section discusses using show routines, **syscallmonitor( )**, and hooks for obtaining information about, and debugging, system calls.

If show routines are included in your VxWorks configuration (with the component **INCLUDE_SHOW_ROUTINES**), the set of system calls currently available can be displayed with the **syscallShow( )** shell command with the shell's C interpreter:

```
-> syscallShow
Group Name             GroupNo   NumRtns   Rtn Tbl Addr
--------------------   -------   -------   ------------
TEMPGroup                7         6        0x001dea50
STANDARDGroup            8         48       0x001deab0
VXWORKSGroup             9         31       0x001dedb0
value = 55 = 0x37 = '7'


-> syscallShow 9,1
System Call Group name: VXWORKSGroup
Group Number          : 9

Routines provided     :
Rtn#   Name                      Address     # Arguments
----   ----------------------   ----------   -----------
0      (null)                   0x00000000      0
1      (null)                   0x00000000      0
2      (null)                   0x00000000      0
3      msgQSend                 0x001d9464      5
4      msgQReceive              0x001d94ec      4
5      _msgQOpen                0x001d9540      5
6      objDelete                0x001d95b8      2
7      objInfoGet               0x001d9bf8      4
8      _semTake                 0x001d9684      2
9      _semGive                 0x001d96d0      1
10     _semOpen                 0x001d970c      5
11     semCtl                   0x001d9768      4
12     _taskOpen                0x001d98b8      1
13     taskCtl                  0x001d99dc      4
14     taskDelay                0x001d99d4      1
15     rtpSpawn                 0x001a2e14      6
16     rtpInfoGet               0x001a2e60      2
17     taskKill                 0x001a2ec8      2
18     taskSigqueue             0x001a2f00      3
19     _timer_open              0x0018a860      4
20     timerCtl                 0x0018a8c0      4
21     pxOpen                   0x0018a960      4
22     pxClose                  0x0018acf4      1
23     pxUnlink                 0x0018ae44      2
24     pxCtl                    0x0018b334      4
25     pxMqReceive              0x0018aea0      6
26     pxMqSend                 0x0018afcc      6
27     pxSemWait                0x0018b1fc      3
28     pxSemPost                0x0018b0f8      1
29     pipeDevCreate            0x001971a8      3
30     pipeDevDelete            0x001971c4      2
value = 50 = 0x32 = '2'
->
```

The **syscallMonitor( )** routine allows truss style monitoring of system calls from kernel mode, on a global, or per-process basis. It lists (on the console) every system call made, and their arguments. The routine synopsis is:

```
syscallMonitor(level, RTP_ID)
```

If the *level* argument is set to 1, the system call monitor is turned on; if it is set to 0, it is turned off. If the *RTP_ID* is set to an RTP_ID, it will monitor only the system calls made from that process; if it is set to 0, it will monitor all system calls.

The **sysCallHookLib** library provides routines for adding extensions to the VxWorks system call library with hook routines. Hook routines can be added without modifying kernel code. The kernel provides call-outs whenever system call groups are registered, and on entry and exit from system calls. Each hook type is represented as an array of function pointers. For each hook type, hook functions are called in the order they were added. For more information, see the **syscallHookLib** API reference.

2.9.6 Documenting Custom System Calls

Since system calls are not functions written in C, the **apigen** documentation generation utility cannot be used to generate API references from source code comments. You can, however, create a function header in a C file that can be read by **apigen**. The function header for system calls is no different from that for other C functions.

Here is a function header for **getpid( )**:

```
/***********************************************************************
*
* getpid - Get the process identifier for the calling process.
*
* SYNOPSIS
* \cs
* int getpid
*    (
*    void
*    )
* \ce
*
* DESCRIPTION
*
* This routine gets the process identifier for the calling process.
* The ID is guaranteed to be unique and is useful for constructing
* uniquely named entities such as temporary files etc.
*
* RETURNS: Process identifier for the calling process.
*
* ERRNO: N/A.
*
* SEE ALSO:
* .pG "Multitasking"
*
*/
```

No code or C declaration should follow the header. The compiler treats it as a comment block, but **apigen** uses it to generate API documentation. All fields in the header above (**SYNOPSIS**, **DESCRIPTION**, **RETURNS**, and so on) must to be present in the code.

You have two choices for the location of the comments:

- You may add system call function headers to an existing C source file (one that has code for other functions). Be sure that this source file is part of the **DOC_FILES** list in the makefile for that directory. The **apigen** utility will not process it otherwise.

- You may create a C file that contains only function headers and no C code. Such files must be part of the **DOC_FILES** list in the makefile, but not part of the **OBJS** list (because there is no code to compile).

For more information about the coding conventions that are required for API documentation generation, and the apigen tool, see the *VxWorks BSP Developer's Guide* and the **apigen** entry in the *Wind River Host Utilities API* reference.

2.10 Custom Scheduler

VxWorks provides a scheduler framework that can be used to implement a custom scheduler. This section describes the requirements for creating a custom scheduler for VxWorks, and for configuring the operating system for its use. It also describes key features of the traditional VxWorks scheduler that provide a useful reference for designing a custom scheduler.

Before you consider implementing a custom scheduler, however, you should be familiar with the schedulers provided by Wind River. They are the following:

- The traditional VxWorks scheduler, which provides for priority-based preemptive scheduling, plus a round-robin extension. The traditional scheduler is included in VxWorks by default. For information, see *4.3.3 VxWorks Traditional Scheduler*, p.168.
- The VxWorks POSIX thread scheduler, which provides additional features for pthreads running in processes. For information, see *5.12 POSIX and VxWorks Scheduling*, p.277.

WARNING: The scheduler is fundamental to an operating system's behavior. VxWorks is not guaranteed to function as expected if a custom scheduler is used. You should ensure that all VxWorks components behave appropriately when a custom scheduler is used in place of a Wind River scheduler.

NOTE: The scheduler framework is not supported for the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

2.10.1 Requirements for a Custom Scheduler

The VxWorks scheduler framework allows you to implement a custom kernel scheduler for VxWorks. This section describes the code and configuration requirements for custom schedulers.

Code Requirements

A custom scheduler must manage the set of tasks that are in the **READY** state; that is, the tasks that are eligible for execution. At a minimum, a custom scheduler must define a ready queue structure for all ready tasks or a hook-routine that is executed at every clock tick.

A custom scheduler may also specify other class-specific structures, manage data in the task control block, and so on.

Q_HEAD and Q_NODE Structures

A user-specified ready queue class must define class-specific **Q_NODE** and **Q_HEAD** structures. The size of these structures together must not be more than 16 bytes. For more information, see *Multi-way Queue Structure*, p.124.

Q_CLASS Structure and Associated Routines

Users must define a ready-queue class for all **READY** tasks. A set of routines required by the **Q_CLASS** structure must be implemented. For more information about the **Q_CLASS** structure, see *Multi-way Queue Structure*, p.124.

Task Control Block Data

For a custom scheduler that must store user specific information in tasks, the **pSchedInfo** member of the task control block (TCB) may be used. The **pSchedInfo** member of the TCB is a **void** pointer.

There are two ways to access **pSchedInfo**, as follows:

- If the **qNode** is given, the **TASK_QNODE_TO_PSCHEDINFO( )** macro may be used to get the address of **pSchedInfo**. The file *installDir*/**vxworks-6.***x*/**target/h/taskLib.h** provides the definition of this macro. The macro is typically used in the user-defined queue management functions. For example:

```
void customQPut
    (
    CUSTOM_Q_HEAD     *pQHead, /* head of readyQ */
    CUSTOM_NODE       *pQNode, /* mode of insert */
    ULONG              key     /* key for insert */
    )
    {
    void          **ppSchedInfo;

    /* get the address to the pSchedInFo */

    ppSchedInfo = (void **) TASK_QNODE_TO_PSCHEDINFO (pQNode);
    }
```

- If the task ID **tid** is given, the **TASK_SCHED_INFO_SET( )** macro can be used to set the **pSchedInfo** field in the TCB. The macro **TASK_SCHED_INFO_GET( )**

can be used for getting the value of **pSchedInfo**. Both macros are defined in *installDir*/**vxworks-6.*x*/target/h/taskUtilLib.h**.

The custom scheduler may use **pSchedInfo** as the pointer to the user-specific data structure for tasks. If so, it must allocate memory for the data structure using a task hook routine that calls **malloc( )** or **memalign( )**. This approach, however, makes the task creation process less deterministic.

The memory can also be statically allocated (using global variables) for user-specified storage, and then used it during task initialization.

Tick Hook Routine

If a custom scheduler performs operations at each tick interrupt, the **tickAnnounceHookAdd( )** routine can be used to register a hook routine that is called at each tick interrupt. The hook routine must obey the same rules as ISRs, because it runs in interrupt context. Any VxWorks kernel service that should not be called in an interrupt context should not be called in this hook. For information about restrictions on ISRs, see *4.20 Interrupt Service Routines*, p.241.

The following pseudo-code example illustrates hook use:

```
void usrTickHook
    (
    int tid    /* task ID */
    )
    {
    update the statistics information if needed;
    update interrupted task's time slice if needed;
    resort the interrupted task location in the ready queue if needed.
    }
```

Custom Round-Robin Scheduling

VxWorks provides a round-robin scheduling policy based on a task hook routine. A custom scheduler may use the VxWorks round-robin scheme by incorporating the **kernelRoundRobinHook( )** routine in the user-specific tick hook routine.

The **kernelRoundRobinHook( )** routine places a task at the tail of the task list for its priority in the ready queue, and resets its time slice, if all of the following are true:

1. The interrupted task has not locked preemption.
2. The interrupted task is still in the **READY** state.
3. The interrupted task has consumed its allowed time slice.

To take advantage of the VxWorks's implementation of round-robin scheduling, the **kernelRoundRobinInstall( )** routine should be called in the **usrCustomSchedulerInit( )** routine to install the **kernelRoundRobinHook( )** call. For more information see *Modify usrCustomSchedulerInit( ) Routine*, p.122.

The routine **_func_kernelRoundRobinHook( )** can then be called within the user defined hook for the round robin policy to take effect. The **_func_kernelRoundRobinHook( )** takes the task ID (*tid*) of the interrupted task as its argument. The following code example takes advantage of the VxWorks round-robin scheduling scheme:

```
void usrTickHook

    int tid     /* task interrupted by tick */
    )
    {
    /* statistic information */

    /* call kernelRoundRobinHook() */

    if (_func_kernelRoundRobinHook != NULL)
        _func_kernelRoundRobinHook (tid);

    /* other work */
    ...
    }
```

If the custom scheduler does not take advantage of the VxWorks round-robin scheduling policy using **kernelRoundRobinHook( )**, the routine **kernelTimeSlice( )** must not be used to adjust system time slice nor to enable or disable round-robin scheduling. The **kernelTimeSlice( )** routine is used to dynamically enable round robin scheduling and to set the system time slice, or to disable it.

For more information about VxWorks round-robin scheduling, see *Round-Robin Scheduling*, p.169.

Configuration Requirements

To use a custom scheduler with VxWorks, the following must be done:

- Configure VxWorks for use with a custom scheduler
- Modify an initialization routine in a configuration file.
- Link the custom ready queue management code with VxWorks.

Add INCLUDE_CUSTOM_SCHEDULER Component

To enable the custom scheduler framework, VxWorks must be configured with the **INCLUDE_CUSTOM_SCHEDULER** component.

Modify usrCustomSchedulerInit( ) Routine

The **usrCustomSchedulerInit( )** routine must be modified to specify the custom ready queue structure and any hook routines that are executed at each tick interrupt. The following code illustrates this modification:

```
void usrCustomSchedulerInit (void)
    {
    vxKernelSchedDesc.readyQClassId     = qUserClassId; /* user's class ID */
    vxKernelSchedDesc.readyQInitArg1    = (void *) &usrReadyQBMap; /* arg1 */
    vxKernelSchedDesc.readyQInitArg2    = (void *) 256; /* arg2 */

    tickAnnounceHookAdd ((FUNCPTR)usrTickHook);
    kernelRoundRobinInstall();
    }
```

The **usrTickHook** argument is the hook to be called at each tick interrupt. The **kernelRoundRobinInstall( )** call is for the user VxWorks round-robin scheduling scheme.

The **usrCustomSchedulerInit( )** routine is in the *installDir*/**vxworks-6.*x*/target/config/comps/src/usrCustomerScheduler.c** file.

For information about the **vxKernelSchedDesc** variable, see *Scheduler Initialization*, p.123. This variable must be initialized for a custom scheduler.

Link Custom Scheduler Code

There are several ways for users to link the definition and implementation of **Q_NODE**, **Q_HEAD**, and **Q_CLASS** structure to VxWorks. For example, the custom scheduler configuration file *installDir*/**vxworks-6.*x*/target/config/comps/src/usrCustomerScheduler.c** can be the placeholder for the **Q_NODE** and **Q_HEAD** type definitions and user specified **Q_CLASS** implementation.

Another way is to create a new header file for **Q_NODE** and **Q_HEAD** definitions and a new source file for **Q_CLASS** implementation, and then link the new object file to VxWorks using the **EXTRA_MODULES** macro. For example:

```
EXTRA_MODULES = qUserPriLib.o
```

For information about using the **EXTRA_MODULES** macro, see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64.

2.10.2 Traditional VxWorks Scheduler

The traditional VxWorks scheduler is the default, and is included in the system with the **INCLUDE_VX_TRADITIONAL_SCHEDULER** component. The traditional scheduler has a priority-based preemptive scheduling policy. A round-robin scheduling extension can be enabled with the **kernelTimeSlice( )** routine. For more information about these options, see *4.3.3 VxWorks Traditional Scheduler*, p.168.

This section provides information about key features of the traditional VxWorks scheduler that can be useful in designing a custom scheduler.

Scheduler Initialization

The kernel scheduler description structure is initialized in the **usrKernelInit( )** routine. The following is an example for configuring the VxWorks traditional scheduler:

```
#ifdef INCLUDE_VX_TRADITIONAL_SCHEDULER

    /* install the traditional priority based preemptive scheduler */

#if (VX_TRADITIONAL_SCHED_CONSTANT_RDY_Q == TRUE)
    vxKernelSchedDesc.readyQClassId     = Q_PRI_BMAP;
    vxKernelSchedDesc.readyQInitArg1    = (void *) &readyQBMap;
    vxKernelSchedDesc.readyQInitArg2    = (void *) 256;
#else
    vxKernelSchedDesc.readyQClassId     = Q_PRI_LIST;
#endif /* VX_TRADITIONAL_SCHED_CONSTANT_RDY_Q == TRUE */

#endif /* INCLUDE_VX_TRADITIONAL_SCHEDULER */
```

This code is from *installDir***/vxworks-6.***x***/target/config/comps/src/usrKernel.c**.

The **vxKernelSchedDesc** variable is the kernel scheduler description structure, which is defined in *installDir***/vxworks-6.***x***/target/h/kernelLib.h** as follows:

```
typedef struct wind_sched_desc
     {
     Q_CLASS_ID readyQClassId;  /* readyQ Id    */
     void *     readyQInitArg1; /* readyQ init arg 1 */
     void *     readyQInitArg2; /* readyQ init arg 2 */
     } WIND_SCHED_DESC;
```

The **readyQClassId** is a pointer to a ready queue class. The ready queue class is a structure with a set of pointers to routines that manage tasks that are in the **READY** state. The **readyQInitArg1** and **readyQInitArg2** are the input arguments for the **initRtn( )** routine of the ready queue class.

The **Q_PRI_BMAP** value (which is assigned to **vxKernelSchedDesc.readyQClassId**) is the priority-based bit-mapped ready queue class ID. See *Multi-way Queue Structure*, p.124 for more information about the ready queue class and its associated members.

The **vxKernelSchedDesc** can be initialized with the user-specified ready queue class for customized kernel schedulers. See *2.10.1 Requirements for a Custom Scheduler*, p.118 for more information how to install custom schedulers.

After the initialization of **vxKernelSchedDesc** variable, VxWorks invokes the **qInit( )** routine to initialize the ready queue class, as follows:

```
/* kernel scheduler ready queue init */

    qInit (&readyQHead, vxKernelSchedDesc.readyQClassId,
                        (int)(vxKernelSchedDesc.readyQInitArg1),
                        (int)(vxKernelSchedDesc.readyQInitArg2));
```

The **qInit( )** routine invokes **vxKernelSchedDesc.readyQClassId->initRtn( )** to set up the ready queue and the **readyQHead** variable (which is of type **Q_HEAD**). It is described below in *Multi-way Queue Structure*, p.124.

Multi-way Queue Structure

The VxWorks scheduler data structure consists of **Q_HEAD**, **Q_NODE**, and **Q_CLASS** elements. The type definitions of **Q_HEAD** and **Q_NODE** structures are flexible so that they can be used for different types of ready queues.

The **readyQHead** variable is the head of a so-called *multi-way queue*, and the aforementioned **Q_PRI_BMAP** queue classes comply with the multi-way queue data structures.

The multi-way queue *head* structure (**Q_HEAD**) is defined in **qLib.h** as follows:

```
typedef struct          /* Q_HEAD */
    {
    Q_NODE  *pFirstNode;    /* first node in queue based on key */
    UINT     qPriv1;        /* use is queue type dependent */
    UINT     qPriv2;        /* use is queue type dependent */
    Q_CLASS *pQClass;       /* pointer to queue class */
    } Q_HEAD;
```

The first field in the **Q_HEAD** contains the highest priority node.

NOTE: Both the **qFirst( )** routine and **Q_FIRST( )** macro simply read the first four bytes of the **Q_HEAD** structure (the **pFirstNode** field) to determine the head of the queue. There is, therefore, no need for a queue-class specific routine to determine which node is the head of the queue.

The kernel scheduler performs a **Q_FIRST( )** on **readyQHead** to determine which task should be allocated to the CPU. For the **Q_PRI_BMAP** and **Q_PRI_LIST** queue classes, this represents the highest priority ready task.

The multi-way queue node structure (**Q_NODE**) is also defined in **qLib.h** as follows:

```
typedef struct          /* Q_NODE */
    {
    UINT     qPriv1;                    /* use is queue type dependent */
    UINT     qPriv2;                    /* use is queue type dependent */
    UINT     qPriv3;                    /* use is queue type dependent */
    UINT     qPriv4;                    /* use is queue type dependent */
    } Q_NODE;
```

Each task control block contains a **Q_NODE** structure for use by a multi-way queue class to manage the set of ready tasks. This same **Q_NODE** is used to manage a task when it is in a pend queue.

Note that a custom implementation of a multi-way queue class may define class-specific **Q_HEAD** and **Q_NODE** structures. The size of the class-specific structures must not exceed 16 bytes, which is the current size of both the **Q_HEAD** and **Q_NODE** structures.

Q_CLASS Structure

The kernel interacts with a multi-way queue class through a **Q_CLASS** structure. A **Q_CLASS** structure contains function pointers to the class-specific operators. For example, the address of the class specific *put* routine is stored in the **putRtn** field.

As described in *Scheduler Initialization*, p.123, the **qInit( )** routine is used to initialize a multi-way queue head to a specified queue type. The second parameter specifies the *queue class* (that is, the type of queue), and is merely a pointer to a **Q_CLASS** structure. All kernel invocations of the queue class operators are performed indirectly through the **Q_CLASS** structure.

The **Q_CLASS** structure is defined in **qClass.h** as follows:

```
typedef struct q_class     /* Q_CLASS */
    {
    FUNCPTR createRtn;      /* create and initialize a queue */
    FUNCPTR initRtn;        /* initialize a queue */
    FUNCPTR deleteRtn;      /* delete and terminate a queue */
    FUNCPTR terminateRtn;   /* terminate a queue */
    FUNCPTR putRtn;         /* insert a node into q with insertion key */
    FUNCPTR getRtn;         /* return and remove lead node routine */
    FUNCPTR removeRtn;      /* remove routine */
    FUNCPTR resortRtn;      /* resort node to new priority */
    FUNCPTR advanceRtn;     /* advance queue by one tick routine */
    FUNCPTR getExpiredRtn;  /* return and remove an expired Q_NODE */
    FUNCPTR keyRtn;         /* return insertion key of node */
    FUNCPTR calibrateRtn;   /* calibrate every node in queue by an offset */
    FUNCPTR infoRtn;        /* return array of nodes in queue */
    FUNCPTR eachRtn;        /* call a user routine for each node in queue */
    FUNCPTR restoreRtn;     /* restore a node position in queue */
    struct q_class *valid;  /* valid == pointer to queue class */
    } Q_CLASS;
```

The **restoreRtn** operator is used in VxWorks SMP, which does not support custom schedulers. The operator must therefore be set to **NULL**.

The following operators are not applicable to a queue class that is used to manage the set of ready tasks: **advanceRtn**, **getExpiredRtn**, and **calibrateRtn**.

The signatures of the expected **Q_CLASS** operators are as follows:

```
Q_HEAD * createRtn    (... /* optional arguments */);
STATUS   initRtn      (Q_HEAD *pQHead, ... /* optional arguments */);
STATUS   deleteRtn    (Q_HEAD *pQHead);
STATUS   terminateRtn (Q_HEAD *pQHead);
void     putRtn       (Q_HEAD *pQHead, Q_NODE *pQNode, ULONG key);
Q_NODE * getRtn       (Q_HEAD *pQHead);
STATUS   removeRtn    (Q_HEAD *pQHead, Q_NODE *pQNode);
void     resortRtn    (Q_HEAD *pQHead, Q_NODE *pQNode, ULONG newKey);
ULONG    keyRtn       (Q_HEAD *pQHead, Q_NODE *pQNode, int keyType);
int      infoRtn      (Q_HEAD *pQHead, Q_NODE *nodeArray [ ], int maxNodes);
Q_NODE * eachRtn      (Q_HEAD *pQHead, FUNCPTR routine, int routineArg);
```

As noted above, a custom scheduler may define class-specific **Q_HEAD** and **Q_NODE** structures.

Q_CLASS Operators

This section provides descriptions of each **Q_CLASS** operator that pertains to the management of ready tasks. Each description provides information about when the kernel invokes the operator for managing ready tasks.

Descriptions of the **advanceRtn**, **getExpiredRtn**, and **calibrateRtn** operators are not provided as they are not applicable to managing the set of ready tasks.

Some **Q_CLASS** operators are invoked within the *kernel context*. The operator description indicate whether the operator is invoked within kernel context or not. The operators that are invoked within kernel context do not have access to all VxWorks facilities. Table 2-8 lists the routines that are available from within kernel context.

Table 2-8 **Kernel Context Routines**

VxWorks Library	Available Routines
blib	All routines
fppArchLib	**fppSave( )** and **fppRestore( )**
intLib	**intContext( )**, **intCount( )**, **intVecSet( )**, **intVecGet( )**
lstLib, dllLib, sllLib	All routines except *xxx***Create( )** and *xxx***Delete( )**
mathALib	All routines, if **fppSave( )** and **fppRestore( )** are used
rngLib	All routines except **rngCreate( )**
taskLib	**taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**,**taskIsSuspended( )**and **taskTcb( )**
vxLib	**vxTas( )**

WARNING: The use of any VxWorks APIs that are *not* listed Table 2-8 from an operator that is invoked from kernel context results in unpredictable behavior. Typically the target will hang or reboot.

createRtn

Allocates a multi-way queue head structure from the system memory pool. The dynamically-allocated head structure is subsequently initialized. Currently the kernel does not utilize this operator. Instead, the ready task queue is initialized by statically allocating the head structure, and using the **initRtn** operator.

initRtn

Initializes a multi-way queue head. Up to ten optional arguments can be passed to the **initRtn**. The kernel initializes the ready task queue from the **usrKernelInit( )** routine as described in *Scheduler Initialization*, p.123. This operator is not called from within kernel context.

deleteRtn

Deallocates (frees) the multi-way queue head. All queued nodes are lost. Currently the kernel does not utilize this operator.

terminateRtn

Terminates a multi-way queue head. All queued nodes will be lost. Currently the kernel does not utilize this operator.

putRtn

Inserts a node into a multi-way queue. The insertion is based on the key and the underlying queue class. The second parameter is the **Q_NODE** structure pointer of the task to be inserted into the queue. Recall that each task control block contains a **Q_NODE** structure for use by a multi-way queue class to manage the set of ready tasks.

The third parameter, (the key), is the task's current priority. Note that a task's current priority may be different than a task's *normal* priority due to the mutex semaphore priority inheritance protocol.

The **pFirstNode** field of the **Q_HEAD** structure must be updated to contain the first node in the queue (if any change has occurred).

The **putRtn** operator is called whenever a task becomes ready; that is, a task is no longer suspended, pended, delayed, or stopped (or a combination thereof).

The VxWorks round-robin policy performs a **removeRtn** operation followed by a **putRtn** when a task has exceeded its time slice. In this case, the task does not change state. However, the expectation after performing a **removeRtn** operation followed by a **putRtn** operation is that the task appears as the last task in the *list* of tasks with the same priority, if there are any.

Performing a **taskDelay(0)** operation also results in a **removeRtn** operation followed by a **putRtn**. Again, in this case the task does not change state, and the expectation after performing a **removeRtn** operation followed by a **putRtn** operation is that the task appears as the last task in the *list* of tasks with the same priority, if there are any.

This operator is called from within kernel context.

getRtn

Removes and returns the first node in a multi-way queue. Currently the kernel does not utilize this operator.

removeRtn

Removes the specified node from the specified multi-way queue.

The **removeRtn** operator is called whenever a task is no longer ready; that is, it is no longer eligible for execution, since it has become suspended, pended, delayed, or stopped (or a combination thereof).

See the discussion of the **putRtn** operator above for more information about situations in which the kernel performs a **removeRtn** operation followed by a **putRtn** without the task's state actually changing.

This operator is called from within kernel context.

resortRtn

Resorts a node to a new position based on a new key.

The **resortRtn** operator is called whenever a task's priority changes, either due to an explicit priority change with the **taskPrioritySet( )** API, or an implicit priority change due to the mutex semaphore priority inheritance protocol.

The difference between invoking the **resortRtn** operator and a **removeRtn**/**putRtn** combination is that the former operator does not change the position of the task in the *list* of tasks with the same priority (if any) when the priority is the same as the old priority.

This operator is called from within kernel context.

keyRtn

Returns the key of a node currently in a multi-way queue. The **keyType** parameter determines key style on certain queue classes. Currently the kernel does not utilize this operator.

infoRtn

Gathers information about a multi-way queue. The information consists of an array, supplied by the caller, filled with all the node pointers currently in the queue. Currently the kernel does not utilize this operator.

eachRtn

Calls a user-supplied routine once for each node in the multi-way queue. The routine should be declared as follows:

```
BOOL routine
    (
    Q_NODE *pQNode,      /* pointer to a queue node */
    int     arg          /* arbitrary user-supplied argument */
    );
```

The user-supplied routine should return **TRUE** if **qEach( )** is to continue calling it for each entry, or **FALSE** if it is done, and **qEach( )** can exit.

Currently the kernel does not utilize this operator.

3
Boot Loader

3.1 Introduction

A VxWorks boot loader is an application whose purpose is to load a VxWorks image onto a target. It is sometimes called the VxWorks *bootrom*, but use of this term is not encouraged (it conflates application and media). Like VxWorks, the boot loader can be configured with various facilities; such a command shell for dynamically setting boot parameters, a network loader, and a file system loader.

The same boot loaders are used for uniprocessor (UP), symmetric multiprocessor (SMP), and asymmetric multiprocessor (AMP), configurations of VxWorks.

In a development environment, boot loaders are useful for loading a VxWorks image from a host system, where VxWorks can be quickly modified and rebuilt. They can also be used in production systems when the boot loader and operating system are stored on a disk or other media.

Self-booting (standalone) VxWorks images do not require a boot loader. These images are commonly used in production systems (stored in non-volatile devices). For more information, see *2.4.1 VxWorks Image Types*, p.15.

Usually, the boot loader is programmed in a non-volatile device (usually flash memory or EEPROM) at an address such that it is the first code that is run by the processor when the target is powered on or rebooted. The procedure to get the boot loader programmed in a non-volatile device or written to a disk is dependent on the target, and is described in the BSP reference documentation.

The VxWorks product installation includes default boot loader images for each installed BSP. If they do not meet your needs, you can create custom boot loaders. For example, you may need to use a different network driver to load the VxWorks image over your development network, or you may want to remove the boot loader shell for deployed systems.

For information beyond what is in this chapter, particularly information about setting up a cross-development environment, see the *Wind River Workbench User's Guide: Setting up Your Hardware*.

3.2 Using a Default Boot Loader

The default boot loader is designed for a networked target and must be configured with parameters such as your host and target network addresses, the full path and name of the file to be loaded, the user name, and more. To use the default boot loader, you must interactively change the default parameters using the boot loader shell so that the loader can find the VxWorks image on the host and load it onto the target.

After you have entered boot loader parameters, the target can be booted with the VxWorks image. For most targets, the new settings are saved (in a non-volatile device or to disk) so you can reboot the target without resetting them.

You interact with the boot loader shell at a terminal console that is usually established by connecting a serial port of the target to a serial port on the host and starting a terminal application on the host. For information about the setup required to establish communication over the serial port of a particular target, see the reference documentation for the BSP in question.

When you apply power to the target (or each time it is reset), the target runs the boot loader (from ROM, flash, disk, or other media). During the boot process, the target uses its serial port to communicate with the host system. The boot loader displays a banner page and then starts a seven-second countdown before booting VxWorks. You can interrupt the boot process by pressing any key in order to set the appropriate boot parameters.

Default boot loader images are in *installDir***/vxworks-6.***x***/target/config/***bspName*. The boot loader commands and parameters are described in *3.4.1 Boot Loader Shell Commands*, p. 136 and *3.5 Boot Loader Parameters*, p. 140. The different types of boot loader images provided as defaults are described in *3.3 Boot Loader Image Types*, p. 133.

3.3 Boot Loader Image Types

Boot loader images can be stored in ROM, flash, on disk, or in storage on a network. Boot loader images are in ELF format. Binary file versions (**.bin**) are used for disks, and the hex record file versions (**.hex**) are used for programming non-volatile devices. For information about which types are available for a specific target, see the reference documentation for the BSP in question.

The varieties of boot loader images are described below. The first of each pair of file names listed for each image type is produced with the **PROFILE_BOOTAPP** profile, and the second by the legacy *bspDir***/config.h** method. For more information about build methods, see *3.7 Customizing and Building Boot Loaders*, p. 146.

Compressed Image

vxWorks_romCompress and **bootrom** files
The image is almost entirely compressed. It has a small un-compressed portion that the processor runs immediately after power is applied or the target is reset. This portion initializes memory and decompresses the compressed section (stored in non-volatile storage) into RAM, and causes the processor to switch execution to RAM. Compression of the image makes it much smaller than other types of boot loader images, therefore it uses less non-volatile storage. However, decompression increases the boot time.

Uncompressed Image

vxWorks_rom and **bootrom_uncmp** files
The image is not compressed. It copies itself to RAM and makes the processor switch execution to RAM. Because the image is not compressed, it is larger than the compressed image type. However, it has faster boot time because there is no decompression operation required.

Resident in Non-Volatile Storage

vxWorks_romResident and **bootrom_res** files
The image copies only the data segment to RAM on startup; the text segment stays in non-volatile storage. This means that the processor always executes instructions out of non-volatile storage. It is therefore sometimes described as being ROM-resident. This type of boot loader image is the one that requires the least amount of RAM to boot load the VxWorks kernel. It is therefore useful for boards with very little RAM, which needs to be saved for the application.

Boot loader images are located in *installDir***/vxworks-6.***x***/target/config/***bspName*. Note that the default images for most BSPs are configured for a networked development environment. For information about creating a custom boot loader, see *3.7 Customizing and Building Boot Loaders*, p. 146.

3.4 Boot Loader Shell

The boot loader shell provides commands for the following activities:

- Changing boot loader parameters (such as the host and target IP addresses)
- Rebooting the target system
- Managing the boot process.

To include the boot loader shell, configure the boot loader with the **INCLUDE_BOOT_SHELL** component.

CAUTION: Do not add any of the following components to the boot loader:

- **INCLUDE_WDB_BANNER**
- **INCLUDE_SIMPLE_BANNER**
- **INCLUDE_SHELL**

They conflict with the boot loader shell. If you include any of them, you will encounter configuration errors.

When the boot loader has initiated booting the system, it prints a banner. After a countdown elapses, the boot loader loads and runs the specified image. (If the boot loader shell is not included, the loader executes with its current parameters and without a countdown.) To reset the boot parameters, interrupt the boot loader during the countdown by pressing any key before the countdown period elapses.

To access the boot-loader shell prompt, power on (or reboot) the target; then stop the boot sequence by pressing any key during the seven-second countdown. The appearance of the boot-loader banner followed by keystroke interruption of the boot process looks like the following:

```
VxWorks System Boot


Copyright 1984-2005  Wind River Systems, Inc.





CPU: PC PENTIUM3
Version: VxWorks 6.3
BSP version: 2.0/6
Creation date: Jun 08 2006, 12:08:39
```

```
Press any key to stop auto-boot...
 7
[VxWorks Boot]:
```

To display a list of available commands, type either **h** or **?** at the prompt, followed by **ENTER**. For information about the commands, see *3.4.1 Boot Loader Shell Commands*, p. 136.

3.4.1 Boot Loader Shell Commands

The VxWorks boot loader provides a set of commands that can be executed from the boot loader shell, which are described in the following tables. For information about the boot loader shell, see *3.4 Boot Loader Shell*, p. 135.

Table 3-1 **Commands Provided with INCLUDE_BOOT_SHELL**

Command	Description
h	Help command. Print a list of available boot commands and flags; the boot device in use, image path, and image file name; and the boot devices available in the boot loader.
?	Same as **h**.
@	Boot (that is, load and execute the VxWorks image file) using the current boot parameters. See *3.5 Boot Loader Parameters*, p.140.
$ [*paramString*]	Boot (that is, load and execute the VxWorks image file). If used without a parameter string, the command is the same as **@.** The parameter string can be used to set boot loader parameter values all at once, instead of interactively. See *3.5 Boot Loader Parameters*, p.140 and *3.5.3 Changing Boot Loader Parameters Interactively*, p.144).
p	Print the current boot parameter values. See *3.5 Boot Loader Parameters*, p.140.
c	Change the boot parameter values. See *3.5 Boot Loader Parameters*, p.140.
l	Load the VxWorks image file using current boot parameters, but without executing.
g *adrs*	Go to (execute at) hex address *adrs*.
e	Display a synopsis of the last occurring VxWorks exception.
v	Display boot-loader banner page with BSP and boot loader version information.

Table 3-2 **Commands Provided with INCLUDE_BOOT_MEM_CMDS**

Command	Description
d *adrs*[**,** *n*]	Display *n* words of memory starting at hex address *adrs*. If *n* is omitted, the default is 64.
m *adrs*	Modify memory at location *adrs* (hex). The system prompts for modifications to memory, starting at the specified address. It prints each address, and the current 16-bit value at that address, in turn. You can respond in any of the following ways at the prompt: **ENTER** Pressing the **ENTER** key alone does not change the address specified with *adrs*, but continues prompting at the next address. *number* Entering a number sets the 16-bit contents at the memory address to that number. **.** Entering a . (period) leaves the address unchanged, and quits.
f *adrs***,** *nbytes***,** *value*	Fill *nbytes* of memory, starting at *adrs* with *value*.
t *adrs1***,** *adrs2***,** *nbytes*	Copy *nbytes* of memory, starting at *adrs1*, to *adrs2*.

Table 3-3 **Command Provided with INCLUDE_BOOT_ETH_MAC_HANDLER**

Command	Description
M [*dev*] [*unitNo*] [*MAC*]	Set and display Ethernet MAC address. For example: `M motfcc0 00:A0:1E:00:10:0A` In this case, the device is motfcc, the unit number is zero, and the MAC address is 00A01E00100A.

Table 3-4 **Command Provided with INCLUDE_BOOT_ETH_ADR_SET**

Command	Description
N [*last3ByteValuesMAC*]	Set (last three bytes) and display Ethernet MAC address.

NOTE: The **M** command (see Table 3-3) is a replacement for the **N** command (Table 3-4), which is maintained for backwards compatibility purposes. For information about which of the two is supported for a given BSP, consult the BSP reference.

The **M** and **N** command are provided by the **INCLUDE_BOOT_ETH_MAC_HANDLER** and **INCLUDE_BOOT_ETH_ADR_SET**, respectively. Do not use both components in the same configuration of VxWorks.

Table 3-5 **Commands Provided with INCLUDE_BOOT_EDR_SUPPORT**

Command	Description
P	Print the error log for the error detection and reporting facility.
C	Clear the error log for the error detection and reporting facility.

For information about the error detection and reporting facility, see *11. Error Detection and Reporting*.

Table 3-6 **Command Provided with BOOT_FILESYSTEMS**

Command	Description
devs	Display a list of all devices known to the I/O system. It performs the same function as the kernel shell C interpreter command of the same name.

Table 3-7 **Command Provided with INCLUDE_BOOT_USB_FS_LOADER**

Command	Description
usbBulkShow *nodeId*	Displays all the logical unit numbers of the USB device specified by *nodeId* (for example, a USB memory stick). It performs the same function as the kernel shell C interpreter command of the same name.

For information about rebooting a system that is running VxWorks, see *3.6 Rebooting VxWorks*, p.145.

3.5 Boot Loader Parameters

The boot loader parameters include all the information needed to locate and load a VxWorks image, as well as other settings used to manage the boot process. As required by the specific boot configuration, they can include host and target IP addresses, the full path and name of the VxWorks image to be booted, the user name, and so on. Boot parameters can be changed interactively at runtime, or statically configured when the boot loader is created. Interactive changes to boot parameters are retained between reboots for most targets (in a non-volatile device or on disk).

3.5.1 Displaying Current Boot Parameters

If the boot loader is configured with its command shell, the current set of boot loader parameters can be displayed interactively with the **p** command, as follows:

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

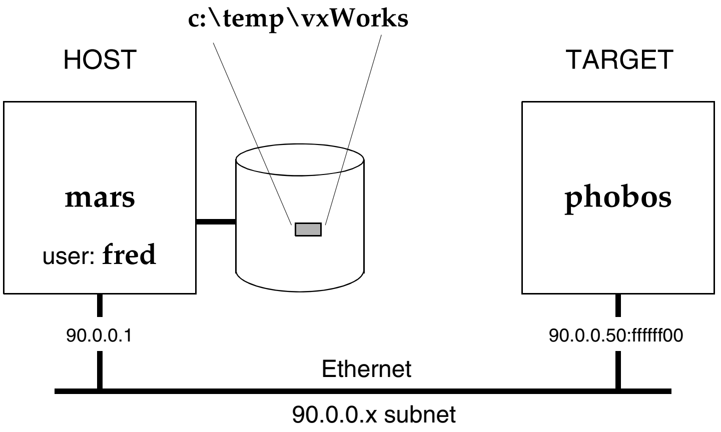
A display similar to the following appears. Note that the **p** command does not actually display unassigned parameters, although this example shows them for completeness.

```
boot device          : ln
unit number          : 0
processor number     : 0
host name            : mars
file name            : c:\tmp\vxWorks
```

```
inet on ethernet (e)    : 90.0.0.50:ffffff00
inet on backplane (b)   :
host inet (h)           : 90.0.0.1
gateway inet (g)        :
user (u)                : fred
ftp password (pw)(blank=use rsh) :secret
flags (f)               : 0x0
target name (tn)        : phobos
startup script (s)      :
other (o)               :
```

This example corresponds to the configuration illustrated in Figure 3-1.

Figure 3-1   **Boot Configuration Example**



3.5.2 Description of Boot Loader Parameters

Each of the boot loader parameters is described below, with reference to the example shown above.

The letters in parentheses after some of the parameters are alternative names used with the *single-string* interactive configuration method described in *3.5.3 Changing Boot Loader Parameters Interactively*, p. 144, and the static configuration method described in *3.7.3 Configuring Boot Loader Parameters Statically*, p. 147.

boot device

The type of device from which to boot. This must be one of the drivers included in the boot loader (for example, **enp** for a CMC controller). Due to limited space in boot media, only a few drivers can be included. A list of the drivers included in the boot loader image can be displayed in the boot loader shell with the **devs** or **h** command. For more information about boot devices, see *3.7.5 Selecting a Boot Device*, p.148.

unit number
The unit number of the boot device, starting at zero.

processor number
A unique numerical target identifier for systems with multiple targets on a backplane. The backplane master must have its processor number set to zero. For boards not connected to a backplane, a value of zero is typically used but is not required.

host name
The name of the host machine to boot from. This is the name by which the host is known to VxWorks; it need not be the name used by the host itself. (The host name is **mars** in the example above.)

file name
The full path name of the VxWorks image to be booted (**c:\myProj\vxWorks** in the example). This path name is also reported to the host when you start a target server, so that it can locate the host-resident image of VxWorks. The path name is limited to a 160 byte string, including the null terminator.

inet on ethernet (e)
The Internet Protocol (IP) address of a target system Ethernet interface, as well as the subnet mask used for that interface. The address consists of the IP address, in dot decimal format, followed by a colon, followed by the mask in hex format (here, 90.0.0.50:ffffff00).

inet on backplane (b)
The Internet address of a target system with a backplane interface (blank in the example).

host inet (h)
The Internet address of the host to boot from (90.0.0.1 in the example).

gateway inet (g)
The Internet address of a gateway node for the target if the host is not on the same network as the target (blank in the example).

user (u)
The user ID used to access the host for the purpose of loading the VxWorks image file (which is **fred** in the example). The user must have host permission to read the VxWorks image file.

On a Windows host, the user specified with this parameter must have FTP access to the host, and the **ftp password** parameter (below) must be used to provide the associated password.

On a UNIX (Linux or Solaris) host, the user must have FTP, TFTP, or **rsh** access. For **rsh**, the user must be granted access by adding the user ID to the host's **/etc/host.equiv** file, or more typically to the user's **.rhosts** file (~*userName*/**.rhosts**).

ftp password (pw)
For FTP or TFTP access, this field is used for the password for the user identified with the **user** parameter (above). For **rsh** access it should be left blank.

NOTE: If this parameter is not used, the boot loader attempts to load the run-time system image using a protocol based on the UNIX **rsh** utility, which is not available for Windows hosts.

flags (f)
Configuration options specified as a numeric value that is the sum of the values of selected option bits defined below. (This field is zero in the example because no special boot options were selected.)

0x01 = Do not enable the system controller, even if the processor number is 0. (This option is board specific; refer to your target documentation.)
0x02 = Load all VxWorks symbols[a], instead of just globals.
0x04 = Do not auto-boot.
0x08 = Auto-boot fast (short countdown).
0x20 = Disable login security.
0x80 = Use TFTP to get boot image.
0x400 = Set system to debug mode for the error detection and reporting facility (depending on whether you are working on kernel modules or user applications, for more information see *11. Error Detection and Reporting*.

[a] Loading a very large group of symbol can cause delays of up to several minutes while Workbench loads the symbols. For information about how to specify the size of the symbol batch to load, see the *Wind River Workbench User's Guide*.

target name (tn)
The name of the target system to be added to the host table (in the example, **phobos**).

startup script (s)
If the kernel shell is included in the downloaded image, this parameter allows you to pass to it the path and filename of a startup script to execute after the

system boots. A startup script file can contain only the shell's C interpreter commands. (Note that you must not add the **INCLUDE_SHELL**, **INCLUDE_WDB_BANNER**, or **INCLUDE_SIMPLE_BANNER** components to a boot loader. These components conflict with the boot loader shell. Doing so causes project configuration errors.)

This parameter can also be used to specify process-based (RTP) applications to run automatically at boot time, if VxWorks has been configured with the appropriate components. See *VxWorks Application Programmer's Guide: Applications and Processes*.

other (o)

This parameter is generally unused and available for applications (blank in the example). It can be used, for example, for specifying the default network interface when booting from a file system device. For more information, see *3.7.4 Enabling Networking for Non-Boot Interfaces*, p.148.

3.5.3 Changing Boot Loader Parameters Interactively

Boot parameters can be entered interactively from the boot loader prompt, either individually or as a string.

NOTE: Interactively-defined boot parameters take precedence over statically-defined parameters for targets that store those changes. Most targets store interactively-defined boot settings in a non-volatile device (or a text file for Pentium BSPs), and do not use the statically-defined values for subsequent reboots.

For information about changing boot parameters statically, see *3.7.3 Configuring Boot Loader Parameters Statically*, p.147.

Changing Parameters Individually

To change parameters on an individual basis, first use the **c** (change) command at the boot prompt, and then enter a new value for each parameter as it is displayed.

If a particular field already has the correct value, simply press **ENTER**. To clear a field, type a period (**.**), then press **ENTER**. To go back to change the previous parameter, type a dash (**-**), then press **ENTER**. If you want to quit before completing all parameters (but saving your changes), press **CTRL+D**.

Changing Multiple Parameters Simultaneously

To change more than one parameter at time, use the **$** boot command at the boot prompt with a parameter string. The syntax is as follows:

```
$dev(0,procnum)host:/file h=# e=# b=# g=# u=usr [pw=passwd] f=# tn=targetname s=script o=other
```

For example:

```
[VxWorks Boot]:$ln(0,0)mars:c:\myProj\vxWorks e=90.0.0.50 h=90.0.0.1 u=fred pw=…
```

The order of the parameters with assignments (those with equal signs) is not important. Omit any assigned fields that are irrelevant. The codes for the assigned fields correspond to the letter codes shown in parentheses by the **p** command and in *3.5.2 Description of Boot Loader Parameters*, p.141.

This method can be particularly useful when booting a target from a host script.

The changes made to boot parameters are retained between reboots for most types of targets (in a non-volatile device or on disk).

3.6 Rebooting VxWorks

When VxWorks is running, any of the following means can be used to reboot it:

- Enter **CTRL+X** in the terminal window.

  Note that some Windows terminal emulators do not pass **CTRL+X** to the target, because of its standard Windows meaning.

- Invoke **reboot( )** from the shell.
- Press the reset button on the target system.
- Turn the target's power off and on.

When you reboot VxWorks in any of these ways, the auto-boot sequence begins again from the countdown.

3.7 Customizing and Building Boot Loaders

A boot loader can be configured with just those VxWorks components that are required for a given system. Boot loaders may need to be customized and rebuilt for a variety of reasons, including the following:

- The target is not on a network.
- The boot loader must use a device or protocol that is not included in the default boot loader image.
- An alternate boot method is preferable (such as booting over the Target Server File System).
- To statically re-define the default boot loader parameters. When the default boot loader image is used with a system that does not have a non-volatile device or disk, the boot loader parameters have to be re-entered manually each time the target is rebooted. (Note that Pentium boot loaders automatically write boot parameter changes back to disk.)
- To create boot loaders for production systems. In addition to setting the boot loader parameters appropriately, features that are not required (such as the network stack) can be removed to reduce the size of the boot loader image.

3.7.1 Configuring Boot Loaders

For most BSPs, boot loaders can be configured and built with Wind River Workbench or the command-line project tool **vxprj**, using the **PROFILE_BOOTAPP** configuration profile.

For some BSPs, the legacy method using *bspDir*/**config.h** and *bspDir*/**make** must be used. Note that the legacy method has been deprecated for most purposes, and cannot be used for multiprocessor development. For information about this method, see the *VxWorks Command-Line Tools User's Guide*.

3.7.2 Boot Loader Components

The **INCLUDE_BOOT_APP** component provides the basic facility for loading and executing a VxWorks image.

The **PROFILE_BOOTAPP** configuration profile can be used with Workbench or **vxprj** to create a boot loader (including **INCLUDE_BOOT_APP**). This profile includes a basic set of boot loader components, such as those for the boot loader shell, drivers, file systems, and so on. Among the components that **PROFILE_BOOTAPP** provides are those for loading and executing a VxWorks image, for booting from a network with various protocols, and for booting from various file systems.

A boot loader needs to be configured appropriately for any device or file system from which you want to boot. Other components, which can be used to create a boot loader for specific boot environments and with various boot management facilities, are described throughout this chapter.

3.7.3 Configuring Boot Loader Parameters Statically

Boot loader parameters include the boot device, IP addresses of the host and target systems, the location of VxWorks image file, and so on. For detailed information about the parameters, see *3.5 Boot Loader Parameters*, p. 140. (For information about configuring boot loader parameters dynamically, see *3.5.3 Changing Boot Loader Parameters Interactively*, p. 144.)

Using Workbench, the **DEFAULT_BOOT_LINE** configuration parameter of the **INCLUDE_BSP_MACROS** component can be used to change the default boot loader parameters.

Using the legacy **config.h** method (which should only be used for BSPs that do not support **PROFILE_BOOTAPP**), edit the **DEFAULT_BOOT_LINE** macro in *installDir*/**vxworks-6.***x***/target/config/***bspName*/**config.h** file to change the default boot loader parameters. The **DEFAULT_BOOT_LINE** macro for a Pentium BSP looks like the following:

```
#define DEFAULT_BOOT_LINE \
    "fd=0,0(0,0)host:/fd0/vxWorks.st h=90.0.0.3 e=90.0.0.50 u=target"
```

For more information about configuration methods, see *3.7.1 Configuring Boot Loaders*, p. 146.

NOTE: Interactively-defined boot parameters take precedence over statically-defined parameters for targets that store those changes. Most targets store interactively-defined boot settings in a non-volatile device (or a text file for Pentium BSPs), and do not use the statically-defined values for subsequent reboots.

3.7.4 Enabling Networking for Non-Boot Interfaces

The **other (o)** boot loader parameter can be used to specify a network interface in addition to whatever device is specified for the boot device. For example, it can be used when booting from a local SCSI disk to specify a network interface to be included. The following example illustrates parameter settings for booting from a SCSI device, and enabling the network with an on-board Ethernet device (here with **ln** for LANCE Ethernet device) with the **other** field.

```
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
```

3.7.5 Selecting a Boot Device

The boot devices that are included in a boot loader image can be identified at run-time with the **devs** or **h** command from the boot loader shell (see *3.4.1 Boot Loader Shell Commands*, p.136).

Boot Device Configuration

In order to boot VxWorks, however, the boot loader must be configured with the appropriate device or devices for your target hardware and desired boot options—they may not be provided by the default boot loader. The process of configuring the boot loader with devices is the same as for VxWorks itself, and the topic of device configuration is covered in *2.4.3 Device Driver Selection*, p.22.

Boot Device Specification

Once a boot loader has been configured with the appropriate boot device (or devices), it must also be instructed as to which device to use. This can be done interactively or statically.

For information about interactive specification using the **boot device** parameter, see *3.5.2 Description of Boot Loader Parameters*, p.141 and *3.5.3 Changing Boot Loader Parameters Interactively*, p.144. For information about static configuration, see *3.7.3 Configuring Boot Loader Parameters Statically*, p.147.

The boot devices that are supported by a given BSP are described in the BSP reference. The syntax used for specifying them with the **boot device** boot loader parameter is provided below.

ATA Device

The syntax for specifying an ATA device is as follows:

ata=*controllerNum*, *driveNum*

where *controllerNum* is the controller number (either 0 or 1) and *driveNum* is the drive number (either 0 or 1). For example:

```
ata=0,0
```

PCMCIA Device

The syntax for specifying a PCMCIA device is as follows:

pcmcia=*socket*

where *socket* is the PCMCIA socket into which the device is plugged.

SCSI Device

The syntax for specifying a SCSI device is as follows:

scsi=*id*,*lun*

where *id* is the SCSI ID of the boot device, and *lun* is its Logical Unit Number (LUN).

TSFS Device

The syntax for specifying a Target Server File System device is simply **tsfs**. No additional boot device arguments are required. The file path and name must be relative to the root of the host file system as defined for the target server on the host. For information about the TSFS, see *8.9 Target Server File System: TSFS*, p.518.

3.7.6 Reconfiguring Memory

The VxWorks boot loader can be customized to meet the size constraints of the non-volatile device on a particular board, as well as the manner in which it retrieves the VxWorks image file.

Persistent Memory Region

The persistent memory region is an area of RAM at the top of system memory specifically reserved for error records and core dumps. For more information about use of persistent memory, see *11.2.2 Configuring the Persistent Memory Region*, p.565.

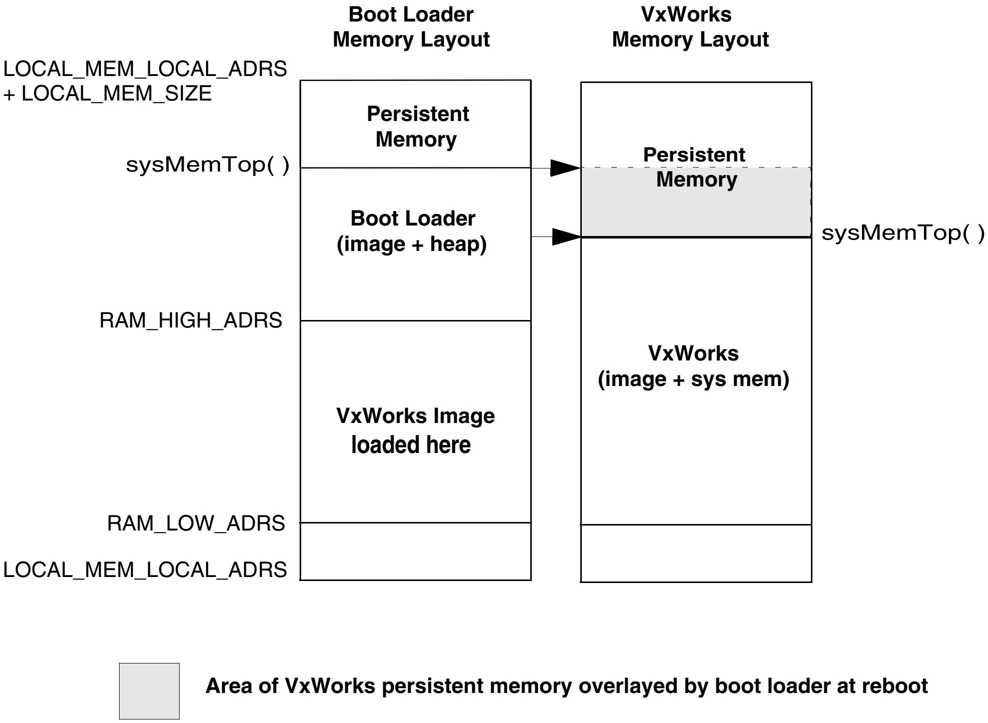
If you increase the size of the persistent memory region for VxWorks beyond the default, you must also create and install a new boot loader with the same **PM_RESERVED_MEM** value.

If you do not, the boot loader (image plus heap) overlays the area of VxWorks persistent memory that extends beyond its own when the system reboots, and any data that may have been stored in the overlapping area will be corrupted. For a simple illustration of this problem, see Figure 3-2.

Note that when you change the value of the **PM_RESERVED_MEM** for the boot loader, you need to change the value of **RAM_HIGH_ADRS** if there is insufficient room for the boot loader itself between **RAM_HIGH_ADRS** and **sysMemTop( )**. If you do so, also be sure that there is sufficient room for the VxWorks image between **RAM_LOW_ADRS** and **RAM_HIGH_ADRS**.

WARNING: Not properly configuring the boot loader (as described above) could corrupt the persistent memory region when the system boots.

Figure 3-2 **RAM Layout Snapshots**


Boot Loader Memory Layout
VxWorks Memory Layout
LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE
Persistent Memory
sysMemTop( )
Boot Loader (image + heap)
RAM_HIGH_ADRS
VxWorks Image loaded here
RAM_LOW_ADRS
LOCAL_MEM_LOCAL_ADRS
Persistent Memory
sysMemTop( )
VxWorks (image + sys mem)
Area of VxWorks persistent memory overlayed by boot loader at reboot

3.7.7 Building Boot Loaders

For boot loaders configured with Workbench or **vxprj**, and the **PROFILE_BOOTAPP** configuration profile, building them is the same as for any other VxWorks image project.

If you have configured a boot loader with the *bspDir*/**config.h** method (which should only be used for BSPs that do not support **PROFILE_BOOTAPP**), use the command **make** *bootLoaderType* in the *installDir*/**vxworks-6.***x*/**target/config/***bspName* directory. For example:

```
% make bootrom
```

The different types of boot loader images that you can build are described in *3.3 Boot Loader Image Types*, p.133.

For information about configuration methods, see *3.7.1 Configuring Boot Loaders*, p.146.

⚠ **CAUTION:** Do not build boot loaders for symmetric multiprocessor (SMP) and asymmetric multiprocessor (AMP) configurations of VxWorks with the SMP or AMP build option—neither with Workbench nor with **vxprj**. The same boot loaders are used for uniprocessor (UP), SMP, and AMP, configurations of VxWorks.

3.8 Installing Boot Loaders

For information about installing boot loaders in various media, see the VxWorks BSP references.

3.9 Booting From a Network

In order to boot from a network, the boot loader must be configured with the appropriate components for the networking protocol and devices, and boot parameters must be set accordingly as well.

For information about boot devices, see *3.7.5 Selecting a Boot Device*, p.148.

Network Protocol Components

The ability to boot over a network is provided with the **INCLUDE_BOOT_NETWORK** component, which registers a network boot loader with the basic loader facility, checks for the network device, and calls the network boot loader for a specific protocol. Support for different protocols is as follows:

INCLUDE_BOOT_FTP_LOADER
FTP boot loader.

INCLUDE_BOOT_TFTP_LOADER
TFTP boot loader.

INCLUDE_BOOT_RSH_LOADER
RSH boot loader.

Parameters for Booting From a Network

The parameters and settings specific to booting from a network with a give protocol are described below.

For general about boot parameters and how to set them, see *3.5.2 Description of Boot Loader Parameters*, p. 141, *3.5.3 Changing Boot Loader Parameters Interactively*, p. 144, and *3.7.3 Configuring Boot Loader Parameters Statically*, p. 147.

FTP

The **user** and **ftp password** boot parameters must be set to match account settings with the FTP server on the host.

TFTP

The **flags** boot parameter must be set to **0x80**, and the **user** and **ftp password** parameters must be set match account settings with the TFTP server on the host.

RSH

The **ftp password** parameter must be set to empty, that is by entering **.** (a period).

Updating Ethernet MAC Settings

The **INCLUDE_BOOT_ETH_MAC_HANDLER** provides the **M** boot loader shell command, which can be used to update (and display) the Ethernet MAC address for the target system. For more information in this regard, see *3.4.1 Boot Loader Shell Commands*, p. 136.

3.10 Booting From a Target File System

In order to boot from a file system, the boot loader must be configured with the appropriate components and devices, and boot parameters must be set accordingly as well.

For information about boot devices, see *3.7.5 Selecting a Boot Device*, p. 148.

For information about booting from TSFS, see *3.11 Booting From the Host File System Using TSFS*, p. 155.

File System Components

The ability to boot from file systems is provided with the **INCLUDE_BOOT_FILESYSTEMS** component, in conjunction with a device-specific file system loader component. The options for file system loaders are as follows:

INCLUDE_BOOT_ATA_LOADER
Support for ATA devices.

INCLUDE_BOOT_FD_LOADER
Support for floppy-disk devices.

Parameters for Booting From a Target File System

The parameters specific to booting from a target file system are described below.

For general about boot parameters and how to set them, see *3.5.2 Description of Boot Loader Parameters*, p. 141, *3.5.3 Changing Boot Loader Parameters Interactively*, p. 144, and *3.7.3 Configuring Boot Loader Parameters Statically*, p. 147.

ATA

The **boot device** boot loader parameter must be set to **ata**.

FD—Floppy Disk

The **boot device** boot loader parameter must be set to **fd**.

3.11 Booting From the Host File System Using TSFS

The simplest way to boot a target from a host without a network is to use the Target Server File System. This does not involve configuring SLIP or PPP. The TSFS can be used to boot a target connected to the host by one or two serial lines. Configure VxWorks with the **INCLUDE_TSFS_BOOT** component.

WARNING: The TSFS boot facility is not compatible with WDB agent network configurations. For information about WDB, see *12.6 WDB Target Agent*, p.626.

To configure a boot loader for TSFS, the boot device parameter must be **tsfs**, and the file path and name must be relative to the root of the host file system defined for the target server.

Regardless of how you specify the boot line parameters, you must reconfigure (as described below) and rebuild the boot image.

If two serial lines connect the host and target (one for the target console and one for WDB communications), the following configuration parameters must be set:

- **CONSOLE_TTY**
- **CONSOLE_TTY 0**
- **WDB_TTY_CHANNEL**
- **WDB_TTY_CHANNEL 1**
- **WDB_COMM_TYPE**
- **WDB_COMM_TYPE WDB_COMM_SERIAL**

If one serial line connects the host and target, the following configuration parameters must be set:

- **CONSOLE_TTY**
- **CONSOLE_TTY NONE**
- **WDB_TTY_CHANNEL**
- **WDB_TTY_CHANNEL 0**
- **WDB_COMM_TYPE**
- **WDB_COMM_TYPE WDB_COMM_SERIAL**

With any of these TSFS configurations, you can also use the target server console to set the boot loader parameters by including the **INCLUDE_TSFS_BOOT_VIO_CONSOLE** component in VxWorks. This disables the auto-boot mechanism, which might otherwise boot the target before the target server could start its virtual I/O mechanism. (The auto-boot mechanism is similarly disabled when **CONSOLE_TTY** is set to **NONE**, or when **CONSOLE_TTY** is set to **WDB_TTY_CHANNEL**.) Using the target server console is particularly useful for a single serial connection, as it provides an otherwise unavailable means of changing boot loader parameters from the command line.

When you build the boot image, select **bootrom.hex** for the image type (see *3.7.7 Building Boot Loaders*, p.151).

For more information about the TSFS, see the *8.9 Target Server File System: TSFS*, p.518.

4
Multitasking

4.1 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.

Tasks are the basic unit of scheduling in VxWorks. All tasks, whether in the kernel or in processes, are subject to the same scheduler. VxWorks processes are not themselves scheduled.

Intertask communication facilities allow tasks to synchronize and communicate in order to coordinate their activity. In VxWorks, the intertask communication facilities include semaphores, message queues, message channels, pipes, network-transparent sockets, and signals.

For interprocess communication, VxWorks semaphores and message queues, pipes, and events (as well as POSIX semaphores and events) can be created as *public* objects to provide accessibility across memory boundaries (between the kernel and processes, and between different processes). In addition, message channels provide a socket-based inter-processor and inter-process communications mechanism.

Hardware interrupt handling is a key facility in real-time systems 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 any task's context.

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. For information about POSIX timers, see *5.6 POSIX Clocks and Timers*, p.259.

This chapter discusses the tasking, intertask communication, and interprocess communication facilities that are at the heart of the VxWorks run-time environment.

For information about POSIX support for VxWorks, see *5. POSIX Facilities*.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

NOTE: This chapter provides information about multitasking facilities that are common to both uniprocessor (UP) and symmetric multiprocessor (SMP) configurations of VxWorks. It also provides information about those facilities that are specific to the UP configuration. In the latter case, the alternatives available for SMP systems are noted.

With few exceptions, the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few routines. Also note that some programming practices—such as implicit synchronization techniques relying on task priority instead of explicit locking—are not appropriate for an SMP system.

For information about SMP programming, see *15. VxWorks SMP*. For information specifically about migration, see *15.15 Migrating Code to VxWorks SMP*, p.702.

4.2 Tasks and Multitasking

VxWorks tasks are the basic unit of code execution in the operating system itself, as well as in applications that it executes as processes. In other operating systems the term *thread* is used similarly. (For information about VxWorks support for POSIX threads, see *5.10 POSIX Threads*, p.264).

Multitasking provides the fundamental mechanism for an application to control and react to multiple, discrete real-world events. The VxWorks real-time kernel 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 policy.

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 tasks' virtual memory context (if process support is included)
- the CPU registers and (optionally) coprocessor registers

- stacks for dynamic variables and function calls
- I/O assignments for standard input, output, and error
- a delay timer
- a time-slice timer
- kernel control structures
- signal handlers
- task private environment (for environment variables)
- error status (errno)
- debugging and performance monitoring values

If VxWorks is configured without process support (the **INCLUDE_RTP** component), the context of a task does not include its virtual memory context. All tasks can only run in a single common address space (the kernel).

However, if VxWorks is configured with process support—regardless of whether or not any processes are active—the context of a kernel task does include its virtual memory context, because the system has the potential to operate with other virtual memory contexts besides the kernel. That is, the system could have tasks running in several different virtual memory contexts (the kernel and one or more processes).

For information about virtual memory contexts, see *6. Memory Management*.

NOTE: The POSIX standard includes the concept of a thread, which is similar to a task, but with some additional features. For details, see *5.10 POSIX Threads*, p.264.

4.2.1 Task States and Transitions

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

Table 4-1 describes the *state symbols* that you see when working with development tools. Example 4-1 shows output from the **i( )** command containing task state information.

Table 4-1 **Task State Symbols**

State Symbol	Description
READY	The task is not waiting for any resource other than the CPU.
PEND	The task is blocked due to the unavailability of some resource.
DELAY	The task is asleep for some duration.
SUSPEND	The task is unavailable for execution (but not pended or delayed). This state is used primarily for debugging. Suspension does not inhibit state transition, only execution. Thus, pended-suspended tasks can still unblock and delayed-suspended tasks can still awaken.
STOP	The task is stopped by the debugger.
DELAY + S	The task is both delayed and suspended.
PEND + S	The task is both pended and suspended.
PEND + T	The a task is pended with a timeout value.
STOP + P	Task is pended and stopped (by the debugger, error detection and reporting facilities, or **SIGSTOP** signal).
STOP + S	Task is stopped by (by the debugger, error detection and reporting facilities, or **SIGSTOP** signal) and suspended.
STOP + T	Task is delayed and stopped (by the debugger, error detection and reporting facilities, or **SIGSTOP** signal).
PEND + S + T	The task is pended with a timeout value and suspended.
STOP +P + S	Task is pended, suspended and stopped by the debugger.
STOP + P + T	Task pended with a timeout and stopped by the debugger.
STOP +T + S	Task is suspended, delayed, and stopped by the debugger.
ST+P+S+T	Task is pended with a timeout, suspended, and stopped by the debugger.
state **+ I**	The task is specified by *state* (any state or combination of states listed above), plus an inherited priority.

The **STOP** state is used by the debugger facilities when a breakpoint is hit. It is also used by the error detection and reporting facilities when an error condition occurs (see *11. Error Detection and Reporting*).

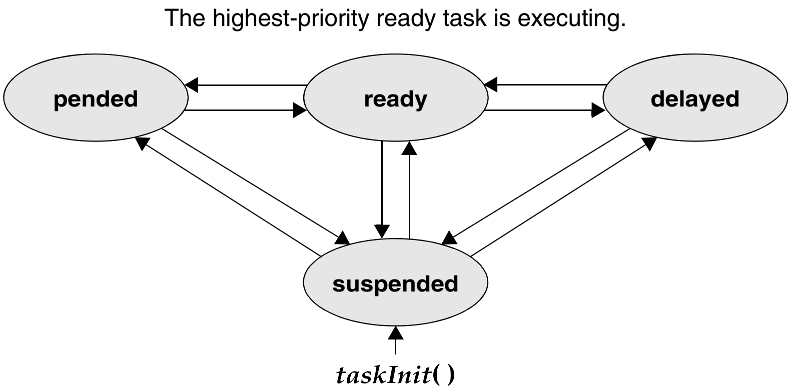
Example 4-1 **Task States in Shell Command Output**

```
-> i

  NAME         ENTRY       TID    PRI   STATUS      PC       SP     ERRNO  DELAY
----------  ------------ -------- --- ---------- -------- -------- ------- -----
tIsr0       42cb40        25b1f74   0 PEND         3bcf54  25b1f2c       0     0
tJobTask    3732d0        25b5140   0 PEND         3bcf54  25b50e8       0     0
tExcTask    372850         4f033c   0 PEND         3bcf54   4ef0f8       0     0
tLogTask    logTask       25b7754   0 PEND         3bb757  25b7670       0     0
tNbioLog    373f28        25bae18   0 PEND         3bcf54  25bad6c       0     0
tShell0     shellTask     2fbdcb4   1 READY        3c2bdc  2fbc0d4       0     0
tWdbTask    wdbTask       2faca28   3 PEND         3bcf54  2fac974       0     0
tErfTask    42e0a0        25bd0a4  10 PEND         3bd3be  25bd03c       0     0
tXbdServic> 36e4b4        25ac3d0  50 PEND+T       3bd3be  25ac36c  3d0004     6
tNet0       ipcomNetTask  25cdb00  50 PEND         3bcf54  25cda88       0     0
ipcom_sysl> 3cba50        27fec0c  50 PEND         3bd3be  27feab0       0     0
ipnetd      3e2170        2fa6d10  50 PEND         3bcf54  2fa6c98  3d0004     0
ipcom_teln> ipcom_telnet  2fa979c  50 PEND         3bcf54  2fa9594       0     0
miiBusMoni> 429420        25a8010 254 DELAY        3c162d  25a7fd0       0    93
value = 0 = 0x0
```

Figure 4-1 illustrates task state transitions for a deployed system—without the **STOP** state associated with development activity. The routines listed are examples of ones that would cause the associated transition. For example, a task that called **taskDelay( )** would move from the ready state to the delayed state.

Figure 4-1 **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( )**

4.3 Task Scheduling

Multitasking requires a task scheduler to allocate the CPU to ready tasks. VxWorks provides the following scheduler options:

- The traditional VxWorks scheduler, which provides priority-based, preemptive scheduling, as well as a round-robin extension. See *4.3.3 VxWorks Traditional Scheduler*, p. 168.

- The VxWorks POSIX threads scheduler, which is designed (and required) for running pthreads in processes (RTPs). See *5.12 POSIX and VxWorks Scheduling*, p. 277.)

- A custom scheduler framework, which allows you to develop your own scheduler. See *2.10 Custom Scheduler*, p. 118.

4.3.1 Task Priorities

Task scheduling relies on a task's priority. The VxWorks kernel provides 256 priority levels, numbered 0 through 255. Priority 0 is the highest and priority 255 is the lowest.

A task is assigned its priority at creation, but you can also change it programmatically thereafter. For information about priority assignment, see *4.4.1 Task Creation and Activation*, p. 172 and *4.3.2 Task Scheduling Control*, p. 167).

Application Task Priorities

All application tasks should be in the priority range from 100 to 255.

Driver Task Priorities

In contrast to application tasks, which should be in the task priority range from 100 to 255, driver *support* tasks (which are associated with an ISR) can be in the range of 51-99.

These tasks are crucial; for example, if a support task fails while copying data from a chip, the device loses that data. Examples of driver support tasks include **tNet0** (the VxWorks network daemon task), an HDLC task, and so on.

The system **tNet0** has a priority of 50, so user tasks should not be assigned priorities below that task; if they are, the network connection could die and prevent debugging capabilities with the host tools.

4.3.2 Task Scheduling Control

The routines that control task scheduling are listed in Table 4-2.

Table 4-2 **Task Scheduling Control Routines**

Routine	Description
kernelTimeSlice( )	Controls round-robin scheduling. See *Round-Robin Scheduling*, p. 169.
taskRotate( )	Rotates tasks on the ready queue.
taskPrioritySet( )	Changes the priority of a task.
taskLock( )	Disables task rescheduling.
taskUnlock( )	Enables task rescheduling.

The **taskRotate( )**routine can be used as an alternative to round-robin scheduling. It allows a program to control sharing of the CPU amongst tasks of the same priority that are ready to run, rather than having the system do so at predetermined equal intervals. For information about round-robin scheduling, see *Round-Robin Scheduling*, p. 169.

Task Priority

Tasks are assigned a priority when they are created (see *4.4.1 Task Creation and Activation*, p. 172). You can change a task's priority level while it is executing by calling **taskPrioritySet( )**. The ability to change task priorities dynamically allows applications to track precedence changes in the real world.

Preemption Locks

The scheduler can be explicitly disabled and enabled on a per-task basis in the kernel 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.

If the task that has disabled the scheduler with **taskLock( )** 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: The **taskLock( )** and **taskUnlock( )** routines are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including task-only spinlocks, which default to **taskLock( )** and **taskUnlock( )** behavior in a UP system. For more information, see *15.6.2 Task-Only Spinlocks*, p.682 and *15.15 Migrating Code to VxWorks SMP*, p.702.

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 *4.11 Mutual Exclusion*, p.196.

A Comparison of taskLock( ) and intLock( )

When using **taskLock( )**, consider that it will not achieve mutual exclusion. Generally, if interrupted by hardware, the system will eventually return to your task. However, if you block, you lose task lockout. Thus, before you return from the routine, **taskUnlock( )** should be called.

When a task is accessing a variable or data structure that is also accessed by an ISR, you can use **intLock( )** to achieve mutual exclusion. Using **intLock( )** makes the operation *atomic* in a single processor environment. It is best if the operation is kept minimal, meaning a few lines of code and no function calls. If the call is too long, it can directly impact interrupt latency and cause the system to become far less deterministic.

For information about interrupts, see *4.11.1 Interrupt Locks and Latency*, p.197 and *4.20 Interrupt Service Routines*, p.241.

4.3.3 VxWorks Traditional Scheduler

The VxWorks traditional scheduler provides priority-based preemptive scheduling as well as the option of programmatically initiating round-robin scheduling. The traditional scheduler may also be referred to as the *original* or *native* scheduler.

The traditional scheduler is included in VxWorks by default with the **INCLUDE_VX_TRADITIONAL_SCHEDULER** component.

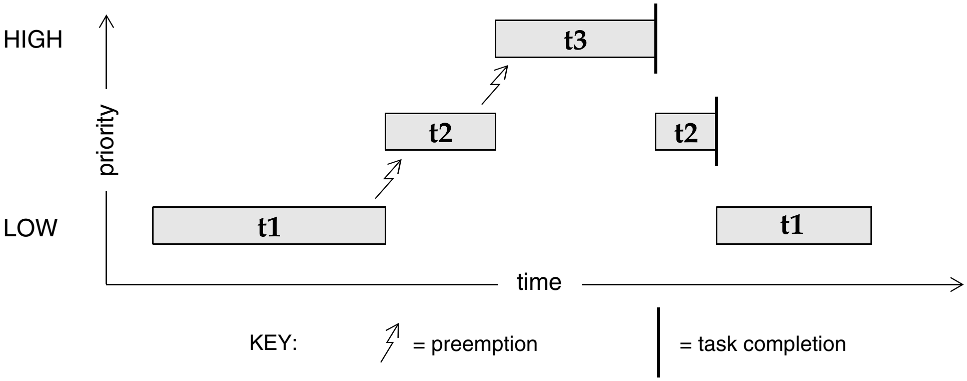
For information about the POSIX thread scheduler and custom schedulers, see *5.12 POSIX and VxWorks Scheduling*, p.277 and *2.10 Custom Scheduler*, p.118, respectively.

Priority-Based Preemptive Scheduling

A priority-based preemptive scheduler *preempts* the CPU when a task has a higher priority than the current task running. Thus, the kernel ensures that the CPU is always allocated to the highest priority task that is ready to run. This means that if a task—with a higher priority than that of 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. For example, in Figure 4-2, 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 disadvantage of this scheduling policy is that, when multiple tasks of equal priority must share the processor, if a single task is never blocked, it can usurp the processor. Thus, other equal-priority tasks are never given a chance to run. Round-robin scheduling solves this problem.

Figure 4-2 **Priority Preemption**



Round-Robin Scheduling

VxWorks provides a round-robin extension to priority-based preemptive scheduling. Round-robin scheduling accommodates instances in which there are more than one task of a given priority that is ready to run. The round-robin algorithm attempts to share the CPU amongst these tasks by using *time-slicing*. Each task in a group of tasks with the same priority executes for a defined interval, or time slice, before relinquishing the CPU to the next task in the group. No one of them, therefore, can usurp the processor until it is blocked. See Figure 4-3 for an illustration of this activity.

Note that while round-robin scheduling is used in some operating systems to provide equal CPU time to all tasks (or processes), regardless of their priority, this is not the case with VxWorks. Priority-based preemption is essentially unaffected by the VxWorks implementation of round-robin scheduling. Any higher-priority task that is ready to run immediately gets the CPU, regardless of whether or not the current task is done with its slice of execution time. When the interrupted task gets to run again, it simply continues using its unfinished execution time.

In most systems, it is not necessary to enable round-robin scheduling, the exception being when multiple copies of the same code are to be run, such as in a user interface task.

Note that the **taskRotate( )**routine can be used as an alternative to round-robin scheduling. It is useful for situations in which you want to share the CPU amongst tasks of the same priority that are ready to run, but to do so as a program requires, rather than at predetermined equal intervals.

Enabling Round-Robin Scheduling

Round-robin scheduling is enabled by calling **kernelTimeSlice( )**, which takes a parameter for a time slice, or interval.

Time-slice Counts and Preemption

The time-slice or interval defined with a **kernelTimeSlice( )** call is the amount of time that each task is allowed to run before relinquishing the processor to another equal-priority task. Thus, the tasks rotate, each executing for an equal interval of time. No task gets a second slice of time before all other tasks in the priority group have been allowed to run.

If round-robin scheduling is enabled, and preemption is enabled for the executing task, the system tick handler increments the task's time-slice count. When the specified time-slice interval is completed, the system tick handler clears the counter and the task is placed at the tail of the list of tasks at its priority level. New tasks joining a given priority group are placed at the tail of the group with their run-time counter initialized to zero.

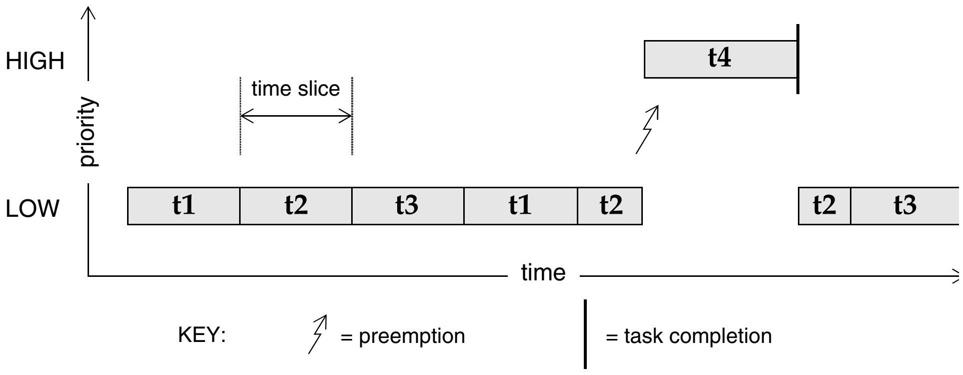
Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.

If a task blocks or is preempted by a higher priority task during its interval, its time-slice count is saved and then restored when the task becomes eligible for execution. In the case of preemption, the task will resume execution once the higher priority task completes, assuming that no other task of a higher priority is ready to run. In the case where the task blocks, it is placed at the tail of the list of tasks at its priority level. If preemption is disabled during round-robin scheduling, the time-slice count of the executing task is not incremented.

Time-slice counts are accrued by the task that is executing when a system tick occurs, regardless of whether or not the task has executed for the entire tick interval. Due to preemption by higher priority tasks or ISRs stealing CPU time from the task, it is possible for a task to effectively execute for either more or less total CPU time than its allotted time slice.

Figure 4-3 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 4-3 **Round-Robin Scheduling**



4.4 Task Creation and Management

The following sections give an overview of the basic VxWorks task routines, which are found in the VxWorks library **taskLib**. These routines provide the means for task creation and control, as well as for retrieving information about tasks. See the VxWorks API reference for **taskLib** for further information.

For interactive use, you can control VxWorks tasks with the host tools or the kernel shell; see the *Wind River Workbench User's Guide*, the *VxWorks Command-Line Tools User's Guide*, and *VxWorks Kernel Programmer's Guide: Target Tools*.

4.4.1 Task Creation and Activation

The routines listed in Table 4-3 are used to create tasks.

The arguments to **taskSpawn( )** are the new task's name (an ASCII string), the task's priority, an *options* word, the stack size, the 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 );
```

Note that a task's priority can be changed after it has been spawned; see *4.3.2 Task Scheduling Control*, p.167.

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.

Table 4-3 **Task Creation Routines**

Call	Description
taskSpawn( )	Spawns (creates and activates) a new task.
taskCreate( )	Creates, but not activates a new task.
taskInit( )	Initializes a new task.
taskInitExcStk( )	Initializes a task with stacks at specified addresses.
taskOpen( )	Open a task (or optionally create one, if it does not exist).
taskActivate( )	Activates an initialized task.

The **taskOpen( )** routine provides a POSIX-like API for creating a task (with optional activation) or obtaining a *handle* on existing task. It also provides for creating a task as either a public or private object (see *4.4.4 Task Names and IDs*, p.177). The **taskOpen( )** routine is the most general purpose task-creation routine.

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

The difference between **taskInit( )** and **taskInitExcStk( )** is that the **taskInit( )** routine allows the specification of the execution stack address, while **taskInitExcStk( )** allows the specification of both the execution and exception stacks.

4.4.2 Task Creation Options

When a task is spawned, you can pass in one or more option parameters, which are listed in Table 4-4. The result is determined by performing a logical OR operation on the specified options.

Table 4-4 **Task Options**

Name	Description
VX_ALTIVEC_TASK	Execute with Altivec coprocessor support.
VX_DEALLOC_EXC_STACK	Deallocates the exception stack.
VX_DEALLOC_STACK	Deallocate stack.
VX_DSP_TASK	Execute with DSP coprocessor support.
VX_FP_TASK	Executes with the floating-point coprocessor.
VX_NO_STACK_FILL	Does not fill the stack with 0xEE.
VX_NO_STACK_PROTECT	Create without stack overflow or underflow guard zones (see *4.4.3 Task Stack*, p.175).
VX_PRIVATE_ENV	Executes a task with a private environment.
VX_TASK_NOACTIVATE	Used with **taskOpen( )** so that the task is not activated.
VX_UNBREAKABLE	Disables breakpoints for the task.

Floating Point Operations

You must include the **VX_FP_TASK** option when creating a task that does any of the following:

- Performs floating-point operations.
- Calls any function that returns a floating-point value.
- Calls any function that takes a floating-point value as an argument.

For example:

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

Some routines perform floating-point operations internally. The VxWorks documentation for each of these routines clearly states the need to use the **VX_FP_TASK** option.

Filling Task Stacks

Note that in addition to using the **VX_NO_STACK_FILL** task creation option for individual tasks, you can use the **VX_GLOBAL_NO_STACK_FILL** configuration parameter (when you configure VxWorks) to disable stack filling for all tasks and interrupts in the system.

By default, task and interrupt stacks are filled with 0xEE. Filling stacks is useful during development for debugging with the **checkStack( )** routine. It is generally not used in deployed systems because not filling stacks provides better performance during task creation (and at boot time for statically-initialized tasks).

After a task is spawned, you can examine or alter task options by using the routines listed in Table 4-5. Currently, only the **VX_UNBREAKABLE** option can be altered.

Table 4-5 **Task Option Routines**

Call	Description
taskOptionsGet( )	Examines task options.
taskOptionsSet( )	Sets task options.

4.4.3 Task Stack

The size of each task's stack is defined when the task is created (see *4.4.1 Task Creation and Activation*, p.172).

It can be difficult, however, to know exactly how much stack space to allocate. To help avoid stack overflow and corruption, you can initially allocated a stack that is much larger than you expect the task to require. Then monitor the stack periodically from the shell with **checkStack( )** or **ti( )**. When you have determined actual usage, adjust the stack size accordingly for testing and for the deployed system.

In addition to experimenting with task stack size, you can also configure and test systems with guard zone protection for task stacks (for more information, see *Task Stack Protection*, p.176).

Task Stack Protection

Task stacks can be protected with guard zones and by making task stacks non-executable.

Task Stack Guard Zones

Systems can be configured with the **INCLUDE_PROTECT_TASK_STACK** component to provide guard zone protection for task stacks. If memory usage becomes an issue, the component can be removed for final testing and the deployed system.

An overrun guard zone prevents a task from going beyond the end of its predefined stack size and corrupting data or other stacks. An under-run guard zone typically prevents buffer overflows from corrupting memory that precedes the base of the stack. The CPU generates an exception when a task attempts to access any of the guard zones. The size of a stack is always rounded up to a multiple of the MMU page size when either a guard zone is inserted or when the stack is made non-executable.

Note that guard zones cannot catch instances in which a buffer that causes an overflow is greater than the page size (although this is rare). For example, if the guard zone is one page of 4096 bytes, and the stack is near its end, and then a buffer of a 8000 bytes is allocated on the stack, the overflow will not be detected.

By default, kernel mode tasks do not have any task stack protection. Configuring VxWorks with the **INCLUDE_PROTECT_TASK_STACK** component provides underflow and overflow guard zones on the *execution stacks,* but none for the *exception stacks*. Stack guard zones in the kernel are mapped to physical memory.

Note that the protection provided for user-mode tasks by configuring the system with the **INCLUDE_RTP** component does not apply to kernel tasks (for information about user-mode task stack protection, see the *VxWorks Application Programmer's Guide: Multitasking*).

Note that the **INCLUDE_PROTECT_TASK_STACK** component does not provide stack protection for tasks that are created with the **VX_NO_STACK_PROTECT** task option (see *4.4.2 Task Creation Options*, p.173). If a task is created with this option, no guard zones are created for that task.

The size of the guard zones are defined by the following configuration parameters:

- **TASK_KERNEL_EXEC_STACK_OVERFLOW_SIZE** for kernel task execution stack overflow size.

- **TASK_KERNEL_EXEC_STACK_UNDERFLOW_SIZE** for kernel task execution stack underflow size.

The value of these parameters can be modified to increase the size of the guard zones on a system-wide basis. The size of a guard zone is rounded up to a multiple of the CPU MMU page size. The insertion of a guard zone can be prevented by setting the parameter to zero.

Stack guard zones in the kernel consume RAM, as guard zones correspond to mapped memory for which accesses are made invalid.

Non-Executable Task Stacks

VxWorks creates kernel task stacks with a non-executable attribute only if the system is configured with the **INCLUDE_TASK_STACK_NO_EXEC** component, and if the CPU supports making memory non-executable on an MMU-page basis. The size of a stack is always rounded up to a multiple of an MMU page size when the stack is made non-executable (as is also the case when guard zones are inserted).

4.4.4 Task Names and IDs

When a task is spawned, you can specify an ASCII string of any length to be the task name, and a task ID is returned.

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. In the kernel, the task ID is a 4-byte handle to the task's data structures.

The following rules and guidelines should be followed when naming tasks:

- The names of public tasks must be unique and must begin with a forward slash; for example **/tMyTask**. Note that public tasks are *visible* throughout the entire system—in the kernel and any processes.
- The names of private tasks should be unique. VxWorks does not require that private task names be unique, but it is preferable to use unique names to avoid confusing the user. (Note that private tasks are *visible* only within the entity in which they were created—either the kernel or a process.)

To use the host 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 any kernel task name started from the target with the letter **t**, and any task name started from the host with the letter **u**. In addition, the name of the initial task of a real-time process is the executable file name (less the extension) prefixed with the letter **i**.

Creating a task as a public object allows other tasks from outside of its process to send signals or events to it (with the **taskKill( )** or the **eventSend( )** routine, respectively).

For more information, see *4.9 Public and Private Objects*, p.194.

You do not have to explicitly name 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 is incremented by one for each unnamed task that is spawned.

The **taskLib** routines listed in Table 4-6 manage task IDs and names.

Table 4-6 **Task Name and ID Routines**

Call	Description
taskName( )	Gets the task name associated with a task ID (restricted to the context—process or kernel—in which it is called).
taskNameToId( )	Looks up the task ID associated with a task name.
taskIdSelf( )	Gets the calling task's ID.
taskIdVerify( )	Verifies the existence of a specified task.

4.4.5 Task Information

The routines listed in Table 4-7 get information about a task by taking a snapshot of a task's context when the routine is called. Because the task state is dynamic, the information may not be current unless the task is known to be dormant (that is, suspended).

Table 4-7 **Task Information Routines**

Call	Description
taskIdListGet( )	Fills an array with the IDs of all active tasks.
taskInfoGet( )	Gets information about a task.
taskPriorityGet( )	Examines the priority of a task.
taskRegsGet( )	Examines a task's registers (cannot use with current task).
taskRegsSet( )	Sets a task's registers (cannot be used with the current task).
taskIsSuspended( )	Checks whether a task is suspended.
taskIsReady( )	Checks whether a task is ready to run.
taskIsPended( )	Checks whether a task is pended.
taskTcb( )	Gets a pointer to a task's control block.

For information about task-specific variables and their use, see *4.7.3 Task-Specific Variables*, p.190.

4.4.6 Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in Table 4-8 to delete tasks and to protect tasks from unexpected deletion.

Table 4-8 **Task-Deletion Routines**

Call	Description
exit( )	Terminates the calling task and frees memory (task stacks and task control blocks only).[a]
taskDelete( )	Terminates a specified task and frees memory (task stacks and task control blocks only).[a] The calling task may terminate itself with this routine.
taskSafe( )	Protects the calling task from deletion.
taskUnsafe( )	Undoes a **taskSafe( )**, which makes calling task available for deletion.

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

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

Tasks implicitly call **exit( )** if the entry routine specified during task creation returns.

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*. Only the calling task is protected. 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 code */
.
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 *4.12.3 Mutual-Exclusion Semaphores*, p.205.

4.4.7 Task Execution Control

The routines listed in Table 4-9 provide direct control over a task's execution.

Table 4-9 **Task Execution Control Routines**

Call	Description
taskSuspend( )	Suspends a task.
taskResume( )	Resumes a task.
taskRestart( )	Restarts a task.
taskDelay( )	Delays a task; delay units are ticks, resolution in ticks.
nanosleep( )	Delays a task; delay units are nanoseconds, resolution in ticks.

Tasks may require restarting during execution in response to some catastrophic error. The restart mechanism, **taskRestart( )**, recreates a task with the original creation arguments.

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 *5.6 POSIX Clocks and Timers*, p.259.

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.

NOTE: ANSI and POSIX APIs are similar.

System clock resolution is typically 60Hz (60 times per second). This is a relatively long time for one clock tick, and would be even at 100Hz or 120Hz. Thus, since periodic delaying is effectively *polling*, you may want to consider using event-driven techniques as an alternative.

4.4.8 Tasking Extensions

To allow additional task-related facilities to be added to the system, VxWorks provides hook routines that 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 4-10; for more information, see the VxWorks API reference for **taskHookLib**.

Table 4-10 **Task Create, Switch, and Delete Hooks**

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

When using switch hook routines, be aware of the following restrictions:

- Do not assume any virtual memory (VM) context is current other than the kernel context (as with ISRs).
- Do not rely on knowledge of the current task or invoke any function that relies on this information, for example **taskIdSelf( )**.
- Do not rely on **taskIdVerify (pOldTcb)** to determine if a delete hook has executed for the self-destructing task case. Instead, other state information must be changed in the delete hook to be detected by the switch hook (for example by setting a pointer to NULL).

Task create hook routines execute in the context of the creator task.

Task create hooks must consider the ownership of any kernel objects (such as watchdog timers, semaphores, and so on) created in the hook routine. Since create hook routines execute in the context of the creator task, new kernel objects will be owned by the creator task's process. It may be necessary to assign the ownership of these objects to the new task's process. This will prevent unexpected object reclamation from occurring if and when the process of the creator task terminates.

When the creator task is a kernel task, the kernel will own any kernel objects that are created. Thus there is no concern about unexpected object reclamation for this case.

User-installed switch hooks are called within the kernel context and therefore do not have access to all VxWorks facilities. Table 4-11 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 4-11 **Routines Callable 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( )**
taskLib	**taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**, **taskIsSuspended( )**, **taskTcb( )**
vxLib	**vxTas( )**

NOTE: For information about POSIX extensions, see *5. POSIX Facilities*.

4.5 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.

NOTE: This section describes the implementation and use of **errno** in UP configurations of VxWorks, which is different from that in SMP configurations.For information about **errno** and other global variables in VxWorks SMP, see *15.15.8 SMP CPU-Specific Variables and Uniprocessor Global Variables*, p.712. For information about migration, see *15.15 Migrating Code to VxWorks SMP*, p.702.

4.5.1 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 host development tools.

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.

4.5.2 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 *4.20.1 Connecting Routines to Interrupts*, p.242).

Thus, regardless of the VxWorks context, an error code can be stored or consulted without direct manipulation of the global variable **errno**.

4.5.3 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);
```

It is recommended 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 VxWorks API reference for **errnoLib** for details on error-status values and building **statSymTbl**.

4.5.4 Assignment of Error Status Values

VxWorks **errno** values encode the module that issues the error, in the most significant two bytes, and uses 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 VxWorks API reference on **errnoLib** for more information about defining and decoding **errno** values with this convention.

4.6 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 (see *11. Error Detection and Reporting*).

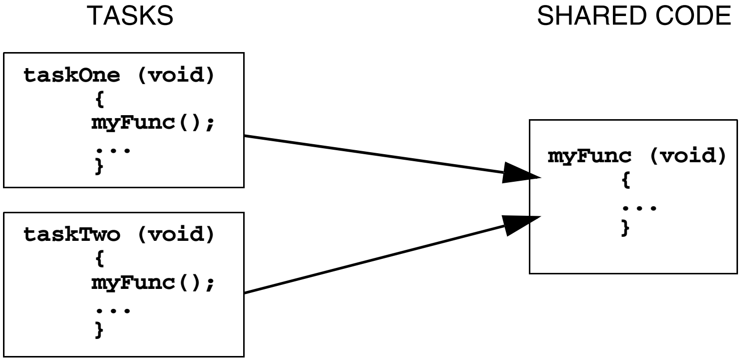
Tasks can also attach their own handlers for certain hardware exceptions through the *signal* facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. A user-defined signal handler is useful for recovering from catastrophic events. Typically, **setjmp( )** is called to define the point in the program where control will be restored, and **longjmp( )** is called in the signal handler to restore that context. Note that **longjmp( )** restores the state of the task's signal mask.

Signals are also used for signaling software exceptions as well as hardware exceptions. They are described in more detail in *4.18 Signals*, p.226 and in the VxWorks API reference for **sigLib**.

4.7 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 especially easy. Shared code makes a system more efficient and easier to maintain; see Figure 4-4.

Figure 4-4 **Shared Code**



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, you should assume that any routine *someName***( )** is not reentrant if there is a corresponding routine named *someName*_**r( )** — the latter is provided as a reentrant version of the routine. 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, Wind River recommends 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 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

Wind River recommends applying these same techniques when writing application code that can be called from several task contexts simultaneously.

NOTE: In some cases reentrant code is not preferable. A critical section should use a binary semaphore to guard it, or use **intLock( )** or **intUnlock( )** if called from by an ISR.

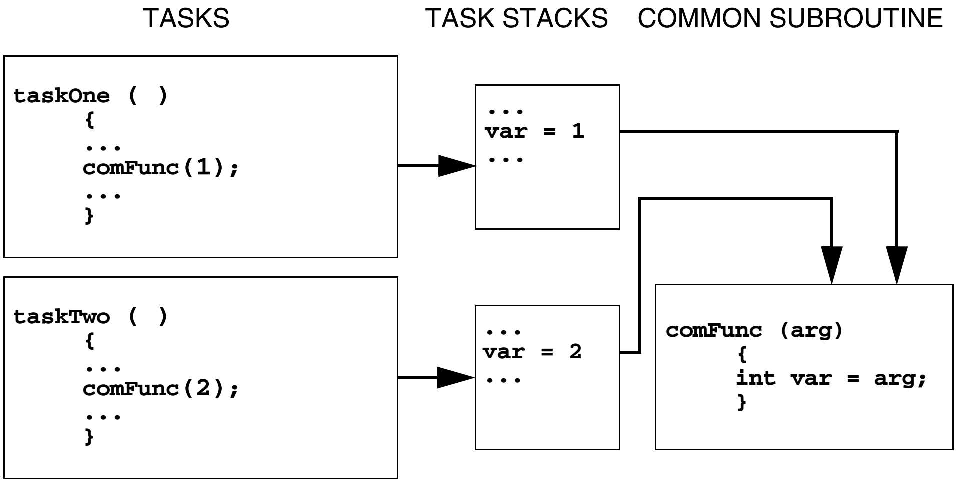
NOTE: Initialization routines should be callable multiple times, even if logically they should only be called once. As a rule, routines should avoid **static** variables that keep state information. Initialization routines are an exception; using a **static** variable that returns the success or failure of the original initialization routine call is appropriate.

4.7.1 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 4-5.

Figure 4-5 **Stack Variables and Shared Code**



4.7.2 Guarded Global and Static Variables

Some libraries encapsulate access to common data. 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 mutex semaphore facility provided by **semMLib** and described in *4.12.3 Mutual-Exclusion Semaphores*, p.205.

4.7.3 Task-Specific Variables

Task-specific variables can be used to ensure that shared code is reentrant by providing task-specific variables of the same name that are located in each task's stack, instead of a standard global or static variables. Each task thereby has its own unique copy of the data item.This allows, for example, several tasks to reference a private buffer of memory and while referring to it with the same global variable name.

NOTE: The **__thread** storage class variables can be used for both UP and SMP configurations of VxWorks, and Wind River recommends its use in both cases as the best method of providing task-specific variables. The **taskVarLib** and **tlsOldLib** (formerly **tlsLib**) facilities—for the kernel-space and user-space respectively— are maintained primarily for backwards-compatibility, are not compatible with VxWorks SMP, and their use is not recommended. In addition to being incompatible with VxWorks SMP, the **taskVarLib** and **tlsOldLib** facilities increase task context switch times. For information about migration, see *15.15 Migrating Code to VxWorks SMP*, p.702.

Also note that each task has a VxWorks events register, which receives events sent from other tasks, ISRs, semaphores, or message queues. See *4.15 VxWorks Events*, p.219 for more information about this register, and the routines used to interact with it.

Thread-Local Variables: __thread Storage Class

Thread-local storage is a compiler facility that allows for allocation of a variable such that there are unique instances of the variable for each thread (or task, in VxWorks terms).

The **__thread** storage class instructs the compiler to make the defined variable a thread-local variable. This means one instance of the variable is created for every task in the system. The compiler key word is used as follows:

```
__thread int i;

extern __thread struct state s;

static __thread char *p;
```

The **__thread** specifier may be used alone, with the **extern** or **static** specifiers, but with no other storage class specifier. When used with **extern** or **static**, **__thread** must appear immediately *after* the other storage class specifier.

The **__thread** specifier may be applied to any global, file-scoped static, function-scoped static, or static data member of a class. It may not be applied to block-scoped automatic or non-static data member.

When the address-of operator is applied to a thread-local variable, it is evaluated at run-time and returns the address of the current task's instance of that variable. The address may be used by any task. When a task terminates, any pointers to thread-local variables in that task become invalid.

No static initialization may refer to the address of a thread-local variable.

In C++, if an initializer is present for a thread-local variable, it must be a *constant-expression*, as defined in 5.19.2 of the ANSI/ISO C++ standard.

taskVarLib and Task Variables

VxWorks provides a task variable facility (with **taskVarLib**) 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.

NOTE: Wind River does not recommend using the **taskVarLib** facility, which is maintained primarily for backwards-compatibility. Use thread-local (**__thread**) storage class variables instead.

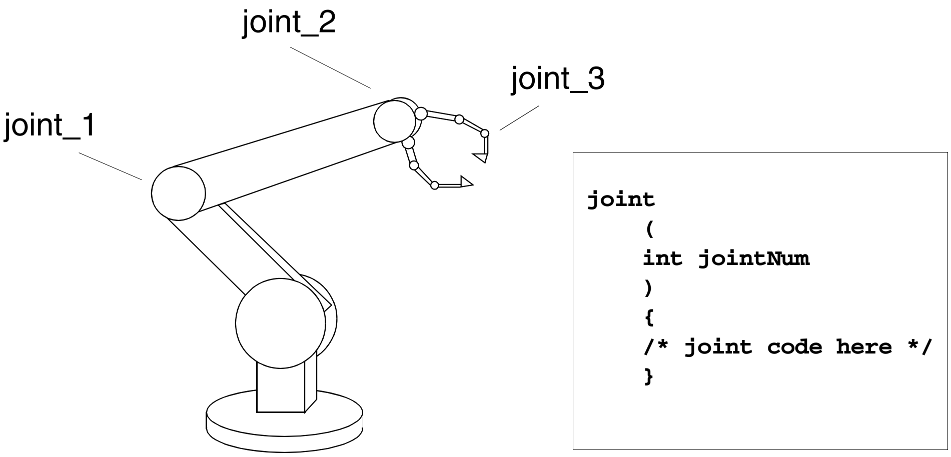
4.7.4 Multiple Tasks with the Same Main Routine

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 *4.7.3 Task-Specific Variables*, p.190 apply to the entire task.

This is useful when the same function must 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 4-6, 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 4-6 **Multiple Tasks Utilizing Same Code**



```
joint
    (
    int jointNum
    )
    {
    /* joint code here */
    }
```

4.8 Intertask and Interprocess Communication

The complement to the multitasking routines described in *4.2 Tasks and Multitasking*, p.160 is the intertask communication facilities. These facilities permit independent tasks to coordinate their actions.

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

- *Shared memory*, for simple sharing of data.
- *Semaphores*, for basic mutual exclusion and synchronization.
- *Mutexes* and *condition variables* for mutual exclusion and synchronization using POSIX interfaces.
- *Message queues* and *pipes*, for intertask message passing within a CPU.
- *VxWorks events*, for communication and synchronization.
- *Message channels*, for socket-based inter-processor and interprocess communication.
- *Sockets* and *remote procedure calls*, for network-transparent intertask communication.
- *Signals*, for exception handling, interprocess communication, and process management.

NOTE: With few exceptions, the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks share the same facilities for intertask and interprocess communications—the difference amounts to only a few routines.

This section provides information about the APIs that are common to both configurations, as well as those APIs that are specific to the UP configuration. In the latter case, the alternatives available for SMP systems are noted. For information about the SMP configuration of VxWorks, see *15. VxWorks SMP*; and for information specifically about migration, see *15.15 Migrating Code to VxWorks SMP*, p.702.

In addition, the VxMP component provides for intertask communication between multiple CPUs that share memory. See *16. Shared-Memory Objects: VxMP*.

4.9 Public and Private Objects

Kernel objects such as semaphores and message queues can be created as either private or public objects. This provides control over the scope of their accessibility—which can be limited to a virtual memory context by defining them as private, or extended to the entire system (the kernel and any processes) by defining them as public. There is no difference in performance between a public and a private object.

An object can only be defined as public or private when it is created—the designation cannot be changed thereafter. Public objects must be named when they are created, and the name must begin with a forward slash; for example, **/foo**. Private objects do not need to be named.

For information about naming tasks in addition to that provided in this section, see *4.4.4 Task Names and IDs*, p.177.

4.9.1 Creating and Naming Public and Private Objects

Public objects are always named, and the name must begin with a forward-slash. Private objects can be named or unnamed. If they are named, the name must not begin with a forward-slash.

Only one public object of a given class and name can be created. That is, there can be only one public semaphore with the name **/foo**. But there may be a public semaphore named **/foo** and a public message queue named **/foo**. Obviously, more distinctive naming is preferable (such as **/fooSem** and **/fooMQ**).

The system allows creation of only one private object of a given class and name in any given memory context; that is, in any given process or in the kernel. For example:

- If process A has created a private semaphore named **bar**, it cannot create a second semaphore named **bar**.
- However, process B could create a private semaphore named **bar**, as long as it did not already own one with that same name.

Note that private tasks are an exception to this rule—duplicate names are permitted for private tasks; see *4.4.4 Task Names and IDs*, p.177.

To create a named object, the appropriate *xyz***Open( )** API must be used, such as **semOpen( )**. When the routine specifies a name that starts with a forward slash, the object will be public.

To delete public objects, the *xyz***Delete( )** API cannot be used (it can only be used with private objects). Instead, the *xyz***Close( )** and *xyz***Unlink( )** APIs must be used in accordance with the POSIX standard. That is, they must be unlinked from the name space, and then the last close operation will delete the object (for example, using the **semUnlink( )** and **semClose( )** APIs for a public semaphore). Alternatively, all close operations can be performed first, and then the unlink operation, after which the object is deleted. Note that if an object is created with the **OM_DELETE_ON_LAST_CLOSE** flag, it is be deleted with the last close operation, regardless of whether or not it was unlinked.

4.9.2 Object Ownership and Resource Reclamation

All objects are owned by the process to which the creator task belongs, or by the kernel if the creator task is a kernel task. When ownership must be changed, for example on a process creation hook, the **objOwnerSet( )** can be used. However, its use is restricted—the new owner must be a process or the kernel.

All objects that are owned by a process are automatically destroyed when the process dies.

All objects that are children of another object are automatically destroyed when the parent object is destroyed.

Processes can share public objects through an object lookup-by-name capability (with the *xyz***Open( )** set of routines). Sharing objects between processes can only be done by name.

When a process terminates, all the private objects that it owns are deleted, regardless of whether or not they are named. All references to public objects in the process are closed (an *xyz***Close( )** operation is performed). Therefore, any public object is deleted during resource reclamation, regardless of which process created them, if there are no more outstanding *xyz***Open( )** calls against it (that is, no other process or the kernel has a reference to it), and the object was already unlinked or was created with the **OM_DELETE_ON_LAST_CLOSE** option. The exception to this rule is tasks, which are always reclaimed when its creator process dies.

When the creator process of a public object dies, but the object survives because it hasn't been unlinked or because another process has a reference to it, ownership of the object is assigned to the kernel.

The **objShowAll( )** show routine can be used to display information about ownership relations between objects.

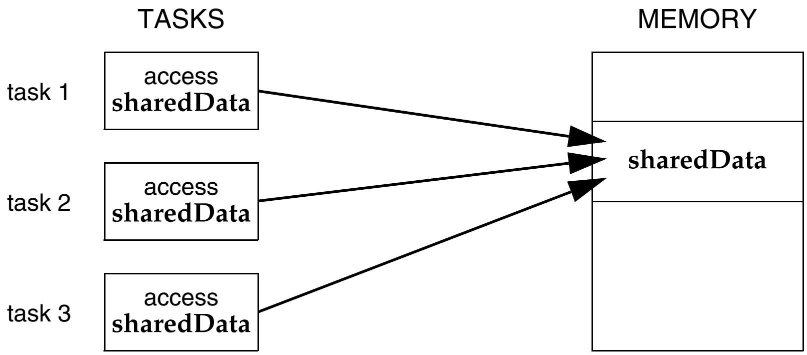
4.10 Shared Data Structures

The most obvious way for tasks executing in the same memory space (either a process or the kernel) to communicate is by accessing shared data structures. Because all the tasks in a single process or in the kernel exist in a single linear address space, sharing data structures between tasks is trivial; see Figure 4-7.

Global variables, linear buffers, ring buffers, linked lists, and pointers can be referenced directly by code running in different contexts.

For information about using shared data regions to communicate between processes, see *VxWorks Application Programmer's Guide: Applications and Processes*.

Figure 4-7 **Shared Data Structures**



4.11 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.

For information about POSIX mutexes, see *5.11 POSIX Thread Mutexes and Condition Variables*, p.273.

4.11.1 Interrupt Locks and Latency

The most powerful method available for mutual exclusion is the disabling of interrupts with the **intLock( )** routine. Such a lock guarantees exclusive access to the CPU:

```
funcA ()
    {
    int lock = intLock();
    .
    .   /* critical region of code 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.

For information about ISRs, see *4.20 Interrupt Service Routines*, p.241.

⚠ **WARNING:** Invoking a VxWorks system routine with interrupts locked may result in interrupts being re-enabled for an unspecified period of time. If the called routine blocks, or results in a higher priority task becoming eligible for execution (**READY**), interrupts will be re-enabled while another task executes, or while the kernel is idle.

NOTE: The **intLock( )** routine is provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including the ISR-callable spinlock, which defaults to **intLock( )** behavior in a UP system. For more information, see *15.6.1 ISR-Callable Spinlocks*, p.682 and *15.15 Migrating Code to VxWorks SMP*, p.702.

4.11.2 Preemptive Locks and Latency

Disabling preemption with the **taskLock( )** routine 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 of code 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, be sure to keep the duration short. Semaphores provide a better mechanism; see *4.12 Semaphores*, p.198.

WARNING: The critical region code should not block. If it does, preemption could be re-enabled.

NOTE: The **taskLock( )** and **taskUnlock( )** routines are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including task-only spinlocks, which default to **taskLock( )** and **taskUnlock( )** behavior in a UP system. For more information, see *15.6.2 Task-Only Spinlocks*, p.682 and *15.15 Migrating Code to VxWorks SMP*, p.702.

4.12 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, as described below:

- 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 *4.11 Mutual Exclusion*, p.196.
- For *synchronization*, semaphores coordinate a task's execution with external events.

NOTE: Semaphores provide full memory barriers, which is of particular significance for the SMP configuration of VxWorks. For more information, see *15.8 Memory Barriers*, p.687.

VxWorks provides the following types of semaphores, which are optimized for different types of uses:

binary
: The fastest, most general-purpose semaphore. Optimized for synchronization or mutual exclusion. For more information, see *4.12.2 Binary Semaphores*, p.201.

mutual exclusion
: A special binary semaphore optimized for problems inherent in mutual exclusion: priority inversion, deletion safety, and recursion. For more information, see *4.12.3 Mutual-Exclusion Semaphores*, p.205.

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. For more information, see *4.12.4 Counting Semaphores*, p.208.

read/write
: A special type of semaphore that provides mutual exclusion for tasks that need write access to an object, and concurrent access for tasks that only need read access to the object. This type of semaphore is particularly useful for SMP systems. For more information, see *4.12.5 Read/Write Semaphores*, p.209.

VxWorks semaphores can be created as private objects, which are accessible only within the memory space in which they were created (kernel or process); or as public objects, which accessible throughout the system. For more information, see *4.9 Public and Private Objects*, p.194.

VxWorks not only provides the semaphores designed expressly for VxWorks, but also POSIX semaphores, designed for portability. An alternate semaphore library provides the POSIX-compliant semaphore interface; see *5.13 POSIX Semaphores*, p.289.

NOTE: The semaphores described here are for use with UP and SMP configurations of VxWorks. The optional product VxMP provides semaphores that can be used in an asymmetric multiprocessor (AMP) system, in the VxWorks kernel (but not in UP or SMP systems). For more information, see *16. Shared-Memory Objects: VxMP*.

4.12.1 Semaphore Control

In most cases, VxWorks provides a single, uniform interface for semaphore control—instead of defining a full set of semaphore control routines specific to each type of semaphore.

The exceptions are the creation routines, which are specific to each semaphore type; and the give and take routines for read/write semaphores, which support read and write modes for each operation.

Table 4-12 lists the semaphore control routines.

Table 4-12 **Semaphore Control Routines**

Call	Description
semBCreate( )	Allocates and initializes a binary semaphore.
semMCreate( )	Allocates and initializes a mutual-exclusion semaphore.
semCCreate( )	Allocates and initializes a counting semaphore.
semRWCreate( )	Allocates and initializes a read/write semaphore.
semDelete( )	Terminates and frees a semaphore.
semTake( )	Takes a binary, mutual-exclusion, or counting semaphore., or a read/write semaphore in write mode.
semRTake( )	Takes a read/write semaphore in read mode.
semWTake( )	Takes a read/write semaphore in write mode.
semGive( )	Gives a binary, mutual -exclusion, or counting semaphore.
semRWGive( )	Gives a read/write semaphore.
semFlush( )	Unblocks all tasks that are waiting for a semaphore.
semExchange( )	Provides for an atomic give and exchange of semaphores in SMP systems.

The creation routines return a semaphore ID that serves as a handle on the semaphore during subsequent use by 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 all 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.

Static Instantiation of Semaphores

The semaphore creation routines listed in Table 4-12 perform a dynamic, two-step operation, in which memory is allocated for the semaphore object at runtime, and then the object is initialized. Semaphores (and other VxWorks objects) can also be statically instantiated—which means that their memory is allocated for the object at compile time—and the object is then initialized at runtime with an initialization routine.

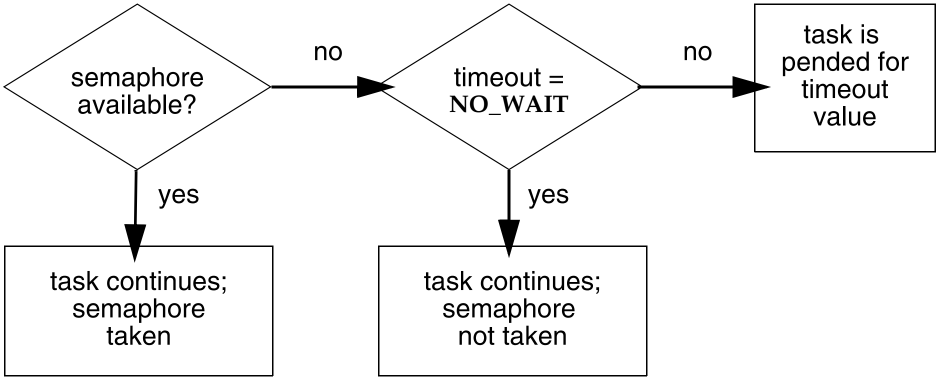
For information about static instantiation, see *2.6.4 Static Instantiation of Kernel Objects*, p.56. For information about semaphore initialization routines, see the VxWorks API references.

4.12.2 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 *4.12.3 Mutual-Exclusion Semaphores*, p.205 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.

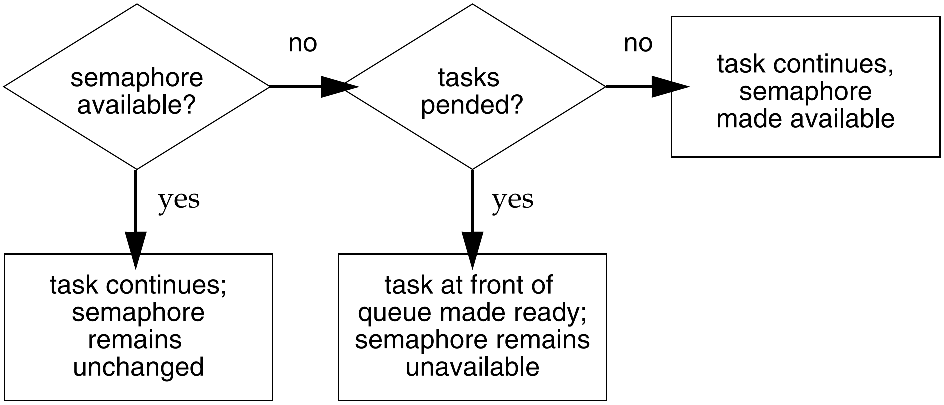
A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with **semTake( )**, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 4-8. If the semaphore is available (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 4-8 **Taking a Semaphore**


semaphore available?
no
timeout = NO_WAIT
no
task is pended for timeout value
yes
yes
task continues; semaphore taken
task continues; semaphore not taken

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 4-9. 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).

Figure 4-9 **Giving a Semaphore**


semaphore available?
no
tasks pended?
no
task continues, semaphore made available
yes
yes
task continues; semaphore remains unchanged
task at front of queue made ready; semaphore remains unavailable

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. Another task waits for the semaphore by calling **semTake( )**. The waiting task blocks until the event occurs and the semaphore is given.

(See *4.20 Interrupt Service Routines*, p.241 for a complete discussion of ISRs)

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 4-2, 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 4-2 **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.

4.12.3 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.

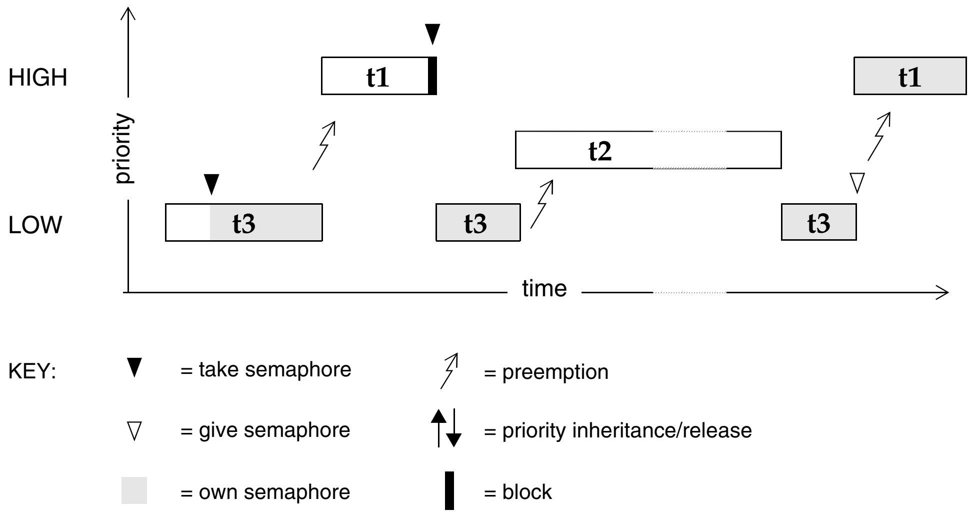
The fundamental behavior of the mutual-exclusion semaphore is identical to the binary semaphore, with the following exceptions:

- 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

Figure 4-10 illustrates a situation called priority inversion.

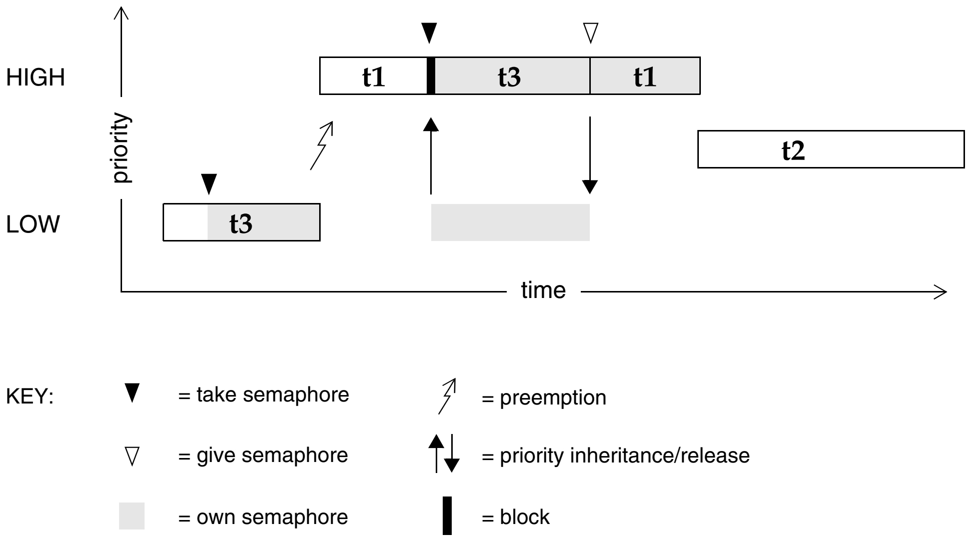
Figure 4-10 **Priority Inversion**



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 4-10: **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* policy. The priority-inheritance policy assures that a task that holds 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 have contributed to the tasks elevated priority are released. 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**).

Figure 4-11 **Priority Inheritance**



In Figure 4-11, 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 policy:

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

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 holds 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 holds 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 4-3 **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");
    }
```

4.12.4 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 4-13 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Table 4-13 **Counting Semaphore Example**

Semaphore Call	Count after Call	Resulting Behavior
semCCreate( )	3	Semaphore initialized with an 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.

4.12.5 Read/Write Semaphores

Read/write semaphores provide enhanced performance for applications that can effectively make use of differentiation between read access to a resource, and write access to a resource. A read/write semaphore can be taken in either read mode or write mode. They are particularly suited to SMP systems (for information about the SMP configuration of VxWorks, see *15. VxWorks SMP*).

A task holding a read/write semaphore in write mode has exclusive access to a resource. On the other hand, a task holding a read/write semaphore in read mode does not have exclusive access. More than one task can take a read/write semaphore in read mode, and gain access to the same resource.

Because it is exclusive, write-mode permits only serial access to a resource, while while read-mode allows shared or concurrent access. In a multiprocessor system, more than one task (running in different CPUs) can have read-mode access to a resource in a truly concurrent manner. In a uniprocessor system, however, access is shared but the concurrency is virtual. More than one task can have read-mode access to a resource at the same time, but since the tasks do not run simultaneously, access is effectively multiplexed.

All tasks that hold a read/write semaphore in read mode must give it up before any task can take it in write mode.

Specification of Read or Write Mode

A read/write semaphore differs from other types of semaphore in that the access mode must be specified when the semaphore is taken. The mode determines whether the access is exclusive (write mode), or if concurrent access is allowed (read mode). Different APIs correspond to the different modes of access, as follows:

- **semRTake( )** for read (exclusive) mode
- **semWTake( )** for write (concurrent) mode

You can also use **semTake( )** on a read/write semaphore, but the behavior is the same as **semWTake( )**. And you can use **semGive( )** on a read/write semaphore as long as the task owns it is in the same mode.

For more information about read/write semaphore APIs, see Table 4-12 and the VxWorks API references.

When a task takes a read/write semaphore in write mode, the behavior is identical to that of a mutex semaphore. The task owns the semaphore exclusively. An attempt to give a semaphore held by one task in this mode by task results in a return value of **ERROR**.

When a task takes a read/write semaphore in read mode, the behavior is different from other semaphores. It does not provide exclusive access to a resource (does not protect critical sections), and the semaphore may be concurrently held in read mode by more than one task.

The maximum number of tasks that can take a read/write semaphore in read mode can be specified when the semaphore is created with the create routine call. The system maximum for all read/write semaphores can also be set with **SEM_RW_MAX_CONCURRENT_READERS** component parameter. By default it is set to 32.

If the number of tasks is not specified when the create routine is called, the system default is used.

Read/write semaphores can be taken recursively in both read and write mode. Optionally, priority inheritance and deletion safety are available for each mode.

Precedence for Write Access Operations

When a read/write semaphore becomes available, precedence is given to pended tasks that require write access, regardless of their task priority relative to pended tasks that require read access. That is, the highest priority task attempting a **semWTake( )** operation gets the semaphore, even if there are higher priority tasks attempting a **semRTake( )**. Precedence for write access helps to ensure that the protected resource is kept current because there is no delay due to read operations occurring before a pending write operation can take place.

Note, however, that all read-mode takes must be given before a read/write semaphore can be taken in write mode.

Read/Write Semaphores and System Performance

The performance of systems that implement read/write semaphores for their intended use should be enhanced, particularly so in SMP systems. However, due to the additional bookkeeping overhead involved in tracking multiple read-mode owners, performance is likely to be adversely affected in those cases where the feature does fit a clear design goal. In particular, interrupt latency in a uniprocessor system and kernel latency in a multiprocessor system may be adversely affected.

4.12.6 Special Semaphore Options

The uniform VxWorks semaphore interface includes three special options: timeouts, queues, and use with VxWorks events. These options are not available for either the read/write semaphores described in *4.12.5 Read/Write Semaphores*, p.209, or the POSIX-compliant semaphores described in *5.13 POSIX Semaphores*, p.289.

Timeouts

As an alternative to blocking until a semaphore becomes available, semaphore take operations can be restricted to a specified period of time. If the semaphore is not taken within that period, the take operation fails.

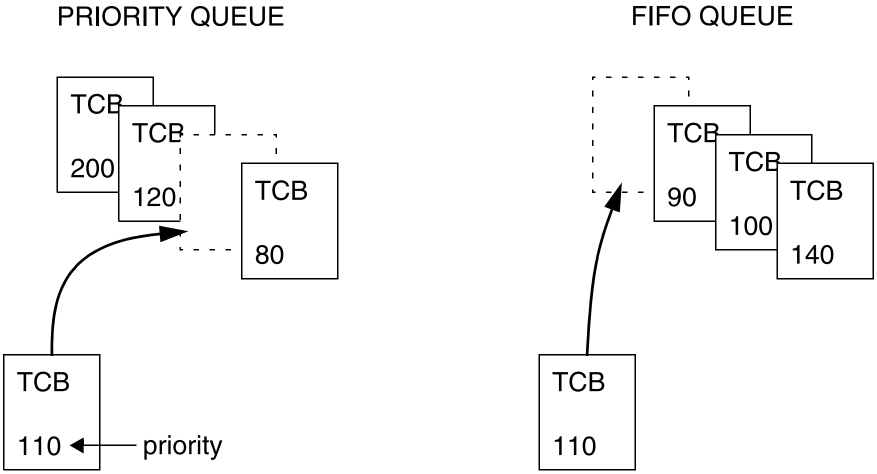
This behavior is controlled by a parameter to **semTake( )** and the take routines for read/write semaphores that specifies the amount of time in ticks that the task is willing to wait in the pended state. If the task succeeds in taking the semaphore within the allotted time, the take routine returns **OK**. The **errno** set when a take routine 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

VxWorks 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 4-12.

Figure 4-12 **Task Queue Types**



Priority ordering better preserves the intended priority structure of the system at the expense of some overhead in take operations because of 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 the semaphore creation routine. Semaphores using the priority inheritance option (**SEM_INVERSION_SAFE**) must select priority-order queuing.

4.12.7 Semaphores and VxWorks Events

Semaphores can send VxWorks events to a specified task when they becomes free. For more information, see *4.15 VxWorks Events*, p.219.

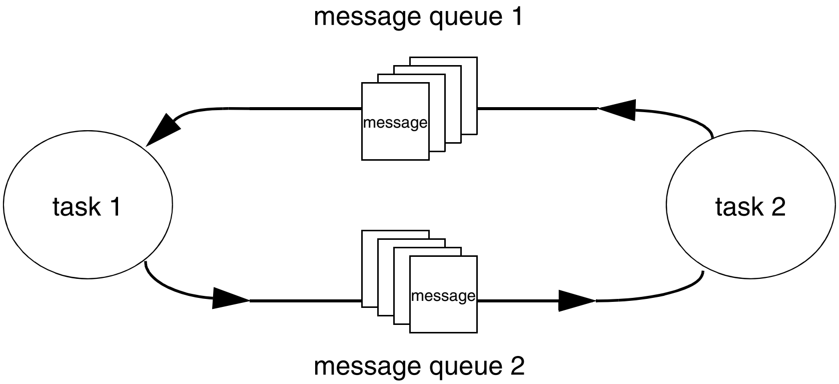
4.13 Message Queues

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*.

For information about socket-based message communication across memory spaces (kernel and processes), and between multiple nodes, see *4.16 Message Channels*, p.226.

Message queues allow a variable number of messages, each of variable length, to be queued. Tasks and ISRs can send messages to a message queue, and tasks can receive messages from a message queue.

Figure 4-13 **Full Duplex Communication Using Message Queues**



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 4-13.

VxWorks message queues can be created as private objects, which accessible only within the memory space in which they were created (process or kernel); or as public objects, which accessible throughout the system. For more information, see *4.9 Public and Private Objects*, p.194.

There are two message-queue subroutine libraries in VxWorks. The first of these, **msgQLib**, provides VxWorks message queues, designed expressly for VxWorks; the second, **mqPxLib**, is compliant with the POSIX standard (1003.1b) for real-time extensions. See *5.13.1 Comparison of POSIX and VxWorks Semaphores*, p.290 for a discussion of the differences between the two message-queue designs.

4.13.1 VxWorks Message Queue Routines

VxWorks message queues are created, used, and deleted with the routines shown in Table 4-14. 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 4-14 **VxWorks Message Queue Control**

Call	Description
msgQCreate( )	Allocates and initializes a message queue.
msgQDelete( )	Terminates and frees a message queue.
msgQSend( )	Sends a message to a message queue.
msgQReceive( )	Receives 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 allocated 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 4-4 **VxWorks 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);
    }
```

Queuing

VxWorks message queues include the ability to select the queuing mechanism employed for tasks blocked on a message queue. The **MSG_Q_FIFO** and **MSG_Q_PRIORITY** options are provided to specify (to the **msgQCreate( )** and **msgQOpen( )** routines) the queuing mechanism that should be used for tasks that pend on **msgQSend( )** and **msgQReceive( )**.

4.13.2 Displaying Message Queue Attributes

The VxWorks **show( )** command produces a display of the key message queue attributes, for either kind of message queue. For example, if **myMsgQId** is a VxWorks message queue, the output is sent to the standard output device, and looks like the following from the shell (using the C interpreter):

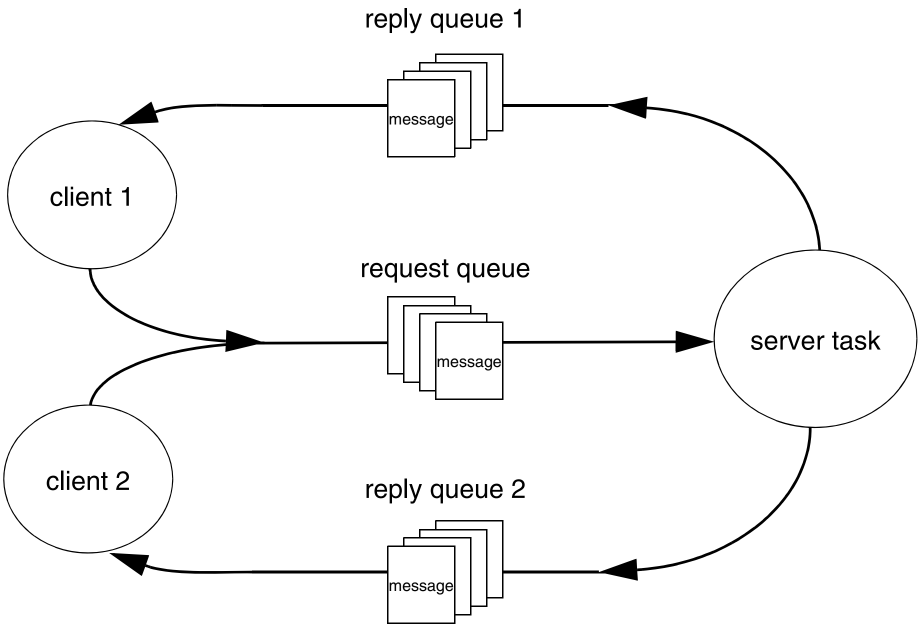
```
-> show myMsgQId
Message Queue Id  : 0x3adaf0
Task Queuing      : FIFO
Message Byte Len  : 4
Messages Max      : 30
Messages Queued   : 14
Receivers Blocked : 0
Send timeouts     : 0
Receive timeouts    : 0
```

4.13.3 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 *4.14 Pipes*, p.218) are a natural way to implement this functionality.

For example, client-server communications might be implemented as shown in Figure 4-14. 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 4-14 **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.

4.13.4 Message Queues and VxWorks Events

Message queues can send VxWorks events to a specified task when a message arrives on the queue and no task is waiting on it. For more information, see *4.15 VxWorks Events*, p.219.

4.14 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 *7.4.9 Pending on Multiple File Descriptors with select( )*, p.374.

Pipes allow you to implement a client-server model of intertask communications; see *4.13.3 Servers and Clients with Message Queues*, p.217.

4.15 VxWorks Events

VxWorks events provide a means of communication and synchronization between tasks and other tasks, interrupt service routines (ISRs) and tasks, semaphores and tasks, and message queues and tasks.[1]

Events can be used as a lighter-weight alternative to binary semaphores for task-to-task and ISR-to-task synchronization (because no object must be created). They can also be used to notify a task that a semaphore has become available, or that a message has arrived on a message queue.

The events facility provides a mechanism for coordinating the activity of a task using up to thirty-two *events* that can be sent to it by other tasks, ISRs, semaphores,

[1] VxWorks events are based on pSOS operating system events. VxWorks introduced functionality similar to pSOS events (but with enhancements) with the VxWorks 5.5 release.

and message queues. A task can wait on multiple events from multiple sources. Events thereby provide a means for coordination of complex matrix of activity without allocation of additional system resources.

Each task has 32 event flags, bit-wise encoded in a 32-bit word (bits 25 to 32 are reserved for Wind River use). These flags are stored in the task's *event register*. Note that an event flag itself has no intrinsic meaning. The significance of each of the 32 event flags depends entirely on how any given task is coded to respond to their being set. There is no mechanism for recording how many times any given event has been received by a task. Once a flag has been set, its being set again by the same or a different sender is essentially an *invisible* operation.

Events are similar to signals in that they are sent to a task asynchronously; but differ in that receipt is synchronous. That is, the receiving task must call a routine to receive at will, and can choose to pend while waiting for events to arrive. Unlike signals, therefore, events do not require a handler.

For a code example of how events can be used, see the **eventLib** API reference.

NOTE: VxWorks events, which are also simply referred to as *events* in this section, should not be confused with System Viewer events.

Configuring VxWorks for Events

To provide events facilities, VxWorks must be configured with the **INCLUDE_VXEVENTS** component.

4.15.1 Preparing a Task to Receive Events

A task can pend on one or more events, or simply check on which events have been received, with a call to **eventReceive( )**. The routine specifies which events to wait for, and provides options for waiting for one or all of those events. It also provides various options for how to manage unsolicited events.

In order for a task to receive events from a semaphore or a message queue, however, it must first register with the specific object, using **semEvStart( )** for a semaphore or **msgQEvStart( )** for a message queue. Only one task can be registered with any given semaphore or message queue at a time.

The **semEvStart( )** routine identifies the semaphore and the events that it should send to the task when the semaphore is free. It also provides a set of options to specify whether the events are sent only the first time the semaphore is free, or each time; whether to send events if the semaphore is free at the time of registration; and whether a subsequent **semEvStart( )** call from another task is allowed to take effect (and to unregister the previously registered task).

Once a task has registered with a semaphore, every time the semaphore is released with **semGive( )**, and as long as no other tasks are pending on it, the semaphore sends events to the registered task.

To request that the semaphore stop sending events to it, the registered task calls **semEvStop( )**.

Registration with a message queue is similar to registration with a semaphore. The **msgQEvStart( )** routine identifies the message queue and the events that it should send to the task when a message arrives and no tasks are pending on it. It provides a set of options to specify whether the events are sent only the first time a message is available, or each time; whether a subsequent call to **msgQEvStart( )** from another task is allowed to take effect (and to unregister the previously registered task).

Once a task has registered with a message queue, every time the message queue receives a message and there are no tasks pending on it, the message queue sends events to the registered task.

To request that the message queue stop sending events to it, the registered task calls **msgQEvStop( )**.

4.15.2 Sending Events to a Task

Tasks and ISRs can send specific events to a task using **eventSend( )**, whether or not the receiving task is prepared to make use of them.

Semaphores and message queues send events automatically to tasks that have registered for notification with **semEvStart( )** or **msgQEvStart( )**, respectively. These objects send events when they are *free*. The conditions under which objects are free are as follows:

Mutex Semaphore
: A mutex semaphore is considered free when it no longer has an owner and no task is pending on it. For example, following a call to **semGive( )**, the semaphore will not send events if another task is pending on a **semTake( )** for the same semaphore.

Binary Semaphore
: A binary semaphore is considered free when no task owns it and no task is waiting for it.

Counting Semaphore

A counting semaphore is considered free when its count is nonzero and no task is pending on it. Events cannot, therefore, be used as a mechanism to compute the number of times a semaphore is released or given.

Message Queue

A message queue is considered free when a message is present in the queue and no task is pending for the arrival of a message in that queue. Events cannot, therefore, be used as a mechanism to compute the number of messages sent to a message queue.

Note that just because an object has been released does not mean that it is free. For example, if a semaphore is *given*, it is released; but it is not free if another task is waiting for it at the time it is released. When two or more tasks are constantly exchanging ownership of an object, it is therefore possible that the object never becomes free, and never sends events.

Also note that when a semaphore or message queue sends events to a task to indicate that it is free, it does not mean that the object is in any way *reserved* for the task. A task waiting for events from an object unpends when the resource becomes free, but the object may be taken in the interval between notification and unpending. The object could be taken by a higher priority task if the task receiving the event was pended in **eventReceive( )**. Or a lower priority task might *steal* the object: if the task receiving the event was pended in some routine other than **eventReceive( )**, a low priority task could execute and (for example) perform a **semTake( )** after the event is sent, but before the receiving task unpends from the blocking call. There is, therefore, no guarantee that the resource will still be available when the task subsequently attempts to take ownership of it.

WARNING: Because events cannot be reserved for an application in any way, care should be taken to ensure that events are used uniquely and unambiguously. Note that events 25 to 32 (VXEV25 to VXEV32) are reserved for Wind River's use, and should not be used by customers. Third parties should be sure to document their use of events so that their customers do not use the same ones for their applications.

Events and Object Deletion

If a semaphore or message queue is deleted while a task is waiting for events from it, the task is automatically unpended by the **semDelete( )** or **msgQDelete( )** implementation. This prevents the task from pending indefinitely while waiting for events from an object that has been deleted. The pending task then returns to the ready state (just as if it were pending on the semaphore itself) and receives an **ERROR** return value from the **eventReceive( )** call that caused it to pend initially.

If, however, the object is deleted between a tasks' registration call and its **eventReceive( )** call, the task pends anyway. For example, if a semaphore is deleted while the task is between the **semEvStart( )** and **eventReceive( )** calls, the task pends in **eventReceive( )**, but the event is never sent. It is important, therefore, to use a timeout other than **WAIT_FOREVER** when object deletion is expected.

Events and Task Deletion

If a task is deleted before a semaphore or message queue sends events to it, the events can still be sent, but are obviously not received. By default, VxWorks handles this event-delivery failure silently.

It can, however, be useful for an application that created an object to be informed when events were not received by the (now absent) task that registered for them. In this case, semaphores and message queues can be created with an option that causes an error to be returned if event delivery fails (the **SEM_EVENTSEND_ERROR_NOTIFY** and **MSG_Q_EVENTSEND_ERROR_NOTIFY** options, respectively). The **semGive( )** or **msgQSend( )** call then returns **ERROR** when the object becomes free.

The error does not mean the semaphore was not given or that the message was not properly delivered. It simply means the resource could not send events to the registered task. Note that a failure to send a message or give a semaphore takes precedence over an events failure.

4.15.3 Accessing Event Flags

When events are sent to a task, they are stored in the task's events register (see *4.15.5 Task Events Register*, p.224), which is not directly accessible to the task itself.

When the events specified with an **eventReceive( )** call have been received and the task unpends, the contents of the events register is copied to a variable that is accessible to the task.

When **eventReceive( )** is used with the **EVENTS_WAIT_ANY** option—which means that the task unpends for the first of any of the specified events that it receives—the contents of the events variable can be checked to determine which event caused the task to unpend.

The **eventReceive( )** routine also provides an option that allows for checking which events have been received prior to the full set being received.

4.15.4 Events Routines

The routines used for working with events are listed in Table 4-15.

Table 4-15 **Events Routines**

Routine	Description
eventSend( )	Sends specified events to a task.
eventReceive( )	Pends a task until the specified events have been received. Can also be used to check what events have been received in the interim.
eventClear( )	Clears the calling task's event register.
semEvStart( )	Registers a task to be notified of semaphore availability.
semEvStop( )	Unregisters a task that had previously registered for notification of semaphore availability.
msgQEvStart( )	Registers a task to be notified of message arrival on a message queue when no recipients are pending.
msgQEvStop( )	Unregisters a task that had previously registered for notification of message arrival on a message queue.

For more information about these routines, see the VxWorks API references for **eventLib**, **semEvLib**, and **msgQEvLib**.

4.15.5 Task Events Register

Each task has its own *task events register*. The task events register is a 32-bit field used to store the events that the task receives from other tasks (or itself), ISRs, semaphores, and message queues.

Events 25 to 32 (VXEV25 or 0x01000000 to VXEV32 or 0x80000000) are reserved for Wind River use only, and should not be used by customers.

As noted above (*4.15.3 Accessing Event Flags*, p.223), a task cannot access the contents of its events registry directly.

Table 4-16 describes the routines that affect the contents of the events register.

Table 4-16 **Routines That Modify the Task Events Register**

Routine	Effect on the Task Events Register
eventReceive( )	Clears or leaves the contents of the task's events register intact, depending on the options selected.
eventClear( )	Clears the contents of the task's events register.
eventSend( )	Writes events to a tasks's events register.
semGive( )	Writes events to the tasks's events register, if the task is registered with the semaphore.
msgQSend( )	Writes events to a task's events register, if the task is registered with the message queue.

4.15.6 Show Routines and Events

For the purpose of debugging systems that make use of events, the **taskShow**, **semShow**, and **msgQShow** libraries display event information.

The **taskShow** library displays the following information:

- the contents of the event register
- the desired events
- the options specified when **eventReceive( )** was called

The **semShow** and **msgQShow** libraries display the following information:

- the task registered to receive events
- the events the resource is meant to send to that task
- the options passed to **semEvStart( )** or **msgQEvStart( )**

4.16 Message Channels

Message channels are a socket-based facility that provides for inter-task communication within a memory boundary, between memory boundaries (kernel and processes), between nodes (processors) in a multi-node cluster, and between between multiple clusters.

In addition to providing a superior alternative to TCP for multi-node intercommunication, message channels provide a useful alternative to message queues for exchanging data between two tasks on a single node. Message channels can be used in both kernel and user (RTP) space.

For a comparison of message channels and message queues, see *18.8 Comparison of Message Channels and Message Queues*, p.806. For detailed information about message channels, see *18. Message Channels*.

4.17 Network Communication

To communicate with a peer on a remote networked system, you can use an Internet domain socket or RPC. For information on working with Internet domain sockets or RPC under VxWorks, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide.* For information about TIPC networking, see *Wind River TIPC for VxWorks 6 Programmer's Guide*.

4.18 Signals

Signals are an operating system facility designed for handling exceptional conditions and asynchronously altering the flow of control. In many respects signals are the software equivalent to hardware interrupts. Signals generated by the operating system include those produced in response to bus errors and floating point exceptions. The signal facility also provides APIs that can be used to generate and manage signals programmatically.

In applications, signals are most appropriate for error and exception handling, and not for a general-purpose inter-task communication. Common uses include using signals to kill processes and tasks, to send signal events when a timer has fired or message has arrived at a message queue, and so on.

In accordance with POSIX, VxWorks supports 63 signals, each of which has a unique number and default action (defined in **signal.h**). The value 0 is reserved for use as the **NULL** signal.

Signals can be *raised* (sent) from tasks to tasks or to processes. Signals can be either *caught* (received) or ignored by the receiving task or process. Whether signals are caught or ignored generally depends on the setting of a *signal mask*. In the kernel, signal masks are specific to tasks, and if no task is set up to receive a specific signal, it is ignored. In user space, signal masks are specific to processes; and some signals, such as **SIGKILL** and **SIGSTOP**, cannot be ignored.

To manage responses to signals, you can create and register signal handling routines that allow a task to respond to a specific signal in whatever way is useful for your application.

A kernel task or interrupt service routine (ISR) can raise a signal for a specific task or process. In the kernel, signal generation and delivery runs in the context of the task or ISR that generates the signal. In accordance with the POSIX standard, a signal sent to a process is handled by the first available task that has been set up to handle the signal in the process.

Each kernel task has a signal mask associated with it. The signal mask determines which signals the task accepts. By default, the signal mask is initialized with all signals unblocked (there is no inheritance of mask settings in the kernel). The mask can be changed with **sigprocmask( )**.

Signal handlers in the kernel can be registered for a specific task. A signal handler executes in the receiving task's context and makes use of that task's execution stack. The signal handler is invoked even if the task is blocked (suspended or pended).

VxWorks provides a software signal facility that includes POSIX routines, UNIX BSD-compatible routines, and native VxWorks routines. The POSIX-compliant signal interfaces include both the basic signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b.

Additional, non-POSIX APIs provide support for signals between kernel and user applications. These non-POSIX APIs are: **taskSigqueue( )**, **rtpSigqueue( )**, **rtpTaskSigqueue( )**, **taskKill( )**, **rtpKill( )**, **rtpTaskKill( ),** and **taskRaise( )**.

In the VxWorks kernel—for backward compatibility with prior versions of VxWorks—the POSIX API that would take a process identifier as one of their parameters, take a task identifier instead.

NOTE: Wind River recommends that you do not use both POSIX APIs and VxWorks APIs in the same application. Doing so may make a POSIX application non-conformant.

NOTE: POSIX signals are handled differently in the kernel and in real-time processes. In the kernel the target of a signal is always a task; but in user space, the target of a signal may be either a specific task or an entire process.

NOTE: The VxWorks implementation of **sigLib** does not impose any special restrictions on operations on **SIGKILL**, **SIGCONT**, and **SIGSTOP** signals such as those imposed by UNIX. For example, the UNIX implementation of **signal( )** cannot be called on **SIGKILL** and **SIGSTOP**.

For information about using signals in processes, see *VxWorks Application Programmer's Guide: Multitasking*.

In addition to signals, VxWorks also provides another type of event notification with the VxWorks events facility. While signal events are completely asynchronous, VxWorks events are sent asynchronously, but received synchronously, and do not require a signal handler. For more information, see *4.15 VxWorks Events*, p.219.

4.18.1 Configuring VxWorks for Signals

By default, VxWorks includes the basic signal facility component **INCLUDE_SIGNALS**. This component automatically initializes signals with **sigInit( )**.

To use the signal event facility, configure VxWorks with the **INCLUDE_SIGEVENT** component. Note that **SIGEV_THREAD** option is only supported in processes, and that it requires that VxWorks also be configured with the **INCLUDE_SIGEVENTS_THREAD** component and full POSIX thread support (the **BUNDLE_RTP_POSIX_PSE52** bundle includes everything required for this option).

To include POSIX queued signals in the system, configure VxWorks with the **INCLUDE_POSIX_SIGNALS** component. This component automatically initializes POSIX queued signals with **sigqueueInit( )**. The **sigqueueInit( )** routine allocates buffers for use by **sigqueue( )**, which requires a buffer for each currently queued signal. A call to **sigqueue( )** fails if no buffer is available.

The maximum number of queued signals in the kernel is set with the configuration parameter **NUM_SIGNAL_QUEUES**. The default value is 16.

4.18.2 Basic Signal Routines

Signals are in many ways analogous to hardware interrupts. The basic signal facility provides a set of 63 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( )** or **sigqueue( )**. This is similar to the occurrence of an interrupt. The **sigprocmask( )** routine 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.

For a list and description of basic POSIX and BSD signal routines provided by VxWorks in the kernel, see Table 4-17.

Table 4-17 **Basic Signal Calls**

POSIX 1003.1b Compatible Routine	UNIX BSD Compatible Routine	Description
signal( )	**signal( )**	Specifies the handler associated with a signal.
raise( )	N/A	Sends a signal to yourself.
sigaction( )	**sigvec( )**	Examines or sets the signal handler for a signal.
sigsuspend( )	**pause( )**	Suspends a task until a signal is delivered.
sigpending( )	N/A	Retrieves a set of pending signals blocked from delivery.

sigemptyset( )
sigfillset( )
sigaddset( )
sigdelset( )
sigismember( )
	N/A	Manipulates a signal mask.

sigprocmask( )	N/A	Sets the mask of blocked signals.
sigprocmask( )	N/A	Adds to a set of blocked signals.

VxWorks also provides a POSIX and BSD-like **kill( )** routine, which sends a signal to a task.

VxWorks also provides additional routines that serve as aliases for POSIX routines, such as **rtpKill( )**, that provide for sending signals from the kernel to processes.

For more information about signal routines, see the VxWorks API reference for **sigLib** and **rtpSigLib**.

4.18.3 Queued Signal Routines

The **sigqueue( )** family of routines provides an alternative to the **kill( )** family of routines for sending signals. The important differences between the two are the following:

- The **sigqueue( )** routine includes an application-specified value that is sent as part of the signal. This value supplies whatever context is appropriate for the signal handler. 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**.
- The **sigqueue( )** routine 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 signals reserved for application use, numbered consecutively from **SIGRTMIN** to **SIGRTMAX**. The number of signals reserved is governed by the **RTSIG_MAX** macro (with a value of 16), which defined in the POSIX 1003.1 standard. The signal values themselves are not specified by POSIX. For portability, specify these signals as offsets from **SIGRTMIN** (for example, use **SIGRTMIN+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 1003.1 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.

The basic queued signal routines are described in Table 4-18. For detailed information on signals, see the API reference for **sigLib**.

Table 4-18 **POSIX 1003.1b Queued Signal Routines**

Routine	Description
sigqueue( )	Sends a queued signal to a task.
sigwaitinfo( )	Waits for a signal.
sigtimedwait( )	Waits for a signal with a timeout.

Additional non-POSIX VxWorks queued signal routines are described in Table 4-19. These routines are provided for assisting in porting VxWorks 5.x kernel applications to processes. The POSIX routines described in Table 4-18 should be used for developing new applications that execute as real-time processes.

Note that a parallel set of non-POSIX APIs are provided for the **kill( )** family of POSIX routines—**taskKill( )**, **rtpKill( )**, and **rtpTaskKill( )**.

Table 4-19 **Non-POSIX Queued Signal Routines**

Routine	Description
taskSigqueue( )	Sends a queued signal from a task in a process to another task in the same process, to a public task in another process, or from a kernel task to a process task.
rtpSigqueue( )	Sends a queued signal from a kernel task to a process or from a process to another process.
rtpTaskSigqueue( )	Sends a queued signal from a kernel task to a specified task in a process (kernel-space only).

Example 4-5 **Queued Signals**

```
#include <stdio.h>
#include <signal.h>
#include <taskLib.h>
#include <rtpLib.h>
#ifdef _WRS_KERNEL
#include <private/rtpLibP.h>
#include <private/taskLibP.h>
#include <errnoLib.h>
#endif


typedef void (*FPTR) (int);

void sigMasterHandler
    (
    int     sig,                /* caught signal */
#ifdef _WRS_KERNEL
    int      code,
#else
    siginfo_t * pInfo,          /* signal info */
#endif
    struct sigcontext *pContext /* unused */
    );

/*****************************************************************************
*
* main - entry point for the queued signal demo
*
* This routine acts the task entry point in the case of the demo spawned as a
* kernel task. It also can act as a RTP entry point in the case of RTP based
* demo.
*/

STATUS main (void)
    {
    sigset_t sig = sigmask (SIGUSR1);
    union sigval sval;
    struct sigaction in;

    sigprocmask (SIG_UNBLOCK, &sig, NULL);

    in.sa_handler = (FPTR) sigMasterHandler;
    in.sa_flags = 0;
    (void) sigemptyset (&in.sa_mask);

    if (sigaction (SIGUSR1, &in, NULL) != OK)
        {
        printf ("Unable to set up handler for task (0x%x)\n", taskIdCurrent);
        return (ERROR);
    }

    printf ("Task 0x%x installed signal handler for signal # %d.\
    Ready for signal.\n", taskIdCurrent,  SIGUSR1);
    for (;;);

    }

/*************************************************************************
*
* sigMasterHandler - signal handler
*
* This routine is the signal handler for the SIGUSR1 signal
*/

void sigMasterHandler
    (
    int     sig,                /* caught signal */
#ifdef _WRS_KERNEL
    int     code,
#else
    siginfo_t * pInfo ,         /* signal info */
#endif
    struct sigcontext *pContext /* unused */
    )
    {
    printf ("Task 0x%x got signal # %d  signal value %d \n",
taskIdCurrent, sig,
#ifdef _WRS_KERNEL
            code
#else
            pInfo->si_value.sival_int
#endif
           );
    }

/*************************************************************************
*
* sig - helper routine to send a queued signal
*
* This routine can send a queued signal to a kernel task or RTP task or RTP.
* <id> is the ID of the receiver entity. <value> is the value to be sent
* along with the signal. The signal number being sent is SIGUSR1.
*/

#ifdef _WRS_KERNEL
STATUS sig
    (
    int id,
    int val
    )
    {
    union sigval        valueCode;

    valueCode.sival_int = val;
    if (TASK_ID_VERIFY (id) == OK)
        {
        if (IS_KERNEL_TASK (id))
            {
            if (sigqueue (id, SIGUSR1, valueCode) == ERROR)
            {
            printf ("Unable to send SIGUSR1 signal to 0xx%x, errno = 0x%x\n",
                    id, errnoGet());
        return ERROR;
        }
            }
        else
            {
            rtpTaskSigqueue ((WIND_TCB *)id, SIGUSR1, valueCode);
            }
        }
    else if (OBJ_VERIFY ((RTP_ID)id, rtpClassId) != ERROR)
        {
        rtpSigqueue ((RTP_ID)id, SIGUSR1, valueCode);
        }
    else
        {
        return (ERROR);
        }

    return (OK);
    }
#endif
```

The code provided in this example can be used to do any of the following:

- Send a queued signal to a kernel task.
- Send a queued signal to a task in a process (RTP).
- Send a queued signal to a process.

The **sig( )** routine provided in this code is a helper routine used to send a queued signal.

To use the code as a kernel application, VxWorks must be configured with **BUNDLE_NET_SHELL** and **BUNDLE_POSIX**.

To send a queued signal to a kernel task:

1. Build the code with the VxWorks image or as a downloadable kernel module. Boot the system. If the code is not linked to the system image, load the module; for example, from the kernel shell:

   ```
   -> ld < signal_ex.o
   value = 8382064 = 0x7fe670
   ```

2. Spawn a task with **main( )** as the entry point. For example:

   ```
   -> sp main
   Task spawned: id = 0x7fd620, name = t1
   value = 8377888 = 0x7fd620
   -> Task 0x7fd620 installed signal handler for signal # 30.  Ready for
   signal.
   sp main
   ```

3. Send a queued signal to the spawned kernel task. From the kernel shell use the command:

   sig *kernelTaskId*, *signalValue*

   For example:

   ```
   -> sig 0x7fd620, 20
   value = 0 = 0x0
   -> Task 0x7fd620 got signal # 30  signal value 20
   sig 0x7fd620, 20
   value = 0 = 0x0
   -> Task 0x7fd620 got signal # 30  signal value 20
   ```

For information on using the code in a process (as an RTP application), see the *VxWorks Application Programmer's Guide: Multitasking*.

4.18.4 Signal Events

The signal event facility allows a pthread or task to receive notification that a particular event has occurred (such as the arrival of a message at a message queue, or the firing of a timer) by way of a signal.

The following routines can be used to register for signal notification of their respective event activities: **mq_notify( )**, **timer_create( )**, **timer_open( )**, **aio_read( )**, **aio_write( )** and **lio_listio( )**.

The POSIX 1003.1-2001 standard defines three signal event notification types:

SIGEV_NONE
Indicates that no notification is required when the event occurs. This is useful for applications that use asynchronous I/O with polling for completion.

SIGEV_SIGNAL
Indicates that a signal is generated when the event occurs.

SIGEV_THREAD
Provides for callback functions for asynchronous notifications done by a function call within the context of a new thread. This provides a multi-threaded process with a more natural means of notification than signals. VxWorks supports this option in user space (processes), but not in the kernel.

The notification type is specified using the **sigevent** structure, which is defined in *installDir***/vxworks-6.x/target/h/sigeventCommon.h**. A pointer the structure is used in the call to register for signal notification; for example, with **mq_notify( )**.

To use the signal event facility, configure VxWorks with the **INCLUDE_SIGEVENT** component.

4.18.5 Signal Handlers

Signals are more appropriate for error and exception handling than as a general-purpose intertask communication mechanism. And normally, signal handlers should be treated like ISRs: no routine should be called from a signal handler that might cause the handler to block. Because signals are asynchronous, it is difficult to predict which resources might be unavailable when a particular signal is raised.

To be perfectly safe, call only those routines listed in Table 4-20. Deviate from this practice only if you are certain that your signal handler cannot create a deadlock situation.

In addition, you should be particularly careful when using C++ for a signal handler or when invoking a C++ method from a signal handler written in C or assembly. Some of the issues involved in using C++ include the following:

- The VxWorks **intConnect( )** and **signal( )** routines require the address of the function to execute when the interrupt or signal occurs, but the address of a non-static member function cannot be used, so static member functions must be implement.
- Objects cannot be instantiated or deleted in signal handling code.
- C++ code used to execute in a signal handler should restrict itself to Embedded C++. No exceptions nor run-time type identification (RTTI) should be used.

Table 4-20 **Routines Callable by Signal Handlers**

Library	Routines
bLib	All routines
errnoLib	**errnoGet( )**, **errnoSet( )**
eventLib	**eventSend( )**
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( )**
rngLib	All routines except **rngCreate( )** and **rngDelete( )**
semLib	**semGive( )** except mutual-exclusion semaphores, **semFlush( )**
sigLib	**kill( )**
taskLib	**taskSuspend( )**, **taskResume( )**, **taskPrioritySet( )**, **taskPriorityGet( )**, **taskIdVerify( )**, **taskIdDefault( )**, **taskIsReady( )**, **taskIsSuspended( )**, **taskTcb( )**

Table 4-20 **Routines Callable by Signal Handlers** (cont'd)

Library	Routines
tickLib	**tickAnnounce( )**, **tickSet( )**, **tickGet( )**
tyLib	**tyIRd( )**, **tyITx( )**
vxLib	**vxTas( )**, **vxMemProbe( )**
wdLib	**wdStart( )**, **wdCancel( )**

Most signals are delivered asynchronously to the execution of a program. Therefore programs must be written to account for the unexpected occurrence of signals, and handle them gracefully. Unlike ISR's, signal handlers execute in the context of the interrupted task.

VxWorks does not distinguish between normal task execution and a signal context, as it distinguishes between a task context and an ISR. Therefore the system has no way of distinguishing between a task execution context and a task executing a signal handler. To the system, they are the same.

When you write signal handlers make sure that they:

- Release resources prior to exiting:
  - Free any allocated memory.
  - Close any open files.
  - Release any mutual exclusion resources such as semaphores.
- Leave any modified data structures in a sane state.
- Notify the kernel with an appropriate error return value.

Mutual exclusion between signal handlers and tasks must be managed with care. In general, users should avoid the following activity in signal handlers:

- Taking mutual exclusion (such as semaphores) resources that can also be taken by any other element of the application code. This can lead to deadlock.
- Modifying any shared data memory that may have been in the process of modification by any other element of the application code when the signal was delivered. This compromises mutual exclusion and leads to data corruption.
- Using **longjmp( )** to change the flow of task execution. If **longjmp( )** is used in a signal handler to re-initialize a running task, you must ensure that the signal is not sent to the task while the task is holding a critical resource (such as a kernel mutex). For example, if a signal is sent to a task that is executing

malloc( ), the signal handler that calls **longjmp( )** could leave the kernel in an inconsistent state.

These scenarios are very difficult to debug, and should be avoided. One safe way to synchronize other elements of the application code and a signal handler is to set up dedicated flags and data structures that are set from signal handlers and read from the other elements. This ensures a consistency in usage of the data structure. In addition, the other elements of the application code must check for the occurrence of signals at any time by periodically checking to see if the synchronizing data structure or flag has been modified in the background by a signal handler, and then acting accordingly. The use of the **volatile** keyword is useful for memory locations that are accessed from both a signal handler and other elements of the application.

Taking a mutex semaphore in a signal handler is an especially bad idea. Mutex semaphores can be taken recursively. A signal handler can therefore easily re-acquire a mutex that was taken by any other element of the application. Since the signal handler is an asynchronously executing entity, it has thereby broken the mutual exclusion that the mutex was supposed to provide.

Taking a binary semaphore in a signal handler is an equally bad idea. If any other element has already taken it, the signal handler will cause the task to block on itself. This is a deadlock from which no recovery is possible. Counting semaphores, if available, suffer from the same issue as mutexes, and if unavailable, are equivalent to the binary semaphore situation that causes an unrecoverable deadlock.

On a general note, the signal facility should be used only for notifying/handling exceptional or error conditions. Usage of signals as a general purpose IPC mechanism or in the data flow path of an application can cause some of the pitfalls described above.

4.19 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. Functions invoked by watchdog timers execute as interrupt service code at the interrupt level of the system clock. Restrictions on ISRs apply to routines connected to watchdog timers. The functions in Table 4-21 are provided by the **wdLib** library.

Table 4-21 **Watchdog Timer Calls**

Call	Description
wdCreate( )	Allocates and initializes a watchdog timer.
wdDelete( )	Terminates and deallocates a watchdog timer.
wdStart( )	Starts a watchdog timer.
wdCancel( )	Cancels 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 4-6 **Watchdog Timers**

```
/* 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);
    /* ... */
    }
```

For information about POSIX timers, see *5.6 POSIX Clocks and Timers*, p.259.

4.20 Interrupt Service Routines

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, VxWorks runs interrupt service routines (ISRs) in a special context outside of any task's context. Thus, interrupt handling involves no task context switch. Table 4-22 lists the interrupt routines provided in **intLib** and **intArchLib**.

Table 4-22 **Interrupt Routines**

Call	Description
intConnect( )	Connects a C routine to an interrupt vector.
intContext( )	Returns TRUE if called from interrupt level.
intCount( )	Gets the current interrupt nesting depth.
intLevelSet( )	Sets the processor interrupt mask level.
intLock( )	Disables interrupts.
intUnlock( )	Re-enables interrupts.
intVecBaseSet( )	Sets the vector base address.
intVecBaseGet( )	Gets the vector base address.
intVecSet( )	Sets an exception vector.
intVecGet( )	Gets an exception vector.

For information about interrupt locks and latency, see *4.11.1 Interrupt Locks and Latency*, p.197.

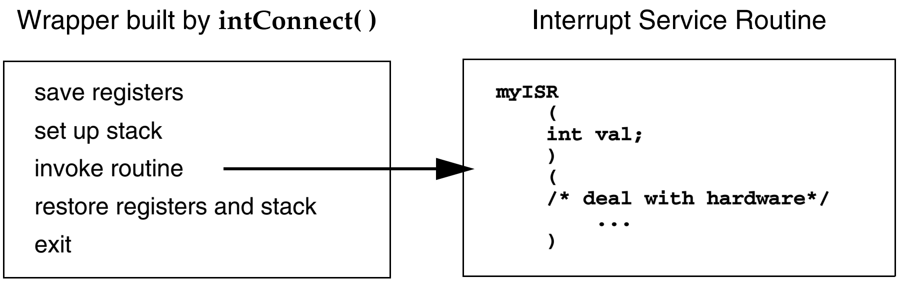
NOTE: The **intLock( )** and **intUnlock( )** routines are provided for the UP configuration of VxWorks, but not the SMP configuration. Several alternative are available for SMP systems, including the ISR-callable spinlock, which default to **intLock( )** and **intUnlock( )** behavior in a UP system. For more information, see *15.6.1 ISR-Callable Spinlocks*, p.682 and *15.15 Migrating Code to VxWorks SMP*, p.702.

4.20.1 Connecting Routines 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 4-15.

Figure 4-15 **Routine Built by intConnect( )**


Wrapper built by intConnect( )
Interrupt Service Routine
save registers
set up stack
invoke routine
restore registers and stack
exit

```
myISR
    (
    int val;
    )
    (
    /* deal with hardware*/
       ...
    )
```

```
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( )**.

4.20.2 Interrupt Stack

All ISRs use the same *interrupt stack*. This stack is allocated and initialized by the system at startup according to specified configuration parameters. It must be large enough to handle the worst possible combination of nested interrupts.

⚠ **CAUTION:** Some architectures do not permit using a separate interrupt stack, and ISRs use the stack of the interrupted task. With such architectures, make sure to 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 VxWorks reference for your BSP to determine whether your architecture supports a separate interrupt stack. If it does not, also see *Task Stack Protection*, p.176.

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

In addition to experimenting with stack size, you can also configure and test systems with guard zone protection for interrupt stacks (for more information, see *Interrupt Stack Protection*, p.243).

Filling Interrupt Stacks

By default, interrupt (and task) stacks are filled with 0xEE. Filling stacks is useful during development for debugging with the **checkStack( )** routine. It is generally not used in deployed systems because not filling stacks provides better performance. You can use the **VX_GLOBAL_NO_STACK_FILL** configuration parameter (when you configure VxWorks) to disable stack filling for all interrupts (and tasks) in the system.

Interrupt Stack Protection

Systems can be configured with the **INCLUDE_PROTECT_INTERRUPT_STACK** component to provide guard zone protection at the start and end of the interrupt stack. If the size of the system becomes an issue, the component can be removed for final testing and the deployed system.

An overrun guard zone prevents a task from going beyond the end of its predefined stack size and ensures that the integrity of the system is not compromised. An under-run guard zone typically prevents buffer overflows from corrupting memory above the stack. The CPU generates an exception when a task attempts to access any of the guard zones. The size of a stack is always rounded up to a multiple the MMU page size when a guard zone is inserted.

The sizes of the guard zones are defined by the following configuration parameters:

- **INTERRUPT_STACK_OVERFLOW_SIZE** for interrupt stack overflow size.
- **INTERRUPT_STACK_UNDERFLOW_SIZE** for interrupt stack underflow size.

The value of these parameters can be modified to increase the size of the guard zone. The size of a guard zone is rounded up to the CPU MMU page size. The insertion of a guard zone can be prevented by setting the parameter to zero.

4.20.3 Writing and Debugging ISRs

There are some restrictions on the routines you can call from an ISR. For example, you cannot use routines like **printf( )**, **malloc( )**, and **semTake( )** in your ISR. You can, however, use **semGive( )**, **logMsg( )**, **msgQSend( )**, and **bcopy( )**. For more information, see *4.20.5 Special Limitations of ISRs*, p.245.

Two basic techniques for debugging an ISR are to use **logMsg( )** and to have global variables that are incremented each time though the ISR. You can then print the globals from the shell. You can also use global variables from the shell to turn on different condition flows through the ISR.

4.20.4 ISRs and the Kernel Work Queue

The VxWorks kernel reduces interrupt latency to a minimum by protecting portions of its critical sections using a work deferral mechanism as opposed to locking interrupts. Work deferral consists of storing kernel work requests performed by ISRs that interrupt the kernel while it is in one of its critical sections. For example, an ISR that performs a **semGive( )** after having interrupted the kernel in a critical section would cause work to be stored in the work queue. This work is processed after the ISR returns and immediately after the kernel exits its critical section. This process involves a static buffer, also known as work queue, used to store work requests until they are processed. This is all internal to the VxWorks kernel and users must never make use of the work queue. However, it is possible for the work queue to overflow in situations where a large number of interrupts cause work deferral without allowing the kernel to complete execution of the critical section that was originally interrupted. These situations are uncommon and are often symptoms of ill-behaved interrupt service routines. A work queue overflow is also known as a work queue panic in reference to the message the kernel displays as a result of an overflow:

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

In order to help reduce the occurrences of work queue overflows, system architects can use the **WIND_JOBS_MAX** kernel configuration parameter to increase the size of the kernel work queue. However in most cases this is simply hiding the root cause of the overflow.

4.20.5 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 and has no task control block, so all ISRs share a single stack.

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 for ISR use, and is the most common way to print messages from ISRs. For more information, see the VxWorks API reference 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**.

In addition, you should be particularly careful when using C++ for an ISR or when invoking a C++ method from an ISR written in C or assembly. Some of the issues involved in using C++ include the following:

- The VxWorks **intConnect( )** routine require the address of the function to execute when the interrupt occurs, but the address of a non-static member function cannot be used, so static member functions must be implement.
- Objects cannot be instantiated or deleted in ISR code.
- C++ code used to execute in an ISR should restrict itself to Embedded C++. No exceptions nor run-time type identification (RTTI) should be used.

All VxWorks utility libraries, such as the linked-list and ring-buffer libraries, can be used by ISRs. As discussed earlier (*4.5 Task Error Status: errno*, p.184), 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 4-23 lists routines that can be called from ISRs.

Table 4-23 **Routines Callable by ISRs**

Library	Routine
bLib	All routines
errnoLib	**errnoGet( )**, **errnoSet( )**
eventLib	**eventSend( )**
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( )**
rngLib	All routines except **rngCreate( )** and **rngDelete( )**
pipeDrv	**write( )**
selectLib	**selWakeup( )**, **selWakeupAll( )**
semLib	**semGive( )** except mutual-exclusion semaphores, **semFlush( )**
semPxLib	**sem_post( )**
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( )**

4.20.6 Exceptions at Interrupt Level

When a task causes a hardware exception such as an 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 loader tests for the presence of the exception description in low memory and if it is detected, display it on the system console. The boot loader's **e** command re-displays the exception description; see *Wind River Workbench User's Guide: Setting up Your Hardware*.

One example of such an exception is the following 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. (See *4.20.4 ISRs and the Kernel Work Queue*, p.244.)

4.20.7 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. For information about architecture-specific implementations of **intLockLevelSet( )**, see the *VxWorks Architecture Supplement*.

⚠ **CAUTION:** Some hardware prevents masking certain interrupt levels; check the hardware manufacturer's documentation.

4.20.8 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. The effective result is that the ISR cannot safely make any call to any VxWorks function, except reboot.

For more information, see the *VxWorks Architecture Supplement* for the architecture in question.

WARNING: The use of NMI with any VxWorks functionality, other than reboot, is not recommended. Routines marked as *interrupt safe* do not imply they are NMI safe and, in fact, are usually the very ones that NMI routines must not call (because they typically use **intLock( )** to achieve the interrupt safe condition).

4.20.9 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.

- **VxWorks Events**

  ISRs can send VxWorks events to tasks.

5
POSIX Facilities

5.1 Introduction

VxWorks provides extensive POSIX support in many of its native kernel libraries. To facilitate application portability, VxWorks provides additional POSIX interfaces as optional components. In the kernel, VxWorks implements some of the traditional interfaces described by the POSIX standard IEEE Std 1003.1 (POSIX.1) as well as many of the real-time interfaces in the POSIX.1 optional functionality.

For detailed information about POSIX standards and facilities, see The Open Group Web sites at **http://www.opengroup.org/** and **http://www.unix.org/**.

POSIX and Real-Time Systems

While VxWorks provides many POSIX compliant APIs, not all POSIX APIs are suitable for embedded and real-time systems, or are entirely compatible with the VxWorks operating system architecture. In a few cases, therefore, Wind River has imposed minor limitations on POSIX functionality to serve either real-time systems or VxWorks compatibility. For example:

- Swapping memory to disk is not appropriate in real-time systems, and VxWorks provides no facilities for doing so. It does, however, provide POSIX page-locking routines to facilitate porting code to VxWorks. The routines otherwise provide no useful function—pages are always locked in VxWorks systems (for more information see *5.9 POSIX Page-Locking Interface*, p.264).
- VxWorks tasks are scheduled on a system-wide basis; processes themselves cannot be scheduled. As a consequence, while POSIX access routines allow two values for contention scope (**PTHREAD_SCOPE_SYSTEM** and **PTHREAD_SCOPE_PROCESS**), only system-wide scope is implemented in VxWorks for these routines (for more information, see *5.10 POSIX Threads*, p.264 and *5.12 POSIX and VxWorks Scheduling*, p.277).

Any such limitations on POSIX functionality are identified in this chapter, or in other chapters of this guide that provide more detailed information on specific POSIX APIs.

POSIX and VxWorks Facilities

This chapter describes the POSIX support provided by VxWorks and VxWorks-specific POSIX extensions. In addition, it compares native VxWorks facilities with similar POSIX facilities that are also available with VxWorks.

The qualifier *VxWorks* is used in this chapter to identify native non-POSIX APIs for purposes of comparison with POSIX APIs. For example, you can find a discussion of VxWorks semaphores contrasted to POSIX semaphores in *5.13.1 Comparison of*

POSIX and VxWorks Semaphores, p.290, although POSIX semaphores are also implemented in VxWorks.

VxWorks extensions to POSIX are identified as such.

NOTE: This chapter provides information about POSIX facilities available in the kernel. For information about facilities available for real-time processes (RTPs), see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

5.2 Configuring VxWorks with POSIX Facilities

VxWorks provides extensive POSIX support for the kernel with many of its libraries (see *5.3 General POSIX Support*, p.255), but the default configuration of VxWorks does not include many other POSIX facilities that may be used. The optional VxWorks components that provide support for individual POSIX libraries are described in this section.

General POSIX support for kernel space can be provided with the **BUNDLE_POSIX** component bundle. If memory constraints require a finer-grained configuration, individual components can be used for selected features. See the configuration instructions for individual POSIX features throughout this chapter for more information in this regard.

⚠ **CAUTION:** The set of components used for POSIX support in kernel space is not the same as the set of components used for POSIX support in user space. For information about the components for user space, see Table 5-1, and see the *VxWorks Application Programmer's Guide: POSIX Facilities* for the appropriate component bundle.

5.2.1 VxWorks Components for POSIX Facilities

Table 5-1 provides an overview of the individual VxWorks components that must be configured in the kernel to provide support for the specified POSIX facilities.

Networking facilities are described in the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

Table 5-1 **VxWorks Components Providing POSIX Facilities**

POSIX Facility	**Required VxWorks Component**	
	for Kernel	**for Processes**
Standard C library	**INCLUDE_ANSI_*** components	Dinkum C library (libc)
Asynchronous I/O with system driver	**INCLUDE_POSIX_AIO**, **INCLUDE_POSIX_AIO_SYSDRV** and **INCLUDE_PIPES**	**INCLUDE_POSIX_CLOCKS** and **INCLUDE_POSIX_TIMERS**
Clocks	**INCLUDE_POSIX_CLOCKS**	**INCLUDE_POSIX_CLOCKS**
Directory and file utilities	**INCLUDE_POSIX_DIRLIB**	N/A
ftruncate( )	**INCLUDE_POSIX_FTRUNC**	N/A
Memory locking	**INCLUDE_POSIX_MEM**	N/A
Memory management	N/A	**INCLUDE_RTP**
Memory-mapped files	N/A	**INCLUDE_POSIX_MAPPED_FILES**
Shared memory objects	N/A	**INCLUDE_POSIX_MAPPED_FILES** and **INCLUDE_POSIX_SHM**
Message queues	**INCLUDE_POSIX_MQ**	**INCLUDE_POSIX_MQ**
pthreads	**INCLUDE_POSIX_THREADS**	**INCLUDE_POSIX_CLOCKS**, **INCLUDE_POSIX_PTHREAD_SCHEDULE**, and **INCLUDE_PTHREAD_CPUTIME**
Process Scheduling API	**INCLUDE_POSIX_SCHED**	N/A
Semaphores	**INCLUDE_POSIX_SEM**	**INCLUDE_POSIX_SEM**
Signals	**INCLUDE_POSIX_SIGNALS**	N/A
Timers	**INCLUDE_POSIX_TIMERS**	**INCLUDE_POSIX_TIMERS**
Trace	N/A	**INCLUDE_POSIX_TRACE**

Table 5-1 **VxWorks Components Providing POSIX Facilities** (cont'd)

POSIX Facility	**Required VxWorks Component**	
	for Kernel	**for Processes**
POSIX PSE52 support	N/A	**BUNDLE_RTP_POSIX_PSE52**

5.3 General POSIX Support

Many POSIX-compliant libraries are provided for VxWorks. These libraries are listed in Table 5-2; see the API references for these libraries for detailed information.

Table 5-2 **POSIX Libraries**

Functionality	Library
Asynchronous I/O	**aioPxLib**
Buffer manipulation	**bLib**
Clock facility	**clockLib**
Directory handling	**dirLib**
Environment handling	C Library
File duplication	**iosLib**
File management	**fsPxLib** and **ioLib**
I/O functions	**ioLib**
Options handling	**getopt**
POSIX message queues	**mqPxLib**
POSIX semaphores	**semPxLib**
POSIX timers	**timerLib**
POSIX threads	**pthreadLib**
Standard I/O and some ANSI	C Library
Math	C Library
Memory allocation	**memLib** and **memPartLib**
Network/Socket APIs	network libraries
String manipulation	C Library
Trace facility	**pxTraceLib**

The following sections of this chapter describe the optional POSIX API components that are provided in addition to the native VxWorks APIs.

CAUTION: Wind River advises that you do not use both POSIX libraries and native VxWorks libraries that provide similar functionality. Doing so may result in undesirable interactions between the two, as some POSIX APIs manipulate resources that are also used by native VxWorks APIs. For example, do not use **tickLib** routines to manipulate the system's tick counter if you are also using **clockLib** routines, do not use the **taskLib** API to change the priority of a POSIX thread instead of the **pthread** API, and so on.

Checking for POSIX Support at Run-time

A POSIX application can use the following APIs at run-time to determine the status of POSIX support in the system:

- The **sysconf( )** routine returns the current values of the configurable system variables, allowing an application to determine whether an optional feature is supported or not, and the precise value of system's limits.
- The **confstr( )** routine returns a string associated with a system variable. With this release, the **confstr( )** routine returns a string only for the system's default path.
- The **uname( )** routine lets an application get information about the system on which it is running. The identification information provided for VxWorks is the system name (**VxWorks**), the network name of the system, the system's release number, the machine name (BSP model), the architecture's endianness, the kernel version number, the processor name (CPU family), the BSP revision level, and the system's build date.

5.4 POSIX Header Files

The POSIX 1003.1 standard defines a set of header files as part of the application development environment. The VxWorks user-side development environment provides more POSIX header files than the kernel’s, and their content is also more in agreement with the POSIX standard than the kernel header files.

The POSIX header files available for the kernel development environment are listed in Table 5-3.

Table 5-3 **POSIX Header Files**

Header File	Description
aio.h	asynchronous input and output
assert.h	verify program assertion
ctype.h	character types
dirent.h	format of directory entries
errno.h	system error numbers
fcntl.h	file control options
limits.h	implementation-defined constants
locale.h	category macros
math.h	mathematical declarations
mqueue.h	message queues
pthread.h	pthreads
sched.h	execution scheduling
semaphore.h	semaphores
setjmp.h	stack environment declarations
signal.h	signals
stdio.h	standard buffered input/output
stdlib.h	standard library definitions
string.h	string operations
sys/mman.h	memory management declarations
sys/resource.h	definitions for XSI resource operations
sys/stat.h	data returned by the **stat( )** function
sys/types.h	data types
sys/un.h	definitions for UNIX domain sockets

Table 5-3 **POSIX Header Files** (cont'd)

Header File	Description
time.h	time types
trace.h	trace facility
unistd.h	standard symbolic constants and types
utime.h	access and modification times structure

5.5 POSIX Namespace

POSIX namespace isolation can be provided for user-mode (RTP) applications. For information in this regard, see the *VxWorks Application Programmer's Guide: POSIX Facilities*.

5.6 POSIX Clocks and Timers

VxWorks provides POSIX 1003.1b standard clock and timer interfaces.

POSIX Clocks

POSIX defines various software (virtual) clocks, which are identified as the **CLOCK_REALTIME** clock, **CLOCK_MONOTONIC** clock, process CPU-time clocks, and thread CPU-time clocks. These clocks all use one system hardware timer.

The real-time clock and the monotonic clock are system-wide clocks, and are therefore supported for both the VxWorks kernel and processes. The process CPU-time clocks are not supported in VxWorks. The thread CPU-time clocks are supported for POSIX threads running in processes. A POSIX thread can use the real-time clock, the monotonic clock, and a thread CPU-time clock for its application.

For information about thread CPU-time clocks, see the *VxWorks Application Programmer's Guide: POSIX Facilities*.

The real-time clock can be reset (but only from the kernel). The monotonic clock cannot be reset, and provides the time that has elapsed since the system booted.

The real-time clock can be accessed with the POSIX clock and timer routines by using the *clock_id* parameter **CLOCK_REALTIME**. A real-time clock can be reset at run time with a call to **clock_settime( )** from within the kernel (not from a process).

The monotonic clock can be accessed by calling **clock_gettime( )** with a *clock_id* parameter of **CLOCK_MONOTONIC**. A monotonic clock keeps track of the time that has elapsed since system startup; that is, the value returned by **clock_gettime( )** is the amount of time (in seconds and nanoseconds) that has passed since the system booted. A monotonic clock cannot be reset. Applications can therefore rely on the fact that any measurement of a time interval that they might make has not been falsified by a call to **clock_settime( )**.

Both **CLOCK_REALTIME** and **CLOCK_MONOTONIC** are defined in **time.h**.

See Table 5-4 for a list of the POSIX clock routines. The obsolete VxWorks-specific POSIX extension **clock_setres( )** is provided for backwards-compatibility purposes. For more information about clock routines, see the API reference for **clockLib**.

Table 5-4 **POSIX Clock Routines**

Routine	Description
clock_getres( )	Get the clock resolution (**CLOCK_REALTIME** and **CLOCK_MONOTONIC**).
clock_setres( )	Set the clock resolution. Obsolete VxWorks-specific POSIX extension.
clock_gettime( )	Get the current clock time (**CLOCK_REALTIME** and **CLOCK_MONOTONIC**).
clock_settime( )	Set the clock to a specified time for **CLOCK_REALTIME** (fails for **CLOCK_MONOTONIC**; not supported for a thread CPU-time clock in the kernel).

To include the **clockLib** library in the system, configure VxWorks with the **INCLUDE_POSIX_CLOCKS** component. For thread CPU-time clocks, the **INCLUDE_POSIX_PTHREAD_SCHEDULER** and **INCLUDE_POSIX_THREAD_CPUTIME** components must be used as well.

POSIX Timers

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.

Timers are created based on clocks. In the kernel, the **CLOCK_REALTIME** and **CLOCK_MONOTONIC** clocks are supported for timers. In processes, the **CLOCK_REALTIME** clock, **CLOCK_MONOTONIC** clock, and thread CPU-time clocks (including **CLOCK_THREAD_CPUTIME_ID** clock) are supported.

When a timer goes off, the default signal, **SIGALRM**, is sent to the task. To install a signal handler that executes when the timer expires, use the **sigaction( )** routine (see *4.18 Signals*, p.226).

See Table 5-5 for a list of the POSIX timer routines. The VxWorks **timerLib** library includes a set of VxWorks-specific POSIX extensions: **timer_open( )**, **timer_close( )**, **timer_cancel( )**, **timer_connect( )**, and **timer_unlink( )**. These routines allow for an easier and more powerful use of POSIX timers on VxWorks. For more information, see the VxWorks API reference for **timerLib**.

Table 5-5 **POSIX Timer Routines**

Routine	Description
timer_create( )	Allocate a timer using the specified clock for a timing base (**CLOCK_REALTIME** or **CLOCK_MONOTONIC**).
timer_delete( )	Remove a previously created timer.
timer_open( )	Open a named timer. VxWorks-specific POSIX extension.
timer_close( )	Close a named timer. VxWorks-specific POSIX extension.
timer_gettime( )	Get the remaining time before expiration and the reload value.
timer_getoverrun( )	Return the timer expiration overrun.
timer_settime( )	Set the time until the next expiration and arm timer.
timer_cancel( )	Cancel a timer. VxWorks-specific POSIX extension.

Table 5-5 **POSIX Timer Routines** (cont'd)

Routine	Description
timer_connect( )	Connect a user routine to the timer signal. VxWorks-specific POSIX extension.
timer_unlink( )	Unlink a named timer. VxWorks-specific POSIX extension.
nanosleep( )	Suspend the current pthread (task) until the time interval elapses.
sleep( )	Delay for a specified amount of time.
alarm( )	Set an alarm clock for delivery of a signal.

Example 5-1 **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);
    }
```

The POSIX **nanosleep( )** routine provides specification of sleep or delay time in units of seconds and nanoseconds, in contrast to the ticks used by the VxWorks **taskDelay( )** function. Nevertheless, the precision of both is the same, and is determined by the system clock rate; only the units differ.

To include the **timerLib** library in a system, configure VxWorks with the **INCLUDE_POSIX_TIMERS** component.

5.7 POSIX Asynchronous I/O

POSIX asynchronous I/O (AIO) routines are provided by the **aioPxLib** library. The VxWorks AIO implementation meets the specification of the POSIX 1003.1 standard. For more information, see *7.7 Asynchronous Input/Output*, p.381.

5.8 POSIX Advisory File Locking

POSIX advisory file locking provides byte-range locks on POSIX-conforming files (for VxWorks, this means files in an HRFS file system). The VxWorks implementation meets the specification of the POSIX 1003.1 standard.

POSIX advisory file locking is provided through the **fcntl( )** file control function. To include POSIX advisory file locking facilities in VxWorks, configure the system with the **INCLUDE_POSIX_ADVISORY_FILE_LOCKING** component.

The VxWorks implementation of advisory file locking involves a behavioral difference with regard to deadlock detection because VxWorks processes are not scheduled. Note that this distinction only matters if you have multiple pthreads (or tasks) within one process (RTP).

According to POSIX, advisory locks are identified by a process ID, and when a process exits all of its advisory locks are destroyed, which is true for VxWorks. But because VxWorks processes cannot themselves be scheduled, individual advisory locks on a given byte range of a file have two owners: the pthread (or task) that actually holds the lock, and the process that contains the pthread. In addition, the calculation of whether one lock would deadlock another lock is done on a pthread basis, rather than a process basis.

This means that deadlocks are detected if the pthread requesting a new lock would block on any pthread (in any given process) that is currently blocked (whether directly or indirectly) on any advisory lock that is held by the requesting pthread. Immediate-blocking detection (**F_SETLK** requests) always fail immediately if the requested byte range cannot be locked without waiting for some other lock, regardless of the identity of the owner of that lock.

5.9 POSIX Page-Locking Interface

The real-time extensions of the POSIX 1003.1 standard are used with operating systems that perform paging and swapping. On such systems, applications that attempt real-time performance can use the POSIX *page-locking* facilities to protect certain blocks of memory from paging and swapping.

VxWorks does not support memory paging and swapping because the serious delays in execution time that they cause are undesirable in a real-time system. However, page-locking routines can be included in VxWorks to facilitate porting POSIX applications to VxWorks.

These routines do not perform any function, as all pages are always kept in memory.

The POSIX page-locking routines are part of the memory management library, **mmanPxLib**, and are listed in Table 5-6.

Table 5-6 **POSIX Page-Locking Routines**

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

To include the **mmanPxLib** library in the system, configure VxWorks with the **INCLUDE_POSIX_MEM** component.

5.10 POSIX Threads

POSIX threads (also known as *pthreads*) are similar to VxWorks tasks, but with additional characteristics. In VxWorks pthreads are implemented on top of native tasks, but maintain pthread IDs that differ from the IDs of the underlying tasks.

The main reasons for including POSIX thread support in VxWorks are the following:

- For porting POSIX applications to VxWorks.
- To make use of the POSIX thread scheduler in real-time processes (including concurrent scheduling policies).

For information about POSIX thread scheduler, see *5.12 POSIX and VxWorks Scheduling*, p.277.

5.10.1 POSIX Thread Attributes

A major difference between VxWorks tasks and POSIX threads is the way in which options and settings are specified. For VxWorks tasks these options are set with the task creation API, usually **taskSpawn( )**.

POSIX threads, on the other hand, have characteristics that are called *attributes*. Each attribute contains a set of values, and a set of *access routines* to retrieve and set those values. You specify all pthread attributes before pthread creation in the attributes object **pthread_attr_t**. In a few cases, you can dynamically modify the attribute values of a pthread after its creation.

5.10.2 VxWorks-Specific Pthread Attributes

The VxWorks implementation of POSIX threads provides two additional pthread attributes (which are POSIX extensions)—pthread *name* and pthread *options*—as well as routines for accessing them.

Pthread Name

While POSIX threads are not named entities, the VxWorks tasks upon which they are based are named. By default the underlying task elements are named **pthr***Number* (for example, **pthr3**). The number part of the name is incremented each time a new thread is created (with a roll-over at 2^32 - 1). It is, however, possible to name these tasks using the thread name attribute.

- Attribute Name: **threadname**
- Possible Values: a null-terminated string of characters
- Default Value: none (the default naming policy is used)
- Access Functions (VxWorks-specific POSIX extensions): **pthread_attr_setname( )** and **pthread_attr_getname( )**

Pthread Options

POSIX threads are agnostic with regard to target architecture. Some VxWorks tasks, on the other hand, may be created with specific options in order to benefit from certain features of the architecture. For example, for the Altivec-capable PowerPC architecture, tasks must be created with the **VX_ALTIVEC_TASK** in order to make use of the Altivec processor. The pthread options attribute can be used to set such options for the VxWorks task upon which the POSIX thread is based.

- Attribute Name: **threadoptions**
- Possible Values: the same as the VxWorks task options. See **taskLib.h**
- Default Value: none (the default task options are used)
- Access Functions (VxWorks-specific POSIX extensions): **pthread_attr_setopt( )** and **pthread_attr_getopt( )**

5.10.3 Specifying Attributes when Creating Pthreads

The following examples create a pthread using the default attributes and use explicit attributes.

Example 5-2 **Creating a pthread Using Explicit Scheduling Attributes**

```
pthread_t tid;
pthread_attr_t attr;
int ret;

pthread_attr_init(&attr);

/* set the inheritsched attribute to explicit */
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

/* set the schedpolicy attribute to SCHED_FIFO */
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);

/* create the pthread */
ret = pthread_create(&tid, &attr, entryFunction, entryArg);
```

Example 5-3 **Creating a pthread Using Default Attributes**

```
pthread_t tid;
int ret;

/* create the pthread with NULL attributes to designate default values */
ret = pthread_create(&tid, NULL, entryFunction, entryArg);
```

Example 5-4 **Designating Your Own Stack for a pthread**

```
pthread_t threadId;
pthread_attr_t attr;
void * stackaddr = NULL;
int stacksize = 0;

/* initialize the thread's attributes */

pthread_attr_init (&attr);

/*
 * Allocate memory for a stack region for the thread. Malloc() is used
 * for simplification since a real-life case is likely to use
memPartAlloc()
 * on the kernel side, or mmap() on the user side.
 */

stacksize = 2 * 4096 /* let's allocate two pages */ stackaddr = malloc
(stacksize);

if (stackbase == NULL)
    {
    printf ("FAILED: mystack: malloc failed\n");
    return (-1);
    }

/* set the stackaddr attribute */

pthread_attr_setstackaddr (&attr, stackaddr);

/* set the stacksize attribute */

pthread_attr_setstacksize (&attr, stacksize);

/* set the schedpolicy attribute to SCHED_FIFO */

pthread_attr_setschedpolicy (&attr, SCHED_FIFO);

/* create the pthread */

ret = pthread_create (&threadId, &attr, mystack_thread, 0);
```

5.10.4 POSIX Thread Creation and Management

VxWorks provides many POSIX thread routines. Table 5-7 lists a few that are directly relevant to pthread creation or execution. See the VxWorks API reference for information about the other routines, and more details about all of them.

Table 5-7 **POSIX Thread Routines**

Routine	Description
pthread_create( )	Create a pthread.
pthread_cancel( )	Cancel the execution of a pthread
pthread_detach( )	Detach a running pthread so that it cannot be joined by another pthread.
pthread_join( )	Wait for a pthread to terminate.
pthread_getschedparam( )	Dynamically set value of scheduling priority attribute.
pthread_setschedparam( )	Dynamically set scheduling priority and policy parameter.
sched_get_priority_max( )	Get the maximum priority that a pthread can get.
sched_get_priority_min( )	Get the minimum priority that a pthread can get.
sched_rr_get_interval( )	Get the time quantum of execution of the round-robin policy.
sched_yield( )	Relinquishes the CPU.

5.10.5 POSIX Thread Attribute Access

The POSIX attribute-access routines are described in Table 5-8. The VxWorks-specific POSIX extension routines are described in section *5.10.2 VxWorks-Specific Pthread Attributes*, p.265.

Table 5-8 **POSIX Thread Attribute-Access Routines**

Routine	Description
pthread_attr_getstacksize( )	Get value of the stack size attribute.
pthread_attr_setstacksize( )	Set the stack size attribute.
pthread_attr_getstackaddr( )	Get value of stack address attribute.
pthread_attr_setstackaddr( )	Set value of stack address attribute.
pthread_attr_getdetachstate( )	Get value of *detachstate* attribute (joinable or detached).
pthread_attr_setdetachstate( )	Set value of *detachstate* attribute (joinable or detached).
pthread_attr_getscope( )	Get contention scope. Only **PTHREAD_SCOPE_SYSTEM** is supported for VxWorks.
pthread_attr_setscope( )	Set contention scope. Only **PTHREAD_SCOPE_SYSTEM** is supported for VxWorks.
pthread_attr_getinheritsched( )	Get value of scheduling-inheritance attribute.
pthread_attr_setinheritsched( )	Set value of scheduling-inheritance attribute.
pthread_attr_getschedpolicy( )	Get value of the scheduling-policy attribute (which is not used by default).
pthread_attr_setschedpolicy( )	Set scheduling-policy attribute (which is not used by default).
pthread_attr_getschedparam( )	Get value of scheduling priority attribute.
pthread_attr_setschedparam( )	Set scheduling priority attribute.

Table 5-8 **POSIX Thread Attribute-Access Routines** (cont'd)

Routine	Description
pthread_attr_getopt( )	Get the task options applying to the pthread. VxWorks-specific POSIX extension.
pthread_attr_setopt( )	Set non-default task options for the pthread. VxWorks-specific POSIX extension.
pthread_attr_getname( )	Get the name of the pthread. VxWorks-specific POSIX extension.
pthread_attr_setname( )	Set a non-default name for the pthread. VxWorks-specific POSIX extension.

5.10.6 POSIX Thread Private Data

POSIX threads can store and access private data; that is, pthread-specific data. They use a *key* maintained for each pthread by the pthread library to access that data. A key corresponds to a location associated with the data. It is created by calling **pthread_key_create( )** and released by calling **pthread_key_delete( )**. The location is accessed by calling **pthread_getspecific( )** and **pthread_setspecific( )**. This location represents a pointer to the data, and not the data itself, so there is no limitation on the size and content of the data associated with a key.

The pthread library supports a maximum of 256 keys for all the pthreads in the kernel.

The **pthread_key_create( )** routine has an option for a destructor function, which is called when the creating pthread exits or is cancelled, if the value associated with the key is non-NULL.

This destructor function frees the storage associated with the data itself, and not with the key. It is important to set a destructor function for preventing memory leaks to occur when the pthread that allocated memory for the data is cancelled. The key itself should be freed as well, by calling **pthread_key_delete( )**, otherwise the key cannot be reused by the pthread library.

5.10.7 POSIX Thread Cancellation

POSIX provides a mechanism, called *cancellation*, to terminate a pthread gracefully. There are two types of cancellation: *deferred* and *asynchronous*.

Deferred cancellation causes the pthread to explicitly check to see if it was cancelled. This happens in one of the two following ways:

- The code of the pthread executes calls to **pthread_testcancel( )** at regular intervals.
- The pthread calls a function that contains a *cancellation point* during which the pthread may be automatically cancelled.

Asynchronous cancellation causes the execution of the pthread to be forcefully interrupted and a handler to be called, much like a signal.[1]

Automatic cancellation points are library routines that can block the execution of the pthread for a lengthy period of time.

NOTE: While the **msync( )**, **fcntl( )**, and **tcdrain( )** routines are mandated POSIX 1003.1 cancellation points, they are not provided with VxWorks for this release.

The POSIX cancellation points provided in VxWorks libraries are described in Table 5-9.

[1] Asynchronous cancellation is actually implemented with a special signal, **SIGCNCL**, which users should be careful not to block or to ignore.

Table 5-9 **Pthread Cancellation Points in VxWorks Libraries**

Library	Routines
aioPxLib	**aio_suspend( )**
ioLib	**creat( )**, **open( )**, **read( )**, **write( )**, **close( )**, **fsync( )**, **fdatasync( )**
mqPxLib	**mq_receive( )**, **mq_send( )**
pthreadLib	**pthread_cond_timedwait( )**, **pthread_cond_wait( )**, **pthread_join( )**, **pthread_testcancel( )**
semPxLib	**sem_wait( )**
sigLib	**pause( )**, **sigsuspend( )**, **sigtimedwait( )**, **sigwait( )**, **sigwaitinfo( )**
timerLib	**sleep( )**, **nanosleep( )**

Routines that can be used with cancellation points of pthreads are listed in Table 5-10.

Table 5-10 **Pthread Cancellation Routines**

Routine	Description
pthread_cancel( )	Cancel execution of a pthread.
pthread_testcancel( )	Create a cancellation point in the calling pthread.
pthread_setcancelstate( )	Enables or disables cancellation.
pthread_setcanceltype( )	Selects deferred or asynchronous cancellation.
pthread_cleanup_push( )	Registers a function to be called when the pthread is cancelled, exits, or calls **pthread_cleanup_pop( )** with a non-null *run* parameter.
pthread_cleanup_pop( )	Unregisters a function previously registered with **pthread_cleanup_push( )**. This function is immediately executed if the *run* parameter is non-null.

5.11 POSIX Thread Mutexes and Condition Variables

Pthread mutexes (mutual exclusion variables) and condition variables provide compliance with the POSIX 1003.1c standard. Like POSIX threads, mutexes and condition variables have *attributes* associated with them. Mutex attributes are held in a data type called **pthread_mutexattr_t**, which contains two attributes, **protocol** and **prioceiling**.

The routines used to manage these attributes are described below. For more information about these and other mutex routines, see the API reference for **pthreadLib**.

5.11.1 Thread Mutexes

The routines that can be used to act directly on a mutex object and on the mutex attribute object are listed in Table 5-11 and Table 5-12 (respectively).

Table 5-11 **POSIX Routines Acting on a Mutex Object**

Routine	Description
pthread_mutex_destroy( )	Destroy a mutex.
pthread_mutex_init( )	Initialize a mutex.
pthread_mutex_getprioceiling( )	Get the priority ceiling of a mutex.
pthread_mutex_setprioceiling( )	Set the priority ceiling of a mutex.
pthread_mutex_lock( )	Lock a mutex.
pthread_mutex_trylock( )	Check and lock a mutex if available.
pthread_mutex_unlock( )	Unlock a mutex.

Table 5-12 **POSIX Routines Acting on a Mutex Attribute Object**

Routine	Description
pthread_mutexattr_init( )	Initialize mutex attributes object.
pthread_mutexattr_destroy( )	Destroy mutex attributes object.
pthread_mutexattr_getprioceiling( )	Get **prioceiling** attribute of mutex attributes object.
pthread_mutexattr_setprioceiling( )	Set **prioceiling** attribute of mutex attributes object.
pthread_mutexattr_getprotocol( )	Get **protocol** attribute of mutex attributes object.
pthread_mutexattr_setprotocol( )	Set **protocol** attribute of mutex attributes object.

Protocol Mutex Attribute

The **protocol** mutex attribute defines how the mutex variable deals with the priority inversion problem (which is described in the section for VxWorks mutual-exclusion semaphores; see *4.12.3 Mutual-Exclusion Semaphores*, p.205).

- Attribute Name: **protocol**
- Possible Values: **PTHREAD_PRIO_NONE**, **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**
- Access Routines: **pthread_mutexattr_getprotocol( )** and **pthread_mutexattr_setprotocol( )**

The **PTHREAD_PRIO_INHERIT** option is the default value of the protocol attribute for pthreads created in the kernel (unlike pthreads created in processes, for which the default is **PTHREAD_PRIO_NONE**).

The **PTHREAD_PRIO_INHERIT** value is used to create a mutex with priority inheritance—and is equivalent to the association of **SEM_Q_PRIORITY** and **SEM_INVERSION_SAFE** options used with **semMCreate( )**. A pthread owning a mutex variable created with the **PTHREAD_PRIO_INHERIT** value inherits the priority of any higher-priority pthread waiting for the mutex and executes at this elevated priority until it releases the mutex, at which points it returns to its original priority.

Because it might not be desirable to elevate a lower-priority pthread to a priority above a certain level, POSIX defines the notion of priority ceiling, described below. Mutual-exclusion variables created with *priority protection* use the **PTHREAD_PRIO_PROTECT** value.

Priority Ceiling Mutex Attribute

The **prioceiling** attribute is the POSIX priority ceiling for mutex variables created with the **protocol** attribute set to **PTHREAD_PRIO_PROTECT**.

- Attribute Name: **prioceiling**
- Possible Values: any valid (POSIX) priority value (0-255, with zero being the lowest).
- Access Routines: **pthread_mutexattr_getprioceiling( )** and **pthread_mutexattr_setprioceiling( )**
- Dynamic Access Routines: **pthread_mutex_getprioceiling( )** and **pthread_mutex_setprioceiling( )**

Note that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *5.12.2 POSIX and VxWorks Priority Numbering*, p.279.

A priority ceiling is defined by the following conditions:

- Any pthread attempting to acquire a mutex, whose priority is higher than the ceiling, cannot acquire the mutex.
- Any pthread whose priority is lower than the ceiling value has its priority elevated to the ceiling value for the duration that the mutex is held.
- The pthread's priority is restored to its previous value when the mutex is released.

5.11.2 Condition Variables

A pthread condition variable corresponds to an object that permits pthreads to synchronize on an event or state represented by the value of a variable. This is a more complicated type of synchronization than the one allowed by mutexes only. Its main advantage is that is allows for passive waiting (as opposed to active waiting or polling) on a change in the value of the variable. Condition variables are used in conjunction with mutexes (one mutex per condition variable). The routines that can be used to act directly on a condition variable and on the condition variable attribute object are listed in Table 5-11 and Table 5-12 (respectively).

Table 5-13 **POSIX Routines Acting on a Condition Variable Object**

Routine	Description
pthread_cond_destroy( )	Destroy condition variables.
pthread_cond_init( )	Initialize condition variables.
pthread_cond_broadcast( )	Broadcast a condition.
pthread_cond_signal( )	Signal a condition.
pthread_cond_wait( )	Wait on a condition.
pthread_cond_timedwait( )	Wait on a condition with timeout.

Table 5-14 **POSIX Routines Acting on a Condition Variable Attribute Object**

Routine	Description
pthread_condattr_destroy( )	Destroy condition variable attributes object.
pthread_condattr_init( )	Initialize condition variable attributes object.

5.12 POSIX and VxWorks Scheduling

VxWorks can be configured with either the traditional (native) VxWorks scheduler or with a POSIX thread scheduler. Neither can be used to schedule processes (RTPs). The only entities that can be scheduled in VxWorks are tasks and pthreads.

The VxWorks implementation of a POSIX thread scheduler is an enhancement of the traditional VxWorks scheduler that provides additional scheduling facilities for pthreads running in processes.

With either scheduler, VxWorks tasks and pthreads share a single priority range and the same global scheduling scheme. With the POSIX thread scheduler, however, pthreads running in processes may have individual (concurrent) scheduling policies. Note that VxWorks must be configured with the POSIX thread scheduler in order to run pthreads in processes.

NOTE: Wind River recommends that you do not use both POSIX APIs and VxWorks APIs in the same application. Doing so may make a POSIX application non-compliant.

Table 5-15 provides an overview of how scheduling works for tasks and pthreads, for each of the schedulers, in both the kernel and processes (RTPs). The key differences are the following:

- The POSIX thread scheduler provides POSIX scheduling support for threads running in processes.
- In all other cases, the POSIX thread scheduler schedules pthreads and tasks in the same (non-POSIX) manner as the traditional VxWorks scheduler. (There is a minor difference between how it handles tasks and pthreads whose priorities have been lowered; see *Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities*, p.285.)
- The traditional VxWorks scheduler cannot be used to schedule pthreads in processes. In fact, pthreads cannot be started in processes unless VxWorks is configured with the POSIX thread scheduler.

The information provided in Table 5-15 is discussed in detail in subsequent sections.

Table 5-15 **Task and Pthread Scheduling in the Kernel and in Processes**

Execution Environment	POSIX Thread Scheduler		Traditional VxWorks Scheduler	
	Tasks	Pthreads	Tasks	Pthreads
Kernel	Priority-based preemptive, or round-robin scheduling.	Same as task scheduling. No concurrent scheduling policies.	Priority-based preemptive, or round-robin scheduling.	Same as task scheduling. No concurrent scheduling policies.
Processes	Priority-based preemptive, or round robin scheduling.	POSIX FIFO, round-robin, sporadic, or other (system default). Concurrent scheduling policies available.	Priority-based preemptive, or round-robin scheduling.	N/A. Pthreads cannot be run in processes with traditional VxWorks scheduler.[a]

[a] The traditional VxWorks scheduler cannot ensure behavioral compliance with the POSIX 1 standard.

5.12.1 Differences in POSIX and VxWorks Scheduling

In general, the POSIX scheduling model and scheduling in a VxWorks system differ in the following ways—regardless of whether the system is configured with the Wind River POSIX thread scheduler or the traditional VxWorks scheduler:

- POSIX supports a two-level scheduling model that includes the concept of *contention scope*, by which the scheduling of pthreads can apply system wide or on a process basis. In VxWorks, on the other hand, processes (RTPs) cannot themselves be scheduled, and tasks and pthreads are scheduled on a system-wide (kernel and processes) basis.

- POSIX applies scheduling policies on a process-by-process and thread-by-thread basis. VxWorks applies scheduling policies on a system-wide basis, for all tasks and pthreads, whether in the kernel or in processes. This means that all tasks and pthreads use either a preemptive priority scheme or a round-robin scheme. The only exception to this rule is that pthreads executing in processes can be subject to concurrent (individual) scheduling policies, including sporadic scheduling (note that the POSIX thread scheduler must be used in this case).

- POSIX supports the concept of *scheduling allocation domain*; that is, the association between processes or threads and processors. Since VxWorks does not support multi-processor hardware, there is only one domain on VxWorks and all the tasks and pthreads are associated to it.

- The POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *5.12.2 POSIX and VxWorks Priority Numbering*, p.279.

- VxWorks does not support the POSIX thread-concurrency feature, as all threads are scheduled. The POSIX thread-concurrency APIs are provided for application portability, but they have no effect.

5.12.2 POSIX and VxWorks Priority Numbering

The POSIX priority numbering scheme is the inverse of the VxWorks priority numbering scheme. In POSIX, the higher the number, the higher the priority. In VxWorks, the *lower* the number, the higher the priority, where 0 is the highest priority.

The priority numbers used with the POSIX scheduling library, **schedPxLib**, do not, therefore, match those used and reported by all other components of VxWorks. You can change the default POSIX numbering scheme by setting the global variable **posixPriorityNumbering** to **FALSE**. If you do so, **schedPxLib** uses the

VxWorks numbering scheme (a smaller number means a higher priority) and its priority numbers match those used by the other components of VxWorks.

In the following sections, discussions of pthreads and tasks *at the same priority level* refer to functionally equivalent priority levels, and not to priority numbers.

5.12.3 Default Scheduling Policy

All VxWorks tasks and pthreads are scheduled according to the system-wide default scheduling policy. The only exception to this rule is for pthreads running in user mode (in processes). In this case, concurrent scheduling policies that differ from the system default can be applied to pthreads.

Note that pthreads can be run in processes only if VxWorks is configured with the POSIX thread scheduler; they cannot be run in processes if VxWorks is configured with the traditional scheduler.

The system-wide default scheduling policy for VxWorks, regardless of which scheduler is used, is priority-based preemptive scheduling—which corresponds to the POSIX **SCHED_FIFO** scheduling policy.

At run-time the active system-wide default scheduling policy can be changed to round-robin scheduling with the **kernelTimeSlice( )** routine. It can be changed back by calling **kernelTimeSlice( )** with a parameter of zero. VxWorks round-robin scheduling corresponds to the POSIX **SCHED_RR** policy.

The **kernelTimeSlice( )** routine cannot be called in user mode (that is, from a process). A call with a non-zero parameter immediately affects all kernel and user tasks, all kernel pthreads, and all user pthreads using the **SCHED_OTHER** policy. Any user pthreads running with the **SCHED_RR** policy are unaffected by the call; but those started after it use the newly defined timeslice.

5.12.4 VxWorks Traditional Scheduler

The VxWorks traditional scheduler can be used with both tasks and pthreads in the kernel. It cannot be used with pthreads in processes. If VxWorks is configured with the traditional scheduler, a **pthread_create( )** call in a process fails and the errno is set to **ENOSYS**.

The traditional VxWorks scheduler schedules pthreads as if they were tasks. All tasks and pthreads executing in a system are therefore subject to the current default scheduling policy (either the priority-based preemptive policy or the round-robin scheduling policy; see *5.12.3 Default Scheduling Policy*, p.280), and concurrent policies cannot be applied to individual pthreads. For general information about the traditional scheduler and how it works with tasks, see *4.3 Task Scheduling*, p.166.

The scheduling options provided by the traditional VxWorks scheduler are similar to the POSIX ones. The following pthreads scheduling policies correspond to the traditional VxWorks scheduling policies:

- **SCHED_FIFO** is similar to VxWorks priority-based preemptive scheduling. There are differences as to where tasks or pthreads are placed in the ready queue if their priority is lowered; see *Caveats About Scheduling Behavior with the POSIX Thread Scheduler*, p.284.
- **SCHED_RR** corresponds to VxWorks round-robin scheduling.
- **SCHED_OTHER** corresponds to the current system-wide default scheduling policy. The **SCHED_OTHER** policy is the default policy for pthreads in VxWorks.

There is no VxWorks traditional scheduler policy that corresponds to **SCHED_SPORADIC**.

Configuring VxWorks with the Traditional Scheduler

VxWorks is configured with the traditional scheduler by default. This scheduler is provided by the **INCLUDE_VX_TRADITIONAL_SCHEDULER** component.

Caveats About Scheduling Behavior with the VxWorks Traditional Scheduler

Concurrent scheduling policies are not supported for pthreads in the kernel, and care must therefore be taken with pthread scheduling-inheritance and scheduling policy attributes.

If the scheduling-inheritance attribute is set to **PTHREAD_EXPLICIT_SCHED** and the scheduling policy to **SCHED_FIFO** or **SCHED_RR**, and this policy does not match the current system-wide default scheduling policy, the creation of pthreads fails.

Wind River therefore recommends that you always use **PTHREAD_INHERIT_SCHED** (which is the default) as a scheduling-inheritance attribute. In this case the current VxWorks scheduling policy applies, and the parent pthread's priority is used. Or, if the pthread must be started with a different priority than its parent, the scheduling-inheritance attribute can be set to **PTHREAD_EXPLICIT_SCHED** and the scheduling policy attribute set to be **SCHED_OTHER** (which corresponds to the current system-wide default scheduling policy.).

In order to take advantage of the POSIX scheduling model, VxWorks must be configured with the POSIX thread scheduler, and the pthreads in question must be run in processes (RTPs). See *5.12.5 POSIX Threads Scheduler*, p.282.

5.12.5 POSIX Threads Scheduler

The POSIX thread scheduler can be used to schedule both pthreads and tasks in a VxWorks system. Note that the purpose of the POSIX thread scheduler is to provide POSIX scheduling support for pthreads running in processes. There is no reason to use it in a system that does not require this support (kernel-only systems, or systems with processes but without pthreads).

The POSIX thread scheduler is *required* for running pthreads in processes, where it provides compliance with POSIX 1003.1 for pthread scheduling (including concurrent scheduling policies). If VxWorks is not configured with the POSIX thread scheduler, pthreads cannot be created in processes.

NOTE: The POSIX priority numbering scheme is the inverse of the VxWorks scheme, so references to *a given priority level* or *same level* in comparisons of these schemes refer to functionally equivalent priority levels, and not to priority numbers. For more information about the numbering schemes see *5.12.2 POSIX and VxWorks Priority Numbering*, p.279.

Scheduling in the Kernel

The POSIX thread scheduler schedules *kernel tasks* and *kernel pthreads* in the same manner as the traditional VxWorks task scheduler. See *4.3 Task Scheduling*, p.166 for information about the traditional scheduler and how it works with VxWorks tasks, and *5.12.4 VxWorks Traditional Scheduler*, p.281 for information about how

POSIX scheduling policies correspond to the traditional VxWorks scheduling policies.

Scheduling in Processes

When VxWorks is configured with the POSIX thread scheduler, *tasks* executing in processes are scheduled according to system-wide default scheduling policy. On the other hand, *pthreads* executing in processes are scheduled according to POSIX 1003.1. Scheduling policies can be assigned to each pthread and changed dynamically. The scheduling policies are as follows:

- **SCHED_FIFO** is a preemptive priority scheduling policy. For a given priority level, pthreads scheduled with this policy are handled as peers of the VxWorks tasks at the same level. There is a slight difference in how pthreads and tasks are handled if their priorities are lowered (for more information; see *Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities*, p.285).
- **SCHED_RR** is a per-priority round-robin scheduling policy. For a given priority level, all pthreads scheduled with this policy are given the same time of execution (time-slice) before giving up the CPU.
- **SCHED_SPORADIC** is a policy used for aperiodic activities, which ensures that the pthreads associated with the policy are served periodically at a high priority for a bounded amount of time, and a low background priority at all other times.
- **SCHED_OTHER** corresponds to the scheduling policy currently in use for VxWorks tasks, which is either preemptive priority or round-robin. Pthreads scheduled with this policy are submitted to the system's global scheduling policy, exactly like VxWorks tasks or kernel pthreads.

Note the following with regard to the VxWorks implementation of the **SCHED_SPORADIC** policy:

- The system periodic clock is used for time accounting.
- Dynamically changing the scheduling policy to **SCHED_SPORADIC** is not supported; however, dynamically changing the policy from **SCHED_SPORADIC** to another policy is supported.
- VxWorks does not impose an upper limit on the maximum number of replenishment events with the **SS_REPL_MAX** macro. A default of 40 events is set with the **sched_ss_max_repl** field of the thread attribute structure, which can be changed.

Configuring VxWorks with the POSIX Thread Scheduler

To configure VxWorks with the POSIX thread scheduler, add the **INCLUDE_POSIX_PTHREAD_SCHEDULER** component to the kernel.

Note that only the **SCHED_FIFO**, **SCHED_RR**, and **SCHED_OTHER** scheduling policies are provided with the **INCLUDE_POSIX_PTHREAD_SCHEDULER** component. For the **SCHED_SPORADIC** scheduling policy, the **INCLUDE_PX_SCHED_SPORADIC_POLICY** component must be included as well. The bundle **BUNDLE_RTP_POSIX_PSE52** includes the **INCLUDE_PX_SCHED_SPORADIC_POLICY** component as well as the **INCLUDE_POSIX_PTHREAD_SCHEDULER** component.

The configuration parameter **POSIX_PTHREAD_RR_TIMESLICE** may be used to configure the default time slicing interval for pthreads started with the **SCHED_RR** policy. To modify the time slice at run time, call **kernelTimeSlice( )** with a different time slice value. The new time slice value only affects pthreads created after the **kernelTimeSlice( )** call.

NOTE: The **INCLUDE_POSIX_PTHREAD_SCHEDULER** component is a standalone component. It is not dependent on any other POSIX components nor is it automatically included with any other components.

The POSIX thread scheduler must be added explicitly with either the **INCLUDE_POSIX_PTHREAD_SCHEDULER** component or the **BUNDLE_RTP_POSIX_PSE52** bundle.

The POSIX thread scheduler component is independent because it is intended to be used *only* with pthreads in processes; kernel-only systems that use pthreads, have no need to change from the default VxWorks traditional scheduler.

Caveats About Scheduling Behavior with the POSIX Thread Scheduler

Using the POSIX thread scheduler involves a few complexities that should be taken into account when designing your system. Care should be taken with regard to the following:

- Using both round-robin and priority-based preemptive scheduling policies.
- Running pthreads with the individual **SCHED_OTHER** policy.
- Differences in re-queuing pthreads and tasks with lowered priorities.
- Backwards compatibility issues for POSIX applications designed for the VxWorks traditional scheduler.

Using both Round-Robin and Priority-Based Preemptive Policies

Using a combination of round-robin and priority-based preemptive policies for tasks and pthreads of the same priority level can lead to task or pthread CPU starvation for the entities running with the round-robin policy.

When VxWorks is running with round-robin scheduling as the system default, tasks may not run with their expected time slice if there are pthreads running at the same priority level with the concurrent (individual) **SCHED_FIFO** policy. This is because one of the pthreads may monopolize the CPU and starve the tasks. Even if the usurper pthread is preempted, it stays at the head of its priority lists (as POSIX mandates), and continues to monopolize the CPU when that priority level can run again. Pthreads scheduled with the **SCHED_RR** or **SCHED_OTHER** policy are at the same disadvantage as the tasks scheduled with the round-robin policy.

Similarly, when VxWorks is running with preemptive scheduling as the system default, tasks may starve pthreads with the same priority level if the latter have the concurrent (individual) **SCHED_RR** policy.

Running pthreads with the Concurrent SCHED_OTHER Policy

Pthreads created with the concurrent (individual) **SCHED_OTHER** policy behave the same as the system-wide default scheduling policy, which means that:

- If the system default is currently priority-based preemptive scheduling, the **SCHED_OTHER** pthreads run with the preemptive policy.
- If the system default is currently round-robin scheduling, the **SCHED_OTHER** pthreads run with the round-robin policy.

While changing the default system policy from priority-based preemptive scheduling to round-robin scheduling (or the opposite) changes the effective scheduling policy for pthreads created with **SCHED_OTHER**, it has no effect on pthreads created with **SCHED_RR** or **SCHED_FIFO**.

Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities

The POSIX thread scheduler re-queues pthreads that have had their priority lowered differently than it re-queues tasks that have had their priority lowered. The difference is as follows:

- When the priority of a pthread is lowered (with the **pthread_setschedprio( )** routine), the POSIX thread scheduler places it at the *head* of the priority list.
- When the priority of a task is lowered (with the **taskPrioritySet( )** routine), the POSIX thread scheduler places it at the *tail* of the priority list—which is the same as what the traditional VxWorks scheduler would do.

What this means is that lowering the priority of a task and a pthread may have a different effect on when they will run (if there are other tasks or pthreads in their priority list). For example, if a task and a pthread each have their priority lowered to effectively the same level, the pthread will be at the head of the priority list and the task at the end. The pthread will run before any other pthreads or tasks at this level, and the task after any other pthreads or tasks.

Backwards Compatibility Issues for Applications

Using the POSIX thread scheduler changes the behavior of POSIX applications that were written to run with the traditional VxWorks scheduler. For existing POSIX applications that require backward-compatibility, the scheduling policy can be changed to **SCHED_OTHER** for all pthreads. This causes their policy to default to the active VxWorks task scheduling policy (as was the case before the introduction of the POSIX thread scheduler).

5.12.6 POSIX Scheduling Routines

The POSIX 1003.1b scheduling routines provided by the **schedPxLib** library for VxWorks are described in Table 5-16.

Table 5-16 **POSIX Scheduling Routines**

Routine	Description
sched_get_priority_max( )	Gets the maximum pthread priority.
sched_get_priority_min( )	Gets the minimum pthread priority.
sched_rr_get_interval( )	If round-robin scheduling is in effect, gets the time slice length.
sched_yield( )	Relinquishes the CPU.

For more information about these routines, see the **schedPxLib** API reference.

NOTE: Several scheduling routines that were provided with **schedPxLib** for VxWorks 5.*x* and early versions of VxWorks 6.*x* are not POSIX compliant, and are maintained only for backward compatibility in the kernel. The use of these routines is deprecated: **sched_setparam( )**, **sched_getparam( )**, **sched_setscheduler( )**, and **sched_getscheduler( )**.

The native VxWorks routines **taskPrioritySet( )** and **taskPriorityGet( )** should be used for task priorities. The POSIX routines **pthread_setschedparam( )** and **pthread_getschedparam( )** should be used for pthread priorities.

For information about changing the default system scheduling policy, see *5.12.3 Default Scheduling Policy*, p.280. For information about concurrent scheduling policies, see *5.12.5 POSIX Threads Scheduler*, p.282.

Note that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *5.12.2 POSIX and VxWorks Priority Numbering*, p.279.

To include the **schedPxLib** library in the system, configure VxWorks with the **INCLUDE_POSIX_SCHED** component.

5.12.7 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, 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 5-5 **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);
    }
```

5.13 POSIX Semaphores

POSIX defines both *named* and *unnamed* semaphores, which have the same properties, but which use slightly different interfaces. The POSIX semaphore library provides routines for creating, opening, and destroying both named and unnamed semaphores.

When opening a named semaphore, you assign a symbolic name,[2] which the other named-semaphore routines accept as an argument. The POSIX semaphore routines provided by **semPxLib** are shown in Table 5-17.

Table 5-17 **POSIX Semaphore Routines**

Routine	Description
sem_init( )	Initializes an unnamed semaphore.
sem_destroy( )	Destroys an unnamed semaphore.
sem_open( )	Initializes/opens a named semaphore.
sem_close( )	Closes a named semaphore.
sem_unlink( )	Removes 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.
sem_timedwait( )	Lock a semaphore with a timeout.

To include the POSIX **semPxLib** library semaphore routines in the system, configure VxWorks with the **INCLUDE_POSIX_SEM** component.

[2] Some 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, named semaphores can be used to share semaphores between real-time processes. In the VxWorks kernel there is no need for named semaphores, because all kernel objects have unique identifiers. However, using named semaphores of the POSIX variety provides a convenient way of determining the object's ID.

VxWorks also provides **semPxLibInit( )**, a non-POSIX (kernel-only) routine that initializes the kernel's POSIX semaphore library. It is called by default at boot time when POSIX semaphores have been included in the VxWorks configuration.

5.13.1 Comparison of POSIX and VxWorks Semaphores

POSIX semaphores are *counting* semaphores; that is, they keep track of the number of times they are given. The VxWorks semaphore mechanism is similar to that specified by POSIX, except that VxWorks semaphores offer these 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
- queuing mechanism options

When these features are important, VxWorks semaphores are preferable to POSIX semaphores. (For information about these features, see *4. Multitasking*.)

The POSIX terms *wait* (or *lock*) and *post* (or *unlock*) correspond to the VxWorks terms *take* and *give*, respectively. The POSIX routines for locking, unlocking, and getting the value of semaphores are used for both named and unnamed semaphores.

The routines **sem_init( )** and **sem_destroy( )** are used for initializing and destroying unnamed semaphores only. The **sem_destroy( )** call terminates an unnamed semaphore and deallocates all associated memory.

The routines **sem_open( )**, **sem_unlink( )**, and **sem_close( )** are for opening and closing (destroying) named semaphores only. The combination of **sem_close( )** and **sem_unlink( )** has the same effect for named semaphores as **sem_destroy( )** does for unnamed semaphores. That is, it terminates the semaphore and deallocates the associated memory.

⚠ **WARNING:** When deleting semaphores, particularly mutual-exclusion semaphores, avoid deleting a semaphore still required by another task. Do not delete a semaphore unless the deleting task first succeeds in locking that semaphore. Similarly for named semaphores, close semaphores only from the same task that opens them.

5.13.2 Using Unnamed Semaphores

When using unnamed semaphores, typically 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( )**, lets you 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( )**.

Semaphores can be used for both synchronization and exclusion. Thus, 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 that is 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, setting the semaphore to 0 (locked). Subsequent tasks will block until the semaphore is released. As with the previous scenario, when the semaphore is released the task with the highest priority is unblocked.

Example 5-6 **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);

/************************************************************************
  * unnameSem - test case for unamed semaphores
  *
  * This routine tests unamed semaphores.
  *
  * RETURNS: N/A
  *
  * ERRNOS: N/A
  */

void unnameSem (void)
     {
     sem_t * pSem;

     /* reserve memory for semaphore */

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

     if (pSem == NULL)
         {
         printf ("pSem allocation failed\n");
         return;
         }

     /* initialize semaphore to unavailable */

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

     /* create sync task */

     printf ("unnameSem: spawning task...\n");
     if (taskSpawn ("tSyncTask", 90, 0, 2000, syncTask, pSem) == ERROR)
         {
         printf ("Failed to spawn tSyncTask\n");
         sem_destroy (pSem);
         free ((char *) pSem);
         return;
         }

     /* 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");
         sem_destroy (pSem);
         free ((char *) pSem);
         return;
         }

     /* all done - destroy semaphore */

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

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 */
     }
```

5.13.3 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 specified.

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

The results, based on the flags and whether the semaphore accessed already exists, are shown in Table 5-18.

Table 5-18 **Possible Outcomes of Calling sem_open( )**

Flag Settings	If Semaphore Exists	If 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.
O_EXCL	Routine fails.	Routine fails.

Once initialized, a semaphore remains usable until explicitly destroyed. Tasks can explicitly mark a semaphore for destruction at any time, but the system only destroys the semaphore when no task has the semaphore open.

If VxWorks is configured with **INCLUDE_POSIX_SEM_SHOW**, you can use **show( )** from the shell (with the C interpreter) to display information about a POSIX semaphore.[3]

[3] The **show( )** routine is not a POSIX routine, nor is it meant to be used programmatically. It is designed for interactive use with the shell (with the shell's C interpreter).

This example shows information about the POSIX semaphore **mySem** with two tasks blocked and waiting for it:

```
-> show semId
value = 0 = 0x0
Semaphore name          :mySem
sem_open() count        :3
Semaphore value         :0
No. of blocked tasks    :2
```

Note that **show( )** takes the semaphore ID as the argument.

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 must 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 then unlocking it with **sem_post( )** when the task is finished with the semaphore.

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. If a task tries to open the semaphore without the **O_CREAT** flag, the operation fails. An unlinked semaphore is deleted by the system when the last task closes it.

NOTE: POSIX named semaphores may be shared between processes only if their names start with a **/** (forward slash) character. They are otherwise private to the process in which they were created, and cannot be accessed from another process. See *4.9 Public and Private Objects*, p.194.

Example 5-7 **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 <taskLib.h>
#include <stdio.h>
#include <semaphore.h>
#include <fcntl.h>

/* forward declaration */

void syncSemTask (char * name);

/****************************************************************************
*
* nameSem - test program for POSIX semaphores
*
* This routine opens a named semaphore and spawns a task, tSyncSemTask, which
* waits on the named semaphore.
*
* RETURNS: N/A
*
* ERRNO: N/A
*/

void 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");
    if (taskSpawn ("tSyncSemTask", 90, 0, 4000, (FUNCPTR) syncSemTask,
                   (int) name, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ("nameSem: unable to spawn tSyncSemTask\n");
        sem_close(semId);
        return;
        }

    /* 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");
        sem_close(semId);
        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");
    }

/****************************************************************************
*
* syncSemTask - waits on a named POSIX semaphore
*
* This routine waits on the named semaphore created by nameSem().
*
* RETURNS: N/A
*
* ERRNO: N/A
*/

void 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;
    }
}
```

5.14 POSIX Message Queues

The POSIX message queue routines, provided by **mqPxLib**, are shown in Table 5-19.

Table 5-19 **POSIX Message Queue Routines**

Routine	Description
mq_open( )	Opens a message queue.
mq_close( )	Closes a message queue.
mq_unlink( )	Removes a message queue.
mq_send( )	Sends a message to a queue.
mq_receive( )	Gets a message from a queue.
mq_notify( )	Signals a task that a message is waiting on a queue.
mq_setattr( )	Sets a queue attribute.
mq_getattr( )	Gets a queue attribute.
mq_timedsend( )	Sends a message to a queue, with a timeout.
mq_timedreceive( )	Gets a message from a queue, with a timeout.

Note that there are behavioral differences between the kernel and user space versions of **mq_open( )**. The kernel version allows for creation of a message queue for any permission specified by the *oflags* parameter. The user-space version complies with the POSIX PSE52 profile, so that after the first call, any subsequent calls in the same process are only allowed if an equivalent or lower permission is specified.

For information about the use of permissions with the user-space version of **mq_open( )**, see the *VxWorks Application Programmer's Guide: POSIX Facilities*.

The VxWorks initialization routine **mqPxLibInit( )** initializes the kernel's POSIX message queue library (this is a kernel-only routine). It is called automatically at boot time when the **INCLUDE_POSIX_MQ** component is part of the system.

For information about the VxWorks message queue library, see the **msgQLib** API reference.

5.14.1 Comparison of POSIX and VxWorks Message Queues

POSIX message queues are similar to VxWorks message queues, except that POSIX message queues provide messages with a range of priorities. The differences are summarized in Table 5-20.

Table 5-20 **Message Queue Feature Comparison**

Feature	VxWorks Message Queues	POSIX Message Queues
Maximum Message Queue Levels	1 (specified by **MSG_PRI_NORMAL | MSG_PRI_URGENT**)	32 (specified by **MAX_PRIO_MAX**)
Blocked Message Queues	FIFO or priority-based	Priority-based
Received with Timeout	**msgQReceive( )** option	**mq_timedreceive( )** (user-space only)
Task Notification	With VxWorks message queue events	**mq_notify( )**
Close/Unlink Semantics	With **msgQOpen** library	Yes
Send with Timeout	**msgQsend( )** option	**mq_timesend( )** (user-space only)

5.14.2 POSIX Message Queue Attributes

A POSIX message queue has the following attributes:

- an optional **O_NONBLOCK** flag, which prevents a **mq_receive( )** call from being a blocking call if the message queue is empty
- 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 using **mq_setattr( )**, and get the values of all the attributes using **mq_getattr( )**. (As allowed by POSIX, this implementation of message queues makes use of a number of internal flags that are not public.)

Example 5-8 **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);
    }
```

5.14.3 Displaying Message Queue Attributes

The **mqPxShow( )** routine can be used to display information about POSIX message queues, as illustrated below.

```
-> mq_open ("mymq4", 0x4201, 0)
value = 8380448 = 0x7fe020
-> mqPxShow 0x7fe020
Message queue name                : mymq4
No. of messages in queue          : 0
Maximum no. of messages           : 16
Maximum message size              : 16
Flags                             : O_WRONLY  O_NONBLOCK   (0x4001      )
```

5.14.4 Communicating Through a Message Queue

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 (blocking) 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.

NOTE: In VxWorks, a POSIX message queue whose name does not start with a forward-slash (/) character is considered private to the process that has opened it and can not be accessed from another process. A message queue whose name starts with a forward-slash (/) character is a public object, and other processes can access it (as according to the POSIX standard). See *4.9 Public and Private Objects*, p.194.

Example 5-9 **POSIX Message Queues**

```
/*
 * In this example, the mqExInit() routine spawns two tasks that
 * communicate using the message queue.
 * To run this test case on the target shell:
 *
 * -> sp mqExInit
 */

/* 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 <taskLib.h>
#include <stdio.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>
#include <mqEx.h>

/* defines */

#define HI_PRIO 31
#define MSG_SIZE 16
#define MSG "greetings"

/****************************************************************************
*
* mqExInit - main for message queue send and receive test case
*
* This routine spawns to tasks to perform the message queue send and receive
* test case.
*
* RETURNS: OK, or ERROR
*
* ERRNOS: N/A
*/
```

```
int mqExInit (void)
    {
    /* create two tasks */

    if (taskSpawn ("tRcvTask", 151, 0, 4000, (FUNCPTR) receiveTask,
                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
    printf ("taskSpawn of tRcvTask failed\n");
    return (ERROR);
    }

    if (taskSpawn ("tSndTask", 152, 0, 4000, (FUNCPTR) sendTask,
                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0) == ERROR)
    {
    printf ("taskSpawn of tSendTask failed\n");
    return (ERROR);
    }
    return (OK);
    }


/****************************************************************************
*
* receiveTask - receive messages from the message queue
*
* This routine creates a message queue and calls mq_receive() to wait for
* a message arriving in the message queue.
*
* RETURNS: OK, or ERROR
*
* ERRNOS: N/A
*/

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 - send a message to a message queue
*
* This routine opens an already created message queue and
* calls mq_send() to send a message to the opened message queue.
*
* RETURNS: OK, or ERROR
*
* ERRNOS: N/A
*/
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");
    }
```

5.14.5 Notification of Message Arrival

A pthread (or task) can use the **mq_notify( )** routine to request notification of the arrival of a message at an empty queue. The pthread can thereby avoid blocking or polling to wait for a message.

Each queue can register only one pthread for notification at a time. Once a queue has a pthread to notify, no further attempts to register with **mq_notify( )** can succeed until the notification request is satisfied or cancelled.

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

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

The **mq_notify( )** mechanism does not send notification:

- When additional messages arrive at a message queue that is not empty. That is, notification is only sent when a message arrives at an empty message queue.
- If another pthread was blocked on the queue with **mq_receive( )**.
- After a response has been made to the call to **mq_notify( )**. That is, only one notification is sent per **mq_notify( )** call.

Example 5-10 **Message Queue Notification**

```
/*
 * In this example, a task uses mq_notify() to discover when a message
 * has arrived on a previously empty queue. To run this from the shell:
 *
 * -> ld < mq_notify_test.o
 * -> sp exMqNotify, "greetings"
 * -> mq_send
 *
 */

/* includes */

#include <vxWorks.h>
#include <signal.h>
#include <mqueue.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <string.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.
*
* RETURNS: 0 on success, or -1
*
* ERRNOS: N/A
*/

int exMqNotify
    (
    char * pMessage,         /* text for message to self */
    int loopCnt                 /* number of times to send a msg */
    )
    {
    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 */
    int cnt = 0;

    /* Minor sanity check; avoid exceeding msg buffer */

    if (MSG_SIZE <= strlen (pMessage))
    {
    printf ("exMqNotify: message too long\n");
    return (-1);
    }

    /*
     * Install signal handler for the notify signal and 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 = 0;
    attr.mq_maxmsg = 2;
    attr.mq_msgsize = MSG_SIZE;
    if ((exMqId = mq_open (QNAM, O_CREAT | O_RDWR | O_NONBLOCK, 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, pMessage, 1 + strlen (pMessage), 0) == -1)
    {
    printf ("mq_send failed\n");
    }

    /* 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.
*
* RETURNS: N/A
*
* ERRNOS: N/A
*/

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.
*
* RETURNS: N/A
*
* ERRNOS: N/A
*/

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: mqId (0x%x) received message: %s\n", exMqId, msg);
    }

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

5.15 POSIX Signals

VxWorks provides POSIX signal routines, as well as BSD-compatible routines and native VxWorks routines in the kernel. For information about these facilities, see *4.18 Signals*, p.226.

5.16 POSIX Memory Management

The VxWorks kernel provides POSIX memory management support for dynamic memory allocation with **calloc( )**, **malloc( )**, **realloc( )**, and **free( )**. For more information in this regard, see *6.7 Kernel Heap and Memory Partition Management*, p.329.

The kernel also provides the following POSIX memory locking routines: **mlock( )**, **munlock( )**, **mlockall( )**, and **munlockall( )**. However, memory mappings in VxWorks are always memory-resident. This ensures deterministic memory access for mapped files, but it also means that physical memory is continuously associated with mappings, until it is unmapped. Therefore, these POSIX memory locking routines do not do anything, and are provided simply for application portability.

For information about the POSIX memory management facilities that are available only in processes, see the *VxWorks Application Programmer's Guide: POSIX Facilities*.

6
Memory Management

Kernel Facilities

6.1 Introduction

VxWorks provides memory management facilities for all code that executes in the kernel, as well as memory management facilities for applications that execute as real-time processes. This chapter deals primarily with kernel-space memory management, although it also provides information about what memory maps look like for systems that include support for processes (and related facilities).

This chapter discusses the following topics:

- The VxWorks components required for different types of memory management support.
- The layout of memory for different configurations of VxWorks.
- Excluding memory from VxWorks use.
- Using run-time memory autosizing.
- The kernel heap and memory partition management facilities that are available in the kernel.
- Memory error detection facilities, including instrumentation provided by VxWorks components and the Wind River compiler.
- Virtual memory management, both automated and programmatic.
- Using the real-time process environment without an MMU.

For information about the memory management facilities available to process-based applications, see *VxWorks Application Programmer's Guide: Memory Management*.

For information about additional error detection facilities useful for debugging software faults, see *11. Error Detection and Reporting*.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

6.2 Configuring VxWorks With Memory Management Facilities

Information about configuring VxWorks with various memory management facilities is provided in the context of the discussions of those facilities. See:

- *6.4 Shell Commands*, p.327
- *6.5 System RAM Autosizing*, p.327
- *6.6 Reserved Memory*, p.328
- *6.7 Kernel Heap and Memory Partition Management*, p.329
- *6.8 Memory Error Detection*, p.331
- *6.9 Virtual Memory Management*, p.343
- *6.11 Processes Without MMU Support*, p.355

6.3 System Memory Maps

This section describes the VxWorks memory map as it appears with different configurations and run-time activity:

- A system without process support.
- A system with process support, but without processes running.
- A system with process support and two processes running, as well as a shared library and a shared data region.

In addition, it describes various memory views within a single system.

6.3.1 System Memory Map Without Process Support

In a VxWorks system RAM is delimited by:

- The **LOCAL_MEM_LOCAL_ADRS** BSP configuration parameter, which defines the start of the system RAM.
- The address returned by the routine **sysPhysMemTop( )**, which is at the top of system RAM. This address is either determined at run-time if RAM autosizing is enabled (see *6.5 System RAM Autosizing*, p.327). If autosizing is disabled, then **sysPhysMemTop( )** is calculated using the BSP configuration parameter **LOCAL_MEM_SIZE**; that is **sysPhysMemTop( )** returns **LOCAL_MEM_LOCAL_ADRS** + **LOCAL_MEM_SIZE**.

(**LOCAL_MEM_LOCAL_ADRS** and **LOCAL_MEM_SIZE** are configuration parameters of the **INCLUDE_MEMORY_CONFIG** component.)

System RAM must be contiguous. For systems without an MMU or with the MMU disabled, this means that the system RAM must be in contiguous physical memory. For systems with and MMU enabled, the system RAM must be mapped contiguously in virtual memory. In the latter case, the physical space may be non-contiguous for some architectures that do not require an identity mapped kernel. For the architecture specific requirements, see the *VxWorks Architecture Supplement*.

⚠ **CAUTION:** The SMP configuration of VxWorks does not support MMU-less configurations. For information about VxWorks SMP, see *15. VxWorks SMP*.

Within system RAM, the elements of a VxWorks system are arranged as follows:

- Below **RAM_LOW_ADRS** there is an architecture specific layout of memory blocks used for saving boot parameters, the system exception message area, the exception or interrupt vector table, and so on. For specific details, see the *VxWorks Architecture Supplement*.
- The kernel code (text, data, and bss) starting at address **RAM_LOW_ADRS**. ROM-resident images are an exception, for which the text segment is located outside of the system RAM (see *2.4.1 VxWorks Image Types*, p.15).
- The WDB target agent memory pool is located immediately above the kernel code, if WDB is configured into the system (see *12.6 WDB Target Agent*, p.626).
- The kernel heap follows the WDB memory pool.
- An optional area of persistent memory.

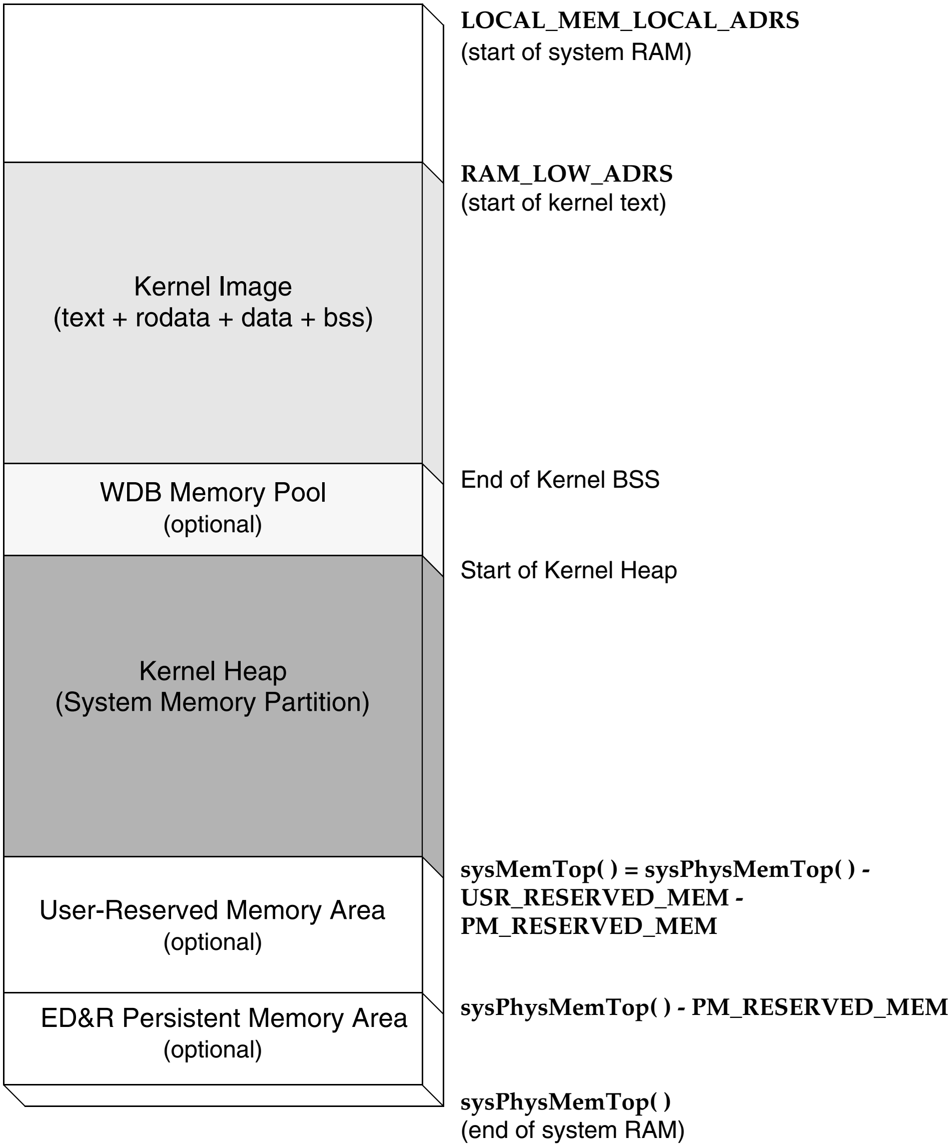
- An optional area of user-reserved memory may be located above the kernel heap.

Figure 6-1 illustrates a typical memory map for a system without process support. For a comparable illustration of a system with process support—which means it has unmapped memory available for processes that have not yet been created—see Figure 6-2.

Note that the memory pool for the WDB target agent is present only if WDB is configured into the kernel. Without WDB, the kernel heap starts right above the end of the kernel BSS ELF segment.

The routine **sysMemTop( )** returns the end of the kernel heap area. If both the user-reserved memory size (**USER_RESERVED_MEM**) and the persistent memory size (**PM_RESERVED_MEM**) are zero, then **sysMemTop( )** returns the same value than **sysPhysMemTop( )**, and the kernel heap extends to the end of the system RAM area. For more information about configuring user-reserved memory and persistent memory. See *6.6 Reserved Memory*, p.328 for more information. Also see *6.3.2 System Memory Map with Process Support*, p.321.

Figure 6-1 **Fig 5-1: Memory Map of a System Without Process Support**



6.3.2 System Memory Map with Process Support

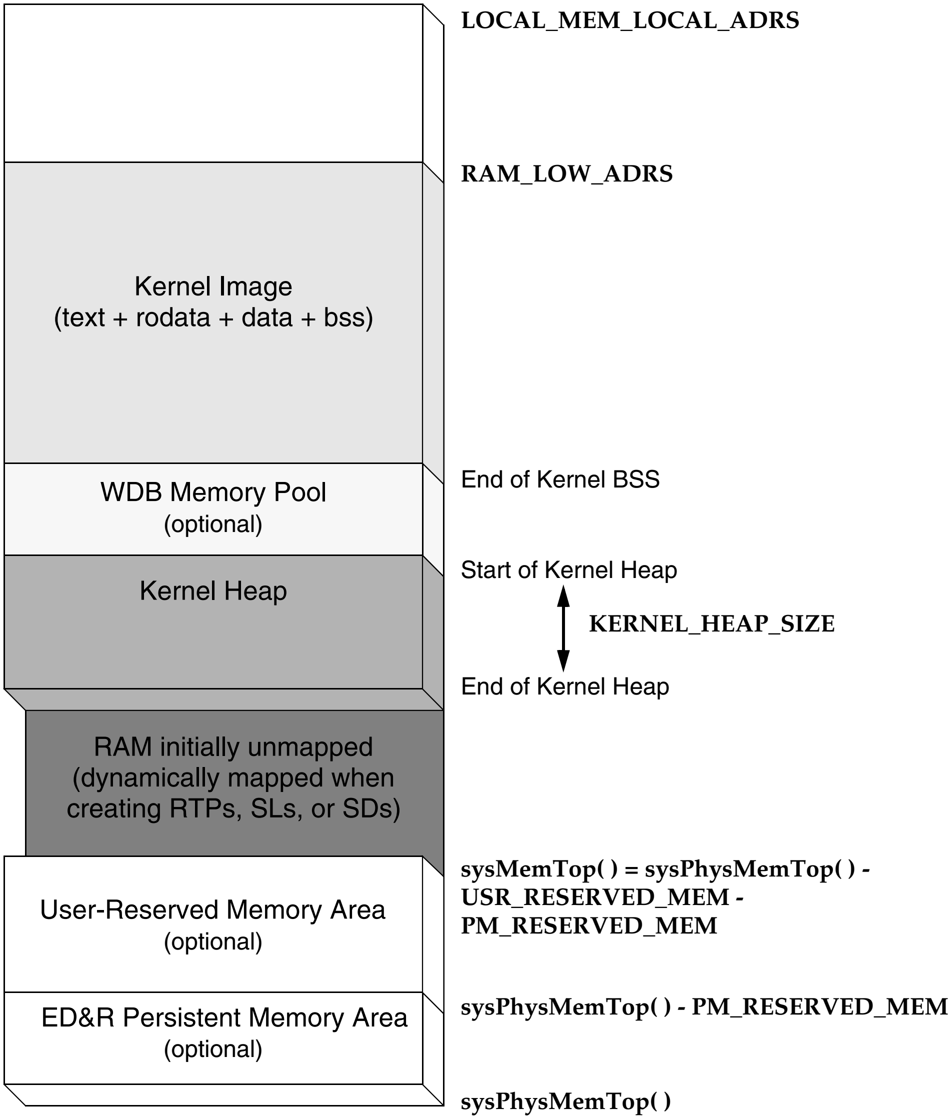
Kernel applications have access to the same memory management facilities as described in *6.3 System Memory Maps*, p.317, whether process support is included or not.

The only difference between the two configurations relates to the size of the kernel heap. Without process support, the kernel heap extends up to **sysMemTop( )**. With process support the kernel heap does not extend up to **sysMemTop( )**, but instead uses the **KERNEL_HEAP_SIZE** parameter (set in the **INCLUDE_RTP** component) as its size. This parameter is disregarded if process support is not included in VxWorks.

By default, **KERNEL_HEAP_SIZE** is set to two-thirds of the RAM located between **sysMemTop( )** and the end of the kernel code, or the end of the WDB memory pool when the WDB component is included into the system configuration.

Figure 6-2 illustrates this configuration. The RAM located between **sysMemTop( )** and the end of the kernel heap is left unmapped. RAM pages are allocated from that unmapped RAM area when process, shared library, or shared data region space must be mapped. For a comparable image of a system without process support, see Figure 6-1.

Figure 6-2 **Fig 5-2: Memory Map of a System With Process Support**



The default setting of **KERNEL_HEAP_SIZE** should be adjusted to meet the requirements of the system.

6.3.3 System Memory Map with Processes Running

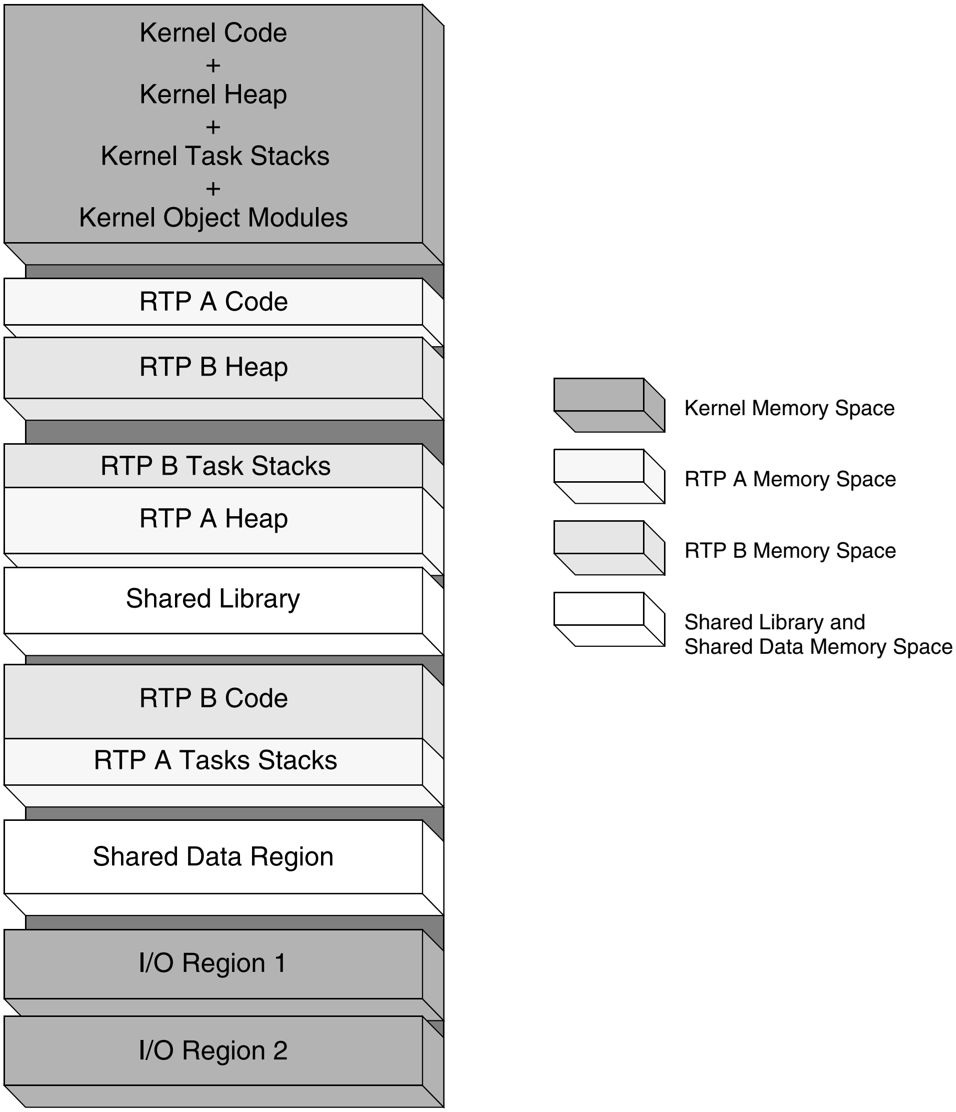
A VxWorks system configured for real-time processes may have one or more applications executing as processes at run-time. It may also have shared libraries and shared data regions instantiated. The kernel, each of the processes, shared libraries, and shared data regions occupy a discrete space in virtual memory.

Each VxWorks process has its own region of virtual memory; processes do not overlap in virtual memory. This flat virtual-memory map provides advantages in speed, in a programming model that accommodates systems with and without an MMU, and in debugging applications (see *6.11 Processes Without MMU Support*, p.355).

The virtual space assigned to a process is not necessarily composed of one large contiguous block of virtual memory. In some cases it will be composed of several smaller blocks of virtual space which are discontinuous from each other.

Figure 6-3 illustrates the memory map of a system with the kernel areas (RAM and I/O), two different processes (RTP A and RTP B), as well as one shared library, and one shared data region.

Figure 6-3 **Fig 5-3: Memory Map of a System With Two Processes**



Each process has its own virtual memory context, defined by its MMU translation table used to map virtual and physical memory, and other information about each page of memory. This memory context describes the virtual space that all of the tasks in a the process can access. In other words, it defines the *memory view* of a process.

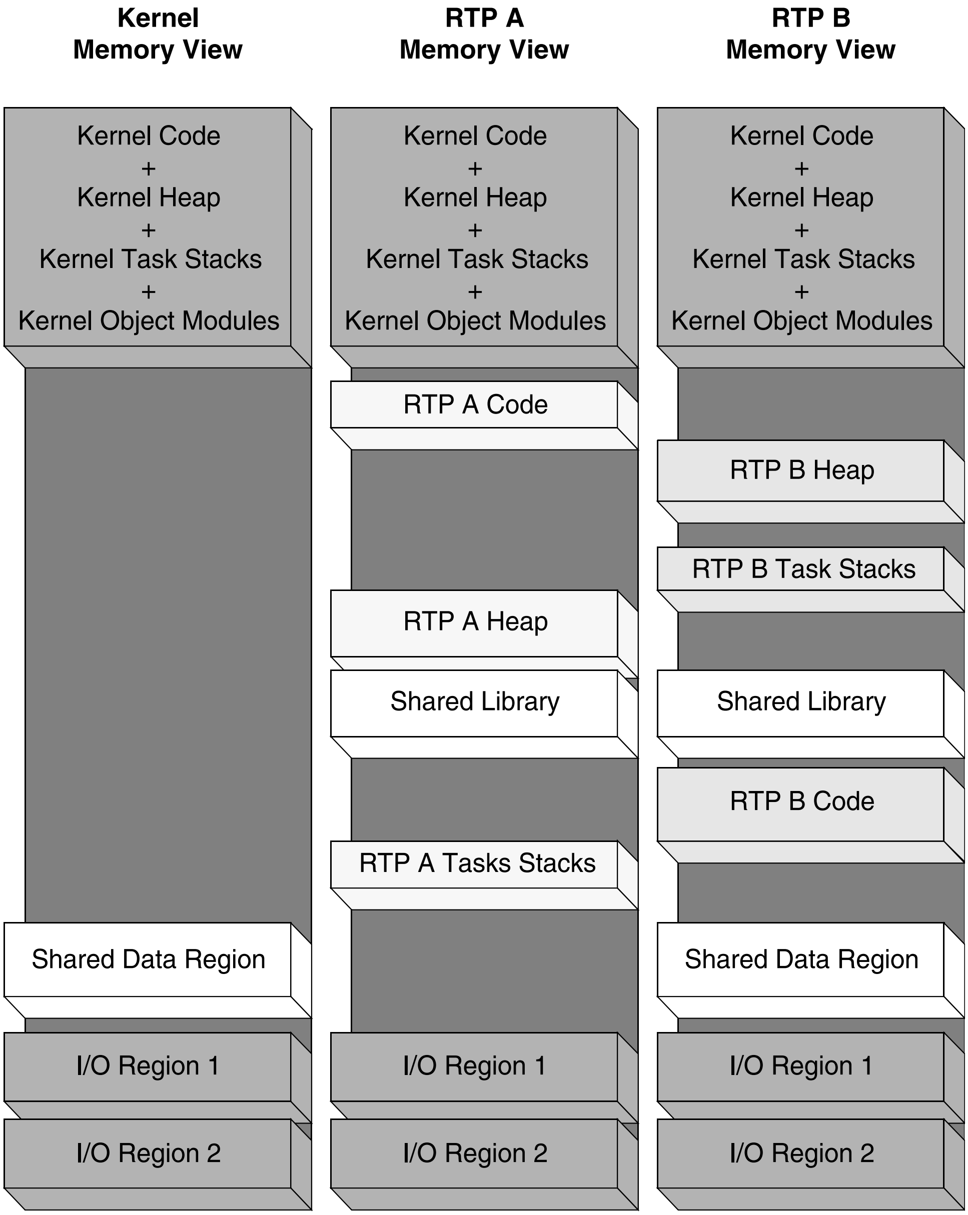
The kernel space is mapped with supervisor access privilege in the memory context of each process (but not with user mode privilege). Therefore tasks executing in a process can access kernel memory space only in system calls, during which the execution is switched to supervisor mode. (For information about system calls, see *VxWorks Application Programmer's Guide: Applications and Processes*.)

A shared library or shared data region is mapped into the virtual context of a process only when the process' application code opens or creates it, and it effectively disappears from the process' memory view when the application closes or deletes the shared library or shared data region.

Figure 6-4 illustrates the different memory views of a system with two processes (RTP A and RTP B), a shared library that both RTP A and RTP B opened, as well as a shared data region that both a kernel application and RTP B opened.

The first memory view corresponds to the memory space accessible by kernel tasks. The second and third memory views correspond to the memory space accessible by tasks executing in process A, respectively process B. Note that the grayed areas are only accessible during system calls.

Figure 6-4 **Fig 5-4: System Memory Views**



Note that on system without an MMU, or with the MMU disabled, there is only one memory view shared by the kernel and all process tasks. This memory view corresponds to Figure 6-3. Any task in the system, whether it is a kernel or a task executing in a process, has access to all the memory: kernel space, I/O regions, any processes memory, shared libraries, and shared data regions. In other words, such configurations do not provide any memory protection. For more information, see *6.11 Processes Without MMU Support*, p.355.

6.4 Shell Commands

The shell's **adrSpaceShow( )** show routine (for the C interpreter) or the **adrsp info** command (for the command interpreter) can be used to display an overview of the address space usage at time of the call. These are included in the kernel with the **INCLUDE_ADR_SPACE_SHOW** and **INCLUDE_ADR_SPACE_SHELL_CMD** components, respectively.

The **rtpMemShow( )** show routine or the **rtp meminfo** command can be used to display the private mappings of a process. These are included with the **INCLUDE_RTP_SHOW** and **INCLUDE_RTP_SHOW_SHELL_CMD** components, respectively.

The kernel mappings can be displayed with the **vmContextShow( )** show routine or the **vm context** command. These are included with the **INCLUDE_VM_SHOW** and **INCLUDE_VM_SHOW_SHELL_CMD** components, respectively.

6.5 System RAM Autosizing

When RAM autosizing is supported by the BSP, defining the configuration parameter **LOCAL_MEM_AUTOSIZE** will enable run time memory sizing. The default definition state for this parameter and the implementation itself is BSP-dependent. Check the BSP reference to see if this feature is supported or not.

When autosizing is supported by the BSP and **LOCAL_MEM_AUTOSIZE** is defined, the top of system RAM as reported by **sysPhysMemTop( )** is the value determined at run-time.

If the **LOCAL_MEM_AUTOSIZE** is not defined, the top of the system RAM as reported by **sysPhysMemTop( )** is the address calculated as:

(**LOCAL_MEM_LOCAL_ADRS** + **LOCAL_MEM_SIZE**)

If the BSP is unable to perform run time memory sizing then a compile time error should be generated, informing the user of the limitation.

LOCAL_MEM_AUTOSIZE, **LOCAL_MEM_LOCAL_ADRS** and **LOCAL_MEM_SIZE** are parameters of the **INCLUDE_MEM_CONFIG** component.

6.6 Reserved Memory

Two types of reserved memory can be configured in VxWorks system RAM: user-reserved memory and persistent memory. Reserved memory is not cleared by VxWorks at startup or during system operation. Boot loaders may or may not clear the area; see *Boot Loaders and Reserved Memory*, p.328.

User-reserved memory, configured with the BSP parameter **USER_RESERVED_MEM**, is part of the system RAM that can managed by kernel applications independently of the kernel heap.

Persistent memory, configured with the parameter **PM_RESERVED_MEM**, is the part of system RAM that is used by the error detection and reporting facilities (see *11. Error Detection and Reporting*).

For the layout of the user-reserved memory and the persistent memory, see figures Figure 6-1 and Figure 6-2.

Boot Loaders and Reserved Memory

Boot loaders may or may not clear reserved memory, depending on the configuration that was used to create them. If the boot loader is built with both **USER_RESERVED_MEM** and **PM_RESERVED_MEM** set to zero, the system RAM is cleared through the address calculated as:

(**LOCAL_MEM_LOCAL_ADRS** + **LOCAL_MEM_SIZE**)

To ensure that reserved memory is not cleared, the boot loader should be created with the **USER_RESERVED_MEM** and the **PM_RESERVED_MEM** parameter set to the desired sizes; that is, the same values that are used to build the downloaded VxWorks image.

For information about VxWorks boot loaders, see *3. Boot Loader*.

NOTE: If autosizing of system RAM is enabled, the top of the system RAM detected at run-time may be different from the address calculated as **LOCAL_MEM_LOCAL_ADRS** + **LOCAL_MEM_SIZE**, resulting in non-identical location of the memory range not being cleared by the boot loader. For more information about autosizing, see *6.5 System RAM Autosizing*, p.327.

6.7 Kernel Heap and Memory Partition Management

VxWorks provides facilities for heap access and memory partition management. The **memLib** and **memPartLib** libraries provide routines to access the kernel heap, including standard ANSI-compatible routines as well as routines to manipulate kernel memory partitions. The kernel heap is used by all code running in the kernel, including kernel libraries and components, kernel applications, and by processes when executing system calls.

Memory partitions consist of areas of memory that are used for dynamic memory allocations by applications and kernel components. Memory partitions may be used to reserve portions of memory for specific applications, or to isolate dynamic memory usage on an application basis.

The kernel heap is a specific memory partition, which is also referred to as the system memory partition.

6.7.1 Configuring the Kernel Heap and the Memory Partition Manager

There are two kernel components for configuring the kernel heap and the memory partition manager. The core functionality for both the kernel heap and memory partition is provided by the **INCLUDE_MEM_MGR_BASIC** component (see the VxWorks API reference for **memPartLib**). The **INCLUDE_MEM_MGR_FULL** component extends the functionality required for a full-featured heap and memory partition manager (see the VxWorks API reference for **memLib**).

The kernel heap is automatically created by the system when either one of these components are included in the VxWorks configuration. The size of the kernel heap is set as described in *6.3 System Memory Maps*, p.317; see Figure 6-1 and Figure 6-2.

Information about allocation statistics in the kernel heap and in kernel memory partitions can be obtained with the show routines provided with the **INCLUDE_MEM_SHOW** component. For more information, see the VxWorks API reference for **memShow**.

6.7.2 Basic Heap and Memory Partition Manager

The **memPartLib** library (**INCLUDE_MEM_MGR_BASIC**) provides the core facilities for memory partition support, including some of the standard ANSI-compatible routines such as **malloc( )**, and **free( )**. The core functionality of **memPartLib** provides for the following API:

- Creation and deletion of memory partitions with **memPartCreate( )** and **memPartDelete( )**.
- Addition of memory to a specific memory partition with **memPartAddToPool( )**, or to the heap with **memAddToPool( )**.
- Allocation and freeing of memory blocks from a specific memory partitions with **memPartAlloc( )**, **memPartAlignedAlloc( )**, and **memPartFree( )**; and from the heap with **malloc( )** and **free( )**.

6.7.3 Full Heap and Memory Partition Manager

The **memLib** library (provided by the **INCLUDE_MEM_MGR_FULL** component) adds a few more routines to provide a full-featured memory partition and heap allocator. The features provided in this library are:

- Allocation of memory aligned to a specific boundary with **memalign( )**, and alignment to a page with **valloc( )**.
- Reallocation of blocks of memory in a specific partition with **memPartRealloc( )**, or in the heap with **realloc( )**.
- The ANSI-compatible routines **calloc( )**, and **cfree( )**.
- Obtaining memory partition statistics with routines **memPartInfoGet( )** and **memPartFindMax( )**, or in the heap with **memFindMax( )** and **memInfoGet( )**.
- Built-in error checking. This feature is controlled with the heap and partition options. Two types of errors can be enabled. The first type, block error, is detected during block validation in **free( )**, **realloc( )**, **memPartFree( )** and **memPartRealloc( )**. The second type, allocation error, is detected by any of the

allocation and re-allocation routines. There are options to enable logging an error message and/or to suspend the task hitting the error. Setting and getting error handling options of a specific memory partition can be done with **memPartOptionsSet( )** and **memPartOptionsGet( )**. The debugging options for the heap are controlled with **memOptionsSet( )** and **memOptionGet( )**. Additional heap and memory partition error detection is provided with heap and partition memory instrumentation (see *6.8.1 Heap and Partition Memory Instrumentation*, p.331).

For more information, see the VxWorks API references for **memPartLib** and **memLib**.

6.8 Memory Error Detection

Support for memory error detection is provided by two optional instrumentation libraries. The **memEdrLib** library performs error checks of operations in the kernel heap and memory partitions in the kernel. The Run-Time Error Checking (RTEC) feature of the Wind River Compiler can be used to check for additional errors, such as buffer overruns and underruns, static and automatic variable reference checks.

Errors detected by these facilities are reported by the error detection and reporting facility, which must, therefore be included in the VxWorks kernel configuration. See *11. Error Detection and Reporting*.

6.8.1 Heap and Partition Memory Instrumentation

To supplement the error detection features built into **memLib** and **memPartLib** (such as valid block checking), components can be added to VxWorks to perform automatic, programmatic, and interactive error checks on **memLib** and **memPartLib** operations.

The components help detect common programming errors such as double-freeing an allocated block, freeing or reallocating an invalid pointer, memory leaks. In addition, with compiler-assisted code instrumentation, they help detect bounds-check violations, buffer over-runs and under-runs, pointer references to free memory blocks, pointer references to automatic variables outside the scope of the variable, and so on. Note that compiler-assisted instrumentation must be used in order to track buffer underruns and overruns. For information about compiler instrumentation, see *6.8.2 Compiler Instrumentation*, p.338.

Errors detected by the automatic checks are logged by the error detection and reporting facility.

Configuring VxWorks with Memory Partition and Heap Instrumentation

To enable the basic level of memory partition and heap instrumentation, the following components must be included into the kernel configuration:

- **INCLUDE_MEM_EDR**, includes the basic memory partition debugging functionality and instrumentation code.
- **INCLUDE_EDR_ERRLOG**, **INCLUDE_EDR_POLICIES** and **INCLUDE_EDR_SHOW** for error detection, reporting, and persistent memory. For more information see *11. Error Detection and Reporting*.

The following component may also be included:

- **INCLUDE_MEM_EDR_SHOW**, for enabling the show routines.

In addition, the following parameters of the **INCLUDE_MEM_EDR** component can be modified:

MEDR_EXTENDED_ENABLE
Set to **TRUE** to enable logging trace information for each allocated block, but at the cost of increased memory used to store entries in the allocation database. The default setting is **FALSE**.

MEDR_FILL_FREE_ENABLE
Set to **TRUE** to enable pattern-filling queued free blocks. This aids detecting writes into freed buffers. The default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN
Maximum length of the free queue. When a memory block is freed, instead of immediately returning it to the partition's memory pool, it is kept in a queue. This is useful for detecting references to a memory block after it has been freed. When the queue reaches the maximum length allowed, the blocks are returned to the respective memory pool in a FIFO order. Queuing is disabled when this parameter is 0. Default setting for this parameter is 64.

MEDR_BLOCK_GUARD_ENABLE
Enable guard signatures in the front and the end of each allocated block. Enabling this feature aids in detecting buffer overruns, underruns, and some heap memory corruption. The default setting is **FALSE**.

MEDR_POOL_SIZE

Set the size of the memory pool used to maintain the memory block database. Default setting in the kernel is 1MB. The database uses 32 bytes per memory block without extended information enabled, and 64 bytes per block with extended information enabled (call stack trace). This pool is allocated from the kernel heap.

Error Types

During execution, errors are automatically logged when the allocation, free, and re-allocation functions are called. The following error types are automatically identified and logged:

- Allocation returns block address within an already allocated block from the same partition. This would indicate corruption in the partition data structures.
- Allocation returns block address that is in the task's stack space. This would indicate corruption in the partition data structures.
- Allocation returns block address that is in the kernel's static data section. This would indicate corruption in the partition data structures.
- Freeing a pointer that is in the task’s stack space.
- Freeing memory that was already freed and is still in the free queue.
- Freeing memory that is in the kernel’s static data section.
- Freeing memory in a different partition than the one in which it was allocated.
- Freeing a partial memory block.
- Freeing a memory block with the guard zone corrupted, when the **MEDR_BLOCK_GUARD_ENABLE** environment variable is **TRUE**.
- Pattern in a memory block which is in the free queue has been corrupted, when the **MEDR_FILL_FREE_ENABLE** environment variable is **TRUE**.

Shell Commands

The show routines and commands described in Table 6-1 are available for use with the shell’s C and command interpreters to display information.

Table 6-1 **Shell Commands**

C Interpreter	Command Interpreter	Description
edrShow( )	**edr show**	Displays error records.
memEdrPartShow( )	**mem part list**	Displays a summary of the instrumentation information for memory partitions in the kernel.
memEdrBlockShow( )	**mem block list**	Displays information about allocated blocks. Blocks can be selected using a combination of various querying criteria: partition ID, block address, allocating task ID, block type.
memEdrFreeQueueFlush( )	**mem queue flush**	Flushes the free queue. When this routine is called, freeing of all blocks in the free queue is finalized so that all corresponding memory blocks are returned the free pool of the respective partition.
memEdrBlockMark( )	**mem block mark** and **mem block unmark**	Marks or unmarks selected blocks allocated at the time of the call. The selection criteria may include partition ID and/or allocating task ID. This routine can be used to monitor memory leaks by displaying information of unmarked blocks with **memBlockShow( )** or **mem block list**.

Code Example

The following kernel application code is used to demonstrate various errors detected with the heap and partition memory instrumentation (line numbers are included for reference purposes). Its use is illustrated in *Shell Session Example*, p.335.

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

void heapErrors (void)
    {
    char * pChar;

    pChar = malloc (24);
    free (pChar + 2);        /* free partial block */
    free (pChar);

    free (pChar);            /* double-free block */
    pChar = malloc (32);     /* leaked memory */
    }
```

Shell Session Example

The following shell session is executed with the C interpreter. The sample code listed above is compiled and linked in the VxWorks kernel (see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64). The kernel must include the **INCLUDE_MEM_EDR** and **INCLUDE_MEM_EDR_SHOW** components. In order to enable saving call stack information, the parameter **MEDR_EXTENDED_ENABLE** is set **TRUE**. Also, the kernel should be configured with the error detection and reporting facility, including the show routines, as described in *11.2 Configuring Error Detection and Reporting Facilities*, p.564.

First mark all allocated blocks:

```
-> memEdrBlockMark
    value = 6390 = 0x18f6
```

Next, clear the error log. This step is optional, and is done only to start with a clean log:

```
-> edrClear
    value = 0 = 0x0
```

The kernel application may be started in a new task spawned with the **sp( )** utility, as follows:

```
-> taskId = sp (heapErrors)
    New symbol "taskId" added to kernel symbol table.
    Task spawned: id = 0x246d010, name = t1
    taskId = 0x2469ed0: value = 38195216 = 0x246d010
```

At this point the application finished execution. The following command lists the memory blocks allocated, but not freed by the application task. Note that the listing shows the call stack at the time of the allocation:

```
-> memEdrBlockShow 0, 0, taskId, 5, 1

      Addr    Type    Size    Part ID  Task ID  Task Name       Trace
    -------- ------ -------- -------- -------- ------------ ------------
     246d7a0  alloc       32   269888  246d010          -t1 heapErrors()
                                                            memPartAlloc()
                                                            0x001bdc88()
```

Errors detected while executing the application are logged in persistent memory region.

Display the log using **edrShow( )**. The first error corresponds to line 9 in the test code; the second error corresponds to line 12.

```
-> edrShow
    ERROR LOG
    =========
    Log Size:         524288 bytes (128 pages)
    Record Size:      4096 bytes
    Max Records:      123
    CPU Type:         0x5a
    Errors Missed:    0 (old) + 0 (recent)
    Error count:      2
    Boot count:       20
    Generation count: 94

    ==[1/2]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          20
    OS Version:          6.0.0
    Time:                THU JAN 01 00:00:31 1970 (ticks = 1880)
    Task:                "t1" (0x0246d010)

    freeing part of allocated memory block
        PARTITION: 0x269888
        PTR=0x246bea2
        BLOCK: allocated at 0x0246bea0, 24 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : heapErrors ()
    0x00111364 heapErrors   +0x24 : free ()
    0x001c26f8 memPartFree  +0xa4 : 0x001bdbb4 ()
    0x001bdc6c memEdrItemGet+0x588: 0x001bd71c ()

    ==[2/2]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          20
    OS Version:          6.0.0
    Time:                THU JAN 01 00:00:31 1970 (ticks = 1880)
    Task:                "t1" (0x0246d010)

    freeing memory in free list
        PARTITION: 0x269888
        PTR=0x246bea0
        BLOCK: free block at 0x0246bea0, 24 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : heapErrors ()
    0x00111374 heapErrors   +0x34 : free ()
    0x001c26f8 memPartFree  +0xa4 : 0x001bdbb4 ()
    0x001bdc6c memEdrItemGet+0x588: 0x001bd71c ()
    value = 0 = 0x0
```

6.8.2 Compiler Instrumentation

Additional errors are detected if the application is compiled using the Run-Time Error Checking (RTEC) feature of the Wind River Compiler (Diab). The following flag should be used:

-Xrtc=*option*

NOTE: This feature is not available with the GNU compiler.

Code compiled with the **-Xrtc** flag is instrumented for run-time checks such as pointer reference check and pointer arithmetic validation, standard library parameter validation, and so on. These instrumentations are supported through the memory partition run-time error detection library. Table 6-2 lists the **-Xrtc** options that are supported.

Note that using the **-Xrtc** flag without specifying any options is the same as using them all.

Table 6-2 **-Xrtc Options**

Option	Description
0x01	register and check static (global) variables
0x02	register and check automatic variables
0x08	pointer reference checks
0x10	pointer arithmetic checks
0x20	pointer increment/decrement checks
0x40	standard function checks; for example **memset( )** and **bcopy( )**
0x80	report source code filename and line number in error logs

The errors and warnings detected by the RTEC compile-in instrumentation are logged by the error detection and reporting facility (see *11. Error Detection and Reporting*). The following error types are identified:

- Bounds-check violation for allocated memory blocks.
- Bounds-check violation of static (global) variables.
- Bounds-check violation of automatic variables.
- Reference to a block in the free queue.
- Reference to the free part of the task's stack.
- De-referencing a NULL pointer.

Configuring VxWorks for RTEC Support

Support for this feature in the kernel is enabled by adding the **INCLUDE_MEM_EDR_RTC** component, as well as the components described in section *Configuring VxWorks with Memory Partition and Heap Instrumentation*, p.332.

Shell Commands

The compiler provided instrumentation automatically logs errors detected in applications using the error detection and reporting facility. For a list of the shell commands available for error logs see *11.4 Displaying and Clearing Error Records*, p.568.

Code Example

Application code built with the RTEC instrumentation has compiler-generated constructors. To ensure that the constructors are called when a module is dynamically downloaded, the module must be processed similarly to a C++ application. For example, the following make rule can be used:

```
TGT_DIR=$(WIND_BASE)/target

%.out : %.c
    @ $(RM) $@
    $(CC) $(CFLAGS) -Xrtc=0xfb $(OPTION_OBJECT_ONLY) $<
    @ $(RM) ctdt_$(BUILD_EXT).c
    $(NM) $(basename $@).o | $(MUNCH) > ctdt_$(BUILD_EXT).c
    $(MAKE) CC_COMPILER=$(OPTION_DOLLAR_SYMBOLS) ctdt_$(BUILD_EXT).o
    $(LD_PARTIAL) $(LD_PARTIAL_LAST_FLAGS) $(OPTION_OBJECT_NAME)$@ $(basename
$@).o ctdt_$(BUILD_EXT).o

include $(TGT_DIR)/h/make/rules.library
```

The the following application code generates various errors that can be recorded and displayed (line numbers are included for reference purposes). Its use is illustrated in *Shell Session Example*, p.340.

```
#include <vxWorks.h>
#include <stdlib.h>
#include <string.h>

void refErrors ()
    {
    char name[] = "very_long_name";
    char * pChar;
    int state[] = { 0, 1, 2, 3 };
    int ix = 0;

    pChar = malloc (13);
    memcpy (pChar, name, strlen (name)); /* bounds check violation */

    /* of allocated block */

    for (ix = 0; ix < 4; ix++)
        state[ix] = state [ix + 1];      /* bounds check violation */

    free (pChar);

    memcpy (pChar, "another_name", 12);  /* reference a free block */
    }
```

Shell Session Example

The following shell session log is executed with the C interpreter. The sample code listed above is compiled and linked in the VxWorks kernel (see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64). The kernel must include the **INCLUDE_MEM_EDR** and **INCLUDE_MEM_EDR_RTC** components. Also, the kernel should be configured with the error detection and reporting facility, including the show routines, as described in *11.2 Configuring Error Detection and Reporting Facilities*, p.564.

First, clear the error log to start with a clean log:

```
-> edrClear
    value = 0 = 0x0
```

Start the kernel application in a new task spawned with the **sp( )** utility:

```
-> sp refErrors
    Task spawned: id = 0x246d7d0, name = t2
    value = 38197200 = 0x246d7d0
```

At this point the application finished execution. Errors detected while executing the application are logged in the persistent memory region. Display the log using

edrShow( ). In the example below, the log display is interspersed with description of the errors.

```
-> edrShow
    ERROR LOG
    =========
    Log Size:          524288 bytes (128 pages)
    Record Size:       4096 bytes
    Max Records:       123
    CPU Type:          0x5a
    Errors Missed:     0 (old) + 0 (recent)
    Error count:       3
    Boot count:        21
    Generation count: 97
```

The first error corresponds to line 13 in the test code. A string of length 14 is copied into a allocated buffer of size 13:

```
==[1/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51738)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:13

    memory block bounds-check violation
        PTR=0x246be60  OFFSET=0  SIZE=14
        BLOCK: allocated at 0x0246be60, 13 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x00111470 refErrors    +0xe4 : __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

The second error refers to line 18: the local **state** array is referenced with index 4. Since the array has only four elements, the range of valid indexes is 0 to 3:

```
==[2/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51738)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:18

    memory block bounds-check violation
        PTR=0x278ba94  OFFSET=16  SIZE=4
        BLOCK: automatic at 0x0278ba94, 16 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x001114a0 refErrors    +0x114: __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

The last error is caused by the code on line 22. A memory block that has been freed is being modified:

```
==[3/3]==============================================================
    Severity/Facility:   NON-FATAL/KERNEL
    Boot Cycle:          21
    OS Version:          6.0.0
    Time:                THU JAN 01 00:14:22 1970 (ticks = 51739)
    Task:                "t2" (0x0246d7d0)
    Injection Point:     refErr.c:22

    pointer to free memory block
        PTR=0x246be60  OFFSET=0  SIZE=12
        BLOCK: free block at 0x0246be60, 13 bytes

    <<<<<Traceback>>>>>

    0x0011d240 vxTaskEntry  +0x54 : 0x00111390 ()
    0x00111518 refErrors    +0x18c: __rtc_chk_at ()
    0x001bd02c memEdrErrorLog+0x13c: _sigCtxSave ()
```

6.9 Virtual Memory Management

VxWorks can be configured with an architecture-independent interface to the CPU's memory management unit (MMU) to provide virtual memory support. This support includes the following features:

- Setting up the kernel memory context at boot time.
- Mapping pages in virtual space to physical memory.
- Setting caching attributes on a per-page basis.
- Setting protection attributes on a per-page basis.
- Setting a page mapping as valid or invalid.
- Locking and unlocking TLB entries for pages of memory.
- Enabling page optimization.

The programmable elements of virtual memory (VM) support are provided by the **vmBaseLib** library.

NOTE: There are differences in the **vmBaseLib** library provided for the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks, and special guidelines for its use in optimizing SMP applications. For more information about **vmBaseLib** and SMP, see *vmBaseLib Restrictions*, p.709 and *Using vmBaseLib*, p.701. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

When process (RTP) support is included in VxWorks with the **INCLUDE_RTP** component, the virtual memory facilities also provide system support for managing multiple virtual memory contexts, such as creation and deletion of process memory context.

For information about additional MMU-based memory protection features beyond basic virtual memory support, see *6.10 Additional Memory Protection Features*, p.353.

Also note that errors (exceptions) generated with the use of virtual memory features can be detected and managed with additional VxWorks facilities. See *11. Error Detection and Reporting* for more information.

6.9.1 Configuring Virtual Memory Management

The components listed in Table 6-3 provide basic virtual memory management, as well as show routines for use from the shell.

Table 6-3 **MMU Components**

Constant	Description
INCLUDE_MMU_GLOBAL_MAP	Initialize the kernel's global MMU mappings according to the BSP's **sysPhysMemDesc[ ]** table. See *Configuring the Kernel Virtual Memory Context*, p.344.
INCLUDE_MMU_BASIC	Include the **vmBaseLib** API, which is used for programmatic management of virtual memory (see *6.9.2 Managing Virtual Memory Programmatically*, p.346).
INCLUDE_LOCK_TEXT_SECTION	Kernel text TLB locking optimization.
INCLUDE_PAGE_SIZE_OPTIMIZATION	Page size optimization for the kernel.
INCLUDE_VM_SHOW	Virtual memory show routines for the shell C interpreter.
INCLUDE_VM_SHOW_SHELL_CMD	Virtual memory show commands for the shell command interpreter.

For information about related components see *6.10 Additional Memory Protection Features*, p.353.

Configuring the Kernel Virtual Memory Context

The kernel virtual memory context is created automatically at boot time based on configuration data provided by the BSP. The primary data is in the **sysPhysMemDesc[ ]** table, which is usually defined in the BSP's **sysLib.c** file. The table defines the initial kernel mappings and initial attributes. The entries in this table are of **PHYS_MEM_DESC** structure type, which is defined in **vmLib.h**.

There is usually no need to change the default **sysPhysMemDesc[ ]** configuration. However, modification may be required or advisable, for example, when:

- New driver support or new devices (for example, flash memory) are added to the system.
- The protection or cache attributes of certain entries must be changed. For example, entries for flash memory can be read-only if the content of the flash is never written from VxWorks. However, if a flash driver such as TrueFFS is used, the protection attribute has to be set to writable.
- There are unused entries in the table. In general, it is best to keep only those entries that actually describe the system, as each entry may require additional system RAM for page tables (depending on size of the entry, its location relative to other entries, and architecture-specific MMU parameters). The larger the memory blocks mapped, the more memory is used for page tables.

The **sysPhysMemDesc[ ]** table can be modified at run-time. This is useful, for example, with PCI drivers that can be auto-configured, which means that memory requirements are detected at run-time. In this case the size and address fields can be updated programmatically for the corresponding **sysPhysMemDesc[ ]** entries. It is important to make such updates before the VM subsystem is initialized by **usrMmuInit( )**, for example during execution of **sysHwInit( )**.

⚠ **CAUTION:** 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 the MMU page size. Specifying elements of **sysPhysMemDesc[ ]** that are not page-aligned causes the target to reboot during initialization. See the *VxWorks Architecture Supplement* to determine what page size is supported for the architecture in question.

Configuration Example

This example is based on multiple CPUs using the shared-memory network. A separate memory board is used for the shared-memory pool. Because this memory is not mapped by default, it must be added to **sysPhysMemDesc[ ]** for all the boards on the network. The memory starts at 0x4000000 and must be made non-cacheable, as shown in the following code fragment:

```
/* shared memory */
{
(VIRT_ADDR)  0x4000000,          /* virtual address */
(PHYS_ADDR)  0x4000000,          /* physical address */
0x20000,                         /* length */
/* initial state mask */
MMU_ATTR_VALID_MSK | MMU_ATTR_PROT_MSK | MMU_ATTR_CACHE_MSK,
/* initial state */
MMU_ATTR_VALID | MMU_ATTR_PROT_SUP_READ | MMU_ATTR_PROT_SUP_WRITE |
MMU_ATTR_CACHE_OFF
}
```

For some architectures, the system RAM (the memory used for the VxWorks kernel image, kernel heap, and so on) must be identity mapped. This means that for the corresponding entry in the **sysPhysMemDesc[ ]** table, the virtual address must be the same as the physical address. For more information see *6.3 System Memory Maps*, p.317 and the *VxWorks Architecture Supplement*.

6.9.2 Managing Virtual Memory Programmatically

This section describes the facilities provided for manipulating the MMU programmatically using low-level routines in **vmBaseLib**. You can make portions of memory non-cacheable, write-protect portions of memory, invalidate pages, lock TLB entries, or optimize the size of memory pages.

For more information about the virtual memory routines, see the VxWorks API reference for **vmBaseLib**.

NOTE: There are differences in the **vmBaseLib** library provided for the symmetric multiprocessor (SMP) and uniprocessor (UP) configurations of VxWorks, and special guidelines for its use in optimizing SMP applications. For more information about **vmBaseLib** and SMP, see *vmBaseLib Restrictions*, p.709 and *Using vmBaseLib*, p.701. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

Modifying Page States

Each virtual memory page (typically 4 KB) has a state associated with it. A page can be valid/invalid, readable, writable, executable, or cacheable/non-cacheable.

The state of a page can be changed with the **vmStateSet( )** routine. See Table 6-4 and Table 6-5 for lists of the page state constants and page state masks that can be used with **vmStateSet( )**. A page state mask must be used to describe which flags are being changed. A logical OR operator can be used with states and masks to define both mapping protection and cache attributes.

Table 6-4 **Page State Constants**

Constant	Description
MMU_ATTR_VALID	Valid translation
MMU_ATTR_VALID_NOT	Invalid translation
MMU_ATTR_PRO_SUP_READ	Readable memory in kernel mode
MMU_ATTR_PRO_SUP_WRITE	Writable memory in kernel mode
MMU_ATTR_PRO_SUP_EXE	Executable memory in kernel mode
MMU_ATTR_PRO_USR_READ	Readable memory in user mode
MMU_ATTR_PRO_USR_WRITE	Writable memory in user mode
MMU_ATTR_PRO_USR_EXE	Executable memory in user mode
MMU_ATTR_CACHE_OFF	Non-cacheable memory
MMU_ATTR_CACHE_COPYBACK	Cacheable memory, copyback mode
MMU_ATTR_CACHE_WRITETHRU	Cacheable memory, writethrough mode
MMU_ATTR_CACHE_DEFAULT	Default cache mode (equal to either **COPYBACK**, **WRITETHRU**, or **CACHE_OFF**, depending on the setting of **USER_D_CACHE_MODE**)
MMU_ATTR_CACHE_COHERENCY	Memory coherency is enforced (not supported on all architectures; for more information, see the *VxWorks Architecture Supplement*)
MMU_ATTR_CACHE_GUARDED	Prevent out-of-order load operations, and pre-fetches (not supported on all architectures; for more information, see the *VxWorks Architecture Supplement*)
MMU_ATTR_NO_BLOCK	Page attributes can be changed from ISR.

Table 6-4 **Page State Constants** (cont'd)

Constant	Description
MMU_ATTR_SPL_0 ... **MMU_ATTR_SPL_7**	Optional Architecture Specific States (only used by some architectures; for more information, see the *VxWorks Architecture Supplement*)

Table 6-5 **Page State Masks**

Constant	Description
MMU_ATTR_VALID_MSK	Modify valid flag
MMU_ATTR_PROT_MSK	Modify protection flags
MMU_ATTR_CACHE_MSK	Modify cache flags
MMU_ATTR_SPL_MSK	Modify architecture specific flags

Not all combinations of protection settings are supported by all CPUs. For example, many processor types do not provide setting for execute or non-execute settings. On such processors, readable also means executable.

For information about architecture-specific page states and their combination, see the *VxWorks Architecture Supplement*.

Making Memory Non-Writable

Sections of memory can be write-protected using **vmStateSet( )** to prevent inadvertent access. This can be used, for example, 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 **MMU_ATTR_PROT_SUP_READ**.

Nonwritable Memory Example

In this code example, a task calls **dataModify( )** to modify the data structure pointed to by **pData**. This routine makes the memory writable, modifies the data, and sets the memory back to nonwritable. If a task subsequently tries to modify the data without using **dataModify( )**, a data access exception 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 <string.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 ((char *) pData, pageSize);
    if (vmStateSet (NULL, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ) == 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, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ | MMU_ATTR_PROT_SUP_WRITE) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    /* update data*/
    bcopy ((char *) pNewData, (char *) pData, sizeof(MY_DATA));
    /* make memory not writable */
    if (vmStateSet (NULL, (VIRT_ADDR) pData, pageSize, MMU_ATTR_PROT_MSK,
            MMU_ATTR_PROT_SUP_READ) == ERROR)
                {
                semGive (dataSemId);
                return (ERROR);
                }
    semGive (dataSemId);
    return (OK);
    }
```

Invalidating Memory Pages

To invalidate memory on a page basis, use **vmStateSet( )** as follows:

```
vmStateSet (NULL, address, len, MMU_ATTR_VALID_MSK, MMU_ATTR_VALID_NOT);
```

Any access to a mapping made invalid generates an exception whether it is a read or a write access.

To re-validate the page, use **vmStateSet( )** as follows:

```
vmStateSet (NULL, address, len, MMU_ATTR_VALID_MSK, MMU_ATTR_VALID);
```

Locking TLB Entries

For some processors it is possible to force individual entries in the Translation Look-aside Buffer (TLB) to remain permanently in the TLB. When the architecture-specific MMU library supports this feature, the **vmPageLock( )** routine can be used to lock page entries, and **vmPageUnlock( )** to unlock page entries.

The **INCLUDE_LOCK_TEXT_SECTION** component provides facilities for TLB locking. When this component is included in VxWorks, the kernel image text section is automatically locked at system startup.

This feature can be used for performance optimizations in a manner similar to cache locking. When often-used page entries are locked in the TLB, the number of TLB misses can be reduced. Note that the number of TLB entries are generally limited on all processors types, so locking too many entries can result in contention for the remaining entries that are used dynamically.

For more information, see the *VxWorks Architecture Supplement*.

Page Size Optimization

For some processors it is possible to enable larger page sizes than the default (defined by **VM_PAGE_SIZE**) for large, contiguous memory blocks that have homogeneous memory attributes. There are several advantages to using such optimization, including:

- Reducing the number of page table entries (PTE) needed to map memory, resulting in less memory used.
- More efficient TLB entry usage, resulting in fewer TLB misses, therefore potentially better performance.

Optimization of the entire kernel memory space (including I/O blocks) at startup can be accomplished by configuring VxWorks with the **INCLUDE_PAGE_SIZE_OPTIMIZATION** component.

Page size optimization for specific blocks of memory can be accomplished at run-time with the **vmPageOptimize( )** routine.

De-optimization is performed automatically when necessary. For example, if part of a memory block that has been optimized is set with different attributes, the large page is automatically broken up into multiple smaller pages and the new attribute is set to the requested pages only.

Setting Page States in ISRs

For many types of processors, **vmStateSet( )** is a non-blocking routine, and can therefore be called safely from ISRs. However, it may block in some cases, such as on processors that support page size optimization (see *Page Size Optimization*, p.351).

To make sure that **vmStateSet( )** can be called safely from an ISR for specific pages, the page must first have the **MMU_ATTR_NO_BLOCK** attribute set. The following code example shows how this can be done:

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

 #define DATA_SIZE   0x10000

 char * pData;

 void someInitFunction ()
     {
     /* allocate buffer */

     pData = (char *) valloc (DATA_SIZE);

     /* set no-block attribute for the buffer */

     vmStateSet (NULL, (VIRT_ADDR) pData, DATA_SIZE,
                 MMU_ATTR_SPL_MSK, MMU_ATTR_NO_BLOCK);
     }

 void someISR ()
     {
     ...
     /* now it's safe to set any attribute for the buffer in an ISR */

     vmStateSet (NULL, (VIRT_ADDR) pData, DATA_SIZE,
                 MMU_ATTR_PROT_MSK, MMU_ATTR_SUP_RWX);
     ...
     }
```

6.9.3 Troubleshooting

The show routines and commands described in Table 6-6 are available to assist with trouble-shooting virtual memory problems.

Table 6-6 **Virtual Memory Shell Commands**

C Interpreter	Command Interpreter	Description
vmContextShow( )	**vm context**	Lists information about the entire process context, including private mappings and kernel mappings (for supervisor access), as well as any shared data contexts attached to the process.
rtpMemShow( )	**rtp meminfo**	Lists only the process' private mappings.

These routines and commands are provided by the **INCLUDE_VM_SHOW**, **INCLUDE_VM_SHOW_SHELL_CMD**, **INCLUDE_RTP_SHOW**, and **INCLUDE_RTP_SHOW_SHELL_CMD** components.

For more details and usage example of the show routines see the VxWorks shell references.

6.10 Additional Memory Protection Features

VxWorks provides MMU-based features that supplement basic virtual memory support to provide a more reliable run-time environment. These additional memory-protection features are:

- task stack overrun and underrun detection
- interrupt stack overrun and underrun detection
- non-executable task stacks
- text segment write-protection
- exception vector table write-protection

For information about basic virtual memory support, see *6.9 Virtual Memory Management*, p.343.

Errors generated with the use of these features can be detected and managed with additional VxWorks facilities. See *11. Error Detection and Reporting* for more information.

6.10.1 Configuring VxWorks for Additional Memory Protection

The components listed in Table 6-7 provide additional memory-protection features. They can be added to VxWorks as a unit with the **INCLUDE_KERNEL_HARDENING** component. The individual and composite components all include the basic virtual memory component **INCLUDE_MMU_BASIC** by default.

Table 6-7 **Additional Memory Protection Components**

Component	Description
INCLUDE_PROTECT_TASK_STACK	Task stack overrun and underrun protection.
INCLUDE_TASK_STACK_NO_EXEC	Non-executable task stacks.
INCLUDE_PROTECT_TEXT	Text segment write-protection.
INCLUDE_PROTECT_VEC_TABLE	Exception vector table write-protection and NULL pointer reference detection.
INCLUDE_PROTECT_INTERRUPT_STACK	Interrupt stack overrun and underrun protection.

Note that protection of the kernel text segment—and the text segments of kernel modules dynamically loaded into the kernel space—is not provided by default. On the other hand, the text segment of processes and shared libraries is always write-protected, whether or not VxWorks is configured with the **INCLUDE_PROTECT_TEXT** component. Similarly, the execution stack of a process task is not affected by the **INCLUDE_PROTECT_TASK_STACK** or **INCLUDE_TASK_STACK_NO_EXEC** components—it is always protected unless the task is spawned with the **taskSpawn( )** option **VX_NO_STACK_PROTECT**.

6.10.2 Stack Overrun and Underrun Detection

VxWorks can be configured so that guard zones are inserted at the beginning and end of task execution stacks. For more information, see *Task Stack Guard Zones*, p.176.

The operating system can also be configured to insert guard zones at both ends of the interrupt stack. For more information, see *Interrupt Stack Protection*, p.243.

6.10.3 Non-Executable Task Stack

VxWorks can be configured so that task stacks are non-executable. For more information, see *Non-Executable Task Stacks*, p.177.

6.10.4 Text Segment Write Protection

All text segments are write-protected when VxWorks is configured with the **INCLUDE_PROTECT_TEXT** component. When VxWorks is loaded, all text segments are write-protected The text segments of any additional object modules loaded in the kernel space using **ld( )** are automatically marked as read-only. When object modules are loaded, memory that is to be write-protected is allocated in page-size increments. No additional steps are required to write-protect kernel application code.

6.10.5 Exception Vector Table Write Protection

When VxWorks is configured with the **INCLUDE_PROTECT_VEC_TABLE** component, the exception vector table is write-protected during system initialization.

The architecture-specific API provided to modify the vector table automatically write-enables the exception vector table for the duration of the call. For more information about these APIs, see the *VxWorks Architecture Supplement* for the architecture in question.

6.11 Processes Without MMU Support

VxWorks can be configured to provide support for real-time processes on a system based on a processor without an MMU, or based on a processor with MMU but with the MMU disabled.

With this configuration, a software simulation-based memory page management library keeps track of identity mappings only. This means that there is no address translation, and memory page attributes (protection attributes and cache attributes) are not supported.

⚠ **CAUTION:** VxWorks SMP does not support MMU-less configurations. For information about VxWorks SMP, see *15. VxWorks SMP*.

The advantages of a configuration without MMU support are that it:

- Enables the process environment on systems without an MMU. It provides private namespace for applications, for building applications independently from the kernel, and for simple migration from systems without an MMU to those with one.
- Allows application code be run in non-privileged (user) mode.
- Under certain conditions, it may provide increased performance by eliminating overhead of the TLB miss and reload. This assumes, however, that there is no negative impact due to the changed cache conditions.

The limitations of this configuration are:

- Depending on the processor type, BSP configuration, drivers and OS facilities used, disabling the MMU may require disabling the data cache as well. Disabling the data cache results in significant performance penalty that is much greater than the benefit derived from avoiding TLB misses.
- There is no memory protection. That is, memory cannot be write-protected, and neither the kernel or any process are protected from other processes.
- The address space is limited to the available system RAM, which is typically smaller than it would be available on systems with MMU-based address translation enabled. Because of the smaller address space, a system is more likely to run out of large contiguous blocks of memory due to fragmentation.
- Not all processors and target boards can be used with the MMU disabled. For the requirements of your system see the hardware manual of the board and processor used.

For information about architecture and processor-specific limitations, see the *VxWorks Architecture Supplement*.

Configuring VxWorks With Process Support for Systems Without an MMU

There are no special components needed for the process environment with software-simulated paging. As with any configurations that provide process support, the **INCLUDE_RTP** component must be added to the kernel.

The steps required to enable software-simulated paging are:

1. Add the **INCLUDE_RTP** component to include process support. This automatically includes all dependent subsystems, among them **INCLUDE_MMU_BASIC**.

2. Change the **SW_MMU_ENABLE** parameter of the **INCLUDE_MMU_BASIC** component to **TRUE** (the default value is **FALSE**).

In addition, the following optional configuration steps can reduce the footprint of the system:

3. Change the **VM_PAGE_SIZE** parameter of the **INCLUDE_MMU_BASIC** component. The default is architecture-dependent; usually 4K or 8K. Allowed values are 1K, 2K, 4K, 8K, 16K, 32K, 64K. Typically, a smaller page size results in finer granularity and therefore more efficient use of the memory space. However, smaller page size requires more memory needed for keeping track the mapping information.

4. Disable stack guard page protection by changing the **TASK_STACK_OVERFLOW_SIZE** and **TASK_STACK_UNDERFLOW_SIZE** configuration parameters to zero. Without protection provided by an MMU, stack overflow and underflow cannot be detected, so the guard pages serve no purpose.

5. Remove the following components from the VxWorks configuration: **INCLUDE_KERNEL_HARDENING**, **INCLUDE_PROTECT_TEXT**, **INCLUDE_PROTECT_VEC_TABLE**, **INCLUDE_PROTECT_TASK_STACK**, **INCLUDE_TASK_STACK_NO_EXEC**, and **INCLUDE_PROTECT_INTERRUPT_STACK**. Without an MMU, these features do not work. Including them only results in unnecessary consumption of resources.

7
I/O System

7.1 Introduction

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

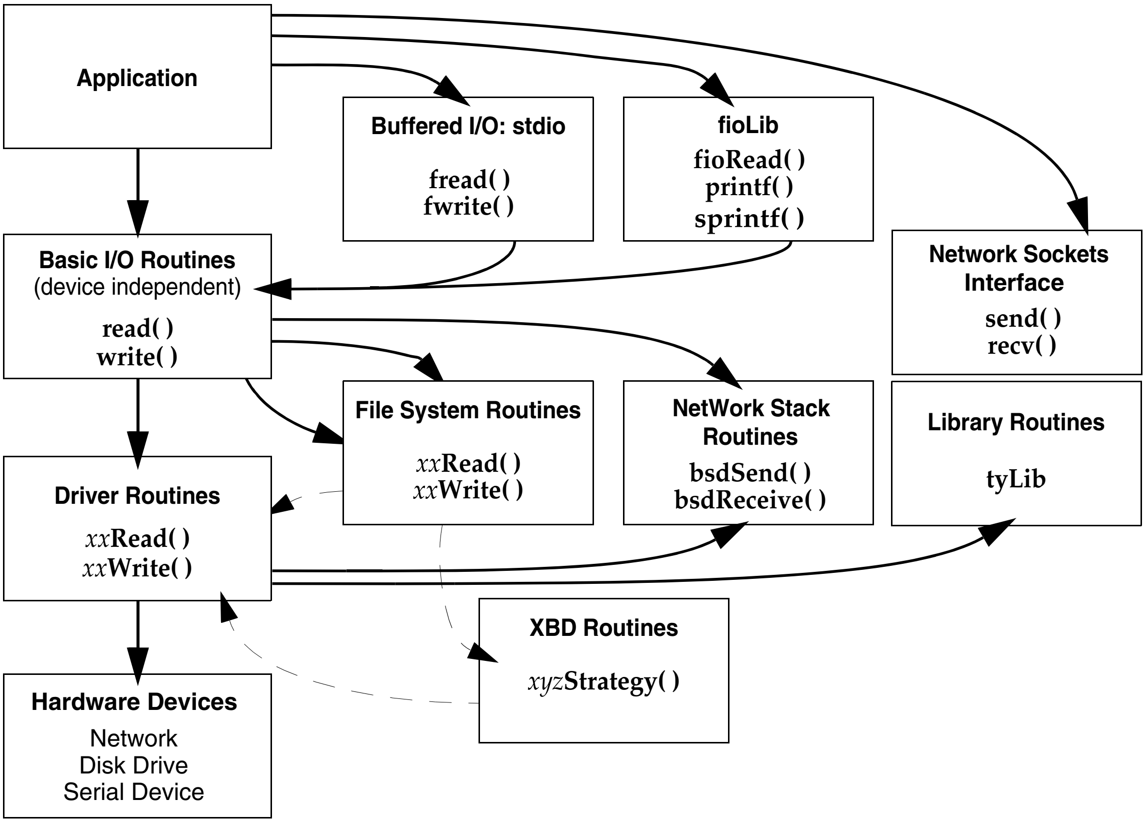
- character-oriented devices such as terminals or communications lines
- random-access block devices such as disks
- virtual devices such as intertask *pipes* and *sockets*
- monitor and control devices such as digital and analog I/O devices
- network devices that give access to remote devices

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.

The diagram in Figure 7-1 illustrates the relationships between the different elements of the VxWorks I/O system. All of these elements are discussed in this chapter, except for file system routines (which are dealt with in *8. Local File Systems*), and the network elements (which are covered in the *Wind River Network Stack for VxWorks 6 Programmer's Guide*).

Figure 7-1 **Overview of the VxWorks I/O System**



NOTE: The dotted lines in Figure 7-1 indicate that the XBD facility is required for some file systems, but not others. For example, HRFS and dosFs require XBD, while ROMFS has its own interface to drivers. See *7.8.8 Extended Block Device Facility: XBD*, p.402.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

7.2 Configuring VxWorks With I/O Facilities

The components providing the primary VxWorks I/O facilities are as follows:

- **INCLUDE_IO_BASIC**—provides basic I/O functionality.
- **INCLUDE_IO_FILE_SYSTEM**—provides file system support.
- **INCLUDE_POSIX_DIRLIB**—provides POSIX directory utilities.
- **INCLUDE_POSIX_FS**—provides POSIX file system APIs.
- **INCLUDE_IO_REMOVABLE**—provides support for removable file systems.
- **INCLUDE_IO_POSIX**—Provides POSIX I/O support.
- **INCLUDE_IO_RTP**—provides I/O support for RTPs.
- **INCLUDE_IO_MISC**—miscellaneous IO functions that are no longer referenced but are provided for backwards compatibility.

The component **INCLUDE_IO_SYSTEM** is provided for backward compatibility. It includes all the components listed above.

Components that provide support for additional features are described throughout this chapter.

7.3 Files, Devices, and Drivers

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

- 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 *device 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 conform to the conventional user view presented here, but can differ in the specifics. See *7.8 Devices in VxWorks*, p.391.

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.

7.3.1 Filenames and the Default Device

A filename 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 filename. Thus, the name **/tyCo/0** might name a particular serial I/O channel, and the name **DEV1:/file1** indicates the file **file1** on the **DEV1:** device.

When a filename 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 filename. 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. You can obtain the current default path by using **ioDefPathGet( )**. You can set the default path by using **ioDefPathSet( )**.

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 filenames in any way, other than during the search for matching device and filenames.

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

NOTE: To be recognized by the virtual root file system, device names must begin with a single leading forward-slash, and must not contain any other slash characters. For more information, see *8.3 Virtual Root File System: VRFS*, p.457.

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 filename in the remote directory on the remote system.

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

```
DEV1:
```

NOTE: Filenames and directory names on dosFs devices are often separated by backslashes (****). These can be used interchangeably with forward slashes (**/**).

CAUTION: Because device names are recognized by the I/O system using simple substring matching, a slash (**/** or ****) should not be used alone as a device name, nor should a slash be used as any part of a device name itself.

7.4 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 Table 7-1.

Table 7-1 **Basic I/O Routines**

Routine	Description
creat( )	Creates a file.
remove( )	Deletes a file.
open( )	Opens a file (optionally, creates a file if it does not already exist.)
close( )	Closes a file.
read( )	Reads a previously created or opened file.
write( )	Writes to a previously created or opened file.
ioctl( )	Performs special control functions on files.

7.4.1 File Descriptors

At the basic I/O level, files are referred to by a *file descriptor*. A file descriptor is a small integer returned by a call to **open( )** or **creat( )**. The other basic I/O calls take a file descriptor as a parameter to specify a file.

File descriptors are not global. The kernel has its own set of file descriptors, and each process (RTP) has its own set. Tasks within the kernel, or within a specific process share file descriptors. The only instance in which file descriptors may be shared across these boundaries, is when one process is a child of another process or of the kernel and it does not explicitly close a file using the descriptors it inherits from its parent. (Processes created by kernel tasks share only the spawning kernel task's standard I/O file descriptors 0, 1 and 2.) For example:

- If task **A** and task **B** are running in process **foo**, and they each perform a **write( )** on file descriptor 7, they will write to the same file (and device).
- If process **bar** is started independently of process **foo** (it is not **foo**'s child) and its tasks **X** and **Y** each perform a **write( )** on file descriptor 7, they will be writing to a different file than tasks **A** and **B** in process **foo**.

- If process **foobar** is started by process **foo** (it is **foo**'s child) and its tasks **M** and **N** each perform a **write( )** on file descriptor 7, they will be writing to the same file as tasks **A** and **B** in process **foo**. However, this is only true as long as the tasks do not close the file. If they close it, and subsequently open file descriptor 7 they will operate on a different file.

When a file is opened, a file descriptor is allocated and returned. When the file is closed, the file descriptor is deallocated.

File Descriptor Table

The number of file descriptors available in the kernel is defined with the **NUM_FILES** configuration macro. This specifies the size of the file descriptor table, which controls how many file descriptors can be simultaneously in use. The default number is 50, but it can be changed to suit the needs of the system.

To avoid running out of file descriptors, and encountering errors on file creation, applications should close any descriptors that are no longer in use.

The size of the file descriptor table for the kernel can also be changed at programmatically. The **rtpIoTableSizeGet( )** routine reads the size of the file descriptor table and the **rtpIoTableSizeSet( )** routine changes it. Note that these routines can be used with both the kernel and processes (the I/O system treats the kernel as a special kind of process).

The calling entity—whether kernel or process—can be specified with an **rtpIoTableSizeSet( )** call by setting the first parameter to zero. The new size of the file descriptor table is set with the second parameter. Note that you can only increase the size.

7.4.2 Standard Input, Standard Output, and Standard Error

Three file descriptors have special meanings:

- 0 is used for standard input (**stdin**).
- 1 is used for standard output (**stdout**).
- 2 is used for standard error output (**stderr**).

All tasks read their standard input—like **getchar( )**—from file descriptor 0. Similarly file descriptor 1 is used for standard output—like **printf( )**. And file descriptor 2 is used for outputting error messages. Using these descriptors, you can manipulate the input and output for many tasks at once by redirecting the files associated with the descriptors.

These standard file descriptors are used to make tasks and modules independent of their actual I/O assignments. If a module sends its output to standard output (file descriptor 1), its output can then 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 file descriptors. Second, individual tasks can override the global assignment of these file descriptors with assignments that apply only to that task.

7.4.3 Standard I/O Redirection

When VxWorks is initialized, the global standard I/O file descriptors, **stdin** (0), **stdout** (1) and **stderr** (2), are set to the system console device file descriptor by default, which is usually the serial tty device.

Each kernel task uses these global standard I/O file descriptors by default. Thus, any standard I/O operations like calls to **printf( )** and **getchar( )** use the global standard I/O file descriptors.

Standard I/O can be redirected, however, either at the individual task level, or globally for the kernel.

The standard I/O of a specific task can be changed with the **ioTaskStdSet( )** routine. The parameters of this routine are the task ID of the task for which the change is to be made (0 is used for the calling task itself), the standard file descriptor to be redirected, and the file descriptor to direct it to. For example, a task can make the following call to write standard output to the **fileFd** file descriptor:

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

The third argument (**fileFd** in this case) can be any valid open file descriptor. If it is a file system file, all the task's subsequent standard output, such as that from **printf( )**, is written to it.

To reset the task standard I/O back to global standard I/O, the third argument can be 0, 1, or 2.

The global standard I/O file descriptors can also be changed from the default setting, which affects all kernel tasks except that have had their task-specific standard I/O file descriptors changed from the global ones.

Global standard I/O file descriptors are changed by calling **ioGlobalStdSet( )**. The parameters to this routine are the standard I/O file descriptor to be redirected, and the file descriptor to direct it to. For example:

```
ioGlobalStdSet (1, newFd);
```

This call sets the global standard output to **newFd**, which can be any valid open file descriptor. All tasks that do not have their individual task standard output redirected are affected by this redirection, and all their subsequent standard I/O output goes to **newFd**.

The current settings of the global and any task's task standard I/O can be determined by calling **ioGlobalStdGet( )** and **ioTaskStdGet( )**. For more information, see the VxWorks API references for these routines.

Issues with Standard I/O Redirection

Be careful with file descriptors used for task standard I/O redirection to ensure that data corruption does not occur. Before any task's standard I/O file descriptors are closed, they should be replaced with new file descriptors with a call to **ioTaskStdSet( )**.

If a task's standard I/O is set with **ioTaskStdSet( )**, the file descriptor number is stored in that task's memory. In some cases, this file descriptor may be closed, released by some other task or the one that opened it. Once it is released, it may be reused and opened to track a different file. Should the task holding it as a task standard I/O descriptor continue to use it for I/O, data corruption is unavoidable.

As an example, consider a task spawned from a telnet or rlogin session. The task inherits the network session task's standard I/O file descriptors. If the session exits, the standard I/O file descriptors of the network session task are closed. However, the spawned task still holds those file descriptors as its task standard I/O continued with input and output to them. If the closed file descriptors are recycled and re-used by other **open( )** call, however, data corruption results, perhaps with serious consequences for the system. To prevent this from happening, all spawned tasks must have their standard I/O file descriptors redirected before the network session is terminated.

The following example illustrates this scenario, with redirection of a spawned task's standard I/O to the global standard I/O from the shell before logout. The **taskspawn( )** call is abbreviated to simplify presentation.

```
-> taskSpawn "someTask",......
Task spawned: id = 0x52a010, name = t4
value = 5414928 = 0x52a010
-> ioTaskStdSet 0x52a010,0,0
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,1,1
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,2,2
value = 0 = 0x0
-> logout
```

The next example illustrates task standard I/O redirection to other file descriptors.

```
-> taskSpawn "someTask",......
Task spawned: id = 0x52a010, name = t4
value = 5414928 = 0x52a010
-> ioTaskStdSet 0x52a010,0,someOtherFdx
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,1,someOtherFdy
value = 0 = 0x0
-> ioTaskStdSet 0x52a010,2,someOtherFdz
value = 0 = 0x0
-> logout
```

7.4.4 Open and Close

Before I/O can be performed on a device, a file descriptor must be opened to the device by invoking the **open( )** routine—or **creat( )**, as discussed in the next section. The arguments to **open( )** are the filename, the type of access, and the mode (file permissions):

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

For **open( )** calls made in the kernel, the mode parameter can be set to zero if file permissions do not need to be specified.

The file-access options that can be used with the *flags* parameter to **open( )** are listed in Table 7-2.

Table 7-2 **File Access Options**

Flag	Description
O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.
O_CREAT	Create a file if it does not already exist.
O_EXCL	Error on open if the file exists and **O_CREAT** is also set.
O_SYNC	Write on the file descriptor complete as defined by synchronized I/O file integrity completion.

Table 7-2 **File Access Options** (cont'd)

Flag	Description
O_DSYNC	Write on the file descriptor complete as defined by synchronized I/O data integrity completion.
O_RSYNC	Read on the file descriptor complete at the same sync level as **O_DSYNC** and **O_SYNC** flags.
O_APPEND	Set the file offset to the end of the file prior to each write, which guarantees that writes are made at the end of the file. It has no effect on devices other than the regular file system.
O_NONBLOCK	Non-blocking I/O.
O_NOCTTY	If the named file is a terminal device, don't make it the controlling terminal for the process.
O_TRUNC	Open with truncation. If the file exists and is a regular file, and the file is successfully opened, its length is truncated to 0. It has no effect on devices other than the regular file system.

⚠ **WARNING:** While the third parameter to **open( )**—*mode*, for file permissions—is usually optional for other operating systems, it is required for the VxWorks implementation of **open( )** in the kernel (but not in processes). When the mode parameter is not appropriate for a given call, it should be set to zero. Note that this can be an issue when porting software from UNIX to VxWorks.

Note the following special cases with regard to use of the file access and mode (file permissions) parameters to **open( )**:

- In general, you can open only preexisting devices and files with **open( )**. However, with NFS network, dosFs, and HRFS devices, you can also create files with **open( )** by OR'ing **O_CREAT** with one of the other access flags.
- HRFS directories can be opened with the **open( )** routine, but only using the **O_RDONLY** flag.
- With both dosFs and NFS devices, you can use the **O_CREAT** flag to create a subdirectory by setting *mode* to **FSTAT_DIR**. Other uses of the mode parameter with dosFs devices are ignored.

- With an HRFS device you cannot use the **O_CREAT** flag and the **FSTAT_DIR** mode option to create a subdirectory. HRFS ignores the mode option and simply creates a regular file.
- The netDrv default file system does not support the **F_STAT_DIR** mode option or the **O_CREAT** flag.
- For NFS devices, the third parameter to **open( )** is normally used to specify the mode of the file. For example:

```
myFd = open ("fooFile", O_CREAT | O_RDWR, 0644);
```

- While HRFS supports setting the permission mode for a file, it is not used by the VxWorks operating system.
- Files can be opened with the **O_SYNC** flag, indicating that each write should be immediately written to the backing media. This flag is currently supported by the dosFs file system, and includes synchronizing the FAT and the directory entries.
- The **O_SYNC** flag has no effect with HRFS because file system is always synchronous. HRFS updates files as though the **O_SYNC** flag were set.

NOTE: Drivers or file systems may or may not honor the flag values or the mode values. A file opened with **O_RDONLY** mode may in fact be writable if the driver allows it. Consult the driver or file system information for specifics.

See the VxWorks file system API references for more information about the features that each file system supports.

The **open( )** routine, if successful, returns a file descriptor. This file descriptor is then used in subsequent I/O calls to specify that file. The file descriptor is an identifier that is not task specific; that is, it is shared by all tasks within the memory space. Within a given process or the kernel, therefore, one task can open a file and any other task can then use the file descriptor. The file descriptor remains valid until **close( )** is invoked with that file descriptor, as follows:

```
close (fd);
```

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

Since the kernel only terminates when the system shuts down, there is no situation analogous to file descriptors being closed automatically when a process terminates. File descriptors in the kernel can only be closed by direct command.

7.4.5 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 filename specifies the name of the new file rather than an existing one; the **creat( )** routine returns a file descriptor identifying the new file.

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

Note that with the HRFS file system the **creat( )** routine is POSIX-compliant, and the second parameter is used to specify file permissions; the file is opened in **O_RDWR** mode.

With dosFs, however, the **creat( )** routine is not POSIX-compliant and the second parameter is used for open mode flags.

The **remove( )** routine deletes a named file on a file-system device:

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

Files should be closed before they are removed.

With non-file-system devices, the **creat( )** routine performs the same function as **open( )**. The **remove( )** routine, however has no effect.

7.4.6 Read and Write

After a file descriptor 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 file descriptor, 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 file descriptor, 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). The **write( )** routine returns the number of bytes written; if the number returned is not equal to the number requested, an error has occurred.

7.4.7 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 a file descriptor and the desired length of the file in bytes:

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

If it succeeds in truncating the file, **ftruncate( )** returns **OK**.

If the file descriptor refers to a device that cannot be truncated, **ftruncate( )** returns **ERROR**, and sets **errno** to **EINVAL**.

If the size specified is larger than the actual size of the file, the result depends on the file system. For both dosFs and HRFS, the size of the file is extended to the specified size; however, for other file systems, **ftruncate( )** returns **ERROR**, and sets **errno** to **EINVAL** (just as if the file descriptor referred to a device that cannot be truncated).

The **ftruncate( )** routine is part of the POSIX 1003.1b standard. It is fully supported as such by the HRFS. The dosFs implementation is, however, only partially compliant: creation and modification times are not changed.

Also note that with HRFS the *seek* position is not modified by truncation, but with dosFs the seek position is set to the end of the file.

7.4.8 I/O Control

The **ioctl( )** routine provides a flexible mechanism for performing I/O functions that are not performed by 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 file descriptor, 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 *7.8 Devices in VxWorks*, p.391 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 or file systems.

7.4.9 Pending on Multiple File Descriptors with select( )

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. An example of using the select facility to pend on multiple file descriptors is 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 7-3.

Table 7-3 **Select Macros**

Macro	Description
FD_ZERO	Zeroes all bits.
FD_SET	Sets the bit corresponding to a specified file descriptor.
FD_CLR	Clears a specified bit.
FD_ISSET	Returns non-zero if the 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.441.

Example 7-1 **Using select( )**

```
/* 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)
        {
        close (fds[0]);
        return (ERROR);
        }
    if ((fds[1] = open (PIPENORM, O_RDONLY, 0)) == ERROR)
```

```
        {
        close (fds[0]);
        close (fds[1]);
        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)
        {
        close (fds[0]);
        close (fds[1]);
        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);
            }
        }
        }
    }
```

7.4.10 POSIX File System Routines

The POSIX **fsPxLib** library provides I/O and file system routines for various file manipulations. These routines are described in Table 7-4.

Table 7-4 **File System Routines**

Routine	Description
unlink( )	Unlink a file.
link( )	Link a file.
fsync( )	Synchronize a file.
fdatasync( )	Synchronize the data of a file.
rename( )	Change the name of a file.
fpathconf( )	Determine the current value of a configurable limit.
pathconf( )	Determine the current value of a configurable limit.
access( )	Determine accessibility of a file.
chmod( )	Change the permission mode of a file.
fcntl( )	Perform control functions over open files.

For more information, see the API references for **fsPxLib** and **ioLib**.

7.5 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. Configure VxWorks with the ANSI Standard component bundle to provide buffered I/O support.

NOTE: 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 *7.6 Other Formatted I/O*, p.380.

7.5.1 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.

This overhead is quite small because the VxWorks primitives are fast. 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* is a handle for the opened file and its associated buffers and pointers. A file pointer 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, which is a small integer. In fact, the **FILE** structure pointed to by the file pointer contains the underlying file descriptor of the open file.

A file descriptor that is already open can be associated subsequently 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.

The **FILE** buffer is deallocated when **fclose( )** is called.

7.5.2 Standard Input, Standard Output, and Standard Error

As discussed in *7.4 Basic I/O*, p.365, there are three special file descriptors (0, 1, and 2) reserved for standard input, standard output, and standard error. Three corresponding *stdio* **FILE** buffers are automatically created when a task uses the standard file descriptors, *stdin*, *stdout*, and *stderr*, to do buffered I/O to the standard file descriptors. Each task using the standard I/O file descriptors has its own *stdio* **FILE** buffers. The **FILE** buffers are deallocated when the task exits.

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

7.6 Other Formatted I/O

This section describes additional formatting routines and facilities.

7.6.1 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*.

7.6.2 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 file descriptor (2). The routine **fdprintf( )** outputs formatted strings to a specified file descriptor.

7.6.3 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( )**.

7.7 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 de-couple I/O operations from the activities of a particular task when these are logically independent.

The VxWorks AIO implementation meets the specification in the POSIX 1003.1b standard.

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.

Include AIO in your VxWorks configuration with the **INCLUDE_POSIX_AIO** and **INCLUDE_POSIX_AIO_SYSDRV** components. The second configuration constant enables the auxiliary AIO system driver, required for asynchronous I/O on all current VxWorks devices.

7.7.1 The POSIX AIO Routines

The VxWorks library **aioPxLib** provides 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 7-5.

The default VxWorks initialization code calls **aioPxLibInit( )** automatically when the POSIX AIO component is included in VxWorks with **INCLUDE_POSIX_AIO**.

The **aioPxLibInit( )** 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**. When the parameter is 0 (the default), the value is taken from **AIO_CLUST_MAX** (defined in *installDir***/vxworks-6.***x***/target/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 included in VxWorks. 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.

Table 7-5 **Asynchronous Input/Output Routines**

Function	Description
aioPxLibInit( )	Initializes the AIO library (non-POSIX).
aioShow( )	Displays the outstanding AIO requests (non-POSIX).[a]
aio_read( )	Initiates an asynchronous read operation.
aio_write( )	Initiates an asynchronous write operation.
lio_listio( )	Initiates a list of up to **LIO_MAX** asynchronous I/O requests.
aio_error( )	Retrieves the error status of an AIO operation.
aio_return( )	Retrieves the return status of a completed AIO operation.
aio_cancel( )	Cancels a previously submitted AIO operation.
aio_suspend( )	Waits until an AIO operation is done, interrupted, or timed out.
aio_fsync( )	Asynchronously forces file synchronization.

[a] This function is not built into the host shell. To use it from the host shell, VxWorks must be configured with the **INCLUDE_POSIX_AIO_SHOW** component. When you invoke the function, its output is sent to the standard output device.

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: **MAX_AIO_SYS_TASKS**, **AIO_TASK_PRIORITY**, and **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 *installDir***/vxworks-6.***x***/target/h/aioSysDrv.h**).

Table 7-6 lists the names of the constants, and shows the constants used within initialization routines when the parameters are left at their default values of 0, and where these constants are defined.

Table 7-6 **AIO Initialization Functions and Related Constants**

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

7.7.2 AIO Control Block

Each of the AIO calls takes an AIO control block (**aiocb**) as an argument. The calling routine must allocate space for the **aiocb**, and this space must remain available for the duration of the AIO operation. (Thus the **aiocb** must not be created on the task's stack unless the calling routine will not return until after the AIO operation is complete and **aio_return( )** has been called.) Each **aiocb** describes a single AIO operation. Therefore, simultaneous asynchronous I/O operations using the same **aiocb** are not valid and produce undefined results.

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

aio_fildes
The file descriptor for I/O.

aio_offset
The offset from the beginning of the file.

aio_buf
The address of the buffer from/to which AIO is requested.

aio_nbytes
The number of bytes to read or write.

aio_reqprio
The priority reduction for this AIO request.

aio_sigevent
The signal to return on completion of an operation (optional).

aio_lio_opcode
An operation to be performed by a **lio_listio( )** call.

aio_sys
The address of VxWorks-specific data (non-POSIX).

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

CAUTION: The **aiocb** structure and the data buffers referenced by it are used by the system to perform the AIO request. Therefore, once the **aiocb** has been submitted to the system, the application must not modify the **aiocb** structure until after a subsequent call to **aio_return( )**. The **aio_return( )** call retrieves the previously submitted AIO data structures from the system. After the **aio_return( )** call, the calling application can modify the **aiocb**, free the memory it occupies, or reuse it for another AIO call. If space for the **aiocb** is allocated from the stack, the task should not be deleted (or complete running) until the **aiocb** has been retrieved from the system with an **aio_return( )** call.

7.7.3 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 this 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( )**. To force all I/O operations to the synchronized I/O completion state, use **aio_fsync( )**.

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 7-2 **Asynchronous I/O**

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

/* includes */

#include <vxWorks.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <aio.h>

/* defines */

#define BUFFER_SIZE 200

struct aiocb    aiocb_read; /* read aiocb */
struct aiocb    aiocb_write; /* write aiocb */

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

STATUS aioExample (const char *exFile)
    {
    int             fd;
    char            buffer [BUFFER_SIZE]; /* buffer for read aiocb */
    static char *   test_string = "testing 1 2 3";
    int             error;

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


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

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

    memset ( &aiocb_write, 0, 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 ((error = aio_error (&aiocb_read) == EINPROGRESS) ||
    (aio_error (&aiocb_write) == EINPROGRESS))
    sleep (1);

    printf ("aioExample: error = %d\n", error);

    /* 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 for notification that the write operation has finished. 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.

Example 7-3 **Asynchronous I/O with Signals**

```
#include <vxWorks.h>
#include <stdio.h>
#include <aio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

/* defines */

#define BUFFER_SIZE   200
#define LIST_SIZE    1
#define WRITE_EXAMPLE_SIG_NO  25 /* signal number */
#define READ_EXAMPLE_SIG_NO  26 /* signal number */

/* forward declarations */

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

static struct aiocb         aiocb_read;   /* read aiocb  */
static struct aiocb         aiocb_write;  /* write aiocb */
static struct sigaction     write_action; /* signal info */
static struct sigaction     read_action;  /* signal info */
static char                 buffer [BUFFER_SIZE]; /* aiocb read buffer */

/************************************************************************
* 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 (const char *exFile)
    {
    int             fd;
    static char *   test_string = "testing 1 2 3";

    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 WRITE_EXAMPLE_SIG_NO */

    write_action.sa_sigaction = writeSigHandler;
    write_action.sa_flags = SA_SIGINFO;
    sigemptyset (&write_action.sa_mask);
    sigaction (WRITE_EXAMPLE_SIG_NO, &write_action, NULL);

    /* set up signal handler for READ_EXAMPLE_SIG_NO */

    read_action.sa_sigaction = readSigHandler;
    read_action.sa_flags = SA_SIGINFO;
    sigemptyset (&read_action.sa_mask);
    sigaction (READ_EXAMPLE_SIG_NO, &read_action, NULL);

    /* initialize read and write aiocbs */

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

    /* set up signal info */

    aiocb_read.aio_sigevent.sigev_signo = READ_EXAMPLE_SIG_NO;
    aiocb_read.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aiocb_read.aio_sigevent.sigev_value.sival_ptr =
    (void *) &aiocb_read;

    memset ( &aiocb_write, 0, 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 = WRITE_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");

    close (fd);
    return (OK);
    }

void writeSigHandler
    (
    int                 sig,
    struct siginfo *    info,
    void *              pContext
    )
    {
    /* print out what was written */
    printf ("writeSigHandler: 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 ("writeSigHandler: aio_return for aiocb_write failed\n");
    }
    }

void readSigHandler
    (
    int                 sig,
    struct siginfo *    info,
    void *              pContext
    )
    {
    /* print out what was read */
    printf ("readSigHandler: Got signal for aio read\n");
```

```
/* write is complete so let's do cleanup for it here */
if (aio_return (info->si_value.sival_ptr) == -1)
{
printf ("readSigHandler: aio_return for aiocb_read failed\n");
}
else
{
printf ("aioExample: message = %s\n", buffer);
}
}
```

7.8 Devices in VxWorks

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

Table 7-7 **Devices Provided with VxWorks**

Device	Driver Description
tty	Terminal device
pty	Pseudo-terminal device
pipe	Pipe device
mem	Pseudo memory device
nfs	NFS client device
net	Network device for remote file access
null	Null device
ram	RAM device for creating a RAM disk
scsi	SCSI interface
romfs	ROMFS device
–	Other hardware-specific device

WARNING: Devices should not be given the same name, or they will overwrite each other in core I/O.

NOTE: Only VxBus-compatible drivers can be used with the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

7.8.1 Serial I/O Devices: Terminal and Pseudo-Terminal Devices

VxWorks provides terminal and pseudo-terminal devices (*tty* and *pty*). The *tty* device is for actual terminals; the *pty* device is for processes that simulate terminals. These pseudo terminals are useful in applications such as remote login facilities.

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.

NOTE: 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. For example, to set all the *tty* options except **OPT_MON_TRAP**:

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL & ~OPT_MON_TRAP);
```

For more information, see *I/O Control Functions*, p.396.

Table 7-8 is a summary of the available options. The listed names are defined in the header file **ioLib.h**. For more detailed information, see the API reference entry for **tyLib**.

Table 7-8 **Tty Options**

Library	Description
OPT_LINE	Selects *line mode*. (See *Raw Mode and Line Mode*, p.393.)
OPT_ECHO	Echoes input characters to the output of the same channel.
OPT_CRMOD	Translates input **RETURN** characters into **NEWLINE** (\n); translates output **NEWLINE** into **RETURN-LINEFEED**.
OPT_TANDEM	Responds to software flow control characters **CTRL+Q** and **CTRL+S** (**XON** and **XOFF**).
OPT_7_BIT	Strips the most significant bit from all input bytes.
OPT_MON_TRAP	Enables the special *ROM monitor trap* character, **CTRL+X** by default.
OPT_ABORT	Enables the special kernel shell abort character, **CTRL+C** by default. (Only useful if the kernel shell is configured into the system)
OPT_TERMINAL	Sets all of the above option bits.
OPT_RAW	Sets 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.392).

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.394.

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 software flow control characters **CTRL+Q** and **CTRL+S** (**XON** and **XOFF**). 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. The software flow control characters are 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.[1] The monitor trap character is enabled by **OPT_MON_TRAP**.

- The special *kernel shell abort* character, by default **CTRL+C**. This character restarts the kernel shell if it gets stuck in an unfriendly routine, such as one that

[1] It will not be possible to restart VxWorks if un-handled external interrupts occur during the boot countdown.

has taken an unavailable semaphore or is caught in an infinite loop. The kernel 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 7-9.

Table 7-9 **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	kernel 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 7-10, defined in **ioLib.h**. For more information, see the reference entries for **tyLib**, **ttyDrv**, and **ioctl( )**.

Table 7-10 **I/O Control Functions Supported by tyLib**

Function	Description
FIOBAUDRATE	Sets the baud rate to the specified argument.
FIOCANCEL	Cancels a read or write.
FIOFLUSH	Discards all bytes in the input and output buffers.
FIOGETNAME	Gets the filename of the file descriptor.
FIOGETOPTIONS	Returns the current device option word.
FIONREAD	Gets the number of unread bytes in the input buffer.
FIONWRITE	Gets the number of bytes in the output buffer.
FIOSETOPTIONS	Sets the device option word.

⚠ **CAUTION:** 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 *installDir***/vxworks-6.***x***/target/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 *installDir***/vxworks-6.***x***/target/h/sioLib.h** provide the POSIX definitions for setting the hardware options.

7.8.2 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 blocked 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 output 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( )**. For more information ISRs, see *4.20 Interrupt Service Routines*, p.241.

I/O Control Functions

Pipe devices respond to the **ioctl( )** functions summarized in Table 7-11. 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 7-11 **I/O Control Functions Supported by pipeDrv**

Function	Description
FIOFLUSH	Discards all messages in the pipe.
FIOGETNAME	Gets the pipe name of the file descriptor.
FIONMSGS	Gets the number of messages remaining in the pipe.
FIONREAD	Gets the size in bytes of the first message in the pipe.

7.8.3 Pseudo I/O Device

The **memDrv** device 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. The device provides a high-level means for reading and writing bytes in absolute memory locations through I/O calls. It is useful when data must be preserved between boots of VxWorks or when sharing data between CPUs.

The **memDrv** driver is initialized automatically by the system with **memDrv( )** when the **INCLUDE_USR_MEMDRV** component is included in VxWorks. The call for device creation must be made from the kernel:

```
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 **memDrv( )**, **memDevCreate( )**, and **memDevCreateDir( )** entries in the VxWorks API reference, as well as the entry for **memdrvbuild** in the online *Wind River Host Utilities API Reference*.

For information about creating a RAM disk, which provides support for file systems,.

I/O Control Functions

The memory device responds to the **ioctl( )** functions summarized in Table 7-12. The functions listed are defined in the header file **ioLib.h**.

Table 7-12 **I/O Control Functions Supported by memDrv**

Function	Description
FIOSEEK	Sets the current byte offset in the file.
FIOWHERE	Returns the current byte position in the file.

For more information, see the reference entries for **memDrv**, **ioLib**, and **ioctl( )**.

7.8.4 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.

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*.

For detailed information about the VxWorks implementation of NFS, see *9. Network File System: NFS*.

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( )**.

To include NFS client support, use the **INCLUDE_NFS** component.

The subject of exporting and mounting NFS file systems and authenticating access permissions is discussed in more detail in *9. Network File System: NFS*. 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 7-13. The functions listed are defined in **ioLib.h**. For more information, see the reference entries for **nfsDrv**, **ioLib**, and **ioctl( )**.

Table 7-13 **I/O Control Functions Supported by nfsDrv**

Function	Description
FIOFSTATGET	Gets file status information (directory entry data).
FIOGETNAME	Gets the filename of the file descriptor.
FIONREAD	Gets the number of unread bytes in the file.
FIOREADDIR	Reads the next directory entry.
FIOSEEK	Sets the current byte offset in the file.
FIOSYNC	Flushes data to a remote NFS file.
FIOWHERE	Returns the current byte position in the file.

7.8.5 Non-NFS Network Devices

VxWorks also supports network access to files on a remote host through the Remote Shell protocol (RSH) or the File Transfer Protocol (FTP).

These implementations of network devices use the driver **netDrv**, which is included in the Wind River Network Stack. Using this driver, you can open, read, write, and close files located on remote systems without needing to manage the details of the underlying protocol used to effect the transfer of information. (For more information, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.)

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 memory-resident copy of the file. When closed, the file is copied back to the original remote file if it was modified.

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 *Wind River Network Stack for VxWorks 6 Programmer's Guide: File Access Applications* and in the API 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 API reference entries for **netDrv** and **ioctl( )**.

7.8.6 Null Devices

VxWorks provides both **/null** and **/dev/null** for null devices. The **/null** device is the traditional VxWorks null device, which is provided by default for backward compatibility. The **/dev/null** device is provided by the **BUNDLE_RTP_POSIX_PSE52** component bundle, and is required for conformance with the POSIX PSE52 profile.

Note that the **devs** shell command lists **/null** and **/dev/null** with other devices, but the **ls** command does not list **/dev/null** under the VRFS root directory (because the name violates the VRFS naming scheme). Applications can, in any case, use **/null** or **/dev/null** as required.

For information about POSIX PSE52, see the *VxWorks Kernel Programmer's Guide: POSIX Facilities*. For information about VRFS, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

7.8.7 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 file descriptor.

VxWorks socket routines are source-compatible with the BSD 4.4 UNIX socket functions and the Windows Sockets (Winsock 1.1) networking standard. Use of these routines is discussed in *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

7.8.8 Extended Block Device Facility: XBD

The extended block device (XBD) facility mediates I/O activity between file systems and block devices. It provides a standard interface between file systems on the one hand, and block drivers on the other.

The XBD facility also provides support for removable file systems, automatic file system detection, and multiple file systems. For more information on these features, see *8. Local File Systems*.

NOTE: The XBD facility is required for some file systems (such as HRFS, dosFs, cdromFs, and rawFs), but not others (such as ROMFS).

For detailed information on developing XBD-compatible device drivers, see the *VxWorks Device Driver Developer's Guide*.

The basic XBD facility is proved with the **INCLUDE_XBD** component, which also provides generic service for following optional components:

INCLUDE_XBD_RAMDRV
Provides support for RAM disks. See *XBD RAM Disk*, p.411.

INCLUDE_XBD_PART_LIB
Provides disk partitioning facilities. See *XBD Disk Partition Manager*, p.403.

INCLUDE_XBD_BLK_DEV
Provides support for legacy block device drivers that were designed to work with the predecessor to XBD—the cache block I/O (CBIO) facility. These devices include floppy drives, SCSI, and TrueFFS (the disk-access emulator for flash). See *XBD Block Device Wrapper*, p.404.

INCLUDE_XBD_TRANS
Provides a transaction-based file system (TRFS) facility, which can be used with dosFs. It provides fault-tolerant file system consistency and fast recovery in response to power loss. See *7.8.9 Transaction-Based Reliable File System Facility: TRFS*, p.405.

XBD Disk Partition Manager

VxWorks provides support for PC-style disk partitioning with the **INCLUDE_XBD_PART_LIB** component, which facilitates sharing fixed disks and removable cartridges between VxWorks target systems and PCs running Windows. This component includes two modules: **xbdPartition** and **partLib**.

xbdPartition Module

The **xbdPartition** facility creates a device for each partition that is detected on media. Each partition that is detected is probed by the file system monitor and an I/O device is added to VxWorks. This device is an instantiation of the file system found by the file system monitor (or rawFs if the file system is not recognized or detected). If no partitions are detected, a single device is created to represent the entire media. There can be up to four partitions on a single media. For information about the file system monitor, see *8.2 File System Monitor*, p.455.

The partition facility also names the partitions. The names are derived from the base device name and the partition number. The base device name is derived from the device driver name (for more information in this regard, see the *VxWorks Device Driver Developer's Guide*). For example, the XBD-compatible device for an ATA hard disk would have a base name of /ata00. If it had four partitions, they would be named as follows:

```
/ata00:1
/ata00:2
/ata00:3
/ata00:4
```

If there were no partitions, the name would be /ata00:0. For an example of how the facility is used, see Example 8-1.

partLib Library

The partLib library provides facilities for creating PC-style partitions on media. It can create up to four primary partitions. Note that when partitions are created, any existing information on the media is lost. For more information see the VxWorks API reference for **xbdCreatePartition( )**.

XBD Block Device Wrapper

The **INCLUDE_XBD_BLKDEV** component provides support for legacy block devices that were designed to work the predecessor to XBD—the cache block I/O (CBIO) facility. It provides a wrapper XBD facility that converts the block I/O driver interface based on the **BLK_DEV** logical block device structure into an XBD API-compliant interface.

NOTE: The Wind River devices that require the **INCLUDE_XBD_BLKDEV** component in addition to **INCLUDE_XBD** are floppy, SCSI, and TrueFFS (the disk-access emulator for flash) drivers. Any third-party device drivers based on the **BLK_DEV** interface also require **INCLUDE_XBD_BLKDEV**.

The Wind River drivers that do not require the **INCLUDE_XBD_BLK_DEV** component are USB block storage, ATA, and the XBD RAM disk.

CAUTION: Depending on the implementation of the driver, the **INCLUDE_XBD_BLK_DEV** component may not properly detect media insertion and removal. It may, therefore remove the file system when the media is removed, or not instantiate a file system when media is inserted.

XBD TRFS Component

The **INCLUDE_XBD_TRANS** component is an XBD-compatible transaction-based reliable file system (TRFS) facility. TRFS an I/O facility that provides fault-tolerant file system layer for the dosFs file system. See *7.8.9 Transaction-Based Reliable File System Facility: TRFS*, p.405 for more information.

7.8.9 Transaction-Based Reliable File System Facility: TRFS

The transaction-based reliable file system (TRFS) facility provides a fault-tolerant file system I/O layer for the dosFs file system. It is provided with the **INCLUDE_XBD_TRANS** component.

TRFS provides both file system consistency and fast recovery for the dosFs file system—DOS-compatible file systems are themselves neither reliable nor transaction-based. It is designed to operate with XBD-compliant device drivers for hard disks, floppy disks, compact flash media, TrueFFS flash devices, and so on. It can also be used with the XBD wrapper component for device drivers that are not XBD-compliant.

TRFS provides reliability in resistance to sudden power loss: files and data that are already written to media are unaffected, they will not be deleted or corrupted because data is always written either in its entirety or not at all.

TRFS provides additional guarantees in its transactional feature: data is always maintained intact up to a given commit transaction. User applications set transaction points on the file system. If there is an unexpected failure of the system, the file system is returned to the state it was in at the last transaction point. That is, if data has changed on the media after a commit transaction but prior to a power loss, it is automatically restored to the its state at the last commit transaction to further ensure data integrity. On mounting the file system, TRFS detects any failure and rolls back data to the last secure transaction.

Unlike some facilities that provide data integrity on a file-by-file basis, TRFS protects the medium as a whole. It is transactional for a file system, which means that setting transaction points will commit all files, not just the one used to set the transaction point.

NOTE: While TRFS is a I/O layer added to dosFs, it uses a modified on-media format that is not compatible with other FAT-based file systems, including Microsoft Windows and the VxWorks dosFs file system without the TRFS layer. It should not, therefore, be used when compatibility with other systems is a requirement

For information about dosFs, see *8.5 MS-DOS-Compatible File System: dosFs*, p.478.

Configuring VxWorks With TRFS

Configure VxWorks with the **INCLUDE_XBD_TRANS** component to provide TRFS functionality for your dosFs file system.

Automatic Instantiation of TRFS

TRFS is automatically detected and instantiated if the media has already been formatted for use with TRFS, in a manner very similar to the instantiation of the dosFs or HRFS file system. The primary difference is that when TRFS is detected by the file system monitor, it calls the TRFS creation function, and the creation function then creates another XBD instance and generates an insertion event for it. The monitor then detects the new XBD and begins probing. In this case, however, the monitor does not examine the media directly—all commands are routed through TRFS, which performs the appropriate translations. If a file system is detected, such as dosFs, the dosFs creation function is called by the monitor and dosFs is instantiated. If not, rawfs is instantiated.

For information about how file systems are automatically instantiated, see *8.2 File System Monitor*, p.455.

Formatting a Device for TRFS

TRFS low-level formatting is accomplished with the call:

```
usrFormatTrans(device, overHead, type);
```

The arguments are:

device
The volume name to format. For example, **"/ata"**.

overHead

An integer that identifies the portion of the disk to use as transactional workspace in parts-per-thousand of the disk.

type

An integer with the values of either **FORMAT_REGULAR** (0), which does not reserve any blocks from the disk; or **FORMAT_TFFS** (1), which reserves the first block.

Once the TRFS format is complete, a dosFs file system can be created by calling the dosFs formatter on the same volume.

When a FAT file system is created using the function **dosFsVolFormat( )** in conjunction with TRFS, a transaction point is automatically inserted following the format. One cannot, therefore, *unformat* by rolling back a transaction point.

Example 7-4 **Formatting a Device for TRFS**

```
/* Create a RAM disk with 512 byte sized sectors and 1024 sectors.*/
if (xbdRamDiskDevCreate (512, 1024 * 512, 0, "/trfs") == NULL)
    {
    printf ("Could not create RAM disk\n");
    return;
    }
/* Put TRFS on the RAM disk */
if (usrFormatTrans ("/trfs", 100, 0) != OK)
    {
    printf ("Could not format\n");
    return;
    }

/* Now put dosFs on TRFS */
if (dosFsVolFormat ("/trfs", DOS_OPT_BLANK, 0) != OK)
    {
    printf ("Could not format for dos\n");
    return;
    }

/* Create a file on the TRFS/DosFS volume */
fd = open ("/trfs/myfile", O_CREAT | O_RDWR, 0666);
if (fd < 0)
    {
    printf ("Couldn't create file\n");
    return;
    }

/* Commit the file creation to media */
ioctl (fd, CBIO_TRANS_COMMIT, 0);
```

Using TRFS in Applications

Once TRFS and dosFs are created, the dosFs file system may be used with the ordinary file creation and manipulation commands. No changes to the file system become permanent, however, until TRFS is used to commit them.

It is important to note that the entire dosFs file system—and not individual files—are committed. The entire disk state must therefore be consistent before executing a commit; that is, there must not be a file system operation in progress (by another task, for example) when the file system is committed. If multiple tasks update the file system, care must be taken to ensure the file data is in a known state before setting a transaction point.

There are two ways to commit the file system:

- Using the volume name of the device formatted for TRFS.
- Using a file descriptor which is open on TRFS.

The function **usrTransCommit( )** takes the volume name of the TRFS device and causes it to commit. The function **usrTransCommitFd( )** takes a file descriptor open on TRFS and causes a commit of the entire file system.

TRFS Code Examples

The following code examples illustrate creating a file system with TRFS and setting a transaction point. The first routine creates a new TRFS layer and dosFs file system; and the second sets a transaction point.

```
void createTrfs
    (
    void
    )
    {

    /* Create an XBD RAM disk with 512 byte sized sectors and 1024 sectors.*/
    if (xbdRamDiskDevCreate (512, 1024 * 512, 0, "/trfs") == NULL)
        {
        printf ("Could not create XBD RAM disk\n");
        return;
        }

    /* Put TRFS on the RAM disk */
    /* Use 10% of the disk as overhead */
    if (usrFormatTrans ("/trfs", 100, 0) != OK)
        {
        printf ("Could not format for TRFS\n");
        return;
        }

    /* Now put dosFs on TRFS */
    if (dosFsVolFormat ("/trfs", DOS_OPT_BLANK, 0) != OK)
        {
        printf ("Could not format for dos\n");
        return;
        }


void transTrfs
    (
    void
    )
    {
    /* This assumes a TRFS with DosFs on "/trfs" */

    ... /* Perform file operations here */
    usrTransCommit ("/trfs");

    ... /* Perform more file operations here */
    usrTransCommit ("/trfs");
    }
```

7.8.10 Block Devices

A physical *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, the I/O activity of block device drivers is mediated by the extended block device (XBD) facility and a file system. The XBD facility provides a standard interface for block device drivers on the one hand, and for file systems on the other.

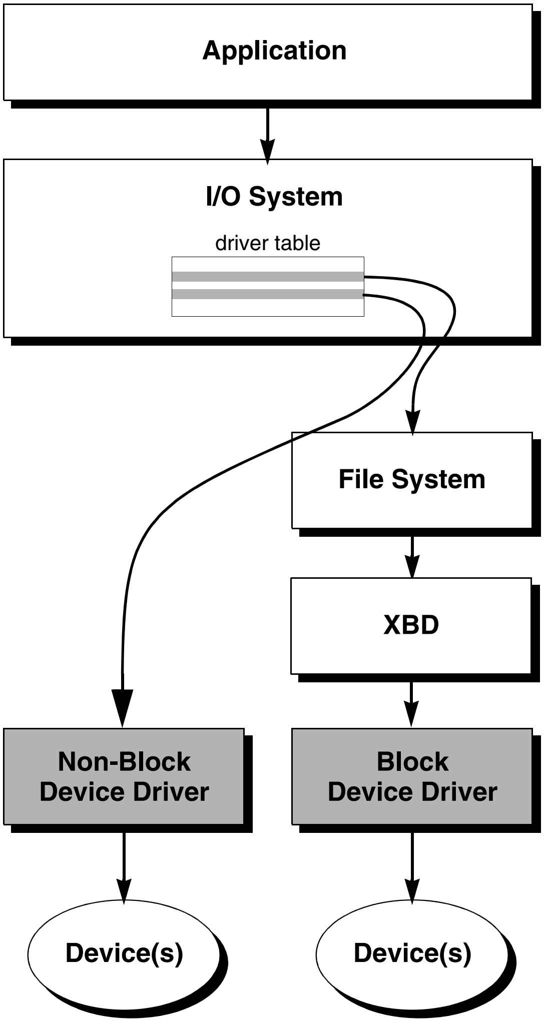
Figure 7-2 shows a layered model of I/O for both block and non-block (character) devices. This architecture allows the same block device driver to be used with different file systems, and reduces the number of I/O functions that must be supported in the driver.

For information about the XBD facility, see *7.8.8 Extended Block Device Facility: XBD*, p.402.

For information about the file systems that can be used with block devices, see *8Local File Systems*, p.451.

For information about information about block device drivers and how to develop them, see the *VxWorks Device Driver Developer's Guide*.

Figure 7-2 **Non-Block Devices and Block Devices**



XBD RAM Disk

A RAM driver emulates a disk device, but keeps all data in memory. The **INCLUDE_XBD_RAMDRV** component allows the use of a file system to access data stored in RAM memory. RAM disks can be created using volatile as well a non-volatile RAM. A RAM disk can be used with the HRFS, dosFs, and rawFs file systems. The RAM disk links into the file system monitor and event framework.

For more about information about RAM disks, see the API reference for **xbdRamDisk**, as well as Example 7-4, Example 8-3, and Example 8-7. For information about compatible file systems, see *8.4 Highly Reliable File System: HRFS*, p. 459, *8.5 MS-DOS-Compatible File System: dosFs*, p. 478, and *8.6 Raw File System: rawFs*, p. 505.

Note that the XBD-compatible RAM disk facility supersedes the **ramDrv** facility.

SCSI Drivers

SCSI is a standard peripheral interface that allows connection with a wide variety of hard disks, optical disks, floppy disks, tape drives, and CD-ROM devices. SCSI block drivers are compatible with the dosFs 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. There are also additional support routines for individual targets in **sysLib.c**.

Configuring SCSI Drivers

Components associated with SCSI drivers are listed in Table 7-14.

Table 7-14 **SCSI and Related Components**

Component	Description
INCLUDE_SCSI	Includes SCSI interface.
INCLUDE_SCSI2	Includes SCSI-2 extensions.
INCLUDE_SCSI_DMA	Enables DMA for SCSI.
INCLUDE_SCSI_BOOT	Allows booting from a SCSI device.
SCSI_AUTO_CONFIG	Auto-configures and locates all targets on a SCSI bus.
INCLUDE_DOSFS	Includes the dosFs file system.
INCLUDE_HRFS	Includes the HRFS file system.
INCLUDE_CDROMFS	Includes CD-ROM file system support.

To include SCSI-1 functionality in VxWorks, use the **INCLUDE_SCSI** component. To include SCSI-2 functionality, you must use **INCLUDE_SCSI2** in addition to **INCLUDE_SCSI**.

Auto-configuration, DMA, and booting from a SCSI device are defined appropriately for each BSP. If you must change these settings, see the VxWorks API reference for **sysScsiConfig( )** and the source file *installDir***/vxworks-6.***x***/target/src/config/usrScsi.c**.

⚠ **CAUTION:** Including SCSI-2 in your VxWorks image can significantly increase the image size.

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 the **INCLUDE_SCSI_BOOT** component is included, larger ROMs may be required for some boards.

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 used (see *Configuring SCSI Drivers*, p.412).

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 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

When VxWorks is built with the **INCLUDE_SCSI** component, the system startup code initializes the SCSI interface by executing **sysScsiInit( )** and **usrScsiConfig( )**. The call to **sysScsiInit( )** initializes the SCSI controller and sets up interrupt handling. The physical device configuration is specified in **usrScsiConfig( )**, which is in *installDir*/**vxworks-6.*x*/target/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( )**
- creation of an XBD block wrapper driver with **xbdBlkDevCreate( )**.

If a recognized file system exists on the SCSI media, it is instantiated automatically when **xbdBlkDevCreate( )** returns. If not, the file system formatter must be called to create the file system. See the **dosFsVolFormat( )** API reference for information about creating a dosFs file system; see the **hrfsFormat( )** API reference for creating an HRFS file system.

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( )**, and **xbdBlkDevCreate( )**.

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 must 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 7-6 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 7-5 is a simple block device configuration setup. Example 7-6 involves selecting special options and demonstrates the use of **scsiTargetOptionsSet( )**. Example 7-7 configures a SCSI device for synchronous data transfer. Example 7-8 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 7-5 **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 ((xbdBlkDevCreate (pSbd0, "/sd0") == NULL) ||
        (xbdBlkDevCreate (pSbd1,"/sd1") == 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.

Example 7-6 **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 7-7 **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 7-8 **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 *xxx***CtrlInitScsi2( )**, 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.

7.9 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. But only in the kernel space.

- **File Descriptors**

  In VxWorks, file descriptors are unique to the kernel and to each process—as in UNIX and Windows. The kernel and each process has its own universe of file descriptors, distinct from each other. When the process is created, its universe of file descriptors is initially populated by duplicating the file descriptors of its creator. (This applies only when the creator is a process. If the creator is a kernel task, only the three standard I/O descriptors 0, 1 and 2 are duplicated.) Thereafter, all open, close, or *dup* activities affect only that process' universe of descriptors.

  In kernel and in each process, file descriptors are global to that entity, meaning that they are accessible by any task running in it.

  In the kernel, however, standard input, standard output, and standard error (0, 1, and 2) can be made task specific.

  For more information see *7.4.1 File Descriptors*, p.365 and *7.4.3 Standard I/O Redirection*, p.367.

- **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 cannot be preempted. In VxWorks, driver routines can be preempted because they execute within the context of the task that invoked them.

7.10 Internal I/O System Structure

The VxWorks I/O system differs from most I/O systems in the way that the work of performing user I/O requests is distributed 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 reading a sequence of bytes from, or writing them 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. 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 provides 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 7-2). 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 typically of the tty/sio type.

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 to block devices. Because block drivers must interact with VxWorks file systems, they use a slightly different organization.

NOTE: This discussion is designed to clarify the structure of VxWorks I/O facilities and to highlight some considerations relevant to writing I/O drivers for VxWorks. For detailed information about writing device drivers, see the *VxWorks Device Driver Developer's Guide*.

Example 7-9 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 7-9 **Hypothetical Driver**

```
/*
 * xxDrv - driver initialization routine
 * xxDrv() init's 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 svced 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)
  ...
```

7.10.1 Drivers

A driver for a non-block device generally implements the seven basic I/O functions—**creat( )**, **remove( )**, **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**—for a particular kind of device. The driver implements these general functions with corresponding device-specific routines that are installed with **iosDrvInstall( )**.

NOTE: Only VxBus-compatible drivers can be used with the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

Not all of the general I/O functions are implemented if they are not supported by a particular device. For example, **remove( )** is usually not supported for devices that are not used with file systems.

If any of the seven basic I/O routines are not implemented by a driver, a null function pointer should be used for the corresponding **iosDrvInstall( )** parameter when the driver is installed. Any call to a routine that is not supported will then fail and return an **ENOTSUP** error.

Drivers may (optionally) allow tasks to wait for activity on multiple file descriptors. This functionality is implemented with the driver's **ioctl( )** routine; see *Implementing select( )*, p.441.

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.

When an application invokes one of the basic I/O functions, the I/O system routes the request to the appropriate routine of a specific driver, as described 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. This routine is typically named *xx***Drv( )**.

- A routine to add devices that are to be serviced by the driver to the I/O system. This routine is typically named *xx***DevCreate( )**.
- Interrupt-level routines that are connected to the interrupts of the devices serviced by the driver.

The Driver Table and Installing Drivers

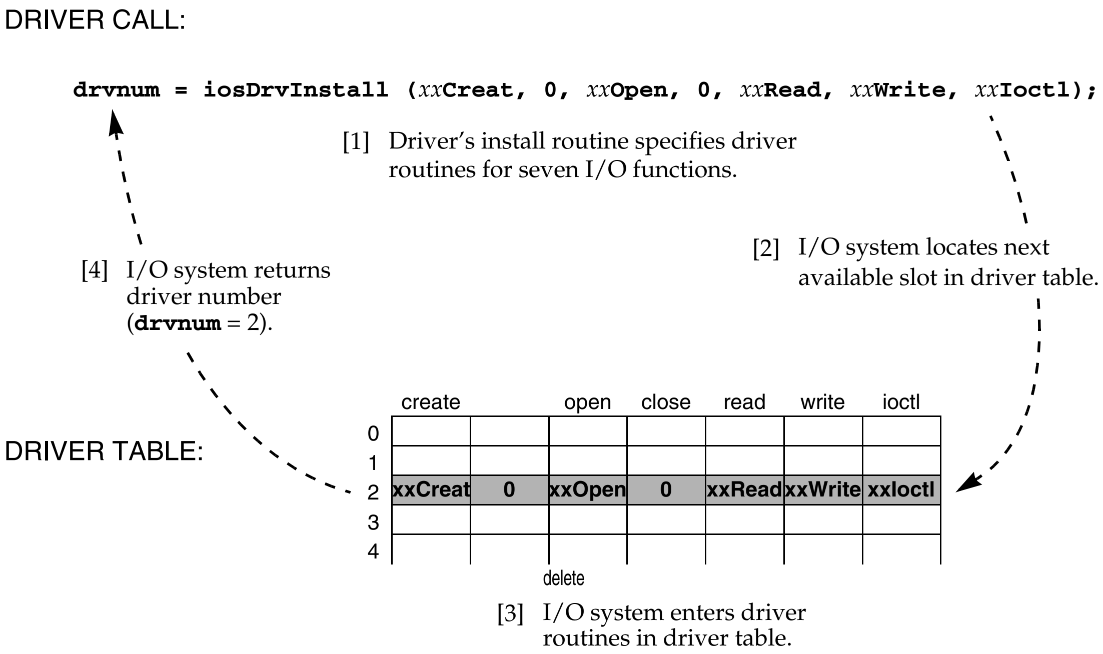
The function of the I/O system is to route user I/O requests to the appropriate routine of the appropriate driver. The I/O system does this by maintaining a table that contains the address of each routine for each driver. Drivers are installed dynamically by calling the I/O system internal routine **iosDrvInstall( )**. 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 any of the seven basic I/O routines that are not supported by a device. For example, **remove( )** is usually not supported for non-file-system devices, and a null is specified for the driver's remove function.

When a user I/O call matches a null driver routine, the call fails and an **ENOTSUP** error is returned.

VxWorks file systems (such as **dosFsLib**) contain their own entries in the driver table, which are created when the file system library is initialized.

Figure 7-3 **Example – Driver Initialization for Non-Block Devices**



```
drvnum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
```

[1] Driver's install routine specifies driver routines for seven I/O functions.

[2] I/O system locates next available slot in driver table.

[3] I/O system enters driver routines in driver table.

[4] I/O system returns driver number (**drvnum** = 2).

	create	delete	open	close	read	write	ioctl
0							
1							
2	xxCreat	0	xxOpen	0	xxRead	xxWrite	xxIoctl
3							
4							

Example of Installing a Driver

Figure 7-3 shows the actions taken by the example driver and by the I/O system when the initialization routine *xx***Drv( )** runs.

The driver calls **iosDrvInstall( )**, specifying the addresses of the driver's routines for the seven basic I/O functions. Then, the I/O system:

1. Locates the next available slot in the driver table, in this case slot 2.
2. Enters the addresses of the driver routines in the driver table.
3. Returns the slot number as the driver number of the newly installed driver.

7.10.2 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—for example, **dosFsDevCreate( )**. The device initialization routine then calls **iosDevAdd( )** automatically.

The routine **iosDevFind( )** can be used to locate the device structure (by obtaining a pointer to the **DEV_HDR**, which is the first member of that structure) and to verify that a device name exists in the table.

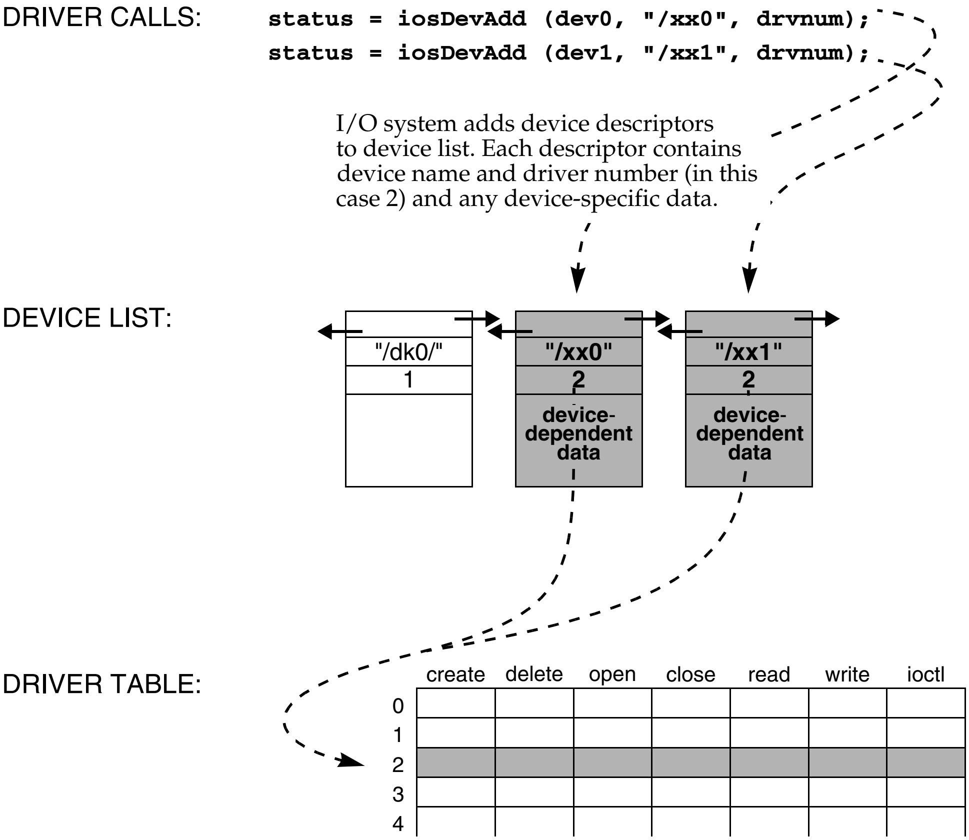
The following is an example using **iosDevFind( )**:

```
char *  pTail;                              /* pointer to tail of devName */
char devName[6] = "DEV1:";                  /* name of device */
DOS_VOLUME_DESC *  pDosVolDesc;             /* first member is DEV_HDR */
    ...
    pDosVolDesc = iosDevFind(devName, (char**)&pTail);
    if (NULL == pDosVolDesc)
        {
        /* ERROR: device name does not exist and no default device */
        }
    else
        {
        /*
         * pDosVolDesc is a valid DEV_HDR pointer
         * and pTail points to beginning of devName.
         * Check devName against pTail to determine if it is
         * the default name or the specified devName.
         */
        }
```

Example of Adding Devices

In Figure 7-4, the example driver's device creation routine *xx***DevCreate( )** adds devices to the I/O system by calling **iosDevAdd( )**.

Figure 7-4 **Example – Addition of Devices to I/O System**


DRIVER CALLS:
status = iosDevAdd (dev0, "/xx0", drvnum);
status = iosDevAdd (dev1, "/xx1", drvnum);
I/O system adds device descriptors to device list. Each descriptor contains device name and driver number (in this case 2) and any device-specific data.
DEVICE LIST:
"/dk0/"
1
"/xx0"
2
device-dependent data
"/xx1"
2
device-dependent data
DRIVER TABLE:
create
delete
open
close
read
write
ioctl
0
1
2
3
4

Deleting Devices

A device can be deleted with **iosDevDelete( )** and the associated driver removed with **iosDrvRemove( )**.

Note that a device-deletion operation causes the file descriptors that are open on the device to be *invalidated*, but not closed. The file descriptors can only be closed by an explicit act on the part of an application. If this were not the case, and file descriptors were closed automatically by the I/O system, the descriptors could be reassigned to new files while they were still being used by an application that was unaware of the deletion of the device. The new files could then be accessed unintentionally by an application attempting to use the files associated with the deleted device, as well as by an application that was correctly using the new files. This would result in I/O errors and possible device data corruption.

Because the file descriptors of a device that has been deleted are invalid, any subsequent I/O calls that use them—except **close( )**—will fail. The behavior of the I/O routines in this regard is as follows:

- **close( )** releases the file descriptor at I/O system level and the driver close routine is not called.
- **read( )**, **write( )**, and **ioctl( )** fail with error **ENXIO** (no such device or address).
- While **open( )**, **remove( )**, and **create( )** do not take an open file descriptor as input, they fail because the device name is no longer in the device list.

Note that even if a device is deleted and immediately added again with the same device name, the file descriptors that were invalidated with the deletion are not restored to valid status. The behavior of the I/O calls on the associated file descriptors is the same as if the device had not been added again.

Applications that are likely to encounter device deletion should be sure to check for **ENXIO** errors from **read( )**, **write( )**, and **ioctl( )** calls, and to then close the relevant file descriptors.

Using Callback Routines to Manage Device Deletion

For situations in which devices are dynamically installed and deleted, the **iosDevDelCallback( )** routine provides the means for calling a post-deletion handler after all driver invocations are exited.

A common use of a device deletion callback is to prevent a race condition that would result from a device descriptor being deleted in one thread of execution while it was still being used in another.

A device descriptor belongs to an application, and the I/O system cannot control its creation and release. It is a user data structure with **DEV_HDR** data structure embedded at the front of it, followed by any specific member of the device. Its pointer can be used to pass into any **iosDev***Xyz***( )** routine as a **DEV_HDR** pointer, or used as the device descriptor for user device handling.

When a device is deleted, an application should not immediately release the device descriptor memory after **iosDevDelete( )** and **iosDrvRemove( )** calls because a driver invocation of the deleted device might still be in process. If the device descriptor is deleted while it is still in use by a driver routine, serious errors could occur.

For example, the following would produce a race condition: task A invokes the driver routine *xyz***Open( )** by a call to **open( )** and the *xyz***Open( )** call does not return before task B deletes the device and releases the device descriptor.

However, if descriptor release is not performed by task B, but by a callback function installed with **iosDevDelCallback( )**, then the release occurs only after task A's invocation of the driver routine has finished.

A device callback routine is called immediately when a device is deleted with **iosDevDelete( )** or **iosDrvRemove( )** as long as no invocations of the associated driver are operative (that is, the device driver reference counter is zero). Otherwise, the callback routine is not executed until the last driver call exits (and the device driver reference counter reaches zero).

A device deletion callback routine should be called with only one parameter, the pointer to the **DEV_HDR** data structure of the device in question. For example:

```
    devDeleteCallback(pDevHdr)
```

The callback should be installed with **iosDevDelCallback( )** after the **iosDevAdd( )** call.

The following code fragments illustrate callback use. The file system device descriptor **pVolDesc** is installed into the I/O device list. Its device deletion callback, **fsVolDescRelease( )** performs the post-deletion processing, including releasing memory allocated for the device volume descriptor.

```
void fsVolDescRelease
   (
   FS_VOLUME_DESC * pVolDesc
   )
   {
   . . . . . .
   free (pVolDesc->pFsemList);
   free (pVolDesc->pFhdlList);
   free (pVolDesc->pFdList);
   . . . . . .
   }


STATUS fsDevCreate
   (
   char *  pDevName,   /* device name */
   device_t  device,   /* underlying block device */
   u_int   maxFiles,   /* max no. of simultaneously open files */
   u_int   devCreateOptions /* write option & volume integrity */
   )
   {
   FS_VOLUME_DESC  *pVolDesc = NULL; /* volume descriptor ptr */
   . . . . . .
   pVolDesc = (FS_VOLUME_DESC *) malloc (sizeof (*pVolDesc));
   pVolDesc->device = device;
   . . . . . .
   if (iosDevAdd((void *)pVolDesc, pDevName, fsDrvNum ) == ERROR)
      {
      pVolDesc->magic = NONE;
      goto error_iosadd;
      }
   /* Device deletion callback installed to release memory resource. */
   iosDevDelCallback((DEV_HDR *) pVolDesc, (FUNCPTR) fsVolDescRelease);
   . . . . . .
   }

STATUS fsDevDelete
   (
   FS_VOLUME_DESC *pVolDesc    /* pointer to volume descriptor */
   )
   {
   . . . . . .
   /*
    * Delete the file system device from I/O device list. Callback
    * fsVolDescRelease will be called from now on at a
    * safe time by I/O system.
    */
   iosDevDelete((DEV_HDR *) pVolDesc);
   . . . . . .
   }
```

The application should check the error returned by a deleted device, as follows:

```
if (write (fd, (char *)buffer, nbytes) == ERROR)
   {
   if (errno == ENXIO)
       {
       /* Device is deleted. fd must be closed by application. */
       close(fd);
       }
   else
       {
       /* write failure due to other reason. Do some error dealing. */
       . . . . . .
       }
       }
```

7.10.3 File Descriptors

Several file descriptors can be open to a single device at one time. A device driver can maintain additional information associated with a file descriptor 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 file descriptor. You can also have several file descriptors open to a non-block device, such as a *tty*; typically there is no additional information, and thus writing on any of the file descriptors produces identical results.

File Descriptor Table

Files are opened with **open( )** or **creat( )**. The I/O system searches the device list for a device name that matches the filename (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 *file descriptor 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 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 file descriptor in the application-level I/O call.

Example of Opening a File

In Figure 7-5 and Figure 7-6, a user calls **open( )** to open the file */xx***0**. The I/O system takes the following series of actions:

1. It searches the device list for a device name that matches the specified filename (or an initial substring). In this case, a complete device name matches.

2. It reserves a slot in the file descriptor table and creates a new file descriptor object, which is used if the open is successful.

3. It then looks up the address of the driver's open routine, *xx***Open( )**, and calls that routine. Note that the arguments to *xx***Open( )** are transformed by the I/O system from the user's original arguments to **open( )**. The first argument to *xx***Open( )** is a pointer to the device descriptor the I/O system located in the full filename search. The next parameter is the *remainder* of the filename specified by the user, after removing the initial substring that matched the device name. In this case, because the device name matched the entire filename, 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 third parameter is the file access flag, in this case **O_RDONLY**; that is, the file is opened for reading only. The last parameter is the mode, as passed to the original **open( )** routine.

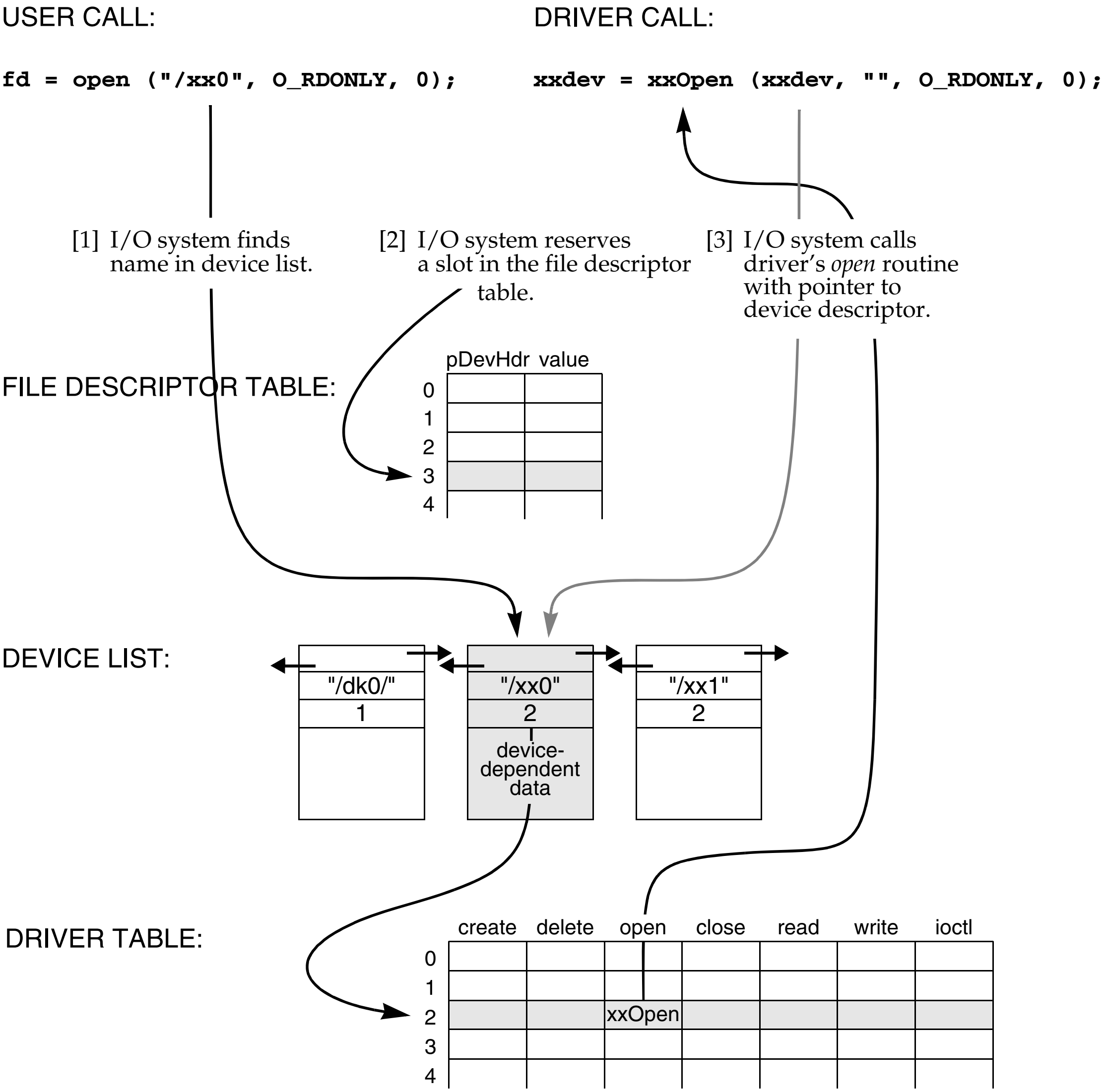
4. It executes *xx***Open( )**, which returns a value that subsequently identifies the newly opened file. In this case, the value is the pointer to the device descriptor. This value is supplied to the driver in subsequent I/O calls that refer to the file being opened. Note that if the driver returns 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 *xx***Open( )** in the new file descriptor object.

   Again, the value entered in the file descriptor object 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 file descriptor table, in this case 3.

Figure 7-5 **Example: Call to I/O Routine open( ) [Part 1]**



USER CALL:

```
fd = open ("/xx0", O_RDONLY, 0);
```

DRIVER CALL:

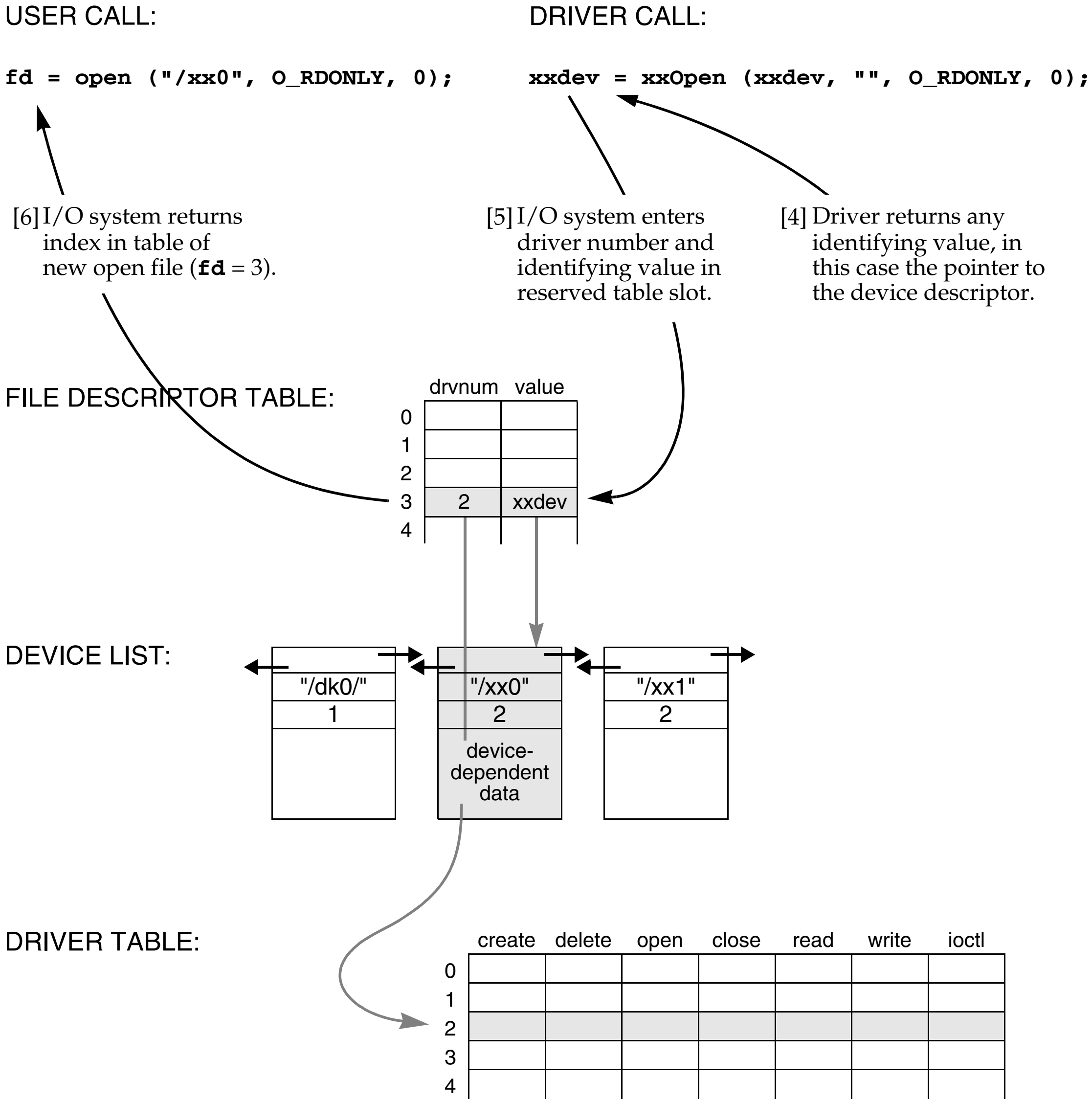
```
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
```

[1] I/O system finds name in device list.

[2] I/O system reserves a slot in the file descriptor table.

[3] I/O system calls driver's *open* routine with pointer to device descriptor.

Figure 7-6 **Example: Call to I/O Routine open( ) [Part 2]**



USER CALL:

```
fd = open ("/xx0", O_RDONLY, 0);
```

DRIVER CALL:

```
xxdev = xxOpen (xxdev, "", O_RDONLY, 0);
```

[6] I/O system returns index in table of new open file (**fd** = 3).

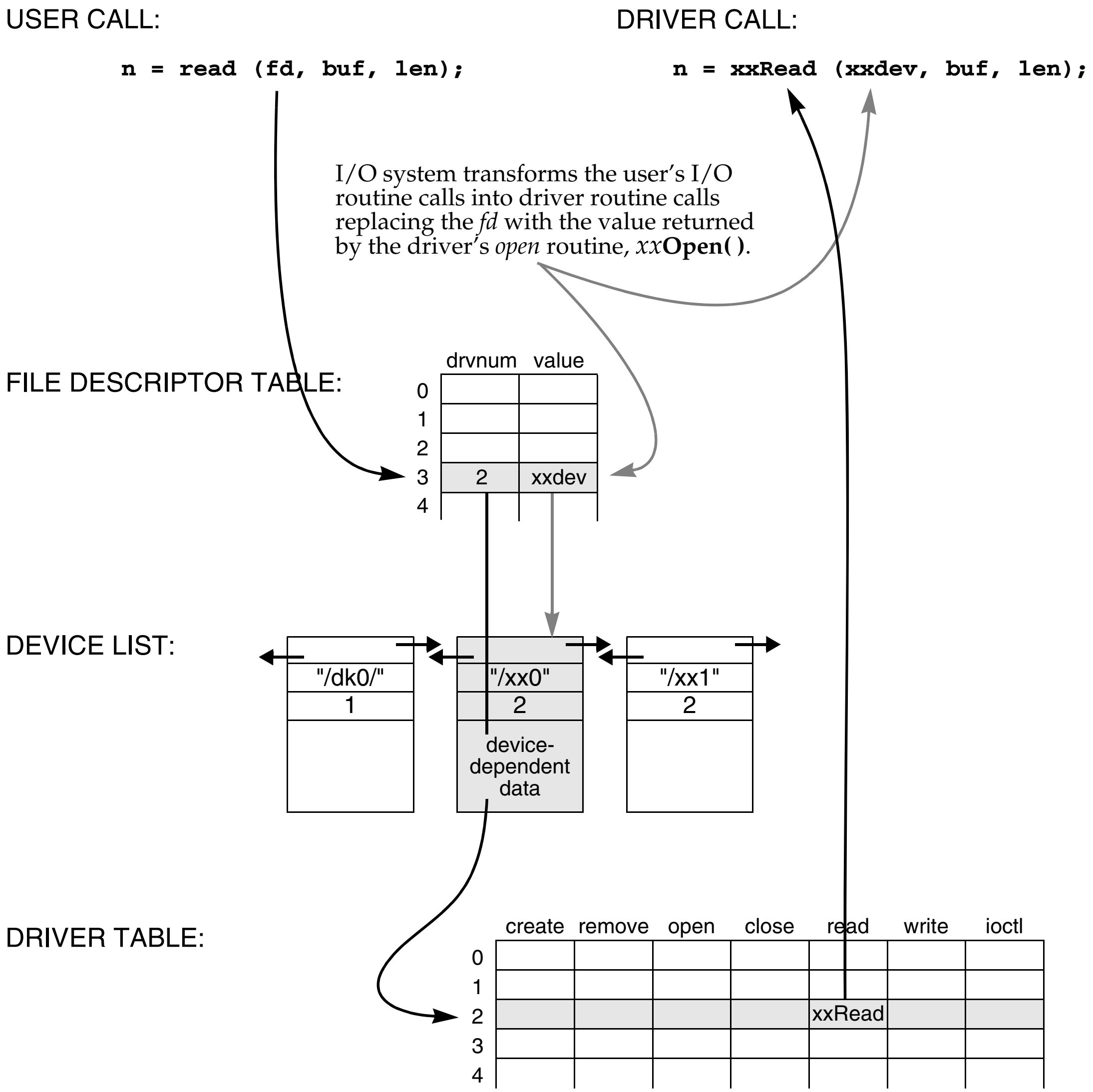
[5] I/O system enters driver number and identifying value in reserved table slot.

[4] Driver returns any identifying value, in this case the pointer to the device descriptor.

Example of Reading Data from the File

In Figure 7-7, the user calls **read( )** to obtain input data from the file. The specified file descriptor is the index into the file descriptor table for this file. The I/O system uses the driver number contained in the table to locate the driver's read routine, *xx***Read( )**. The I/O system calls *xx***Read( )**, passing it the identifying value in the file descriptor table that was returned by the driver's open routine, *xx***Open( )**. 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.

Figure 7-7 **Example: Call to I/O Routine read( )**



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 file descriptor 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 file descriptor table as being available. Any subsequent references to that file descriptor cause an error. 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 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 *xx***DevCreate( )** 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 *xx***Read( )** 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 *xx***Write( )** routine might call **selWakeupAll( )** to unblock all the tasks waiting to read, now that there is data in the pipe.

Example 7-10 **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.
 */

/* myDrvLib.h - header file for driver */

typedef struct     /* MY_DEV */
    {
    DEV_HDR     devHdr;                 /* device header */
    BOOL       myDrvDataAvailable;      /* data is available to read */
    BOOL       myDrvRdyForWriting;      /* device is ready to write */
    SEL_WAKEUP_LIST selWakeupList;      /* list of tasks pended in select */
    } MY_DEV;
```

```
/* myDrv.c - code fragments for supporting select() in a driver */

#include <vxWorks.h>
#include <selectLib.h>

/* First create and initialize the device */

STATUS myDrvDevCreate
    (
    char *  name,                      /* name of device to create */
    )

    {
    MY_DEV * pMyDrvDev;                /* pointer to device descriptor*/
    ... additional driver code ...

    /* allocate memory for MY_DEV */
    pMyDrvDev = (MY_DEV *) malloc (sizeof MY_DEV);
    ... additional driver code ...

    /* initialize MY_DEV */
    pMyDrvDev->myDrvDataAvailable=FALSE
    pMyDrvDev->myDrvRdyForWriting=FALSE

    /* initialize wakeup list */
    selWakeupListInit (&pMyDrvDev->selWakeupList);
    ... additional driver code ...
    }

/* ioctl function to request reading or writing */

STATUS myDrvIoctl
    (
    MY_DEV * pMyDrvDev,                /* pointer to device descriptor */
    int          request,                  /* ioctl function */
    int          arg                       /* where to send answer */
    )
    {
    ... additional driver code ...

    switch (request)
        {
        ... additional driver code ...

        case FIOSELECT:

            /* add node to wakeup list */

            selNodeAdd (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELREAD
                && pMyDrvDev->myDrvDataAvailable)
                {
                /* data available, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELWRITE
                && pMyDrvDev->myDrvRdyForWriting)
                {
                /* device ready for writing, make sure task does not pend */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            break;

        case FIOUNSELECT:

            /* delete node from wakeup list */
            selNodeDelete (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);
            break;

            ... additional driver code ...
        }
    }

/* code that actually uses the select() function to read or write */

void myDrvIsr
    (
    MY_DEV * pMyDrvDev;
    )
    {
    ... additional driver code ...

    /* if there is data available to read, wake up all pending tasks */

    if (pMyDrvDev->myDrvDataAvailable)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELREAD);
```

```
/* if the device is ready to write, wake up all pending tasks */

if (pMyDrvDev->myDrvRdyForWriting)
    selWakeupAll (&pMyDrvDev->selWakeupList, SELWRITE);
}
```

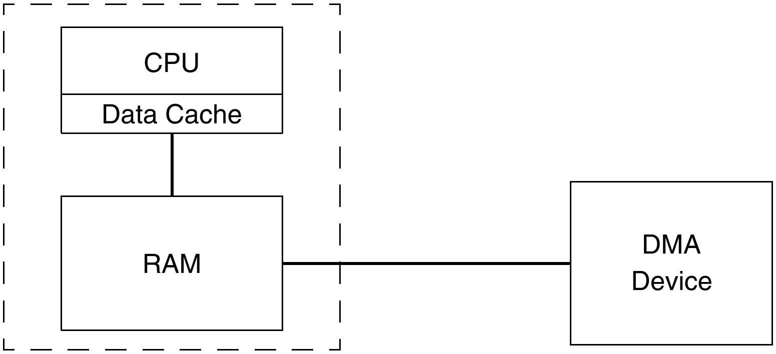
Cache Coherency

NOTE: The cache facilities described in this section are provided for the uniprocessor (UP) configuration of VxWorks, some of which are not appropriate—and not provided—for the symmetric multiprocessor (SMP) configuration. For more information in this regard, see *cacheLib Restrictions*, p.708. For general information about VxWorks SMP and about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

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 7-8 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 7-8 **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 incoherence 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 incoherence 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 7-11 **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).

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 must use the virtual address.

The device driver should use **CACHE_DMA_VIRT_TO_PHYS** to translate a virtual address to a physical address before passing it to the device. It may also use **CACHE_DMA_PHYS_TO_VIRT** to translate a physical address to a virtual one, but this process is time-consuming and non-deterministic, and should be avoided whenever possible.

Example 7-12 **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);
    }
```

7.11 PCMCIA

A PCMCIA card can be plugged into notebook computers to connect devices such as modems and external hard drives.[2] VxWorks provides PCMCIA facilities for **pcPentium, pcPentium2**, and **pcPentium3** BSPs and PCMCIA drivers that allow VxWorks running on these targets to support PCMCIA hardware.

PCMCIA support is at the PCMCIA Release 2.1 level. It does not include socket services or card services, which are not required by VxWorks. It does include chip drivers and libraries. The PCMCIA libraries and drivers are also available in source code form for VxWorks systems based on CPU architectures other than Intel Pentium.

To include PCMCIA support in your system, configure VxWorks with the **INCLUDE_PCMCIA** component. For information about PCMCIA facilities, see the API references for **pcmciaLib** and **pcmciaShow**.

7.12 Peripheral Component Interconnect: PCI

Peripheral Component Interconnect (PCI) is a bus standard for connecting peripherals to a PC, and is used in Pentium systems, among others. PCI includes buffers that de-couple the CPU from relatively slow peripherals, allowing them to operate asynchronously.

For information about PCI facilities, see the API references for **pciAutoConfigLib**, **pciConfigLib**, **pciInitLib**, and **pciConfigShow**.

[2] PCMCIA stands for Personal Computer Memory Card International Association, and refers to both the association and the standards that it has developed.

8
Local File Systems

8.1 Introduction

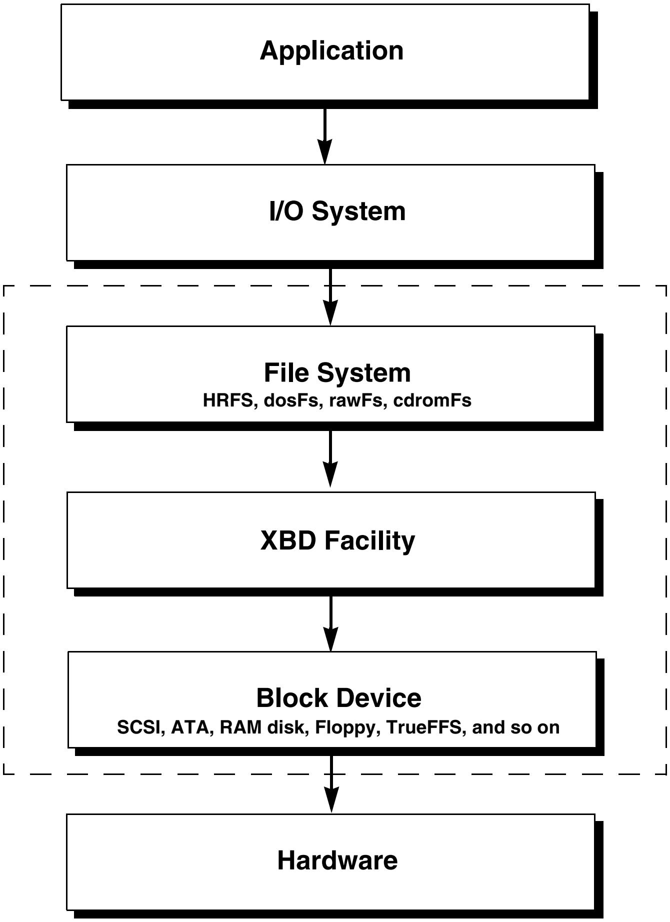
VxWorks provides a variety of file systems that are suitable for different types of applications. The file systems can be used simultaneously, and in most cases in multiple instances, for a single VxWorks system.

Most VxWorks file systems rely on the extended block device (XBD) facility for a a standard I/O interface between the file system and device drivers. This standard interface allows you to write your own file system for VxWorks, and freely mix file systems and device drivers.

File systems used for removable devices make use of the file system monitor for automatic detection of device insertion and instantiation of the appropriate file system on the device.

The relationship between applications, file systems, I/O facilities, device drivers and hardware devices is illustrated in Figure 8-1. Note that this illustration is relevant for the HRFS, dosFs, rawFs, and cdromFs file systems. The dotted line indicates the elements that must be configured and instantiated to create a specific, functional run-time file system.

Figure 8-1 **File Systems in a VxWorks System**


Application
I/O System
File System
HRFS, dosFs, rawFs, cdromFs
XBD Facility
Block Device
SCSI, ATA, RAM disk, Floppy, TrueFFS, and so on
Hardware

For information about the XBD facility, see *7.8.8 Extended Block Device Facility: XBD*, p.402.

This chapter discusses the file system monitor and the following VxWorks file systems, describing how they are organized, configured, and used:

- **VRFS**

  A virtual root file system for use with applications that require a POSIX root file system. The VRFS is simply a root directory from which other file systems and devices can be accessed. See *8.3 Virtual Root File System: VRFS*, p.457.

- **HRFS**

  A POSIX-compliant transactional file system designed for real-time use of block devices (disks). Can be used on flash memory in conjunction with TrueFFS and the XBD block wrapper component. See *8.4 Highly Reliable File System: HRFS*, p.459.

- **dosFs**

  An MS-DOS compatible file system designed for real-time use of block devices. Can be used with flash memory in conjunction with the TrueFFS and the XBD block wrapper component. Can also be used with the transaction-based reliable file system (TRFS) facility. See *8.5 MS-DOS-Compatible File System: dosFs*, p.478.

- **rawFS**

  Provides a simple raw file system that treats an entire disk as a single large file. See *8.6 Raw File System: rawFs*, p.505.

- **cdromFs**

  Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system. See *8.7 CD-ROM File System: cdromFs*, p.510.

- **ROMFS**

  Designed for bundling applications and other files with a VxWorks system image. No storage media is required beyond that used for the VxWorks boot image. See *8.8 Read-Only Memory File System: ROMFS*, p.516.

- **TSFS**

  Uses the host target server to provide the target with access to files on the host system. See *8.9 Target Server File System: TSFS*, p.518.

For information about the XBD facility, see *7.8.8 Extended Block Device Facility: XBD*, p.402).

File Systems and Flash Memory

VxWorks can be configured with file-system support for flash memory devices using TrueFFS and the HRFS or dosFs file system. For more information, see *8.5 MS-DOS-Compatible File System: dosFs*, p. 478 and *10. Flash File System Support: TrueFFS*.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the *VxWorks Application Programmer's Guide: Local File Systems*.

8.2 File System Monitor

The file system monitor provides for automatic detection of device insertion, and instantiation of the appropriate file system on the device. The monitor is required for all file systems that are used with the extended block device (XBD) facility. It is provided with the **INCLUDE_FS_MONITOR** component.

The file systems that require both the XBD and the file system monitor components are HRFS, dosFs, rawFs, and cdromFs.

The process by which devices are detected, and file systems created, is as follows:

1. When file systems are initialized (at boot time), they register probe routines and instantiation routines with the file system monitor.

2. When a device is detected or inserted (for example, when a driver is initialized, or media is inserted into an existing device—such as a floppy disk into a floppy drive) the block device associated with it generates a primary insertion event. (See *Device Insertion Events*, p. 456.)

3. In response to the primary insertion event, the file system monitor creates an XBD partition manager if the device can support partitions. (For information about the partition manager, see *XBD Disk Partition Manager*, p. 403.)

4. If the partition manager finds partitions on the physical device, it creates a device for each partition; and whether or not partitions are found, the manager generates a secondary insertion event.

5. When the file system monitor receives a secondary event, all the registered file system probe functions are run.

6. When a file system's probe routine returns success, that file system's instantiation routine is executed. If none of the probes are successful, or if the file system instantiation routine fails, a rawFs file system is created on the device by default.

When a device is removed, the following occurs:

1. The block device detects the removal of the hardware device associated with it and generates a removal event.
2. The block device removes itself, freeing all its resources.
3. The file system associated with the block device removes itself from core I/O, invalidating its file handles.
4. The file system removes itself, freeing all its resources.

Device Insertion Events

The types of device insertion events to which the file system monitor responds are described in more detail below.

XBD Primary Insertion Event

An XBD-compliant block device generates a primary insertion event when media that can support partitions is inserted (that is, if a partition table is found). In response, the file system monitor creates a partition manager, which in turn generates secondary insertion events for each partition that it finds on the media (see below).

Note that for block devices used with the XBD wrapper component (**INCLUDE_XBD_BLK_DEV**), a primary insertion event is always generated, regardless of the media. The wrapper element is essentially hardware agnostic; it cannot know if the device might include partitions. For example, the device could be a hard disk—for which partitions are expected—or it could be a floppy device.

Note also that a RAM disk device can generate a primary insertion event, depending on the parameters used when it was created (see *XBD RAM Disk*, p.411 and the API reference for **XbdRamDisk**).

XBD Secondary Insertion Event

A secondary insertion event is generated by either by a block device whose media does not support partitions, or by an XBD partition manager. The secondary event signals the file system manager to run the probe routines that identify the file system on the device. If a probe routine returns **OK**, the associated file system creation routine is executed. If none of the probe routines identifies a file system, or if a file system creation routine fails, the rawFs file system is created by default.

XBD Soft Insert Event

Unlike the other events, an XBD soft insert event is produced by application directive rather than by physical media being swapped. When **ioctl( )** is called with the **XBD_SOFT_EJECT** control function it tells the file system manager that the current file system has been removed, and that a rawFs file system should be created. This call causes the system to bypass the usual file system detection operations, and ensures that rawFs is instantiated in place of the current file system.

XBD Name Mapping Facility

The file system monitor name mapping facility allows XBD names to be mapped to a more suitable name. It's primary use is for the partition manager which appends **:x** to the base **xbd** name when it detects a partition. By using the **fsm** name facility you can map the partition names to something more useful. For example, the floppy drive configlette uses the name component to map the supplied floppy name plus the **:0** the partition manager will add to **/fd***x*. Where *x* represents the floppy drive number. If this was not done one would see the default device names in the list generated by the **devs** shell command. For more information see the API references for **fsmNameInstall( )**, **fsmNameMap( )**, and **fsmNameUninstall( )**; also see Example 8-2.

8.3 Virtual Root File System: VRFS

VxWorks provides a virtual root file system (VRFS) for use with applications that require a POSIX root file system. The VRFS is simply a "/" or root directory from which other file systems and devices can be accessed. VRFS is not a true file system, as files and directories cannot be created with the sorts of commands that are ordinarily associated with file systems, and it is read-only.

Only devices whose names begin with a single leading forward slash—and which do not contain any other forward slash characters—are recognized by the VRFS.

To include the VRFS in VxWorks, configure the kernel with the **INCLUDE_VRFS** component. The VRFS is created and mounted automatically if the component is included in VxWorks.

This shell session illustrates the relationship between device names and access to devices and file systems with the VRFS.

```
-> devs
drv name
  0 /null
  1 /tyCo/0
  1 /tyCo/1
  2 /aioPipe/0x1817040
  6 /romfs
  7 /
  9 yow-build02-lx:
 10 /vio
 11 /shm
 12 /ram0
value = 25 = 0x19

-> cd "/"
value = 0 = 0x0
-> ll
?---------  0 0        0                0 Jan  1 00:00 null
drwxrwxr-x  0 15179    100             20 Jan 23  2098 romfs/
?---------  0 0        0                0 Jan  1 00:00 vio
drwxrwxrwx  1 0        0                0 Jan  1 00:00 shm/
drwxrwxrwx  1 0        0             2048 Jan  1 00:00 ram0/
value = 0 = 0x0
```

Note that **/tyCo/0**, **/tyCo/1**, **/aioPipe/0x1817040** and **yow-build02-lx** do not show up in the directory listing of the root directory as they do not follow the naming convention required by the VRFS. The first three include forward slashes in the body of the device name and the fourth does not have a leading forward slash in its name.

Also note that the listings of file systems have a trailing forward slash character. Other devices do not, and they have a question mark in the permissions (or attributes) column of the listing because they do not have recognizable file permissions.

NOTE: Configuring VxWorks with support for POSIX PSE52 conformance (using **BUNDLE_RTP_POSIX_PSE52**) provides the **/dev/null** device. Note that the **devs** shell command lists **/dev/null** with other devices, but the **ls** command does not list **/dev/null** under the VRFS root directory (because the name violates the VRFS naming scheme). Applications can, in any case, use **/dev/null** as required. For information about null devices, see *7.8.6 Null Devices*, p.402. For information about POSIX PSE52, see the *VxWorks Application Programmer's Guide: POSIX Facilities*.

CAUTION: VRFS alters the behavior of other file systems because it provides a root directory on VxWorks. Changing directory to an absolute path on a host file system will not work when VRFS is installed without preceding the absolute path with the VxWorks device name. For example, if the current working directory is **hostname**, changing directory to **/home/panloki** will not work— it must be named **hostname:/home/panloki**.

8.4 Highly Reliable File System: HRFS

The Highly Reliable File System (HRFS) is a transactional file system for real-time systems. The primary features of the file system are:

- Fault tolerance. The file system is never in an inconsistent state, and is therefore able to recover quickly from unexpected loses of power.
- Configurable commit policies.
- Hierarchical file and directory system, allowing for efficient organization of files on a volume.
- Compatibility with a widely available storage devices.
- POSIX compliance.

For more information about the HRFS libraries see the VxWorks API references for **hrfsFormatLib**, **hrFsLib**, and **hrfsChkDskLib**.

HRFS and Flash Memory

For information about using HRFS with flash memory, see *10. Flash File System Support: TrueFFS*.

8.4.1 Configuring VxWorks for HRFS

To include HRFS support in VxWorks, configure the kernel with the appropriate required and optional components.

Required Components

The components required for HRFS are the following:

- Either the **INCLUDE_HRFS** or the **INCLUDE_HRFS_READONLY** component is required. As its name indicates, the latter is a read-only version of the main HRFS component. The libraries it provides are smaller as it provides no facilities for disk modifications.
- The appropriate component for your block device; for example, **INCLUDE_ATA**.
- If you are using a device driver that is not designed for use with the XBD facility, you must use the **INCLUDE_XBD_BLK_DEV** wrapper component in addition to **INCLUDE_XBD**.

  See *XBD Block Device Wrapper*, p. 404 for more information.

Optional HRFS Components

The **INCLUDE_HRFS_FORMAT** (formatter), **INCLUDE_HRFS_CHKDSK** (consistency checker), and **INCLUDE_HRFS_ACCESS_TIMESTAMP** (access file stamp) components are optional. Note that the **INCLUDE_HRFS_ACCESS_TIMESTAMP** component is included in the **BUNDLE_RTP_POSIX_PSE52** component bundle.

Optional XBD Components

The **INCLUDE_XBD_PART_LIB (**disk partitioning) and **INCLUDE_XBD_RAMDRV** (RAM disk) components are optional.

For information about the XBD facility, see *7.8.8 Extended Block Device Facility: XBD*, p. 402).

8.4.2 Configuring HRFS

HRFS provides the following component configuration parameters:

HRFS_DEFAULT_MAX_BUFFERS
Defines how many buffers HRFS uses for its caching mechanism. HRFS needs a minimum of 6 buffers. The default setting is 16. This parameter applies to all HRFS volumes. Note that while increasing the number of buffers can increase performance, it does so at the expense of heap memory.

HRFS_DEFAULT_MAX_FILES
Defines how many files can be simultaneously open on an HRFS volume. The minimum is 1. The default setting is 10. Note that is not the same as the maximum number of file descriptors.

HRFS_DEFAULT_COMMIT_POLICY
Defines the default commit policy for an HRFS volume, which is **FS_COMMIT_AUTO**. Commit policies can also be changed at runtime. For more information see *8.4.5 Transactional Operations and Commit Policies*, p.469 and *8.4.6 Configuring Transaction Points at Runtime*, p.471.

HRFS_DEFAULT_COMMIT_PERIOD
Defines the initial commit period of an HRFS volume if it has been configured for periodic commits. This parameter is measured in milliseconds. The default value is 5000 milliseconds (5 seconds). The commit period can also be changed at runtime. For more information see *8.4.5 Transactional Operations and Commit Policies*, p.469 and *8.4.6 Configuring Transaction Points at Runtime*, p.471.

8.4.3 Creating an HRFS File System

This section describes the process of creating an HRFS file system. It first provides a summary overview and then a detailed description of each step. See *8.4.4 HRFS, ATA, and RAM Disk Examples*, p.463 for examples of the steps and code examples.

Overview of HRFS File System Creation

For information operating system configuration, see *8.4.1 Configuring VxWorks for HRFS*, p.459. Note that the file system is initialized automatically at boot time.

The steps involved in creating an HRFS file system are as follows:

1. If you are using a custom driver, create the appropriate block device. See *Step 1:Create a Block Device*, p.462.

   If you are using a standard VxWorks component for the device, it is created automatically.

2. If you are using a device driver that is not XBD-compliant, create an XBD device wrapper. See *Step 2:Create an XBD Device Wrapper*, p.462 (Also see *XBD Block Device Wrapper*, p.404.).

3. Optionally, create and mount partitions. See *Step 3:Create Partitions*, p.463.

4. If you are not using pre-formatted disks, format the volumes. See *Step 4:Formatting the Volume*, p.463.

HRFS File System Creation Steps

Before any other operations can be performed, the HRFS file system library, **hrFsLib**, must be initialized. This happens automatically at boot time, triggered by the required HRFS components that were included in the system.

Initializing HRFS involves the creation of a vnode layer. HRFS installs an number of internal vnode operators into the this layer. The vnode layer invokes **iosDrvInstall( )** when media is detected, which adds the driver to the I/O driver table. The driver number assigned to vnodes—and therefore HRFS—is recorded in a global variable, **vnodeAffDriverNumber**. The table specifies entry points for the vnode file operations that are accessing devices using HRFS.

Step 1: Create a Block Device

If you are using a standard VxWorks component for the device, it is created automatically.

If you are using a custom driver, create the appropriate block device by calling the creation routine for the device driver. The format for this routine is *xxx***DevCreate( )** where *xxx* represents the device driver type; for example, **ataDevCreate( )**.

The driver routine returns a pointer to a block device descriptor structure, **BLK_DEV**. This structure describes the physical attributes of the device and specifies the routines that the device driver provides. The pointer returned is used to create an XBD device wrapper in the next step.

Step 2: Create an XBD Device Wrapper

If you are using a device driver that is not XBD-compliant, it requires an XBD device wrapper.

The wrapper is created automatically if you have configured VxWorks with the **INCLUDE_XBD_BLK_DEV** wrapper component (See *XBD Block Device Wrapper*, p.404). Otherwise, create a wrapper for each block device using **xbdBlkDevCreate( )**.

After the XBD device wrapper is created, the physical device is automatically probed for a file system and partitions. If a disk is already formatted, the disk is mounted. If a file system is found, it is mounted as well. If file system is not HRFS, it must be formatted (see below).

Step 3: Create Partitions

If you have included the **INCLUDE_XBD_PART_LIB** component in your system, you can create partitions on a disk and mount volumes atop the partitions. Use the **xbdCreatePartition( )** routine to create partitions.

This step should only be performed once, when the disk is first initialized. If partitions are already written to the disk, this step should not be performed as it destroys data.

Step 4: Formatting the Volume

If you are using unformatted disk or wish to replace the current file system on the disk, format the disk by calling **hrFsFormat( )**. For more information, see the VxWorks API reference for this routine.

⚠ **CAUTION:** Reformatting a disk destroys any data that may be on it.

8.4.4 HRFS, ATA, and RAM Disk Examples

This section provides examples of the steps discussed in the preceding section. They are meant to be relatively generic, and illustrate the following:

- Creating and working with an HRFS file system on an ATA disk with commands from the shell.
- Code that creates and formats partitions.
- Code that creates and formats a RAM disk volume.

⚠ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name; unexpected results may otherwise occur.

Example 8-1 **Create HRFS in Partitions on an ATA Disk**

This example demonstrates how to initialize an ATA disk with HRFS on two partitions from the shell. While these steps use an ATA device, they are applicable to other block devices.

1. If you are using a custom driver, create an ATA block device that controls the master ATA hard disk (drive zero) on the primary ATA controller (controller zero). This device uses the entire disk.

   ```
   -> xbd = ataXbdDevCreate(0,0,0,0,"/ata")
   New symbol "xbd" added to kernel symbol table.
   Instantiating /ata:0 as rawFs
   xbd = 0xca4fe0: value = 262145 = 0x40001
   ```

   The **xbd** variable is of type **device_t**. A value of zero would indicate an error in the **ataXbdDevCreate( )** call, which usually indicates a BSP configuration or hardware configuration error.

   If you are using the standard **INCLUDE_ATA** device component, the block device is created automatically. Note that in this case the default device name (provided by the component) is **/ata0a**.

2. Display information about devices.

   ```
   -> devs
   drv name
     0 /null
     1 /tyCo/0
     1 /tyCo/1
     8 yow-grand:
     9 /vio
     4 /ata:0
   value = 25 = 0x19
   ```

3. The new ata driver **/ata:0** is listed. The zero in the name indicates that no partitions were detected. Note that if no file system is detected on the device, the rawFs file system is instantiated automatically and appears in the device list. Prepare the disk for first use. Create two partitions on this disk device, specifying 50% of the disk space for the second partition, leaving 50% for the first partition. This step should only be performed once, when the disk is first initialized. If partitions are already written to the disk, this step should not be performed since it destroys any data on the disk.

   ```
   -> xbdCreatePartition ("/ata:0", 2, 50, 0, 0)
   value = 0 = 0x0
   ```

   The four arguments are to **xbdCreatePartition( )** are:

   - the drive name
   - the number of partitions
   - the percent of disk to use for the second partition
   - the percent of disk to use for the third partition
   - the percent of disk to use for the fourth partition

   The remainder of the disk is used for the first partition.

4. Then list the devices to display information about the new partitions.

```
-> devs
drv name
  0 /null
  1 /tyCo/0
  1 /tyCo/1
  8 yow-grand:
  9 /vio
  3 /ata:1
  3 /ata:2
```

Note that **/ata:0** does not appear in this list, and two new devices, **/ata:1** and **/ata:2**, have been added to represent the new partitions. Each volume has rawfs instantiated in it as they are new and unformatted.

5. Format the volumes for HRFS. This step need only be done once, when the volumes are first created. If the volumes have already been formatted, then omit this step. This example formats the file system volumes with default options.

```
-> hrfsFormat ("/ata:1", 0ll, 0, 0)
Formatting /ata:1 for HRFS
Instantiating /ata:1 as rawFs
Formatting...OK.
value = 0 = 0x0

-> hrfsFormat ("/ata:2", 0ll, 0, 0)
Formatting /ata:2 for HRFS
Instantiating /ata:2 as rawFs
Formatting...OK.
value = 0 = 0x0
```

Note that in the **hrfsFormat( )** call, the **ll** (two lower-case L letters) used with the second parameter is required to indicate to the shell that the data type is **long long.**

For more information, see the API reference for **hrFsFormatLib**.

6. Display information about the HRFS volumes.

```
-> ll "/ata:1"

Listing Directory /ata:1:
drwxrwxrwx  1 0       0             8192 Jan  1 00:13 ./
drwxrwxrwx  1 0       0             8192 Jan  1 00:13 ../
value = 0 = 0x0

-> ll "/ata:2"

Listing Directory /ata:2:
drwxrwxrwx  1 0       0             8192 Jan  1 00:13 ./
drwxrwxrwx  1 0       0             8192 Jan  1 00:13 ../
value = 0 = 0x0
```

If you are working with an ATA hard disk or a CD-ROM file system from an ATAPI CD-ROM drive, you can, alternatively, use **usrAtaConfig( )**. This routine processes several steps at once. For more information, see the API reference.

Example 8-2 **Creating and Partitioning a Disk and Creating Volumes**

This code takes the name of a block device that you have already instantiated, creates three partitions, creates the partition handler for these partitions, and creates the HRFS device handler for them. Then it formats the partitions using **hrfsFormat( )**.

```
STATUS usrPartDiskFsInit
    (
    char * xbdName /* device name used during creation of XBD */
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    devname_t xbdPartName;
    int i;


    /* Map partition names */

    for (i = 1; i <= 3; i++)
        {
        sprintf (xbdPartName, "%s:d", devNames[i],i);
        fsmNameInstall (devNames[i-1], xbdPartName);
        }

    /* create partitions */

    if((xbdCreatePartition (xbdName,3,50,45)) == ERROR)
        return ERROR;

    /* Formatting the first partition */

    if(hrfsFormat (devNames[0], 0ll,0, 0) == ERROR)
        return ERROR;

    /* Formatting the second partition */

    if(hrfsFormat (devNames[1], 0ll, 0, 0) == ERROR)
        return ERROR;

    /* Formatting the third partition */

    if(hrfsFormat (devNames[2], 0ll, 0, 0) == ERROR)
        return ERROR;

    return OK;
    }
```

Note that in most cases you would be likely to format the different partitions for different file systems.

Example 8-3 **Creating and Formatting a RAM Disk Volume and Performing File I/O**

The following code creates a RAM disk, formats it for use with the HRFS file system, and performs file system operations.

```
#include <vxWorks.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <hrFsLib.h>
#include <xbdPartition.h>
#include <xbdRamDisk.h>

#define  DEVNAME "/myram" /* name of the RAM disk */
#define  BLOCKSIZE  512
#define  DISKSIZE  (BLOCKSIZE * 2000)

STATUS hrfsSetup
    (
    void
    )
    {
    STATUS error;
    device_t xbd;

    /* Create a RAM disk. Don't support partitions */
    xbd = xbdRamDiskDevCreate (BLOCKSIZE, DISKSIZE, 0, DEVNAME);
    if (xbd == NULLDEV)
        {
        printf("Failed to create RAM disk. errno = 0x%x\n", errno);
        return (ERROR);
        }

    /*
     * Format the RAM disk for HRFS. Allow for upto a 1000 files/directories
     * and let HRFS determine the logical block size.
     */
    error = hrfsFormat (DEVNAME, 0ll, 0, 1000);
    if (error != OK)
        {
        printf("Failed to format RAM disk. errno = 0x%x\n", errno);
        return (ERROR);
        }

    printf ("%s now ready for use.\n", DEVNAME);
    return (OK);
    }
```

```
STATUS hrfsFileExample
    (
    void
    )
    {
    int fd;
    char path[PATH_MAX];
    char *testString = "hello world";
    int size = strlen (testString) + 1; /* size of test string including EOS */
    int len;

    /* Create a file under the root directory */

    /* First build the path */
    sprintf (path, "%s/myfile", DEVNAME);

    fd = open (path, O_RDWR | O_CREAT, 0777);
    if (fd < 0)
        {
        printf ("Couldn’t create file %s. errno = 0x%x\n", path, errno);
        return (ERROR);
        }

    /* Write to the file */
    printf("Writing %d bytes to file.\n", size);
    len = write (fd, testString, size);
    if (len != size)
        {
        printf ("Couldn’t write to file %s. errno = 0x%x\n", path, errno);
        close (fd);
        return (ERROR);
        }

    /* Close and re-open file */
    close (fd);

    fd = open (path, O_RDWR, 0777);
    if (fd < 0)
        {
        printf ("Couldn’t re-open file %s. errno = 0x%x\n", path, errno);
        return (ERROR);
        }

    /* Now read back what we wrote */
    printf("Reading %d bytes from file.\n", size);
    len = read (fd, path, size);
    if (len != 12)
        {
        printf ("Couldn’t read from file %s. errno = 0x%x\n", path, errno);
        close (fd);
        return (ERROR);
        }
    /* Make sure we read what we wrote */
    if ((len = strcmp (path, testString)) != 0)
        {
        printf ("Read data different from written data. errno = 0x%x, %d\n", errno, len);
        close (fd);
        return (ERROR);
        }
    close (fd);
    return (OK);
    }
```

Note that to use this code, you must configure VxWorks with the **INCLUDE_HRFS_FORMAT**, **INCLUDE_XBD_RAMDRV** and **INCLUDE_XBD_PART_LIB** components.

The following illustrates running the example from the shell.

```
-> hrfsSetup
Instantiating /myram as rawFs
Formatting /myram for HRFS
Instantiating /myram as rawFs
Formatting...OK.
/myram now ready for use.
value = 0 = 0x0
-> hrfsFileExample
Writing 12 bytes to file.
Reading 12 bytes from file.
value = 0 = 0x0
-> ll "/myram"
Listing Directory /myram:
drwxrwxrwx  1 0       0             2048 Jan  1 00:00 ./
drwxrwxrwx  1 0       0             2048 Jan  1 00:00 ../
-rwxrwxrwx  1 0       0               12 Jan  1 00:00 myfile
value = 0 = 0x0
->
```

8.4.5 Transactional Operations and Commit Policies

HRFS is a transactional file system. That is, transaction or commit points are set to make disk changes permanent. Commit points can be configured to be set under different conditions, which are referred to as policies. Some disk operations trigger commits regardless of the policy. Under certain circumstances, HRFS rollbacks undo disk changes since the last commit, in order to protect the integrity of the file system.

For information about static and dynamic configuration of commit policies, see *8.4.2 Configuring HRFS*, p.460 and *8.4.6 Configuring Transaction Points at Runtime*, p.471.

Commit Policies

The HRFS commit policies are as follows:

Automatic

Any operation that changes data on the disk results in a transaction point being set. This is the safest policy in terms of the potential for data loss. It is also the slowest in terms of performance, as every write to disk cause a commit. This is the default policy. There is no need for explicit action on the part of an application to commit a change. The following routines, for example, cause modifications to disk and result in a commit when the automatic policy is in force:

- **write( )**
- **remove( )**
- **delete( )**
- **mkdir( )**
- **rmdir( )**
- **link( )**
- **unlink( )**
- **truncate( )**
- **ftruncate( )**
- **ioctl( )** when used with a control function that requires modifying the disk.

Manual

The application decides when a commit is to be performed. The user explicitly sets transaction points. This is the fastest policy in terms of performance but obviously has the potential for greater data loss. The application can, however, decide when critical data has been written and needs to be committed. The **commit( )** routine is used with this policy.

Periodic

Transaction points are set automatically at periodic intervals. This policy is in between automatic and manual in terms of performance and potential data loss.

Mandatory Commits

For both manual and periodic commit policies there are circumstances under which a commit is always performed. Mandatory commits occur under the following circumstances:

- Creation of a file or directory
- Deletion of a file or directory.
- Renaming/moving a file or directory.
- Space in the inode journal is exhausted.
- Commit policy is changed at runtime.

Note that mandatory commits are a subset of automatic commits—they do not, for example, include **write( )** and **truncate( )**.

Rollback

A rollback undoes any disk changes since the last commit. Rollbacks usually occur when the system is unexpectedly powered down or reset. Rollbacks can also occur when the file system encounters errors; for example, the lack of disk space to complete a **write( )**, or an error is reported by the underlying device driver. Rollbacks of this nature only happen on operations that modify the media. Errors on read operations do not force a rollback.

A rollback involves HRFS returning to the state of the disk at the last transaction point, which thereby preserves the integrity of the file system, but at the expense of losing file data that has changed since the last transaction point. If the manual or periodic commit policy is specified, there is the potential for losing a lot of data—although the integrity of the file system is preserved.

8.4.6 Configuring Transaction Points at Runtime

The Highly Reliable File System (HRFS) provides configurable transaction points, which allow for finer control of how and when transaction points are set.

The **HRFS_DEFAULT_COMMIT_POLICY** and **HRFS_DEFAULT_COMMIT_PERIOD** component configuration parameters are used to statically define the default commit policy and period (for more information see *8.4.2 Configuring HRFS*, p.460).

Both kernel and RTP applications can change commit policies at runtime. The following **ioctl( )** functions are used to get and set commit policies:

- **FIOCOMMITPOLICYGETFS**
- **FIOCOMMITPOLICYSETFS**
- **FIOCOMMITPERIODGETFS**
- **FIOCOMMITPERIODSETFS**

The commit policy for each volume can be changed using the **ioctl( )** function **FIOCOMMITPOLICYSETFS** as the second parameter.

The third parameter then specifies the actual commit policy: **FS_COMMIT_POLICY_AUTO**, **FS_COMMIT_POLICY_MANUAL**, or **FS_COMMIT_POLICY_PERIODIC**.

If an HRFS volume has been configured for periodic commits, the commit period can be changed with **ioctl( )** function **FIOCOMMITPERIODSETFS**. The third parameter is used to specify the commit period in milliseconds. If 0 is specified then the default commit period is used.

The **commit( )** routine can be used to commit programmatically. The routine is provided by the **INCLUDE_DISK_UTILS** component.

Example 8-4 **Setting an HRFS Commit Policy at Runtime**

```
/* open root directory of an HRFS volume */
fd = open ("/hrfs", O_READONLY, 0666);
if (fd < 0)
    return ERROR;
/* Set commit policy to manual */
if (ioctl (fd, FIOCOMMITPOLICYSETFS, (void *)FS_COMMIT_POLICY_MANUAL)) ==
           ERROR)
    return ERROR;
/* Policy set to manual. Change policy to periodic */
if (ioctl (fd, FIOCOMMITPOLICYSETFS, (void *)FS_COMMIT_POLICY_PERIODIC)) ==
           ERROR)
    return ERROR;
/* Policy set to periodic. Change commit period to 10 seconds*/
if (ioctl (fd, FIOCOMMITPERIODSETFS, (void *)10000)) == ERROR)
    return ERROR;

/* Period set to 10 seconds. Change commit period  back to default */
if (ioctl (fd, FIOCOMMITPERIODSETFS, (void *)0)) == ERROR)
    return ERROR;

/* Period reset. Change policy back to automatic */
if (ioctl (fd, FIOCOMMITPOLICYSETFS, (void *)FS_COMMIT_POLICY_AUTO)) ==
           ERROR)
    return ERROR;
```

8.4.7 File Access Time Stamps

Access time stamps can be enabled by configuring VxWorks with the **INCLUDE_HRFS_ACCESS_TIMESTAMP** component. The component is include in the **BUNDLE_RTP_POSIX_PSE52** component bundle.

For access time stamps to be saved to disk, the volume must be formatted with HRFS on-disk format 1.2 or greater. Version 1.2 is the default version for VxWorks 6.3. See API references for **hrfsAdvFormat( )** and **hrfsAdvFormatFd( )** for more information.

When the access timestamp component is included, and the appropriate disk format version is used, reading from a file or directory causes its access time stamp to be updated. This can cause significant performance loss, as a write to disk occurs even on a read operation and a transaction point is set. Only use access time stamps if the application requires it for POSIX compliance.

8.4.8 Maximum Number of Files and Directories

HRFS files and directories are stored on disk in data structures called inodes. During formatting the maximum number of inodes is specified as a parameter to **hrfsFormat( )**. The total number of files and directories can never exceed the number inodes. Attempting to create a file or directory when all inodes are in use generates an error. Deleting a file or directory frees the corresponding inode.

8.4.9 Working with Directories

This section discusses creating and removing directories, and reading directory entries.

Creating Subdirectories

You can create as many subdirectories as there are inodes. Subdirectories can be created in the following ways:

- With **open( )**. To create a directory, the **O_CREAT** option must be set in the flags parameter and the **S_IFDIR** or **FSTAT_DIR** option must be set in the mode parameter. The **open( )** calls returns a file descriptor that describes the new directory. The file descriptor can only be used for reading only and should be closed when it no longer needed.

- With **mkdir( )** from **usrFsLib**.

When creating a directory using either of the above methods, the new directory name must be specified. This name can be either a full pathname or a pathname relative to the current working directory.

Removing Subdirectories

A directory that is to be deleted must be empty (except for the "**.**" and "**..**" entries). The root directory can never be deleted. Subdirectories can be removed in the following ways:

- Using **ioctl( )** with the **FIORMDIR** function and specifying the name of the directory. The file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
- Using the **remove( )**, specifying the name of the directory.
- Use **rmdir( )** from **usrFsLib**.

Reading Directory Entries

You can programmatically search directories on HRFS volumes using the **opendir( )**, **readdir( )**, **rewinddir( )**, and **closedir( )** routines.

To obtain more detailed information about a specific file, use the **fstat( )** or **stat( )** routine. Along with standard file information, the structure used by these routines also provides the file-attribute byte from a directory entry.

For more information, see the API reference for **dirLib**.

8.4.10 Working with Files

This section discusses file I/O and file attributes.

File I/O Routines

Files on an HRFS file system device are created, deleted, written, and read using the standard VxWorks I/O routines: **creat( )**, **remove( )**, **write( )**, and **read( )**. For more information, see *7.4 Basic I/O*, p.365, and the **ioLib** API references.

Note that and **remove( )** is synonymous with **unlink( )** for HRFS.

File Linking and Unlinking

When a link is created an inode is not used. Another directory entry is created at the location specified by the parameter to **link( )**. In addition, a reference count to the linked file is stored in the file's corresponding inode. When unlinking a file, this reference count is decremented. If the reference count is zero when **unlink( )** is called, the file is deleted except if there are open file descriptors open on the file. In this case the directory entry is removed but the file still exists on the disk. This prevents tasks and processes (RTPs) from opening the file. When the final open file descriptor is closed the file is fully deleted freeing its inode.

Note that you cannot create a link to a subdirectory only to a regular file.

File Permissions

HRFS files have POSIX-style permission bits (unlike dosFs files, which have attributes). The bits can be changed using the **chmod( )** and **fchmod( )** routines. See the API references for more information.

8.4.11 Crash Recovery and Volume Consistency

HRFS is a transactional based file system that is designed to be consistent at all times.

Crash Recovery

If a system unexpectedly loses power or crashes, HRFS rolls back to the last transaction point when the system reboots. The rollback occurs automatically when the file system is mounted. Any changes made after the last complete transaction are lost, but the disk remains in a consistent state.

Consistency Checking

An HRFS file system remains in a consistent state for most media (such as hard drives) as long as the underlying hardware is working correctly and never writes an incomplete sector or physical block.

This is necessarily true for RAM disks, however, because sector writing is simply a copy of one memory location to another. The write operation may be interrupted before completion if the system loses power or crashes.

The **hrfsChkDsk( )** routine can, however, be used to check for inconsistencies in the file system. The execution of the disk checker is not automatic; it must be done programmatically. The **hrfsChkDsk( )** routine is also useful in relation to hardware problems, and is usually run at boot time for this purpose.

8.4.12 I/O Control Functions Supported by HRFS

The HRFS file system supports the **ioctl( )** functions. These functions are defined in the header file **ioLib.h** along with their associated constants; and they are listed in Table 8-1.

Table 8-1 **I/O Control Functions Supported by HRFS**

Function	Decimal Value	Description
FIODISKCHANGE	13	Announces a media change.
FIODISKFORMAT	5	Formats the disk (device driver function).
FIODISKINIT	6	Initializes a file system on a disk volume.
FIOFLUSH	2	Flushes the file output buffer.
FIOFSTATGET	38	Gets file status information (directory entry data).
FIOGETNAME	18	Gets the filename of the *fd*.
FIOMOVE	47	Moves a file (does not rename the file).
FIONFREE	30	Gets the number of free bytes on the volume.
FIONREAD	1	Gets the number of unread bytes in a file.
FIOREADDIR	37	Reads the next directory entry.

Table 8-1 **I/O Control Functions Supported by HRFS** (cont'd)

Function	Decimal Value	Description
FIORENAME	10	Renames a file or directory.
FIORMDIR	32	Removes a directory.
FIOSEEK	7	Sets the current byte offset in a file.
FIOSYNC	21	Same as **FIOFLUSH**, but also re-reads buffered file data.
FIOTRUNC	42	Truncates a file to a specified length.
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position in a file.
FIONCONTIG64	50	Gets the maximum contiguous disk space into a 64-bit integer.
FIONFREE64	51	Gets the number of free bytes into a 64-bit integer.
FIONREAD64	52	Gets the number of unread bytes in a file into a 64-bit integer.
FIOSEEK64	53	Sets the current byte offset in a file from a 64-bit integer.
FIOWHERE64	54	Gets the current byte position in a file into a 64-bit integer.
FIOTRUNC64	55	Set the file's size from a 64-bit integer.

For more information, see the API reference for **ioctl( )** in **ioLib.**

8.5 MS-DOS-Compatible File System: dosFs

The dosFs file system is an MS-DOS-compatible file system that offers considerable flexibility appropriate to the multiple demands of real-time applications. The primary features are:

- Hierarchical files and directories, allowing efficient organization and an arbitrary number of files to be created on a volume.
- A choice of contiguous or non-contiguous files on a per-file basis.
- Compatible with widely available storage and retrieval media (diskettes, hard drives, and so on).
- The ability to boot VxWorks from a dosFs file system.
- Support for VFAT (Microsoft VFAT long file names)
- Support for FAT12, FAT16, and FAT32 file allocation table types.

For information about dosFs libraries, see the VxWorks API references for **dosFsLib** and **dosFsFmtLib**.

For information about the MS-DOS file system, please see the Microsoft documentation.

dosFs and Flash Memory

For information about using dosFs with flash memory, see *10. Flash File System Support: TrueFFS*.

dosFs and the Transaction-Based Reliable File System Facility

The dosFs file system can be used with the transaction-based reliable file system (TRFS) facility; see *7.8.9 Transaction-Based Reliable File System Facility: TRFS*, p.405.

8.5.1 Configuring VxWorks for dosFs

To include dosFs support in VxWorks, configure the kernel with the appropriate required and optional components.

Required Components

The following components are required:

INCLUDE_DOSFS_MAIN	**dosFsLib**
INCLUDE_DOSFS_FAT	dosFs FAT12/16/32 FAT handler
INCLUDE_XBD	XBD component

And, either one or both of the following components are required:

INCLUDE_DOSFS_DIR_VFAT	Microsoft VFAT direct handler
INCLUDE_DOSFS_DIR_FIXED	Strict 8.3 & VxLongNames directory handler

In addition, you must include the appropriate component for your block device; for example, **INCLUDE_ATA**.

If you are using a device driver that is not designed for use with the XBD facility, you must use the **INCLUDE_XBD_BLK_DEV** wrapper component in addition to **INCLUDE_XBD**. See *XBD Block Device Wrapper*, p.404 for more information.

Note that you can use **INCLUDE_DOSFS** to automatically include the following components:

- **INCLUDE_DOSFS_MAIN**
- **INCLUDE_DOSFS_DIR_VFAT**
- **INCLUDE_DOSFS_DIR_FIXED**
- **INCLUDE_DOSFS_FAT**
- **INCLUDE_DOSFS_CHKDSK**
- **INCLUDE_DOSFS_FMT**

Optional dosFs Components

The optional dosFs components are:

INCLUDE_DOSFS_CACHE	disk cache facility
INCLUDE_DOSFS_FMT	dosFs file system formatting module
INCLUDE_DOSFS_CHKDSK	file system integrity checking
INCLUDE_DISK_UTIL	standard file system operations, such as **ls**, **cd**, **mkdir**, **xcopy**, and so on
INCLUDE_TAR	the **tar** utility

Optional XBD Components

Optional XBD components are:

INCLUDE_XBD_PART_LIB	disk partitioning facilities
INCLUDE_XBD_TRANS	TRFS support facility
INCLUDE_XBD_RAMDRV	RAM disk facility

For information about the XBD facility, see *7.8.8 Extended Block Device Facility: XBD*, p.402).

8.5.2 Configuring dosFs

Several dosFs component configuration parameters can be used to define how the file system behaves when a dosfs volume is mounted. These parameters are as follows:

DOSFS_CHK_ONLY
When a dosfs volume is mounted, the media is analyzed for errors, but no repairs are made.

DOSFS_CHK_REPAIR

Similar to **DOSFS_CHK_ONLY,** but an attempt to repair the media is made if errors are found.

DOSFS_CHK_NONE
Media is not checked for errors on mount.

DOSFS_CHK_FORCE
Used in conjunction with **DOSFS_CHK_ONLY** and **DOSFS_CHK_REPAIR** to force a consistency check even if the disk has been marked clean.

DOS_CHK_VERB_SILENT or **DOS_CHK_VERB_0**
dosFs does not to produce any output to the terminal when mounting.

DOS_CHK_VERB_1
dosFs produces a minimal amount of output to the terminal when mounting.

DOS_CHK_VERB_2
dosFs to produces maximum amount output to the terminal when mounting.

Other parameters can be used to configure physical attributes of the file system. They are as follows:

DOSFS_DEFAULT_CREATE_OPTIONS
The default parameter for the **dosFsLib** component. It specifies the action to be taken when a dosFs file system is instantiated. Its default is **DOSFS_CHK_NONE**.

DOSFS_DEFAULT_MAX_FILES
The maximum number of files. The default is 20.

DOSFS_DEFAULT_DATA_CACHE_SIZE
The size of the data cache. The default is 128 KB.

DOSFS_DEFAULT_FAT_CACHE_SIZE
The size of the FAT cache. The default 16 KB.

DOSFS_DEFAULT_DIR_CACHE_SIZE
The directory cache size. The default is 64 KB.

Caches can be tuned dynamically for individual instances of the file system using the **dosFsCacheInfo( )** and **dosFsCacheTune( )** routines.

The routines **dosFsCacheDelete( )** and **dosFsCacheCreate( )** can be used to delete and changes the size of caches. To change the size, first delete, and then create.

8.5.3 Creating a dosFs File System

This section describes the process of creating a dosFs file system. It first provides a summary overview and then a detailed description of each step. See *8.5.4 dosFs, ATA Disk, and RAM Disk Examples*, p.486 for examples of the steps and code examples.

Overview of dosFs File System Creation

For information operating system configuration, see *8.5.1 Configuring VxWorks for dosFs*, p.478. Note that The file system is initialized automatically at boot time.

The steps involved in creating a dosFs file system are as follows:

1. If you are using a custom driver, create the appropriate block device. See *Step 1:Create a Block Device*, p.462.

   If you are using a standard VxWorks component for the device, it is created automatically.

2. If you are using a device driver that is not XBD-compliant, create an XBD device wrapper. See *Step 2:Create an XBD Device Wrapper*, p.483. (Also see *XBD Block Device Wrapper*, p.404.)

3. Optionally, create and mount partitions. See *Step 3:Create Partitions*, p.463.

4. If you are not using pre-formatted disks, format the volumes. See *Step 4:Formatting the Volume*, p.484.

5. Optionally, change the size of the disk cache. See *Step 5:Change the Disk Cache Size*, p.485.

6. Optionally, check the disk for volume integrity. See *Step 6:Check Disk Volume Integrity*, p.485.

dosFs File System Creation Steps

Before any other operations can be performed, the dosFs file system library, **dosFsLib**, must be initialized. This happens automatically at boot time, triggered by the required dosFs components that were included in the system.

Initializing the file system invokes **iosDrvInstall( )**, which adds the driver to the I/O system driver table. The driver number assigned to the dosFs file system is recorded in a global variable, **dosFsDrvNum**. The table specifies the entry points for the dosFs file operations that are accessed by the devices using dosFs.

Step 1: Create a Block Device

If you are using a standard VxWorks component for the device, it is created automatically.

If you are using a custom driver, create the appropriate block device by calling the creation routine for the device driver. The format for this routine is *xxx***DevCreate( )** where *xxx* represents the device driver type; for example, **scsiBlkDevCreate( )** or **ataDevCreate( )**.

The driver routine returns a pointer to a block device descriptor structure, **BLK_DEV**. This structure describes the physical attributes of the device and specifies the routines that the device driver provides. The pointer returned is used to create an XBD block device wrapper. For more information on block devices, see *TRFS Code Examples*, p.408.

Step 2: Create an XBD Device Wrapper

If you are using a device driver that is not XBD-compliant, it requires an XBD device wrapper.

The wrapper is created automatically if you have configured VxWorks with the **INCLUDE_XBD_BLK_DEV** wrapper component (See *XBD Block Device Wrapper*, p.404). Otherwise, create a wrapper for each block device using **xbdBlkDevCreate( )**.

After the XBD device wrapper is created the physical device is automatically probed for a file system and partitions. If a disk is already formatted, the disk is mounted. If a a file system is found, it is mounted. If the file system is not dosFs, it must be formatted (see below).

Step 3: Create Partitions

If you have included the **INCLUDE_XBD_PART_LIB** component in your system, you can create partitions on a disk and mount volumes atop the partitions. Use the **xbdCreatePartition( )** routine to create partitions.

This step should only be performed once, when the disk is first initialized. If partitions are already written to the disk, this step should not be performed as it destroys data.

Step 4: Formatting the Volume

If you are using unformatted disk or wish to replace the current file system on the disk, format the disk by calling **dosFsVolFormat( )**. For more information, see the VxWorks API reference for this routine.

The MS-DOS and dosFs file systems provide options for the format of the File Allocation Table (FAT) and the format of the directory. These options, described below, are completely independent.

⚠ **CAUTION:** Reformatting a disk destroys any data that may be on it.

File Allocation Table (FAT) Formats

A volume FAT format is set during disk formatting, according to either the volume size (by default), or the per-user defined settings passed to **dosFsVolFormat( )**. FAT options are summarized in Table 8-2:

Table 8-2 **FAT Formats**

Format	FAT Table Entry SIze	Usage	Size
FAT12	12 bits per cluster number	Appropriate for very small devices with up to 4,084 KB clusters.	Typically, each cluster is two sectors large.
FAT16	16 bits per cluster number	Appropriate for small disks of up to 65,524 KB clusters.	Typically, used for volumes up to 2 GB; can support up to 8 GB.
FAT32	32 bits (only 28 used) per cluster number	Appropriate for medium and larger disk drives.	By convention, used for volumes larger than 2 GB.

Directory Formats

The options for the directory format are:

- **MSFT Long Names (VFAT)**

  Uses case-insensitive long filenames, with up to 254 characters. This format accepts disks created with short names. MSFT Long Names[1] is the default directory format.

[1] The MSFT Long Names (VFAT) format supports 32-bit file size fields, limiting the file size to a 4 GB maximum.

- **Short Names (8.3)**

  Case-insensitive MS-DOS-style filenames (8.3), with eight uppercase characters for the *name* itself and three for the *extension*.

Step 5: Change the Disk Cache Size

If you have included the **INCLUDE_DOSFS_CACHE** component, disk caches are automatically created. Three parameters define the size of the data, directory entry, and FAT caches: **DOSFS_DEFAULT_DATA_CACHE_SIZE** (default 128 KB), **DOSFS_DEFAULT_FAT_CACHE_SIZE** (default 16 KB), and **DOSFS_DEFAULT_DIR_CACHE_SIZE** (default 64 KB).

In addition, the caches can be tuned dynamically using the **dosFsCacheInfo( )** and **dosFsCacheTune( )** routines.

You can change the size of the cache for a particular instantiation of the file system by first destroying the cache with **dosFsCacheDelete( )** and then re-creating the cache with **dosFsCacheCreate( )**. For more information see the VxWorks API references for these routines.

A disk cache is intended to reduce the to reduce the number of accesses to the media. It is not intended for use with RAM disks or TrueFFS. If the cache component is included with TrueFFS, the cache should be removed programmatically with **dosFsCacheDelete( )**.

Step 6: Check Disk Volume Integrity

Optionally, check the disk for volume integrity using **dosFsChkDsk( )**. Disk checking large disks can be time-consuming.

The parameter **DOSFS_DEFAULT_CREATE_OPTIONS** (of the **INCLUDE_DOSFS_MAIN** component) provides an option for checking the disk, which takes place automatically when the dosfs file system is mounted. This can, however, be a time consuming process and makes the file system inaccessible until the check is complete. Alternatively, the check can be performed programmatically by calling **dosFsChkDsk( )**.

8.5.4 dosFs, ATA Disk, and RAM Disk Examples

This section provides examples of the steps discussed in the preceding section. These examples use a variety of configurations and device types. They are meant to be relatively generic and applicable to most block devices. The examples illustrate the following:

- Creating and working with a dosFs file system on an ATA disk with commands from the shell.
- Code that creates and formats partitions.
- Code that creates and formats a RAM disk volume.

The examples in this section require that VxWorks be configured with the **INCLUDE_DOSFS_FMT** component. One example also relies on the **INCLUDE_DOSFS_CACHE** component.

⚠ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name; unexpected results may occur.

Example 8-5 **Create dosFs for an ATA Disk**

This example demonstrates how to initialize an ATA disk with dosFs from the shell. While these steps use an XBD-compatible ATA block device, they are applicable to any XBD-compatible block device.

1. If you are using a custom driver, create an ATA block device that controls the master ATA hard disk (drive zero) on the primary ATA controller (controller zero). This device uses the entire disk.

```
-> xbd = ataXbdDevCreate(0,0,0,0,"/ata")
New symbol "xbd" added to kernel symbol table.
Instantiating /ata:0 as rawFs
xbd = 0xca4fe0: value = 262145 = 0x40001
```

   The **xbd** variable is of type **device_t**. A value of zero would indicate an error in **ataXbdDevCreate( )**. Such an error usually indicates a BSP configuration or hardware configuration error.

   If you are using the standard **INCLUDE_ATA** device component, the block device is created automatically. Note that in this case the default device name (provided by the component) is **/ata0a**.

2. Display information about devices.

   ```
   -> devs
   drv name
     0 /null
     1 /tyCo/0
     1 /tyCo/1
     8 yow-grand:
     9 /vio
     4 /ata:0
   value = 25 = 0x19
   ```

   The new ata driver **/ata:0** is listed. The zero in the name indicates that no partitions were detected. Note that if no file system is detected on the device, the rawFs file system is instantiated automatically and appears the device list.

3. Create two partitions on this disk device, specifying 50% of the disk space for the second partition, leaving 50% for the first partition. This step should only be performed once, when the disk is first initialized. If partitions are already written to the disk, this step should not be performed since it destroys data.

   ```
   -> xbdCreatePartition ("/ata:0", 2, 50, 0, 0)
   value = 0 = 0x0
   ```

4. Then list the devices to display information about the new partitions.

   ```
   -> devs
   drv name
     0 /null
     1 /tyCo/0
     1 /tyCo/1
     8 yow-grand:
     9 /vio
     3 /ata:1
     3 /ata:2
   ```

   Note that **/ata:0** does not appear in this list, and two new devices, **/ata:1** and **/ata:2**, have been added to represent the new partitions. Each volume has rawfs instantiated in it as they are new and unformatted.

5. Format the volumes for dosFs. This step need only be done once, when the volumes are first initialized. If the volumes have already been initialized (formatted), then omit this step. This example formats the file system volumes with default options.

```
-> dosFsVolFormat ("/ata:1", 0, 0)
Formatting /ata:1 for DOSFS
Instantiating /ata:1 as rawFs
Formatting...Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT32, sectors per cluster 8
  2 FAT copies, 0 clusters, 38425 sectors per FAT
  Sectors reserved 32, hidden 0, FAT sectors 76850
  Root dir entries 0, sysId (null)  , serial number 3a80000
  Label:"           " ...
Disk with 40149184 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT32, sectors per cluster 8
  2 FAT copies, 5008841 clusters, 39209 sectors per FAT
  Sectors reserved 32, hidden 0, FAT sectors 78418
  Root dir entries 0, sysId VX5DOS32, serial number 3a80000
  Label:"           " ...
OK.
value = 0 = 0x0

-> dosFsVolFormat ("/ata:2", 0, 0)
Formatting /ata:2 for DOSFS
Instantiating /ata:2 as rawFs
Formatting...Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT32, sectors per cluster 8
  2 FAT copies, 0 clusters, 19602 sectors per FAT
  Sectors reserved 32, hidden 0, FAT sectors 39204
  Root dir entries 0, sysId (null)  , serial number c78ff000
  Label:"           " ...
Disk with 40144000 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT32, sectors per cluster 8
  2 FAT copies, 5008195 clusters, 39204 sectors per FAT
  Sectors reserved 32, hidden 0, FAT sectors 78408
  Root dir entries 0, sysId VX5DOS32, serial number c78ff000
  Label:"           " ...
OK.
value = 0 = 0x0
```

   For more information, see the API reference for **dosFsFmtLib**.

6. If the **INCLUDE_DOSFS_CACHE** component is included in VxWorks, 128 KB data, 16 KB directory and a 64 KB FAT cache are created by default. The size of these caches can be changed by removing them and creating a new one. The following example deletes the default caches and creates new ones at twice their size.

```
-> dosFsCacheDelete "/ata:1"
value = 0 = 0x0
-> dosFsCacheCreate "/ata:1", 0, 256 * 1024, 0, 32 * 1024, 0, 128 * 1024;
value = 0 = 0x0
```

7. Display information about the dosFs volumes.

```
-> ll "/ata:1"

Listing Directory /ata:1:
value = 0 = 0x0
-> ll "/ata:2"

Listing Directory /ata:2:
value = 0 = 0x0
-> dosFsShow "/ata:2"

volume descriptor ptr (pVolDesc):    0xc7c358
XBD device block I/O handle: 0x60001
auto disk check on mount:      NOT ENABLED
volume write mode:         copyback (DOS_WRITE)
max # of simultaneously open files:    22
file descriptors in use:       0
# of different files in use:       0
# of descriptors for deleted files:    0
# of  obsolete descriptors:        0

current volume configuration:
 - volume label:    NO LABEL ; (in boot sector:              )
 - volume Id:       0xc78ff000
 - total number of sectors:   40,144,000
 - bytes per sector:     512
 - # of sectors per cluster: 8
 - # of reserved sectors:    32
 - FAT entry size:      FAT32
 - # of sectors per FAT copy:    39,204
 - # of FAT table copies:   2
 - # of hidden sectors:      0
 - first cluster is in sector #  78,440
 - Update last access date for open-read-close = FALSE
 - directory structure:         VFAT
 - file name format:            8-bit (extended-ASCII)
 - root dir start cluster:      2

FAT handler information:
------------------------
 - allocation group size:       501 clusters
 - free space on volume:        20,513,562,620 bytes
value = 0 = 0x0
```

Above, we can see the **Volume** parameters for the **/ata:2** volume. The file system volumes are now mounted and ready to be used.

If you are working with an ATA hard disk or a CD-ROM file system from an ATAPI CD-ROM drive, you can, alternatively, use **usrAtaConfig( )**. This routine processes several steps at once. For more information, see the API reference.

Example 8-6 **Creating and Partitioning a Disk and Creating Volumes**

This code example takes a pointer to a block device, creates three partitions, creates the partition handler for these partitions, and creates the dosFs device handler for them. Then, it formats the partitions using **dosFsVolFormat( )**.

```
STATUS usrPartDiskFsInit
    (
    char * xbdName /* device name used during creation of XBD */
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    devname_t xbdPartName;

int newDataCacheSize = 0x40000 /* 256 KB data cache */
int newFatCacheSize = 0x20000 /* 128 KB FAT cache */
int newDirCacheSize = 0x8000 /* 32 KB Dir cache */
DOSFS_CACHE_INFO cacheParams;
int i;


    /* Map partition names */

    for (i = 1; i <= 3; i++)
        {
        sprintf (xbdPartName, "%s:d", devNames[i-1],i);
        fsmNameInstall (devNames[i], xbdPartName);
        }

    /* create partitions */

    if((xbdCreatePartition (xbdName,3,50,45)) == ERROR)
        return ERROR;

    /* Formatting the first partition */

    if(dosFsVolFormat (devNames[0], 2,0) == ERROR)
        return ERROR;

    /* Re-configure the cache for the first partition */
    if (dosFsCacheCreate (devNames[0], NULL, newDataCacheSize, NULL,
        newDirCacheSize, NULL, newFatCacheSize) == ERROR)
    return ERROR;

/* Retrieve the current data cache tuning parameters and double them */
if (dosFsCacheInfoGet (devNames[0], DOS_DATA_CACHE, &cacheParams) == ERROR)
    return ERROR;

cacheParams.bypass = cacheParams.bypass * 2;
cacheParams.readAhead = cacheParams.readAhead * 2;

if (dosFsCacheTune (devNames[0], DOS_DATA_CACHE, &cacheParams) == ERROR)
    return ERROR;
    /* Formatting the second partition */

    if(dosFsVolFormat (devNames[1], 2,0) == ERROR)
        return ERROR;

    /* Formatting the third partition */

    if(dosFsVolFormat (devNames[2], 2,0) == ERROR)
        return ERROR;

    return OK;
    }
```

Note that in most cases you would be likely to format the different partitions for different file systems.

Example 8-7 **Creating and Formatting a RAM Disk Volume**

The following code creates a RAM disk and formats it for use with the dosFs file system.

```
STATUS usrRamDiskInit
    (
    void                              /* no argument */
    )
    {
    int ramDiskSize = 512 * 1024 ;    /* 512KB, 512 bytes per sector */
    char *ramDiskDevName = "/ram0" ;
    device_t  xbd;

/* 512 byte/sec, no partition support */

    xbd = xbdRamDiskDevCreate (512, ramDiskSize, 0, ramDiskDevName);

    if( xbd == NULL )
        return ERROR ;

    /* format the RAM disk, ignore memory contents */

    dosFsVolFormat( ramDiskDevName, DOS_OPT_BLANK | DOS_OPT_QUIET, NULL );

    return OK;
    }
```

8.5.5 Working with Volumes and Disks

This section discusses accessing volume configuration information and synchronizing volumes. For information about **ioctl( )** support functions, see *8.5.10 I/O Control Functions Supported by dosFsLib*, p.500.

Accessing Volume Configuration Information

The **dosFsShow( )** routine can be used to display volume configuration information from the shell. The **dosFsVolDescGet( )** routine can be used programmatically obtain or verify a pointer to the **DOS_VOLUME_DESC** structure. For more information, see the API references for these routines.

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, a disk should be synchronized before it is removed. For more information, see the API references for **close( )** and **dosFsVolUnmount( )**.

8.5.6 Working with Directories

This section discusses creating and removing directories, and reading directory entries.

Creating Subdirectories

For FAT32, subdirectories can be created in any directory at any time. For FAT12 and FAT16, subdirectories can be created in any directory at any time, except in the root directory once it reaches its maximum entry count. Subdirectories can be created in the following ways:

- Using **ioctl( )** with the **FIOMKDIR** function. The name of the directory to be created is passed as a parameter to **ioctl( )**.
- Using **open( )**. To create a directory, the **O_CREAT** option must be set in the *flags* parameter 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.
- Use **mkdir( )** from **usrFsLib**.

When creating a directory using any of the above methods, the new directory name must be specified. This name can be either a full pathname or a pathname relative to the current working directory.

Removing Subdirectories

A directory that is to be deleted must be empty (except for the "**.**" and "**..**" entries). The root directory can never be deleted. Subdirectories can be removed in the following ways:

- Using **ioctl( )** with the **FIORMDIR** function, specifying the name of the directory. The file descriptor used can refer to any file or directory on the volume, or to the entire volume itself.
- Using the **remove( )** function, specifying the name of the directory.
- Use **rmdir( )** from **usrFsLib**.

Reading Directory Entries

You can programmatically search directories on dosFs volumes using the **opendir( )**, **readdir( )**, **rewinddir( )**, and **closedir( )** routines.

To obtain more detailed information about a specific file, use the **fstat( )** or **stat( )** routine. 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 API reference for **dirLib**.

8.5.7 Working with Files

This section discusses file I/O and file attributes.

File I/O Routines

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( )**. For more information, see *7.4 Basic I/O*, p.365, and the **ioLib** API references.

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 8-3.

Table 8-3 **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

DOS_ATTR_RDONLY
If this flag is set, files accessed with **open( )** cannot be written to. If the **O_WRONLY** or **O_RDWR** flags are set, **open( )** returns **ERROR**, setting **errno** to **S_dosFsLib_READ_ONLY**.

DOS_ATTR_HIDDEN
This flag is ignored by **dosFsLib** and produces no special handling. For example, entries with this flag are reported when searching directories.

DOS_ATTR_SYSTEM
This flag is ignored by **dosFsLib** and produces no special handling. For example, entries with this flag are reported when searching directories.

DOS_ATTR_VOL_LABEL
This is a volume label flag, which 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.

DOS_ATTR_DIRECTORY
This is a directory flag, which indicates that this entry is a subdirectory, and not a regular file.

DOS_ATTR_ARCHIVE
This is an archive flag, which 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** operators.

Example 8-8 **Setting DosFs File Attributes**

This example makes a dosFs file read-only, and leaves other attributes intact.

```
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);
    return (OK);
    }
```

NOTE: You can also use the **attrib( )** routine to change file attributes. For more information, see the entry in **usrFsLib**.

8.5.8 Disk Space Allocation Options

The dosFs file system allocates disk space using one of the following methods. The first two methods are selected based upon the size of the write operation. The last method must be manually specified.

- **single cluster allocation**

  Single cluster allocation uses a single cluster, which is the minimum allocation unit. This method is automatically used when the write operation is smaller than the size of a single cluster.

- **cluster group allocation (nearly contiguous)**

  Cluster group allocation uses adjacent (contiguous) groups of clusters, called *extents*. Cluster group allocation is nearly contiguous allocation and is the default method used when files are written in units larger than the size of a disk's cluster.

- **absolutely contiguous allocation**

  Absolutely contiguous allocation uses only absolutely contiguous clusters. Because this type of allocation is dependent upon the existence of such space, it is specified under only two conditions: immediately after a new file is created and when reading from a file assumed to have been allocated to a contiguous space. Using this method risks disk fragmentation.

For any allocation method, you can deallocate unused reserved bytes by using the POSIX-compliant routine **ftruncate( )** or the **ioctl( )** function **FIOTRUNC**.

Choosing an Allocation Method

Under most circumstances, cluster group allocation is preferred to absolutely contiguous file access. Because it is nearly contiguous file access, it achieves a nearly optimal access speed. Cluster group allocation also significantly minimizes the risk of fragmentation posed by absolutely contiguous allocation.

Absolutely contiguous allocation attains raw disk throughput levels, however this speed is only slightly faster than nearly contiguous file access. Moreover, fragmentation is likely to occur over time. This is because after a disk has been in use for some period of time, it becomes impossible to allocate contiguous space. Thus, there is no guarantee that new data, appended to a file created or opened with absolutely continuous allocation, will be contiguous to the initially written data segment.

It is recommended that for a performance-sensitive operation, the application regulate disk space utilization, limiting it to 90% of the total disk space. Fragmentation is unavoidable when filling in the last free space on a disk, which has a serious impact on performance.

Using Cluster Group Allocation

The dosFs file system defines the size of a cluster group based on the media's physical characteristics. That size is fixed for each particular media. Since seek operations are an overhead that reduces performance, it is desirable to arrange files so that sequential portions of a file are located in physically contiguous disk clusters. Cluster group allocation occurs when the cluster group size is considered sufficiently large so that the seek time is negligible compared to the **read**/**write** time. This technique is sometimes referred to as *nearly contiguous* file access because seek time between consecutive cluster groups is significantly reduced.

Because all large files on a volume are expected to have been written as a group of extents, removing them frees a number of extents to be used for new files subsequently created. Therefore, as long as free space is available for subsequent file storage, there are always extents available for use. Thus, cluster group allocation effectively prevents *fragmentation* (where a file is allocated in small units spread across distant locations on the disk). Access to fragmented files can be extremely slow, depending upon the degree of fragmentation.

Using Absolutely Contiguous Allocation

A contiguous file is made up of a series of consecutive disk sectors. Absolutely contiguous allocation is intended to allocate contiguous space to a specified file (or directory) and, by so doing, optimize access to that file. You can specify absolutely contiguous allocation either when creating a file, or when opening a file previously created in this manner.

For more information on the **ioctl( )** functions, see *8.5.10 I/O Control Functions Supported by dosFsLib*, p.500.

Allocating Contiguous Space for a File

To allocate a contiguous area to a newly created file, follow these steps:

1. First, create the file in the normal fashion using **open( )** or **creat( )**.
2. Then, call **ioctl( )**. Use the file descriptor returned from **open( )** or **creat( )** as the file descriptor argument. Specify **FIOCONTIG** as the function code argument and the size of the requested contiguous area, in bytes, as the third argument.

The FAT is then searched for a suitable section of the disk. If found, this space is assigned to the new file. The file can then be closed, or it can be used for further I/O operations. The file descriptor used for calling **ioctl( )** should be the only descriptor open to the file. Always perform the **ioctl( ) FIOCONTIG** operation before writing any data to the file.

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);
```

Allocating Space for Subdirectories

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 subsequently 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.

Opening and Using a Contiguous File

Fragmented files require following cluster chains in the FAT. However, if a file is recognized as contiguous, the system can use an enhanced method that improves performance. This applies to all contiguous files, whether or not they were explicitly created using **FIOCONTIG**. Whenever a file is opened, it is checked for contiguity. If it is found to be contiguous, the file system registers the necessary information about that file to avoid the need for subsequent access to the FAT table. This enhances performance when working with the file by eliminating seek operations.

When you are opening a contiguous file, you can explicitly indicate that the file is contiguous by specifying the **DOS_O_CONTIG_CHK** flag with **open( )**. This prompts the file system to retrieve the section of contiguous space, allocated for this file, from the FAT table.

To find the maximum contiguous area on a device, you can use the **ioctl( )** function **FIONCONTIG**. This information can also be displayed by **dosFsConfigShow( )**.

Example 8-9 **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*).

```
STATUS contigTest
    (
    void                    /* no argument */
    )
    {
    int count;              /* size of maximum contiguous area in bytes */
    int fd;                 /* file descriptor */

    /* 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);
    return (OK);
    }
```

8.5.9 Crash Recovery and Volume Consistency

The DOS file system is inherently susceptible to data structure inconsistencies that result from interruptions during certain types of disk updates. These types of interruptions include power failures, system crashes (for fixed disks), and the manual removal of a disk.

NOTE: The DOS file system is not considered a fault-tolerant file system. The VxWorks dosFs file system, however, can be used in conjunction with the Transaction-Based Reliable File System facility; see *7.8.9 Transaction-Based Reliable File System Facility: TRFS*, p.405.

Inconsistencies occur because the file system data for a single file is stored in three separate regions of the disk. The data stored in these regions are:

- The file chain in the File Allocation Table (FAT), located in a region near the beginning of the disk.
- The directory entry, located in a region that could be anywhere on the disk.
- File clusters containing file data, that could be located anywhere on the disk.

Since all three regions are not always updated before an interruption, dosFs includes an optional integrated consistency-checking mechanism to detect and recover from inconsistencies. For example, if a disk is removed when a file is being deleted, a consistency check completes the file deletion operation. Or, if a file is being created when an interruption occurs, then the file is un-created. In other words, the consistency checker either rolls forward or rolls back the operation that experienced the inconsistency, making whichever correction is possible.

8.5.10 I/O Control Functions Supported by dosFsLib

The dosFs file system supports the **ioctl( )** functions. These functions are defined in the header file **ioLib.h** along with their associated constants, and they are described in Table 8-4.

Table 8-4 **I/O Control Functions Supported by dosFsLib**

Function	Decimal Value	Description
FIOATTRIBSET	35	Sets the file-attribute byte in the dosFs directory entry.
FIOCONTIG	36	Allocates contiguous disk space for a file or directory.
FIODISKCHANGE	13	Announces a media change.
FIODISKFORMAT	5	Formats the disk (device driver function).
FIODISKINIT	6	Initializes a dosFs file system on a disk volume.
FIOFLUSH	2	Flushes the file output buffer.
FIOFSTATGET	38	Gets file status information (directory entry data).
FIOGETNAME	18	Gets the filename of the *fd*.
FIOLABELGET	33	Gets the volume label.

Table 8-4 **I/O Control Functions Supported by dosFsLib** (cont'd)

Function	Decimal Value	Description
FIOLABELSET	34	Sets the volume label.
FIOMKDIR	31	Creates a new directory.
FIOMOVE	47	Moves a file (does not rename the file).
FIONCONTIG	41	Gets the size of the maximum contiguous area on a device.
FIONFREE	30	Gets the number of free bytes on the volume.
FIONREAD	1	Gets the number of unread bytes in a file.
FIOREADDIR	37	Reads the next directory entry.
FIORENAME	10	Renames a file or directory.
FIORMDIR	32	Removes a directory.
FIOSEEK	7	Sets the current byte offset in a file.
FIOSYNC	21	Same as **FIOFLUSH**, but also re-reads buffered file data.
FIOTRUNC	42	Truncates a file to a specified length.
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position in a file.
FIOCONTIG64	49	Allocates contiguous disk space using a 64-bit size.
FIONCONTIG64	50	Gets the maximum contiguous disk space into a 64-bit integer.
FIONFREE64	51	Gets the number of free bytes into a 64-bit integer.
FIONREAD64	52	Gets the number of unread bytes in a file into a 64-bit integer.
FIOSEEK64	53	Sets the current byte offset in a file from a 64-bit integer.

Table 8-4 **I/O Control Functions Supported by dosFsLib** (cont'd)

Function	Decimal Value	Description
FIOWHERE64	54	Gets the current byte position in a file into a 64-bit integer.
FIOTRUNC64	55	Set the file's size from a 64-bit integer.

For more information, see the API references for **dosFsLib** and for **ioctl( )** in **ioLib.**

8.5.11 Booting from a Local dosFs File System Using SCSI

VxWorks can be booted from a local SCSI device (such as a hard drive in the target system). Before you can boot from SCSI, you must make a new boot loader that contains the SCSI library. Configure VxWorks with the **INCLUDE_SCSI**, **INCLUDE_SCSI_BOOT**, and **SYS_SCSI_CONFIG** components.

After creating the SCSI boot loader ROM, 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 system 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 and is incompatible with the NFS requirement that device names not end in a slash.

Step 1: Create the SCSI Device

Create the SCSI device using **scsiPhysDevCreate( )** and initialize the disk with a dosFs file system. Modify the file *installDir***/vxworks-6.***x***/target/***bspName***/sysScsi.c** to reflect your SCSI configuration.

Step 2: Rebuild Your System

Rebuild your system.

Step 3: Copy the VxWorks Run-time Image

Copy the file **vxWorks** to the drive. Below, a VxWorks task spawns the **copy( )** routine, passing it two arguments.

The first argument is the source file for the **copy( )** command. The source file is the VxWorks run-time image, **vxWorks**. The source host name is **tiamat**:, the source filename is **C:/vxWorks**. These are passed to **copy( )** in concatenated form, as the string "**tiamat:C:/vxWorks**."

The second argument is the destination file for the **copy( )** command. The dosFs file system, on the local target SCSI disk device, is named **/sd0**, and the target file name is **vxWorks**. These are, similarly, passed to **copy( )** in concatenated form, as the string "**/sd0/vxWorks**." When booting the target from the SCSI device, the boot loader image should specify the run-time file as "**/sd0/vxWorks**".

```
-> sp (copy, "tiamat:c:/vxWorks","/sd0/vxWorks")
task spawned: id = 0x3f2a200, name = t2
value = 66232832 = 0x3f2a200

Copy OK: 1065570 bytes copied
```

Step 4: Copy the System Symbol Table

Depending upon image configuration, the **vxWorks.sym** file for the system symbol table may also be needed. Therefore, in similar fashion, copy the **vxWorks.sym** file. The run-time image, **vxWorks**, downloads the **vxWorks.sym** file from the same location.

```
-> sp (copy, "tiamat:c:/vxWorks.sym","/sd0/vxWorks.sym")
task spawned: id = 0x3f2a1bc, name = t3
value = 66232764 = 0x3f2a1bc

Copy OK: 147698 bytes copied
```

Step 5: Test the Copying

Now, list the files to ensure that the files were correctly copied.

```
-> sp (ll, "/sd0")
task spawned: id = 0x3f2a1a8, name = t4
value = 66232744 = 0x3f2a1a8
->

Listing Directory /sd0:
-rwxrwxrwx  1 0       0          1065570 Oct 26 2001 vxWorks
-rwxrwxrwx  1 0       0           147698 Oct 26 2001 vxWorks.sym
```

Step 6: Reboot and Change Parameters

Reboot the system, and then change the boot loader 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.

```
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
```

8.6 Raw File System: rawFs

VxWorks provides a *raw file system* (rawFs) for use in systems that require only the most basic disk I/O functions. The rawFs file system, implemented with **rawFsLib**, treats the entire disk volume much like a single large file.

Although the dosFs file system provides this ability to varying degrees, the rawFs file system offers advantages in size and performance if more complex functions are not required.

The rawFs file system imposes no organization of the data on the disk. It maintains no directory information; and there is therefore no division of the disk area into specific files. All **open( )** operations on rawFs devices specify only the device name; no additional filenames are possible.

The entire disk area is treated as a single file and 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.

A rawFs file system is created by default if inserted media does not contain a recognizable file system.

8.6.1 Configuring VxWorks for rawFs

To use the rawFs file system, configure VxWorks with the **INCLUDE_RAWFS** and **INCLUDE_XBD** components.

If you are using a device driver that is not designed for use with the XBD facility, you must use the **INCLUDE_XBD_BLK_DEV** wrapper component in addition to **INCLUDE_XBD**. See *XBD Block Device Wrapper*, p.404 for more information.

Set the **NUM_RAWFS_FILES** parameter of the **INCLUDE_RAWFS** component to the desired maximum open file descriptor count. For information about using multiple file descriptors with what is essentially a single large file, see *8.6.4 rawFs File I/O*, p.509.

8.6.2 Creating a rawFs File System

The rawFs file system is the default file system. It is created automatically when VxWorks cannot instantiate a known file system such as dosFs, HRFS, or cdromFs. Unlike dosFs and HRFS, rawFs does not have a formatter. There are no particular data structures on the media that signify the disk as being raw. To create a rawFs file system manually, the current file system must be un-instantiated and replaced with rawFs. Having two or more file systems on the same media can produce instabilities in the VxWorks system. Hence, when instantiating a new file system the previous one must be removed.

See *Example 8-10Creating a rawFs File System*, p.508 for code that illustrates how this can be done. (See *8.2 File System Monitor*, p.455 for information about default creation of rawFs.)

The rawFs library **rawFsLib** is initialized automatically at boot time. The **rawFsInit( )** routine is called by the **usrRoot( )** task after starting the VxWorks system. The **rawFsInit( )** 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 parameter can be set with the **NUM_RAWFS_FILES** configuration parameter of the **INCLUDE_RAWFS** component

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, for all devices that use the rawFs file system. The driver number assigned to the rawFs file system is placed in a global variable, **rawFsDrvNum**.

After the rawFs file system is initialized, one or more devices must be created. Devices are created with the device driver's device creation routine (*xx***DevCreate( )**). 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.

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.

NOTE: 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**.

Example 8-10 **Creating a rawFs File System**

This example illustrates creating a rawFs file system.

```
int fd;
device_t xbd;


/* Map some XBD names. Use :0 and :1 since the disk may or may not have    partitions */

fsmNameMap ("/ata:0", "/rawfs");
fsmNameMap ("/ata:1", "/rawfs");
xbd = ataXbdDevCreate (0,0,0,0,"/ata");

/* Get an file descriptor to the current file system */
fd = open ("/rawfs", 0, 0);

/* Register on the path instantiator event */

/* The ejection of the current file system is asynchronous and is handled by    another task.
Depending on relative priorities this may not happen    immediately so the path wait even
facility is used. Each file system will    trip this event when they instatiate to let
waiting task that it is ready.
*/

    fsPathAddedEventSetup (&waitData, "/rawfs");

fd = open ("/rawfs", 0, 0);

/* Eject the current file system and put rawfs in its place */
ioctl (fd, XBD_SOFT_EJECT, (int)XBD_TOP);

    /* Our FD is now invalid */
/* Wait for the path to instantiate */

fsWaitForPath(&waitData);
```

Once the call to **fsWaitForPath( )** returns the rawfs file system is ready.

8.6.3 Mounting rawFs 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.

⚠ **CAUTION:** Because device names are recognized by the I/O system using simple substring matching, file systems should not use a slash (**/**) alone as a name or unexpected results may occur.

8.6.4 rawFs File I/O

To begin I/O operations upon a rawFs device, first open the device using the standard **open( )** routine (or the **creat( )** routine). Data on the rawFs device is written and read using the standard I/O routines **write( )** and **read( )**. For more information, see *7.4 Basic I/O*, p.365.

The character pointer associated with a file descriptor (that is, the byte offset where the read and write operations 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.

8.6.5 I/O Control Functions Supported by rawFsLib

The rawFs file system supports the **ioctl( )** functions shown in Table 8-5. The functions listed are defined in the header file **ioLib.h**. For more information, see the API references for **rawFsLib** and for **ioctl( )** in **ioLib**.

Table 8-5 **I/O Control Functions Supported by rawFsLib**

Function	Decimal Value	Description
FIODISKCHANGE	13	Announces a media change.
FIODISKFORMAT	5	Formats the disk (device driver function).
FIOFLUSH	2	Same as **FIOSYNC**.
FIOGETNAME	18	Gets the device name of the *fd*.
FIONREAD	1	Gets the number of unread bytes on the device.
FIOSEEK	7	Sets the current byte offset on the device.
FIOSYNC	21	Writes out all modified file descriptor buffers.
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position on the device.

8.7 CD-ROM File System: cdromFs

The VxWorks CD-ROM file system, cdromFs allows applications to read data from CDs formatted according to the ISO 9660 standard file system with or without the Joliet extensions. This section describes how cdromFs is organized, configured, and used.

The cdromFs library, **cdromFsLib**, lets applications read any CD-ROMs, CD-Rs, or CD-RWs (collectively called CDs) that are formatted in accordance with the ISO 9660 file system standard, with or without the Joliet extensions. ISO 9660 interchange level 3, implementation level 2, is supported. Note that multi-extent files, interleaved files, and files with extended attribute records are supported.

The following CD features and ISO 9660 features are not supported:

- Multi-volume sets
- Record format files
- CDs with a sector size that is not a power of two[2]
- Multi-session CD-R or CD-RW[3]

After initializing cdromFs and mounting it on a CD-ROM block device, you can access data on that device using the standard POSIX I/O calls: **open( )**, **close( )**, **read( )**, **ioctl( )**, **readdir( )**, and **stat( )**. The **write( )** call always returns an error.

The cdromFs utility supports multiple drives, multiple open files, and concurrent file access. When you specify a pathname, cdromFS accepts both forward slashes (/) and back slashes (\) as path delimiters. However, the backslash is not recommended because it might not be supported in future releases.

The initialization sequence for the cdromFs file system is similar to installing a dosFs file system on a SCSI or ATA device.

After you have created the CD file system device (*8.7.2 Creating and Using cdromFs*, p.512), use **ioctl( )** to set file system options. The files system options are described below:

CDROMFS_DIR_MODE_SET/GET
These options set and get the directory mode. The directory mode controls whether a file is opened with the Joliet extensions, or without them. The directory mode can be set to any of the following:

[2] Therefore, mode 2/form 2 sectors are not supported, as they have 2324 bytes of user data per sector. Both mode 1/form 1 and mode 2/form 1 sectors are supported, as they have 2048 bytes of user data per sector.

[3] The first session (that is, the earliest session) is always read. The most commonly desired behavior is to read the last session (that is, the latest session).

MODE_ISO9660
Do not use the Joliet extensions.

MODE_JOLIET
Use the Joliet extensions.

MODE_AUTO
Try opening the directory first without Joliet, and then with Joliet.

⚠ **CAUTION:** Changing the directory mode un-mounts the file system. Therefore, any open file descriptors are marked as obsolete.

CDROMFS_STRIP_SEMICOLON
This option sets the **readdir( )** strip semicolon setting to **FALSE** if *arg* is 0, and to **TRUE** otherwise. If **TRUE**, **readdir( )** removes the semicolon and following version number from the directory entries retrieved.

CDROMFS_GET_VOL_DESC
This option returns, in *arg*, the primary or supplementary volume descriptor by which the volume is mounted. *arg* must be of type **T_ISO_PVD_SVD_ID**, as defined in **cdromFsLib.h**. The result is the volume descriptor, adjusted for the endianness of the processor (not the raw volume descriptor from the CD). This result can be used directly by the processor. The result also includes some information not in the volume descriptor, such as which volume descriptor is in use.

For information on using **cdromFs( )**, see the API reference for **cdromFsLib**.

8.7.1 Configuring VxWorks for cdromFs

To configure VxWorks with cdromFs, add the **INCLUDE_CDROMFS** and **INCLUDE_XBD** components to the kernel. Add other required components (such as SCSI or ATA) depending on the type of device).

If you are using a device driver that is not designed for use with the XBD facility, you must use the **INCLUDE_XBD_BLK_DEV** wrapper component in addition to **INCLUDE_XBD**. See *XBD Block Device Wrapper*, p.404 for more information.

If you are using an ATAPI device, make appropriate modifications to the **ataDrv**, **ataResources[ ]** structure array (if needed). This must be configured appropriately for your hardware platform.

8.7.2 Creating and Using cdromFs

This section describes the steps for creating a block device for the CD-ROM, creating a **cdromFsLib** device, mounting the file system, and accessing the media. The steps are performed from the shell, and shell show routines are used to display information.

Step 1: Create a Block Device

Create a block device. The following is an example for an ATAPI master device upon the secondary ATA controller:

```
-> xbd = ataXbdDevCreate(1,0,0,0,"/cdrom")
New symbol "xbd" added to kernel symbol table.
xbd = 0xca4fe0: value = 262145 = 0x4000
```

CDROMFS file system is created automatically if a CD is present in the drive.

Step 2: Verify cdromFs is instantiated

If a CD is present in the drive, the device name appears in the **devs** output.

```
-> devs
drv name
  0 /null
  1 /tyCo/0
  1 /tyCo/1
  4 /fd0
  5 /ata0a
  9 yow-grand:
 10 /vio
  3 /cdrom:0
```

Step 3: Open the Root Directory

This step is optional. It is only required if you plan to perform Step 4 or Step 5, both of which use the file descriptor obtained in this step.

```
-> fd = open ("/cdrom:0", 0, 0)
New symbol "fd" added to kernel symbol table.
fd = 0x18cef98: value = 4 = 0x4
```

In the command-line sequence above, the first 0 is the value of **O_RDONLY** in **fcntl.h**.

Remember to close the root directory in Step 6.

Step 4: Set readdir( ) To Omit Version Numbers from Its Output

This step is optional.

The strip semicolon mode controls whether **readdir( )** returns version numbers. After mounting, the strip semicolon mode defaults to **FALSE**, meaning that version numbers will be returned. If you do not want version numbers, type the following:

```
    -> ioctl (fd, 0x740002, 1)
```

In the command-line sequence above, 0x740002 is the value of **CDROMFS_STRIP_SEMICOLON** in **cdromFsLib.h**.

Step 5: Specify Which Volume Descriptor To Use

This step is optional.

The directory mode controls which volume descriptor is used to open a file or directory. After mounting, the directory mode defaults to **MODE_AUTO**, meaning that all volume descriptors will be tried. The directory mode can be changed to **MODE_ISO9660** to use only the ISO 9660 volume descriptors, or to **MODE_JOLIET** to use only the Joliet volume descriptors. If either of these modes is selected, and if the CD does not contain the selected volume descriptor, an **S_cdromFsLib_UNKNOWN_FILE_SYSTEM** error is recorded.

```
    -> ioctl (fd, 0x740000, 0)
```

In the command-line sequence above, 0x740000 is the value of **CDROMFS_DIR_MODE_SET**, and 0 is the value of **MODE_ISO9660**. Both are located in **cdromFsLib.h**.

Step 6: Close the Root Directory

This step is required only if you opened the root directory in Step 3.

```
    -> close (fd)
```

Step 7: Check the Configuration

You can check the CD-ROM configuration using **cdromFsVolConfigShow( )**:

```
-> cdromFsVolConfigShow "/cdrom:0"

device config structure ptr     0x18d4dd8
device name                     /cdrom:0
bytes per physical sector       2048

Primary directory hierarchy:

volume descriptor number        :1
descriptor logical sector       :16
descriptor offset in sector     :0
standard ID                     :CD001
volume descriptor version       :1
UCS unicode level  (0=ISO9660)  :0
system ID                       :
volume ID                       :DELL_P1110
volume size                     :37773312 = 36 MB
number of logical blocks        :18444 = 0x480c
volume set size                 :1
volume sequence number          :1
logical block size              :2048
path table memory size (bytes)  :364
path table size on CD (bytes)   :364
path table entries              :21
volume set ID                   :


volume publisher ID             :


volume data preparer ID         :


volume application ID           :NERO - BURNING ROM


copyright file name             :none
abstract file name              :none
bibliographic file name         :none
creation date                   :13.07.2000  12:30:00:00
modification date               :13.07.2000  12:30:00:00
expiration date                 :00.00.0000  00:00:00:00
effective date                  :00.00.0000  00:00:00:00
value = 0 = 0x0
```

8.7.3 I/O Control Functions Supported by cdromFsLib

The cdromFs file system supports the **ioctl( )** functions. These functions, and their associated constants, are defined in the header files **ioLib.h** and **cdromFsLib.h**.

Table 8-6 describes the **ioctl( )** functions that **cdromFsLib** supports. For more information, see the API references for **cdromFsLib** and for **ioctl( )** in **ioLib**.

Table 8-6 **ioctl( ) Functions Supported by cdromFsLib**

Function Constant	Decimal	Description
CDROMFS_DIR_MODE_GET	7602176	Gets the volume descriptor(s) used to open files.
CDROMFS_DIR_MODE_SET	7602177	Sets the volume descriptor(s) used to open files.
CDROMFS_GET_VOL_DESC	7602179	Gets the volume descriptor that is currently in use.
CDROMFS_STRIP_SEMICOLON	7602178	Sets the **readdir( )** strip version number setting.
FIOFSTATGET	38	Gets file status information (directory entry data).
FIOGETNAME	18	Gets the filename of the file descriptor.
FIOLABELGET	33	Gets the volume label.
FIONREAD	1	Gets the number of unread bytes in a file.
FIONREAD64	52	Gets the number of unread bytes in a file (64-bit version).
FIOREADDIR	37	Reads the next directory entry.
FIOSEEK	7	Sets the current byte offset in a file.
FIOSEEK64	53	Sets the current byte offset in a file (64-bit version).
FIOUNMOUNT	39	Un-mounts a disk volume.
FIOWHERE	8	Returns the current byte position in a file.
FIOWHERE64	54	Returns the current byte position in a file (64-bit version).

8.7.4 Version Numbers

cdromFsLib has a 4-byte version number. The version number is composed of four parts, from most significant byte to least significant byte:

- major number
- minor number
- patch level
- build

The version number is returned by **cdromFsVersionNumGet( )** and displayed by **cdromFsVersionNumDisplay( )**.

8.8 Read-Only Memory File System: ROMFS

ROMFS is a simple, read-only file system that represents and stores files and directories in a linear way (similar to the tar utility). It is installed in RAM with the VxWorks system image at boot time. The name ROMFS stands for *Read-Only Memory File System*; it does not imply any particular relationship to ROM media.

ROMFS provides the ability to bundle VxWorks applications—or any other files for that matter—with the operating system. No local disk or network connection to a remote disk is required for executables or other files. When VxWorks is configured with the ROMFS component, files of any type can be included in the operating system image simply by adding them to a ROMFS directory on the host system, and then rebuilding VxWorks. The build produces a single system image that includes both the VxWorks and the files in the ROMFS directory.

When VxWorks is booted with this image, the ROMFS file system and the files it holds are loaded with the kernel itself. ROMFS allows you to deploy files and operating system as a unit. In addition, process-based applications can be coupled with an automated startup facility so that they run automatically at boot time. ROMFS thereby provides the ability to create fully autonomous, multi-process systems.

ROMFS can also be used to store applications that are run interactively for diagnostic purposes, or for applications that are started by other applications under specific conditions (errors, and so on).

8.8.1 Configuring VxWorks with ROMFS

VxWorks must be configured with the **INCLUDE_ROMFS** component to provide ROMFS facilities.

8.8.2 Building a System With ROMFS and Files

Configuring VxWorks with ROMFS and applications involves the following steps:

1. Create a ROMFS directory in the project directory on the host system, using the name **/romfs**.
2. Copy the application files into the **/romfs** directory.
3. Rebuild VxWorks.

For example, adding a process-based application called **myVxApp.vxe** from the command line would look like this:

```
cd c:\myInstallDir\vxworks-6.1\target\proj\wrSbc8260_diab
mkdir romfs
copy c:\allMyVxApps\myVxApp.vxe romfs
make TOOL=diab
```

The contents of the **romfs** directory are automatically built into a ROMFS file system and combined with the VxWorks image.

The ROMFS directory does not need to be created in the VxWorks project directory. It can also be created in any location on (or accessible from) the host system, and the **make** utility's **ROMFS_DIR** macro used to identify where it is in the build command. For example:

```
    make TOOL=diab ROMFS_DIR="c:\allMyVxApps"
```

Note that any files located in the **romfs** directory are included in the system image, regardless of whether or not they are application executables.

8.8.3 Accessing Files in ROMFS

At run-time, the ROMFS file system is accessed as **/romfs**. The content of the ROMFS directory can be browsed using the **ls** and **cd** shell commands, and accessed programmatically with standard file system routines, such as **open( )** and **read( )**.

For example, if the directory *installDir***/vxworks-6.***x***/target/proj/wrSbc8260_diab/romfs** has been created on the host, the file **foo** copied to it, and the system rebuilt and booted; then using **cd** and **ls** from the shell (with the command interpreter) looks like this:

```
[vxWorks *]# cd /romfs
[vxWorks *]# ls
.
..
foo
[vxWorks *]#
```

And **foo** can also be accessed at run-time as **/romfs/foo** by any applications running on the target.

8.8.4 Using ROMFS to Start Applications Automatically

ROMFS can be used with various startup mechanisms to start process-based applications automatically when VxWorks boots.

See the *VxWorks Application Programmer's Guide: Applications and Processes* for more information.

8.9 Target Server File System: TSFS

The Target Server File System (TSFS) is designed for development and diagnostic purposes. It is a full-featured VxWorks file system, but the files are actually located on the host system.

NOTE: TSFS is not designed for use with large files (whether application executables or other files), and performance may suffer when they are greater than 50 KB. For large files, use FTP or NFS instead of TSFS

TSFS provides all of the I/O features of the network driver for remote file access (see *7.8.5 Non-NFS Network Devices*, p.400), without requiring any target resources—except those required for communication between the target system and the target server on the host. The TSFS uses a WDB target agent driver to transfer requests from the VxWorks I/O system to the target server. The target server reads the request and executes it using the host file system. When you open a file with TSFS, the file being opened is actually on the host. Subsequent **read( )** and **write( )** calls on the file descriptor obtained from the **open( )** call read from and write to the opened host file.

The TSFS VIO driver is oriented toward file I/O rather than toward console operations. TSFS provides all the I/O features that **netDrv** provides, without requiring any target resource beyond what is already configured to support communication between target and target server. It is possible to access host files randomly without copying the entire file to the target, to load an object module from a virtual file source, and to supply the filename to routines such as **ld( )** and **copy( )**.

Each I/O request, including **open( )**, is synchronous; the calling target task is blocked until the operation is complete. This provides flow control not available in the console VIO implementation. In addition, there is no need for WTX protocol requests to be issued to associate the VIO channel with a particular host file; the information is contained in the name of the file.

Consider a **read( )** call. The driver transmits the ID of the file (previously established by an **open( )** call), the address of the buffer to receive the file data, and the desired length of the read to the target server. The target server responds by issuing the equivalent **read( )** call on the host and transfers the data read to the target program. The return value of **read( )** and any **errno** that might arise are also relayed to the target, so that the file appears to be local in every way.

For detailed information, see the API reference for **wdbTsfsDrv**.

Socket Support

TSFS sockets are operated on in a similar way to other TSFS files, using **open( )**, **close( )**, **read( )**, **write( )**, and **ioctl( )**. To open a TSFS socket, use one of the following forms of filename:

```
"TCP:hostIP:port"
"TCP:hostname:port"
```

The *flags* and *permissions* arguments are ignored. The following examples show how to use these filenames:

```
fd = open("/tgtsvr/TCP:phobos:6164",0,0);     /* open socket and connect  */
                                            /* to server phobos         */

fd = open("/tgtsvr/TCP:150.50.50.50:6164",0,0);  /* open socket and     */
                                                 /* connect to server    */
                                                 /* 150.50.50.50        */
```

The result of this **open( )** call is to open a TCP socket on the host and connect it to the target server socket at *hostname* or *hostIP* awaiting connections on *port*. The resultant socket is non-blocking. Use **read( )** and **write( )** to read and write to the TSFS socket. Because the socket is non-blocking, the **read( )** call returns immediately with an error and the appropriate **errno** if there is no data available to read from the socket. The **ioctl( )** usage specific to TSFS sockets is discussed in the API reference for **wdbTsfsDrv**. This socket configuration allows VxWorks to use the socket facility without requiring **sockLib** and the networking modules on the target.

Error Handling

Errors can arise at various points within TSFS and are reported back to the original caller on the target, along with an appropriate error code. The error code returned is the VxWorks **errno** which most closely matches the error experienced on the host. If a WDB error is encountered, a WDB error message is returned rather than a VxWorks **errno**.

Configuring VxWorks for TSFS Use

To use TSFS, configure VxWorks with the **INCLUDE_WDB_TSFS** component. This creates the **/tgtsvr** file system on the target.

The target server on the host system must also be configured for TSFS. This involves assigning a root directory on your host to TSFS (see the discussion of the target server **-R** option in *Security Considerations*, p.520). For example, on a PC host you could set the TSFS root to **c:\myTarget\logs**.

Having done so, opening the file **/tgtsvr/logFoo** on the target causes **c:\myTarget\logs\logFoo** to be opened on the host by the target server. A new file descriptor representing that file is returned to the caller on the target.

Security Considerations

While TSFS has much in common with **netDrv**, the security considerations are different (also see *7.8.5 Non-NFS Network Devices*, p.400). With TSFS, the host file operations are done on behalf of the user that launched the target server. The user name given to the target as a boot parameter has no effect. In fact, none of the boot parameters have any effect on the access privileges of TSFS.

In this environment, it is less clear to the user what the privilege restrictions to TSFS actually are, since the user ID and host machine that start the target server may vary from invocation to invocation. By default, any host tool that connects to a target server which is supporting TSFS has access to any file with the same authorizations as the user that started that target server. However, the target server can be locked (with the **-L** option) to restrict access to the TSFS.

The options which have been added to the target server startup routine to control target access to host files using TSFS include:

-R Set the root of TSFS.
For example, specifying **-R /tftpboot** prepends this string to all TSFS filenames received by the target server, so that **/tgtsvr/etc/passwd** maps to **/tftpboot/etc/passwd**. If **-R** is not specified, TSFS is not activated and no TSFS requests from the target will succeed. Restarting the target server without specifying **-R** disables TSFS.

-RW Make TSFS read-write.
The target server interprets this option to mean that modifying operations (including file create and delete or write) are authorized. If **-RW** is not specified, the default is read only and no file modifications are allowed.

NOTE: For more information about the target server and the TSFS, see the **tgtsvr** command reference. For information about specifying target server options from Workbench, see the *Wind River Workbench User's Guide: Setting Up Your Hardware* and the *Wind River Workbench User's Guide: New Target Server Connections*.

Using the TSFS to Boot a Target

For information about using the TSFS to boot a targets, see *3.11 Booting From the Host File System Using TSFS*, p.155.

9

Network File System: NFS

9.1 Introduction

VxWorks provides an implementation of the Network File System (NFS) application protocol, versions 2 and 3.

The first part of this chapter describes how to configure and use an NFS client, which enables a VxWorks target to mount remote file systems and access the contents of those file systems as if they were local. The second part of the chapter describes how to configure and use an NFS server, which enables a VxWorks target to export local systems to remote network systems.

NOTE: VxWorks does not normally provide authentication services for NFS requests. If you need the NFS server to authenticate incoming requests, see the **nfsdInit( )** and **mountdInit( )** reference entries for information on authorization hooks.

9.2 Configuring VxWorks for an NFS Client

The VxWorks supports the following NFS client configuration components, which you can use to include the modules needed to create an NFS client:

- Core NFS Client
- NFS Client All
- NFS v2 Client
- NFS v3 Client
- NFS Mount All

These components are described in detail below.

Core NFS Client

The **INCLUDE_CORE_NFS_CLIENT** component provides **nfsCommon**, a library that provides core functionality for NFS clients. This library also provides parameters that you can use to specify the NFS user ID, group identifier, and the maximum length of the pathname for a file.

This component contains the following configuration parameters:

NFS_USER_ID

Synopsis: User identifier for NFS access.
Default: 2001

NFS_GROUP_ID

Synopsis: Group identifier for NFS access.
Default: 100

NFS_MAXPATH

Synopsis: Maximum file path length.
Default: 255

NFS Client All

Including the **INCLUDE_NFS_CLIENT_ALL** component is equivalent to selecting both **INCLUDE_NFS2_CLIENT** and **INCLUDE_NFS3_CLIENT**, the components that pull in client support for NFS version 2 and NFS version 3 respectively.

This component requires the **INCLUDE_NFS2_CLIENT** and **INCLUDE_NFS3_CLIENT** components.

NFS v2 Client

The **INCLUDE_NFS2_CLIENT** component provides **nfs2Lib** and **nfs2Drv**, which provide support of an NFS version two client. Using this client, you can mount exported NFS (version two) directories to your VxWorks target.

This component requires the **INCLUDE_CORE_NFS_CLIENT** component.

Initialization

This component also configures the VxWorks image to initialize the NFS v2 client, which includes a call to **nfsAuthUnixSet( )**:

```
nfsAuthUnixSet ( sysBootParams.hostName, NFS_USER_ID, NFS_GROUP_ID, 0,
                 (int *) 0);
```

Values for the **NFS_USER_ID** and **NFS_GROUP_ID** parameters are taken from the required **INCLUDE_CORE_NFS_CLIENT** component.

Parameters

This component contains the following configuration parameters:

NFS2_CLIENT_CACHE_DEFAULT_NUM_LINES

Synopsis: Default number of cache lines.
Default: 16

The NFS client implementation uses a persistent cache, which is structured as a multi-line buffer cache. This parameter configures the default number of lines in the NFS v2 client cache. The value can be decreased to as low as 1, or increased.

You can modify the number of cache lines, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs2CacheInit( )**:

```
usrNfs2CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

NFS2_CLIENT_CACHE_DEFAULT_LINE_SIZE

Synopsis: Default number of bytes in cache line.
Default: 16384

The NFS client implementation uses a persistent cache, which is structured as a multi-line buffer cache. The cache line size must be a power of two.

This parameter configures the default size of the NFS v2 client cache line. The default value for this is 16384 (16 kB). When combined with the default number of cache lines, this yields a default cache size of 256 kB.

You can modify the number of cache line size, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs2CacheInit( ):**

```
usrNfs2CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

Changing the size of the cache will not affect any existing cache. It will only impact future caches.

NFS2_CLIENT_CACHE_DEFAULT_OPTIONS

Synopsis: Default cache options.
Default: 0

This parameter configures the default options for the NFS v2 client cache. The two valid settings for this parameter are:

0
The default value of zero (0) means that the cache will collect the written data and only send it to the server when the cache line is full, or it needs to be flushed (no options), which is the default value

1, **NFS_CACHE_WRITE_THROUGH**
A value of one means that the cache will be write-through.

You can modify the cache options, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs2CacheInit( )**:

```
usrNfs2CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

NFS v3 Client

The **INCLUDE_NFS3_CLIENT** component provides the **nfs3Lib** and **nfs3Drv** libraries, which provide support of an NFS version three client. Using this client, you can mount exported NFS (version three) directories to your VxWorks target.

This component requires the **INCLUDE_CORE_NFS_CLIENT** component.

Initialization

This component also configures the VxWorks image to initialize the NFS v3 client, which includes a call to **nfsAuthUnixSet( )**:

```
nfsAuthUnixSet ( sysBootParams.hostName, NFS_USER_ID, NFS_GROUP_ID, 0,
                 (int *) 0);
```

Values for the **NFS_USER_ID** and **NFS_GROUP_ID** parameters to this routine are taken from the required **INCLUDE_CORE_NFS_CLIENT** component.

Parameters

This component contains the following configuration parameters:

NFS3_CLIENT_CACHE_DEFAULT_NUM_LINES

Synopsis: Default number of cache lines.
Default: 16

The NFS client implementation uses a persistent cache, which is structured as a multi-line buffer cache. This parameter configures the default number of lines in the NFS v3 client cache. The value can be decreased to as low as 1, or increased.

You can modify the number of cache lines, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs3CacheInit( )**:

```
usrNfs3CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

NFS3_CLIENT_CACHE_DEFAULT_LINE_SIZE

Synopsis: Default number of bytes in cache line.
Default: 16384

The NFS client implementation uses a persistent cache, which is structured as a multi-line buffer cache. The cache line size must be a power of two.

This parameter configures the default size of the NFS v3 client cache line. The default value for this is 16384 (16 kB). When combined with the default number of cache lines, this yields a default cache size of 256 kB.

You can modify the number of cache line size, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs3CacheInit( ):**

```
usrNfs3CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

Changing the size of the cache will not affect any existing cache. It will only impact future caches.

NFS3_CLIENT_CACHE_DEFAULT_OPTIONS

Synopsis: Default cache options.
Default: 0

This parameter configures the default options for the NFS v3 client cache. The two valid settings for this parameter are:

0
The default value of zero (0) means that the cache will collect the written data and only send it to the server when the cache line is full, or it needs to be flushed (no options), which is the default value

1, **NFS_CACHE_WRITE_THROUGH**
A value of one means that the cache will be write-through.

You can modify the cache options, either at build time or at run-time. To configure the cache at run time, call the routine **usrNfs3CacheInit( )**:

```
usrNfs3CacheInit (UINT32 numLines, UINT32 lineSize, UINT32 options);
```

Configuring NFS3 Client Writes

The NFS v3 client has one additional configurable parameter that is not available on NFS v2. According to the RFC, the NFS v3 client can dictate to an NFS v3 server how it should perform the write operations. At runtime, the NFS v3 client can be set to inform the server that it should perform writes one of the following styles:

- **UNSTABLE**
- **FILE_SYNC**
- **DATA_SYNC**

The default setting is **UNSTABLE**.

You can use two routines to configure these options at run-time:

- **nfs3StableWriteSet(stable_how mode)** lets you set the mode
- **nfs3StableWriteGet( )** routine gets the current mode

NOTE: This option does not exist in NFS v2.

NFS Mount All

The **INCLUDE_NFS_MOUNT_ALL** component configures a VxWorks target to make a boot-time call to **nfsMountAll( )**. This routine automatically mounts all file systems exported by the boot host system.

This component requires the **INCLUDE_NFS_CLIENT_ALL** component.

This component contains the following configuration parameter:

GROUP_EXPORTS

Synopsis: Remove check for GroupName during MountAll.
Default: **FALSE**

An NFS server can specify that an exported file system is intended for particular group of clients. Before mounting an exported file system, a client typically checks that it is a group member. If the client is not a group member, it does not mount the file system. To configure the client to skip this check, set this parameter to **TRUE**. The default of **FALSE** ensures that the check occurs.

9.3 Creating an NFS Client

To create an NFS client and use it to access a remote networked file system:

1. Call **nfs2Drv( )** or **nfs3Drv( )** to initialize support for NFS v2 or v3 or both.

   Normally, you should not need to call either of these routines explicitly. VxWorks calls these routines automatically at boot time if the image includes **INCLUDE_NFS_CLIENT_ALL**. For NFS v2 only, use **INCLUDE_NFS2_CLIENT**. To initialize NFS v3 only, use **INCLUDE_NFS3_CLIENT**.

2. Call **nfsAuthUnixSet( )** to configure the user name and user ID the device should use when communicating with a particular host.

   VxWorks calls **nfsAuthUnixSet( )** automatically at boot time if the image contains **INCLUDE_NFS_CLIENT_ALL**, **INCLUDE_NFS2_CLIENT**, or **INCLUDE_NFS3_CLIENT**. The values for user name and user ID are supplied by the parameters to **INCLUDE_CORE_NFS_CLIENT**.

3. Call **hostAdd( )** to configure the host table to know of the host system from which you want to mount an exported file system.

VxWorks calls **hostAdd( )** for the host system automatically at boot time. If you want to mount file systems from other remote systems, you need to make and explicit **hostAdd( )** call for those systems.

4. Call **nfsMount( )** or **nfsMountAll( )** to actually mount a remote file system. The **nfsMountAll( )** routine queries the specified remote system for a list of exported file names and then creates NFS client device instances for each exported file system. To unmount a file system, use **nfsUnmount( )**. Use **nfsDevShow( )** to display a list of the mounted NFS devices.

   As a convenience, the **INCLUDE_NFS_MOUNT_ALL** component configures an image to make a boot time call to **nfsMountAll( )** to mount all file systems exported by the boot host.

The following sections supplement and expand on the procedure outlined above.

Exporting File Systems from the Remote NFS Server

For a UNIX NFS server, the **/etc/exports** file specifies which of the server's file systems are exported for mounting by remote NFS clients. If a file system on a UNIX NFS server is not listed in **/etc/exports**, the file system is not exported, which means other machines cannot use NFS to mount it. For example, consider an **/etc/exports** file that contains the line:

```
/usr
```

The server exports **/usr** without restriction. If you want to limit access to this directory, you can include additional parameters on the line. For example:

1. On the UNIX box, login as root (super user).
2. Edit: **/etc/exports**
3. Specify the path and permission for the file system that you would export. For example: **/usr * (rw)**

   For more information on these parameters, consult your UNIX system documentation.
4. Export the file system, run: **exportfs -ra**
5. On the UNIX target, run the NFS daemon: **rpc.nfsd**
6. Run: **rpc.rquotad**
7. To run mount the daemon, run: **rpc.mountd**

To check whether NFS is running, use: **rpcinfo -p**.

Windows systems also support NFS. Thus, it is possible to configure a directory on a Windows system so that it is exported over NFS. The exact procedures for doing so depend upon the particular network package you purchased. For more information, consult the documentation included with your Windows networking package.

Setting Your NFS Client Name, User ID, and Group ID

Internally, NFS depends upon RPC to handle the remote execution of the commands (open, read, write, and others) that access the data in the remote file system. Associated with the RPC protocol is an authentication system known as **AUTH_UNIX**. This authentication system requires RPC peers to provide a user name, a user ID, and a group name. The recipient of an RPC message uses this information to decide whether to honor or ignore the RPC request.

On a VxWorks host, you can set the NFS user name, user ID, and group name using the **NFS_GROUP_ID** and **NFS_USER_ID** parameters included in the **INCLUDE_CORE_NFS_CLIENT** component. You can also set these values by calling **nfsAuthUnixSet( )** or **nfsAuthUnixPrompt( )**. For example, to use **nfsAuthUnixSet( )** to set the NFS user ID to 1000 and the NFS group ID to 200 for the machine **mars**, you would call **nfsAuthUnixSet( )** as follows:

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

The **nfsAuthUnixPrompt( )** routine provides a more interactive way of setting the NFS authentication parameters from the shell.

On UNIX systems, a user ID is specified in the file **/etc/passwd**. A list of groups that a user belongs to is specified in the file **/etc/group**. To configure a default user ID and group ID, set **NFS_USER_ID** and **NFS_GROUP_ID**. The NFS authentication parameters will take on these values at system startup. If NFS file access is unsuccessful, make sure that the configuration is correct.

Mounting a Remote File System

After setting your NFS client name, user ID, and group ID, you are ready to call **nfsMount( )** to mount any file system exported by a known host. To add a system to the list of hosts known to a VxWorks system, call **hostAdd( )**:

```
hostAdd ("host", "IPaddress" )
```

This routine associates a host name with an IP address. Thus, if you wanted to mount a file system exported by a system called *mars*, you would need to have already called **hostAdd( )** for mars. For example, if mars were at 192.168.10.1 you would need to call **hostAdd( )** as follows:

```
hostAdd ("mars", "192.168.10.1" )
```

If mars exports a file system called **/usr**, you can now use a call to **nfsMount( )** to create a local mount of that remotely exported file system. The syntax of an **nfsMount( )** call is as follows:

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

hostName
The host name of the NFS server that exports the file system you want to mount.

hostFileSys
The name of the host file system or subdirectory as it is known on the exporting NFS server system.

localName
The local name to assign to the file system.

For example, if you wanted to mount a remote file system, **/d0/**, on your target, **wrs**, as a device called **/myDevice0/**, you would make the following call to **nfsMount( )**:

```
nfsMount ("wrs", "/d0/", "/myDevice0/");
```

The VxWorks target now has access to the contents of **/d0/**, although using the device name, **/myDevice0/**. For example, if the remote device stores the file, **/d0/bigdog**, you can access this file from the **wrs** target using the pathname, **/myDevice0/bigdog**. If you want the local device to use the same device name as is used on the exporting system, use a NULL as the third parameter of the **nfsMount( )** call. For example:

```
nfsMount ("wrs", "/d0/", NULL);
```

Or, from the kernel shell:

```
-> nfsMount "wrs", "/d0/"
```

On the VxWorks target, the **nfsMount( )** call creates the local device, **/d0/**. Thus, on the target, the pathname to **bigdog** is the same as on the exporting system; that is: **/d0/bigdog**.

If you do not need to mount the remote file system under a new name, you should consider using **nfsMountAll( )** instead of **nfsMount( )**. A call to **nfsMountAll( )** mounts all file systems that are exported from the remote system and that are accessible to the specified client.

The syntax of **nfsMountAll( )** is as follows:

```
nfsMountAll( "hostName", "clientName", quietFlag )
```

hostName
The name of the host from which you want to mount all exported file systems.

clientName
The name of a client specified in an access list, if any. A NULL *clientName* mounts only those file systems that are accessible to any client.

quietFlag
A boolean value that tells **nfsMountAll( )** whether to execute in verbose or silent mode. **FALSE** indicates verbose mode, and **TRUE** indicates quiet mode.

Using ioctl( ) With Open Files from a Mounted Directory

After opening a file in a mounted directory, you can work with the file using the **ioctl( )** control functions listed in Table 9-1.

Table 9-1 **Supported I/O Control Functions for Files Accessed through NFS**

IOCTL	Description
FIOGETNAME	Gets the file name of **fd** and copies it to the buffer referenced by **nameBuf**: `status = ioctl (fd, FIOGETNAME, &nameBuf);`
FIONREAD	Copies to **nBytesUnread** the number of bytes remaining in the file specified by **fd**: `status = ioctl (fd, FIONREAD, &nBytesUnread);`
FIOSEEK	Sets the current byte offset in the file to the position specified by **newOffset**. If the seek goes beyond the end-of-file, the file grows. The end-of-file pointer gets moved to the new position, and the new space is filled with zeros: `status = ioctl (fd, FIOSEEK, newOffset);`
FIOSYNC	Flush data to the remote NFS file. It takes no additional argument: `status = ioctl (fd, FIOSYNC, 0);`
FIOWHERE	Returns the current byte position in the file. This is the byte offset of the next byte to be read or written. It takes no additional argument: `position = ioctl (fd, FIOWHERE, 0);`

FIOREADDIR	Reads the next directory entry. Use the third argument in the **ioctl( )** call to supply a pointer to a directory descriptor of type **DIR**.
	`DIR dirStruct;`
	`fd = open ("directory", O_RDONLY);`
	`status = ioctl (fd, FIOREADDIR, &dirStruct);`
	Normally, you do not use the **FIOREADDIR** functionality directly. Instead, you would call **readdir( )**. See the reference entry for **dirLib**.

Table 9-1 **Supported I/O Control Functions for Files Accessed through NFS** (cont'd)

IOCTL	Description

FIOFSTATGET	Gets file status information (directory entry data). Use the third argument in the **ioctl( )** call to supply a pointer to a **stat** structure that is filled with data describing the specified file. For example:
	`struct stat statStruct;`
	`fd = open ("file", O_RDONLY);`
	`status = ioctl (fd, FIOFSTATGET, &statStruct);`
	Normally, you do not use the **FIOFSTATGET** functionality directly. Instead, you would **stat( )** or **fstat( )** routines get file information. See the manual entry for **dirLib**.

FIOFSTATFSGET	Gets the file system parameters for and open file descriptor. Use the third argument in the **ioctl( )** call to supply a pointer to a **statfs** structure that is filled with data describing the underlying file system.
	`statfs statfsStruct;`
	`fd = open ("directory", O_RDONLY);`
	`status = ioctl (fd, FIOFSTATFSGET, &statfsStruct);`
	Normally, you do not use the **FIOFSTATFSGET** functionality directly. Instead, you would **stat( )** or **fstat( )** routines get file information. See the manual entry for **dirLib**.

9.4 Configuring VxWorks for an NFS Server

VxWorks supports the following NFS server configuration components, which you can use to include the modules needed to create an NFS server:

- NFS Server
- NFS server All
- NFS server V2
- NFS server V3

These components are discussed in detail below.

NFS Server

The **INCLUDE_CORE_NFS_SERVER** component provides **nfsd**, the library that supplies the NFS server daemon initialization routines.

For more information on these routines, see the **nfsd** reference entry. This component also supplies configuration parameters that set basic configuration values for the server.

NOTE: The NFS server maintains a reply cache of up to 64 recently transmitted reply messages. The purpose of this cache is to deal with RPC retransmissions. If the XID of an incoming NFS request matches that of a previously replied message, the cached reply is sent back instead of actually executing the NFS request.

Each entry in the server reply cache requires 8800 bytes for the entry itself and 48 bytes for each cache node. Thus, if there were three entries in the cache, the memory usage would be 26,544 bytes, which is (3 * 8800) + (3 * 48) bytes. If there were the full 64 entries in the cache, the memory usage would be 566,272 bytes.

If you think you have noticed a memory leak in NFS, please consider whether this memory use accounts for the discrepancy.

This component contains the following configuration parameters:

NFS_MAXPATH

Synopsis: Maximum length of a file path string (excludes file name).
Default: 255

NFS_USER_ID

Synopsis: User identifier for NFS access.
Default: 2001

NFS_MAXFILENAME

Synopsis: Maximum file name length. Valid values range from 1 to 99.
Default: 40

This parameter specifies the maximum length of a filename. If a filename is longer than this value then it is truncated before returning it to the client. This parameter should be set carefully since it affects the amount of memory allocated by the server.

NFS_GROUP_ID

Synopsis: Group identifier for NFS access.
Default: 100

NFS server All

Including the **INCLUDE_NFS_SERVER_ALL** component is equivalent to selecting both **INCLUDE_NFS3_SERVER** and **INCLUDE_NFS2_SERVER**.

This component requires the **INCLUDE_NFS3_SERVER**, **INCLUDE_NFS2_SERVER**, and **INCLUDE_NFS_SERVER_INSTALL** components.

NFS server V2

The **INCLUDE_NFS2_SERVER** component provides **nfs2dLib** and **mount1Lib**. The **nfs2dLib** library implements the NFS version 2 procedures as specified in the Network File System Protocol Specification (RFC 1094).

The **mount1Lib** library implements a mount version one server to support mounting VxWorks file systems remotely. The mount server is an implementation of version 1 of the mount protocol as defined in RFC 1094.

The **INCLUDE_NFS2_SERVER** component requires the **INCLUDE_CORE_NFS_SERVER** and **INCLUDE_NFS_SERVER_INSTALL** components.

NOTE: VxWorks does not normally provide authentication services for NFS requests. If you need to authenticate incoming requests, see the documentation for **nfsdInit( )** and **mountdInit( )** for information about authorization hooks.

To actually export a file system, you must call **nfsExport( )**.

Table 9-2 lists the requests that RFC 1813 accepted from clients. For details of their use, see RFC 1094, *NFS: Network File System Protocol Specification*.

NFS server V3

The **INCLUDE_NFS3_SERVER** component provides **nfs3dLib** and **mount3Lib**. The **nfs3dLib** library provides an implementation of the NFS version 3 procedures as specified in the RFC 1813 (Network File System Protocol Specification). The

mount3Lib library implements a mount server. The mount server is an implementation of version 3 mount protocol as defined in RFC 1813.

NFS version 3 requires mount versions 1 and 3. Also required are **INCLUDE_CORE_NFS_SERVER** and **INCLUDE_NFS_SERVER_INSTALL**.

To actually export a file system, you must call **nfsExport( )**.

The **INCLUDE_NFS3_SERVER** component requires the **INCLUDE_CORE_NFS_SERVER** and **INCLUDE_NFS_SERVER_INSTALL** components.

NOTE: VxWorks does not normally provide authentication services for NFS requests. If you need to authenticate incoming requests, see the documentation for **nfsdInit( )** and **mountdInit( )** for information about authorization hooks.

The following requests are accepted from clients. For details of their use, see RFC 1813: *NFS: Network File System Protocol Specification*.

Table 9-2 **RFC 1813 Supported Client Requests**

Procedure Name	Procedure Number
NFSPROC_NULL	0
NFSPROC_GETATTR	1
NFSPROC_SETATTR	2
NFSPROC_LOOKUP	3
NFSPROC_ACCESS	4
NFSPROC_READLINK	5 – not supported, limitation in DOSFS
NFSPROC_READ	6
NFSPROC_WRITE	7
NFSPROC_CREATE	8
NFSPROC_MKDIR	9
NFSPROC_SYMLINK	10 – not supported
NFSPROC_MKNOD	11 – not supported
NFSPROC_REMOVE	12

Table 9-2 **RFC 1813 Supported Client Requests** (cont'd)

Procedure Name	Procedure Number
NFSPROC_RMDIR	13
NFSPROC_RENAME	14
NFSPROC_LINK	15 – not supported
NFSPROC_READDIR	16
NFSPROC_READDIRPLUS	17
NFSPROC_FSSTAT	18
NFSPROC_FSINFO	19
NFSPROC_PATHCONF	20
NFSPROC_COMMIT	21

9.5 Creating an NFS Server

To set up an NFS file server on a VxWorks target, you need to configure the VxWorks target to include the appropriate NFS server configuration components, described in *9.4 Configuring VxWorks for an NFS Server*, p.535. You also need a file system for export; for information about file systems, see *8. Local File Systems*.

After you have a file system to export, call **nfsExport( )** to export the specified file system.

Initializing an NFS File System for Export

The following code fragment creates a RAM drive, initializes it for dosFs, and exports the file system for NFS clients on the network:

```
unsigned myBlockSize; /* block size in bytes */
unsigned myTotalSize; /* disk size in bytes */

myBlockSize = 512;
myTotalSize = 16777216; /* 16Mb */

xbdRamDiskDevCreate (myBlockSize, myTotalSize, FALSE, "/ramDrv");
dosFsVolFormat ("/ramDrv", 2, 0);
nfsExport ("/ramDrv", 0, FALSE, 0);
```

⚠ **CAUTION:** For NFS-exportable file systems, the device name absolutely must *not* end in a slash.

Exporting a File System through NFS

After you have an exportable file system, call **nfsExport( )** to make it available to NFS clients on your network. Then mount the file system from the remote NFS client using the facilities of that system. The following example shows how to export the new file system from a VxWorks target 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 routine 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 placeholder for future extensions.

2. To mount the file system from another machine, see the system documentation for that machine. Specify the name of the VxWorks system that exports the file system, and the name of the desired file system. You can also specify a different name for the file system as seen on the NFS client.

⚠ **CAUTION:** On UNIX systems, you 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
```

10
Flash File System Support: TrueFFS

10.1 Introduction

TrueFFS is a flash management facility that provides access to flash memory by emulating disk access.

It provides VxWorks with block device functionality, which allows either the dosFs file system (with or without TRFS support) or the HRFS file system to be used to access flash memory in the same manner as a disk. For information about the file system facilities, see *8.5 MS-DOS-Compatible File System: dosFs*, p.478), *7.8.9 Transaction-Based Reliable File System Facility: TRFS*, p.405, and *8.4 Highly Reliable File System: HRFS*, p.459.

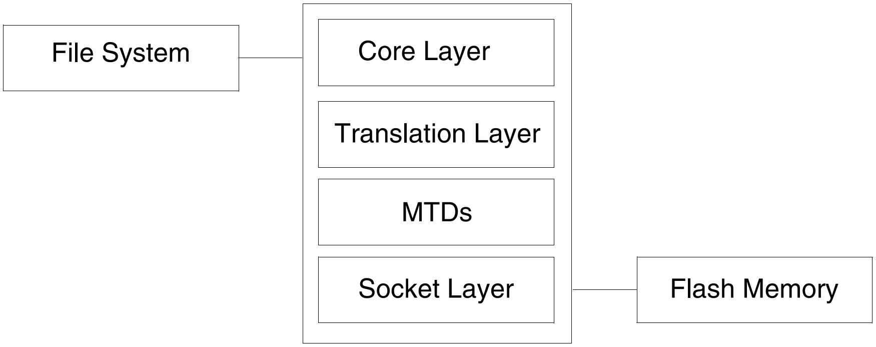
In addition, TrueFFS provides full flash media management capabilities.

TrueFFS is a VxWorks-compatible implementation of M-Systems FLite, version 2.0. This system is reentrant, thread-safe, and supported on all CPU architectures that host VxWorks. TrueFFS consists of the following four layers:

- The *core layer*, which connects the other layers and handles global facilities, such as back-grounding, garbage collection, timers, and other system resources. This layer provides the block device interface for a file system.
- The *flash translation layer*, which maintains the block allocation map that associates the file system's view of the storage medium with erase blocks in flash.
- The *Memory Technology Device (MTD) layer*, which implements the low-level programming of the flash medium (map, read, write, and erase functions).
- The *socket layer*, which provides an interface between TrueFFS and the board hardware with board-specific hardware access routines.

Figure 10-1 illustrates the relationship between the file system, TrueFFS layers, and the flash medium itself.

Figure 10-1 **File System, TrueFFS Layers, and Flash**



This chapter provides instructions for using TrueFFS with the MTDs and drivers that are included in this release. It provides quick-start material for configuring TrueFFS and formatting TrueFFS drives, and thus presents the basic steps required to use the default TrueFFS facilities with your application.

It also provides information about creating a boot image region that excludes TrueFFS, and about writing the boot image to that region.

If you must customize or create new socket drivers or MTDs, or would simply like more detailed information about TrueFFS technology, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

NOTE: This version of the TrueFFS product is a block device driver to VxWorks that, although intended to be file system neutral, is provided for use with the dosFs file system or the HRFS file system. The configuration steps for using TrueFFS with dosFs and HRFS are slightly different.

10.2 Overview of Implementation Steps

This section provides an overview of how to use TrueFFS with VxWorks BSPs that provide support for the TrueFFS component. To determine if your BSP provides TrueFFS support, see the online BSP reference documentation (or the file *installDir*/**vxworks-6.***x***/target/config/***bspName***/target.ref**).

You may need to write certain sub-components for your application. This is most often required for the MTD layer. See the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS* for information in this regard.

To determine if this release provides an MTD suitable for your flash hardware, see *10.3.1 Selecting an MTD*, p.546 and *Including the MTD Component*, p.550.

Step 1: Select an MTD Component

Choose an MTD, appropriate for your hardware, from those provided with the TrueFFS product. See *10.3.1 Selecting an MTD*, p.546.

Step 2: Identify the Socket Driver

Ensure that you have a working socket driver. The socket driver is a source code component, implemented in the file **sysTffs.c**. For BSPs that already support TrueFFS, the socket driver is fully defined and located in the BSP directory. See *10.3.2 Identifying the Socket Driver*, p.547.

Step 3: Configure the System

Configure your system for TrueFFS by adding the appropriate VxWorks components. Minimum support requires components for a file system and the four TrueFFS layers. See *10.3.3 Configuring VxWorks with TrueFFS*, p.548.

Step 4: Build the System

Build the system from Workbench or from the command line with the **vxprj** facility. See *10.3.4 Building the System*, p.551.

Step 5: Boot the Target and Format the Drives

Boot the target and then format the drives. See *10.3.5 Formatting the Flash, p.552*.

Step 6: Create a Flash Region for Boot Code

Optionally, create a boot image region that excludes TrueFFS, and write the boot image to that region (this region can alternatively be used for a system startup log or configuration data). See *10.3.6 Reserving a Region in Flash for a Boot Image*, p.554.

Step 7: Mount the Drive

Mount the file system on a TrueFFS flash drive. See *10.3.7 Mounting the Drive*, p.556.

Step 8: Test the Drive

Test your drive(s). See *10.3.8 Testing the Drive*, p.557.

10.3 Creating a System with TrueFFS

This section provides detailed instructions on configuring VxWorks with the required TrueFFS and file system components, building the system, formatting the flash, mounting the drive, and testing the drive. It also provides information about creating a region for a boot image.

10.3.1 Selecting an MTD

Determine whether any of the MTDs provided with this release support the device that you intend to use for TrueFFS. Devices are usually identified by their JEDEC IDs. If you find an MTD appropriate to your flash device, you can use that MTD. These drivers are also provided in binary form; so you do not need to compile the MTD source code unless you have modified it.

The directory *installDir***/vxworks-6.***x***/target/src/drv/tffs** contains the source code for the following types of MTD components:

- MTDs that work with several of the devices provided by Intel, AMD, Fujitsu, and Sharp.
- Two generic MTDs that can be used for devices complying with CFI.

To better support the out-of-box experience, these MTDs attempt to cover the widest possible range of devices (in their class) and of bus architectures. Consequently, the drivers may not provide the performance you may need for the run-time environment that you want to target. Also note that your hardware configuration may require some changes to the generic drivers in order to work. Consider such requirements as word or byte access limitations, or unusual array configurations.

If the performance and size of the drivers provided do not match your requirements, you can modify them to better suit your needs. For more information, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

NOTE: For the list of the MTD components and details about adding the MTD component to your system, see *Including the MTD Component*, p.550.

10.3.2 Identifying the Socket Driver

The socket driver that you include in your system must be appropriate for your BSP. Some BSPs include socket drivers, others do not. The socket driver file is **sysTffs.c** and, if provided, it is located in your BSP directory.

If your BSP does not provide this file, follow the procedure described in the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*, which explains how to port a stub version to your hardware.

In either case, the build process requires that a working socket driver (**sysTffs.c**) be located in the BSP directory. For more information, see *Adding the Socket Driver*, p.551.

10.3.3 Configuring VxWorks with TrueFFS

To configure a VxWorks systems with TrueFFS, you must include:

- Components to fully support either the dosFs or HRFS file system.
- The **INCLUDE_XBD_BLKDEV** component (see *XBD Block Device Wrapper*, p.404).
- The core layer TrueFFS component, **INCLUDE_TFFS**.

  Some BSPs—like the PowerQuiccII— have a custom MTD in **sysTffs.c** and do not need any additional selections. For this BSP you must identify the flash in which TrueFFS will reside (**INCLUDE_SIMM_TFFS**).
- At least one software module from each of the other three TrueFFS layers (translation, MTD, and socket).

You can configure and build your system either from the command line or with Workbench. For general information on configuration procedures, see the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

For either configuration and build method, special consideration must be given to cases in which either the socket driver or the MTD, or both, are not provided. The drivers must be registered and MTDs need appropriate component descriptions. For more information, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*. For supported BSPs this is usually taken care of automatically by including components. It is more of a concern when adding TrueFFS support to a BSP.

NOTE: Included with TrueFFS are sources for several MTDs and socket drivers. The MTDs are in *installDir***/vxworks-6.***x***/target/src/drv/tffs**. The socket drivers are defined in the **sysTffs.c** files provided in the *installDir***/vxworks-6.***x***/target/config/***bspname* directory for each BSP that supports TrueFFS.

Including File System Components

File system support—for either dosFs or HRFS—must be included in your system for TrueFFS to be useful. For information about these file systems and components upon which they depend, see *8.5 MS-DOS-Compatible File System: dosFs*, p.478 and *8.4 Highly Reliable File System: HRFS*, p.459.

There are other file system components that are not required, but which may be useful. These components add support for the basic functionality needed to use a file system, such as the commands **ls**, **cd**, **copy**, and so forth (which are provided by the **INCLUDE_DISK_UTIL** component).

Including the XBD Wrapper Component

VxWorks must be configured with the **INCLUDE_XBD_BLKDEV** component for the TrueFFS block device driver (which is not XBD-compatible). For more information, see *XBD Block Device Wrapper*, p. 404).

Including the Core TrueFFS Component

VxWorks must be configured with the TrueFFS core component, **INCLUDE_TFFS**. Including this component triggers the correct sequence of events, at boot time, for initializing this product. It also ensures that the socket driver is included in your system (see *Adding the Socket Driver*, p. 551).

Including Utility Components

TrueFFS provides optional utility components for automatic drive detection, show routines, and writing a boot image to flash.

INCLUDE_TFFS_MOUNT
Including this component adds automatic detection (on booting) of existing formatted TrueFFS drives.

INCLUDE_TFFS_SHOW
Including this component adds two TrueFFS configuration display utilities, **tffsShow( )** and **tffsShowAll( )** for use from the shell.

The **tffsShow( )** routine prints device information for a specified socket interface. It is particularly useful when trying to determine the number of erase units required to write a boot image (*10.3.6 Reserving a Region in Flash for a Boot Image*, p. 554). The **tffsShowAll( )** routine provides the same information for all socket interfaces registered with VxWorks. The **tffsShowAll( )** routine can be used from the shell to list the drives in the system. The drives are listed in the order in which they were registered. This component is not included by default.

INCLUDE_TFFS_BOOT_IMAGE

Including this component provides the **tffsBootImagePut( )** routine (in **sysTffs.o**). The routine is used to write a boot image to flash memory (see *Writing the Boot Image to Flash*, p.556)

NOTE: **INCLUDE_TFFS_BOOT_IMAGE** is included by default for some BSPs.

Including the MTD Component

Add the MTD component appropriate to your flash device (*10.3.1 Selecting an MTD*, p.546) to your system. The MTD components for flash devices from Intel, AMD, Fujitsu, and Sharp, are described in Table 10-1. For more information about support for these devices, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

Table 10-1 **Components for TrueFFS MTDs**

Component	Device
INCLUDE_MTD_CFISCS	CFI/SCS device.
INCLUDE_MTD_CFIAMD	CFI-compliant AMD and Fujitsu devices.
INCLUDE_MTD_I28F016	Intel 28f016 device.
INCLUDE_MTD_I28F008	Intel 28f008 device.
INCLUDE_MTD_AMD	AMD, Fujitsu: 29F0{40,80,16} 8-bit devices.

If you have written your own MTD, you must be sure that it is correctly defined for inclusion in the system, and that it explicitly requires the transition layer. See the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS* for information.

Including the Translation Layer Component

Choose the translation layer appropriate to the technology used by your flash medium. The main variants of flash devices are NOR and NAND. TrueFFS provides support for:

- NOR devices.

- NAND devices that conform to the SSFDC specification.

The translation layer is provided in binary form only. The translation layer components are listed in Table 10-2.

Table 10-2 **Components for TrueFFS Translation Layers**

Component	Description
INCLUDE_TL_FTL	The translation layer for NOR flash devices. If you can execute code in flash, your device uses NOR logic.
INCLUDE_TL_SSFDC	The translation layer for devices that conform to Toshiba Solid State Floppy Disk Controller Specifications. TrueFFS supports only those NAND devices that comply with the SSFDC specification.

The component descriptor files specify the dependency between the translation layers and the MTDs; therefore, when configuring through Workbench or the command-line **vxprj** facility, you do not need to explicitly select a translation layer. The build process handles it for you.

For more information about the translation layer, see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

Adding the Socket Driver

Inclusion of the socket driver is automatic for BSPs that provide the driver. By including the core TrueFFS component, **INCLUDE_TFFS**, in VxWorks, the build process checks for a socket driver, **sysTffs.c**, in the BSP directory and includes that file in the system.

If your BSP does not provide a socket driver, follow the procedure described in the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS* for writing a socket driver. To include the socket driver in your system, a working version of the socket driver (**sysTffs.c**) must be located in you BSP directory.

10.3.4 Building the System

Build the system from Workbench or from the command line with **vxprj**.

10.3.5 Formatting the Flash

WARNING: If the flash array for your system is used for a boot image (a boot loader or self-booting VxWorks system) as well as file system media, space must be reserved for the boot image before you format the flash for TrueFFS. For more information, see *10.3.6 Reserving a Region in Flash for a Boot Image*, p.554.

First, boot your system. After the system boots and registers the socket driver(s), start the shell. From the shell, run **tffsDevFormat( )** or **sysTffsFormat( )** to format the flash memory for use with TrueFFS—use the latter if the BSP provides it (**sysTffsFormat( )** performs some setup operations and then calls **tffsDevFormat( )**).

For example, the **tffsDevFormat( )** routine takes two arguments, a drive number and a format argument:

```
tffsDevFormat (int tffsDriveNo, int formatArg);
```

NOTE: You can format the flash medium even if though block device driver has not yet been associated with the flash.

NOTE: The size of the flash media available for use is reduced by one sector for TrueFFS internal use.

Specifying the Drive Number

The first argument for **tffsDevFormat( )**, *tffsDriveNo*, is the drive number (socket driver number), which identifies the flash medium to be formatted. Most systems have a single flash drive, but TrueFFS supports up to five.

The socket registration process determines the drive number. Drive numbers are assigned to the flash devices on the basis of the order in which the socket drivers are registered in **sysTffsInit( )** at boot time. The first registered is drive 0, the second is drive 1, and so on up to 4. Details of this process are described in see the *VxWorks Device Driver Developer's Guide: Flash File System Support with TrueFFS*.

Specifying Format Options

The second argument for **tffsDevFormat( )**, **formatArg**, is a pointer to a **tffsDevFormatParams** structure (cast to an **int**). This structure describes how the volume should be formatted. The **tffsDevFormatParams** structure is defined in *installDir*/**vxworks-6.***x*/**target/h/tffs/tffsDrv.h**:

```
typedef struct
    {
     tffsFormatParams  formatParams;
     unsigned          formatFlags;
    }tffsDevFormatParams;
```

TFFS_STD_FORMAT_PARAMS

To facilitate calling **tffsDevFormat( )** from the shell, you can simply pass zero (or a **NULL** pointer) for the second argument, *formatArg*. Doing makes use of the **TFFS_STD_FORMAT_PARAMS** macro, which defines default values for the **tffsDevFormatParams** structure. The macro defines the default values used in formatting a flash disk device.

Do not use this macro if the flash device is shared with a boot loader. If the BSP provides **sysTffsFormat( )** use that routine instead.

TFFS_STD_FORMAT_PARAMS is defined in **tffsDrv.h** as:

```
#define TFFS_STD_FORMAT_PARAMS {{0, 99, 1, 0x10000l, NULL, {0,0,0,0},
NULL, 2, 0, NULL}, FTL_FORMAT_IF_NEEDED}
```

If the second argument, **formatArg**, is zero, **tffsDevFormat( )** uses the default values from this macro.

The macro passes values for both the first and second members of the **tffsDevFormatParams** structure. These are:

```
formatParams = {0, 99, 1, 0x10000l, NULL, {0,0,0,0}, NULL, 2, 0, NULL}
formatFlags = FTL_FORMAT_IF_NEEDED
```

The meaning of these default values, and other possible arguments for the members of this structure, are described below.

formatParams

The **formatParams** member is of the type **tffsFormatParams**. Both this structure, and the default values used by the **TFFS_STD_FORMAT_PARAMS** macro, are defined in *installDir*/**vxworks-6.***x*/**target/h/tffs/tffsDrv.h**.

If you use the **TFFS_STD_FORMAT_PARAMS** macro, the default values will format the entire flash medium for use with TrueFFS.

If you want to store a boot image in flash, you must change the value of the **bootImageLen** member of the **tffsFormatParams** structure to reserve a region flash for the image, which is separate from that used by TrueFFS (for more information see *10.3.6 Reserving a Region in Flash for a Boot Image*, p.554).

formatFlags

The second member of the **tffsDevFormatParams** structure, **formatFlags**, determines the option used to format the drive. There are several possible values for **formatFlags**, which are listed in Table 10-3.

Table 10-3 **Options for formatFlags**

Macro	Value	Meaning
FTL_FORMAT	1	FAT and FTL formatting
FTL_FORMAT_IF_NEEDED	2	FAT formatting, FTL formatting if needed
NO_FTL_FORMAT	0	FAT formatting only

The default macro **TFFS_STD_FORMAT_PARAMS** passes **FTL_FORMAT_IF_NEEDED** as the value for this argument.

10.3.6 Reserving a Region in Flash for a Boot Image

In order to use flash media for a boot image as well as a file system, a portion of the flash memory (a *fallow region*) must be reserved so that it is excluded from the area subject to formatting and the run-time operations of TrueFFS. Note that the fallow region can be used for purposes other than boot loader code (for example, a system startup log or configuration data).

If TrueFFS is used with the **vxWorks_romResident** VxWorks image type, TrueFFS must be read-only. For information about boot image types, see *3.3 Boot Loader Image Types*, p.133 and *2.4.1 VxWorks Image Types*, p.15.

Reserving a Fallow Region

To create a boot image region, TrueFSS must be configured so that it is excluded from an area of flash that is large enough to accommodate the boot image. If your BSP provides **sysTffsFormat( )**, you can use the routine to do so. Otherwise, you must modify the **tffsFormatParams** structure.

Using sysTffsFormat( )

Some BSPs provide an optional, BSP-specific, helper routine **sysTffsFormat( )**, which can be used to preserve a boot image region.

This routine first sets up a pointer to a **tffsFormatParams** structure that has been initialized with a value for **bootImageLen** (which defines the boot image region); then it calls **tffsDevFormat( )**.

Several BSPs, among them the ads860 BSP, include a **sysTffsFormat( )** routine that reserves 0.5 MB for the boot image. For example:

```
STATUS sysTffsFormat (void)
    {
    STATUS status;
    tffsDevFormatParams params =
        {

#define HALF_FORMAT
/* lower 0.5MB for bootimage, upper 1.5MB for TFFS */

#ifdef  HALF_FORMAT
    {0x80000l, 99, 1, 0x10000l, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#else
    {0x000000l, 99, 1, 0x10000l, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#endif  /* HALF_FORMAT */

    FTL_FORMAT_IF_NEEDED
    };

/* assume that the drive number 0 is SIMM */

    status = tffsDevFormat (0, (int)&params);
    return (status);
    }
```

For more examples of **sysTffsFormat( )** usage, see the socket drivers in *installDir***/vxworks-6.***x***/target/src/drv/tffs/sockets**. If your BSP does not provide a **sysTffsFormat( )** routine, then create a similar routine, or pass the appropriate argument to **tffsDevFormat( )**.

Modifying the tffsFormatParams Structure

If your BSP does not provide **sysTffsFormat( )**, you must modify the **tffsFormatParams** structure to reserve a fallow region before you call **tffsDevFormat( )**.

Change the **bootImageLen** member of the **tffsFormatParams** structure to a value that is at least as large as the boot image. The area defined by **bootImageLen** is excluded from TrueFFS activity (formatting and wear-leveling).

For more information about **bootImageLen** and other members of the structure, see *Specifying Format Options*, p.552.

Writing the Boot Image to Flash

Once you have created a boot image region, you can write the boot image to the flash device using **tffsBootImagePut( )**. This routine bypasses TrueFFS (and its translation layer) and writes directly into any location in flash memory. However, because **tffsBootImagePut( )** relies on a call to **tffsRawio( )**, you cannot use this routine once the TrueFFS volume is mounted.

The arguments to **tffsBootImagePut( )** are the following:

driveNo
The same drive number as the one used as input to the format routine.

offset
Offset from the start of flash at which the image is written (most often specified as zero).

filename
Pointer to the boot image.

WARNING: Because **tffsBootImagePut( )** lets you write directly to any area of flash, it is possible to accidentally overwrite and corrupt the TrueFFS-managed area of flash if you do not specify the parameters correctly. For more information about how to use this utility, see the reference entry for **tffsBootImagePut( )** in the VxWorks API reference.

10.3.7 Mounting the Drive

Use the **usrTffsConfig( )** routine to mount the file system on a TrueFFS flash drive. Its arguments are the following:

drive
Specifies the drive number of the TFFS flash drive; valid values are 0 through the number of socket interfaces in BSP.

removable
Specifies whether the media is removable. Use 0 for non-removable, 1 for removable.

fileName
Specifies the mount point, for example, '**/tffs0/**'.

The following example runs **usrTffsConfig( )** to attach a drive to the file system, and then runs **devs** to list all drivers:

```
-> usrTffsConfig 0,0,"/flashDrive0/"

-> devs
drv     name
0       /null
1       /tyCo/0
1       /tyCo/1
5       host:
6       /vio
2       /flashDrive0/
```

Internally, **usrTffsConfig( )** calls other routines, passing the parameters it is given. Among these routines is **tffsDevCreate( )**, which creates a TrueFFS block device on top of the socket driver. This routine takes, as input, a number (0 through 4) that identifies the socket driver on top of which to construct the TrueFFS block device. The **tffsDevCreate( )** call uses this number as an index into the array of **FLSocket** structures. This number is visible later to the file system as the driver number. The **xbdBlkDevCreateSync( )** routine is then called, which sends a message the file system handler. If the TFFS device has a valid file system installed (dosFS or HRFS) the file system handler mounts the underlying file system. If there is no valid file system on the device, rawfs is instantiated on the device. The user can then format the device with the file system of choice.

After mounting the drive, create the file system (dosFs or HRFS).

10.3.8 Testing the Drive

One way to test your drive is by copying a text file from the host (or from another type of storage medium) to the flash file system on the target. Then, copy the file to the console or to a temporary file for comparison, and verify the content. The following example (using dosFs on TrueFFS) is run from the shell:

```
->@copy "host:/home/myHost/.cshrc" "/flashDrive0/myCshrc"
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
->@copy "/flashDrive0/myCshrc"
...
...
...
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
```

NOTE: The copy command requires the appropriate configuration of dosFs support components. For more information on dosFs, see *8.5.1 Configuring VxWorks for dosFs*, p.478.

10.4 Using TrueFFS Shell Commands

This section illustrates using TrueFFS shell commands (with the C interpreter) to access the flash file system. These examples assume that the flash has been configured into the system in the default configuration for the BSPs shown. For detailed information on creating a system with TrueFFS, *10.2 Overview of Implementation Steps*, p.545 and *10.3 Creating a System with TrueFFS*, p.546.

Target with a Board-Resident Flash Array and a Boot Image Region

This example uses **sysTffsFormat( )** to format board-resident flash, preserving the boot image region. It does not update the boot image, so no call is made to **tffsBootImagePut( )**. Then, it mounts the non-removable RFA medium as drive number 0.

At the shell prompt, type the following commands:

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
```

Target with a Board-Resident Flash Array and a PCMCIA Slot

This example formats RFA and PCMCIA flash for two drives.

The first lines of this example format the board-resident flash by calling the helper routine, **sysTffsFormat( )**, which preserves the boot image region. This example does not update the boot image. It then mounts the drive, numbering it as 0 and passing 0 as the second argument to **usrTffsConfig( )**. Zero is used because RFA is non-removable.

The last lines of the example format PCMCIA flash, passing default format values to **tffsDevFormat( )** for formatting the entire drive. Then, it mounts that drive. Because PCMCIA is removable flash, it passes 1 as the second argument to **usrTffsConfig( )**. (See *10.3.7 Mounting the Drive*, p.556 for details on the arguments to **usrTffsConfig( )**.)

Insert a flash card in the PCMCIA socket. At the shell prompt, type the following commands:

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA1/"
```

Target with a Board-Resident Flash Array and No Boot Image Region Created

This example formats board-resident flash using the default parameters to **tffsDevFormat( )**, as described in *10.3.5 Formatting the Flash, p.552*. Then, it mounts the drive, passing 0 as the drive number and indicating that the flash is non-removable.

At the shell prompt, type the following commands:

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,0,"/RFA/"
```

Target with Two PCMCIA Slots

This example formats PCMCIA flash for two drives. Neither format call preserves a boot image region. Then, it mounts the drives, the first is numbered 0, and the second is numbered 1. PCMCIA is a removable medium.

Insert a flash card in each PCMCIA socket. At the shell prompt, type the following commands:

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,1,"/PCMCIA1/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA2/"
```

10.5 Using TrueFFS With HRFS

This section provides information about implementing TrueFFS with HRFS programmatically, as well as interactively from the kernel shell.

10.5.1 TrueFFS With HRFS Code Example

The following code fragment illustrates the procedure for creating a TrueFFS block device, creating an XBD block device wrapper, and formatting the TrueFFS media for an HRFS file system.

```
/* create block device for the entire disk, */

if ((pBlkDev = tffsDevCreate (0, 0)) == NULL)
{
printErr ("tffsDevCreate failed.");
return (ERROR);
}

/* wrap this block device into an XBD */

xbd = xbdBlkDevCreateSync(pBlkDev, "/tffs0");
printf("Wrap XBD=0x%x

/*
* If the HRFS file system exists already, it will be
* automatically instantiated and we are done.
*/

/*
* But if a file system does not exist or it isn't HRFS
* format it.
*/

hrfsFormat ("/tffs0", 0, 0, 0);
```

For a description of the full set of steps needed to implement HRFS with TrueFFS, see *10.5.2 TrueFFS With HRFS Shell Command Example*, p.560.

10.5.2 TrueFFS With HRFS Shell Command Example

The following steps illustrate the procedure for implementing TrueFFS with HRFS using the kernel shell.

1. Perform a low-level format of the flash device. This step must be performed before you use the flash device for the first time.

   ```
   > sysTffsFormat
   value = 0 = 0x0
   ```

2. Create the TrueFFS block device.

   ```
   > dev = tffsDevCreate (0,0)
   New symbol "dev" added to kernel symbol table.
   dev = 0x1ffb840: value = 33536128 = 0x1ffb880 = dev + 0x40
   ```

3. Create an XBD device wrapper.

   ```
   > xbdBlkDevCreateSync dev, "/tffs"
   Instantiating /tffs:0 as rawFs, device = 0x10001 value = 1 = 0x1
   ```

4. Format the TrueFFS device for HRFS.

   ```
   > hrfsDiskFormat "/tffs:0"
   Formatting /tffs:0 for HRFS v1.2
   Instantiating /tffs:0 as rawFs, device = 0x10001 Formatting...OK.
   value = 0 = 0x0
   ```

   Note that the **hrfsDiskFormat( )** routine is designed for convenient use from the shell; the **hrfsFormat( )** routine is used in code.

5. The TFFS device is now ready for use. List the contents.

   ```
   > ll "/tffs:0"
   Listing Directory /tffs:0:
   drwxrwxrwx 1 0 0 8192 Jan 1 00:05 ./
   drwxrwxrwx 1 0 0 8192 Jan 1 00:05 ../
   value = 0 = 0x0
   ```

After the media and file system and been prepared, for subsequent reboots the procedure is slightly simpler as the flash media does not have to be formatted for TrueFFS and HRFS.

1. Create the TrueFFS block device.

   ```
   > dev = tffsDevCreate (0,0)
   New symbol "dev" added to kernel symbol table.
   dev = 0x461eb8: value = 33535048 = 0x1ffb448
   ```

2. Create an XBD device wrapper.

   ```
   > xbdBlkDevCreateSync dev, "/tffs"
   value = 1 = 0x1
   ```

3. The TFFS device is now ready for use. List the contents.

   ```
   > ll "/tffs:0"
   Listing Directory /tffs:0:
   drwxrwxrwx 1 0 0 8192 Jan 1 00:05 ./
   drwxrwxrwx 1 0 0 8192 Jan 1 00:05 ../
   value = 0 = 0x0
   ```

11
Error Detection and Reporting

11.1 Introduction

VxWorks provides an error detection and reporting facility to help debugging software faults. It does so by recording software exceptions in a specially designated area of memory that is not cleared between warm reboots. The facility also allows for selecting system responses to fatal errors, with alternate strategies for development and deployed systems.

The key features of the error detection and reporting facility are:

- A persistent memory region in RAM used to retain error records across warm reboots.
- Mechanisms for recording various types of error records.
- Error records that provide detailed information about run-time errors and the conditions under which they occur.
- The ability to display error records and clear the error log from the shell.
- Alternative error-handing options for the system's response to fatal errors.
- Macros for implementing error reporting in user code.

The hook routines described in the **edrLib** API reference can be used as the basis for implementing custom functionality for non-RAM storage for error records.

For more information about error detection and reporting routines in addition to that provided in this chapter, see the API reference entries for **edrLib**, **edrShow**, **edrErrLogLib**, and **edrSysDbgLib**.

For information about related facilities, see *6.8 Memory Error Detection*, p.331.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

11.2 Configuring Error Detection and Reporting Facilities

To use the error detection and reporting facilities:

- VxWorks must be configured with the appropriate components.
- A persistent RAM memory region must be configured, and it must be sufficiently large to hold the error records.
- Optionally, users can change the system's default response to fatal errors.

11.2.1 Configuring VxWorks

To use the error detection and reporting facility, the kernel must be configured with the following components:

- **INCLUDE_EDR_PM**
- **INCLUDE_EDR_ERRLOG**
- **INCLUDE_EDR_SHOW**
- **INCLUDE_EDR_SYSDBG_FLAG**

As a convenience, the **BUNDLE_EDR** component bundle may be used to include all of the above components.

11.2.2 Configuring the Persistent Memory Region

The persistent-memory region is an area of RAM at the top of system memory specifically reserved for error records. It is protected by the MMU and the VxWorks **vmLib** facilities. The memory is not cleared by warm reboots, provided a VxWorks 6.*x* boot loader is used.

NOTE: The persistent memory region is not supported for the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and for information about migration, see *15. VxWorks SMP* and *15.15 Migrating Code to VxWorks SMP*, p.702.

A cold reboot always clears the persistent memory region. The **pmInvalidate( )** routine can also be used to explicitly destroy the region (making it unusable) so that it is recreated during the next warm reboot.

The persistent-memory area is write-protected when the target system includes an MMU and VxWorks has been configured with MMU support.

The size of the persistent memory region is defined by the **PM_RESERVED_MEM** configuration parameter. By default the size is set to six pages of memory.

By default, the error detection and reporting facility uses one-half of whatever persistent memory is available. If no other applications require persistent memory, the component may be configured to use almost all of it. This can be accomplished by defining **EDR_ERRLOG_SIZE** to be the size of **PM_RESERVED_MEM** less the size of one page of memory.

If you increase the size of the persistent memory region beyond the default, you must create a new boot loader with the same **PM_RESERVED_MEM** value. The memory area between **RAM_HIGH_ADRS** and **sysMemTop( )** must be big enough to copy the VxWorks boot loader. If it exceeds the **sysMemTop( )** limit, the boot loader may corrupt the area of persistent memory reserved for core dump storage when it loads VxWorks. The boot loader, must therefore be rebuilt with a lower **RAM_HIGH_ADRS** value.

⚠ **WARNING:** If the boot loader is not properly configured (as described above), this could lead into corruption of the persistent memory region when the system boots.

The **EDR_RECORD_SIZE** parameter can be used to change the default size of error records. Note that for performance reasons, all records are necessarily the same size.

The **pmShow( )** shell command (for the C interpreter) can be used to display the amount of allocated and free persistent memory.

For more information about persistent memory, see *6.6 Reserved Memory*, p.328 and the **pmLib** API reference.

⚠ **WARNING:** A VxWorks 6.*x* boot loader must be used to ensure that the persistent memory region is not cleared between warm reboots. Prior versions of the boot loader may clear this area.

11.2.3 Configuring Responses to Fatal Errors

The error detection and reporting facilities provide for two sets of responses to fatal errors. See *11.5 Fatal Error Handling Options*, p.569 for information about these responses, and various ways to select one for a run-time system.

11.3 Error Records

Error records are generated automatically when the system experiences specific kinds of faults. The records are stored in the persistent memory region of RAM in a circular buffer. Newer records overwrite older records when the persistent memory buffer is full.

The records are classified according to two basic criteria:

- event type
- severity level

The event type identifies the context in which the error occurred (during system initialization, or in a process, and so on).

The severity level indicates the seriousness of the error. In the case of fatal errors, the severity level is also associated with alternative system's responses to the error (see *11.5 Fatal Error Handling Options*, p.569).

The event types are defined in Table 11-1, and the severity levels in Table 11-2.

Table 11-1 **Event Types**

Type	Description
INIT	System initialization events.
BOOT	System boot events.
REBOOT	System reboot (warm boot) events.
KERNEL	VxWorks kernel events.
INTERRUPT	Interrupt handler events.
RTP	Process environment events.
USER	Custom events (user defined).

Table 11-2 **Severity Levels**

Severity Level	Description
FATAL	Fatal event.
NONFATAL	Non-fatal event.
WARNING	Warning event.
INFO	Informational event.

The information collected depends on the type of events that occurs. In general, a complete fault record is recorded. For some events, however, portions of the record are excluded for clarity. For example, the record for boot and reboot events exclude the register portion of the record.

Error records hold detailed information about the system at the time of the event. Each record includes the following generic information:

- date and time the record was generated
- type and severity
- operating system version
- task ID
- process ID, if the failing task in a process
- task name
- process name, if the failing task is in a process
- source file and line number where the record was created
- a free form text message

It also optionally includes the following architecture-specific information:

- memory map
- exception information
- processor registers
- disassembly listing (surrounding the faulting address)
- stack trace

11.4 Displaying and Clearing Error Records

The **edrShow** library provides a set of commands for the shell's C interpreter that are used for displaying the error records created since the persistent memory region was last cleared. See Table 11-3.

Table 11-3 **Shell Commands for Displaying Error Records**

Command	Action
edrShow( )	Show all records.
edrFatalShow( )	Show only **FATAL** severity level records.
edrInfoShow( )	Show only **INFO** severity level records.
edrKernelShow( )	Show only **KERNEL** event type records.

Table 11-3 **Shell Commands for Displaying Error Records** (cont'd)

Command	Action
edrRtpShow( )	Show only **RTP** (process) event type records.
edrUserShow( )	Show only **USER** event type records.
edrIntShow( )	Show only **INTERRUPT** event type records.
edrInitShow( )	Show only **INIT** event type records.
edrBootShow( )	Show only **BOOT** event type records.
edrRebootShow( )	Show only **REBOOT** event type records.

The shell's command interpreter provides comparable commands. See the API references for the shell, or use the **help edr** command.

In addition to displaying error records, each of the show commands also displays the following general information about the error log:

- total size of the log
- size of each record
- maximum number of records in the log
- the CPU type
- a count of records missed due to no free records
- the number of active records in the log
- the number of reboots since the log was created

See the **edrShow** API reference for more information.

11.5 Fatal Error Handling Options

In addition to generating error records, the error detection and reporting facility provides for two modes of system response to fatal errors for each event type:

- debug mode, for lab systems (development)
- deployed mode, for production systems (field)

The difference between these modes is in their response to fatal errors in processes (RTP events). In debug mode, a fatal error in a process results in the process being stopped. In deployed mode, as fatal error in a process results in the process being terminated.

The operative error handling mode is determined by the system debug flag (see *11.5.2 Setting the System Debug Flag*, p.571). The default is deployed mode.

Table 11-4 describes the responses in each mode for each of the event types. It also lists the routines that are called when fatal records are created.

The error handling routines are called response to certain fatal errors. Only fatal errors—and no other event types—have handlers associated with them. These handlers are defined in *installDir*/**vxworks-6.***x***/target/config/comps/src/edrStub.c**. Developers can modify the routines in this file to implement different system responses to fatal errors. The names of the routines, however, cannot be changed.

Table 11-4 **FATAL Error-Handling Options**

Event Type	Debug Mode	Deployed Mode (default)	Error Handling Routine
INIT	Reboot	Reboot	**edrInitFatalPolicyHandler( )**
KERNEL	Stop failed task	Stop failed task	**edrKernelFatalPolicyHandler( )**
INTERRUPT	Reboot	Reboot	**edrInterruptFatalPolicyHandler( )**
RTP	Stop process	Delete process	**edrRtpFatalPolicyHandler( )**

Note that when the debugger is attached to the target, it gains control of the system before the error-handling option is invoked, thus allowing the system to be debugged even if the error-handling option calls for a reboot.

11.5.1 Configuring VxWorks with Error Handling Options

In order to provide the option of debug mode error handling for fatal errors, VxWorks must be configured with the **INCLUDE_EDR_SYSDBG_FLAG** component, which it is by default. The component allows a system debug flag to be used to select debug mode, as well as reset to deployed mode (see *11.5.2 Setting the System Debug Flag*, p.571). If **INCLUDE_EDR_SYSDBG_FLAG** is removed from VxWorks, the system defaults to deployed mode (see Table 11-4).

11.5.2 Setting the System Debug Flag

How the error detection and reporting facility responds to fatal errors, beyond merely recording the error, depends on the setting of the system debug flag. When the system is configured with the **INCLUDE_EDR_SYSDBG_FLAG** component, the flag can be used to set the handling of fatal errors to either debug mode or deployed mode (the default).

For systems undergoing development, it is obviously desirable to leave the system in a state that can be more easily debugged; while in deployed systems, the aim is to have them recover as best as possible from fatal errors and continue operation.

The debug flag can be set in any of the following ways:

- Statically, with boot loader configuration.
- Interactively, at boot time.
- Programmatically, using APIs in application code.

When a system boots, the banner displayed on the console displays information about the mode defined by the system debug flag. For example:

```
ED&R Policy Mode: Deployed
```

The modes are identified as **Debug**, **Deployed**, or **Permanently Deployed**. The latter indicates that the **INCLUDE_EDR_SYSDBG_FLAG** component is not included in the system, which means that the mode is deployed and that it cannot be changed to debug.

For more information, see the following sections and the API reference entry for **edrSysDbgLib** (in particular with regard to the **edrSystemDebugModeSet( )** routine).

Setting the Debug Flag Statically

The system can be set to either debug mode or deployed mode with the **f** boot loader parameter when a boot loader is configured and built. The value of 0x000 is used to select deployed mode. The value of 0x400 is used to select debug mode. By default, it is set to deployed mode.

For information about configuring and building boot loaders, see *3.7 Customizing and Building Boot Loaders*, p.146.

Setting the Debug Flag Interactively

To change the system debug flag interactively, stop the system when it boots. Then use the **c** command at the boot-loader command prompt. Change the value of the the **f** parameter: use 0x000 for deployed mode (the default) or to 0x400 for debug mode.

Setting the Debug Flag Programmatically

The state of the system debug flag can also be changed in user code with the **edrSysDbgLib** API.

11.6 Using Error Reporting APIs in Application Code

The **edrLib.h** file provides a set of convenient macros that developers can use in their source code to generate error messages (and responses by the system to fatal errors) under conditions of the developers choosing.

The macros have no effect if VxWorks has not been configured with error detection and reporting facilities. Code, therefore, must not be conditionally compiled to make use of these facilities.

The **edrLib.h** file is in *installDir*/**vxworks-6.***x*/**target/h**.

The following macros are provided:

EDR_USER_INFO_INJECT (trace, msg)
Creates a record in the error log with an event type of USER and a severity of INFO.

EDR_USER_WARNING_INJECT (trace, msg)
Creates a record in the error log with event type of USER and a severity of WARNING.

EDR_USER_FATAL_INJECT (trace, msg)
Creates a record in the error log with event type of USER and a severity of FATAL.

All the macros use the same parameters. The *trace* parameter is a boolean value indicating whether or not a traceback should be generated for the record. The *msg* parameter is a string that is added to the record.

11.7 Sample Error Record

The following is an example of a record generated by a failed kernel task:

```
==[1/1]==============================================================
Severity/Facility:   FATAL/KERNEL
Boot Cycle:          1
OS Version:          6.0.0
Time:                THU JAN 01 05:15:07 1970 (ticks = 1134446)
Task:                "kernelTask" (0x0068c6c8)
Injection Point:     excArchLib.c:2523

fatal kernel task-level exception!

<<<<<Memory Map>>>>>

0x00100000 -> 0x002a48dc: kernel

<<<<<Exception Information>>>>>

data access
Exception current instruction address: 0x002110cc
Machine Status Register: 0x0000b032
Data Access Register: 0x50000000
Condition Register: 0x20000080
Data storage interrupt Register: 0x40000000

<<<<<Registers>>>>>

r0         = 0x00210ff8   sp         = 0x006e0f50   r2         = 0x00000000
r3         = 0x00213a10   r4         = 0x00003032   r5         = 0x00000001
r6         = 0x0068c6c8   r7         = 0x0000003a   r8         = 0x00000000
r9         = 0x00000000   r10        = 0x00000002   r11        = 0x00000002
r12        = 0x0000007f   r13        = 0x00000000   r14        = 0x00000000
r15        = 0x00000000   r16        = 0x00000000   r17        = 0x00000000
r18        = 0x00000000   r19        = 0x00000000   r20        = 0x00000000
r21        = 0x00000000   r22        = 0x00000000   r23        = 0x00000000
r24        = 0x00000000   r25        = 0x00000000   r26        = 0x00000000
r27        = 0x00000000   r28        = 0x00000000   r29        = 0x006e0f74
r30        = 0x00000000   r31        = 0x50000000   msr        = 0x0000b032
lr         = 0x00210ff8   ctr        = 0x0024046c   pc         = 0x002110cc
cr         = 0x20000080   xer        = 0x20000000   pgTblPtr   = 0x00481000
scSrTblPtr = 0x0047fe4c   srTblPtr   = 0x0047fe4c
```

```
<<<<<Disassembly>>>>>

 0x2110ac  2c0b0004    cmpi        crf0,0,r11,0x4 # 4
 0x2110b0  41820024    bc          0xc,2, 0x2110d4 # 0x002110d4
 0x2110b4  2c0b0008    cmpi        crf0,0,r11,0x8 # 8
 0x2110b8  41820030    bc          0xc,2, 0x2110e8 # 0x002110e8
 0x2110bc  4800004c    b           0x211108 # 0x00211108
 0x2110c0  3c600021    lis         r3,0x21 # 33
 0x2110c4  83e1001c    lwz         r31,28(r1)
 0x2110c8  38633a10    addi        r3,r3,0x3a10 # 14864
*0x2110cc  a09f0000    lhz         r4,0(r31)
 0x2110d0  48000048    b           0x211118 # 0x00211118
 0x2110d4  83e1001c    lwz         r31,28(r1)
 0x2110d8  3c600021    lis         r3,0x21 # 33
 0x2110dc  38633a15    addi        r3,r3,0x3a15 # 14869
 0x2110e0  809f0000    lwz         r4,0(r31)
 0x2110e4  48000034    b           0x211118 # 0x00211118
 0x2110e8  83e1001c    lwz         r31,28(r1)

<<<<<Traceback>>>>>

0x0011047c vxTaskEntry  +0x54 : 0x00211244 ()
0x00211258 d            +0x18 : memoryDump ()
```

12
Target Tools

12.1 Introduction

The Wind River host development environment provides tools that reside and execute on the host machine. This approach conserves target memory and resources. However, there are many situations in which it is desirable to make use of target-resident facilities: a target-resident shell, kernel object-module loader, debugging facilities, and system symbol table. The uses for these target-resident tools include the following:

- Debugging a deployed system over a serial connection.
- Developing and debugging network protocols, where it is useful to see the target's view of a network.
- Loading kernel modules from a target disk, from ROMFS, or over the network, and running them interactively (or programmatically).

The target based tools are partially independent of each other. For example, the kernel shell may be used without the kernel object-module loader, and vice versa. However, for any of the other individual tools to be completely functional, the system symbol table is required.

In some situations, it may be useful to use both the host-resident development tools and the target-resident tools at the same time. In this case, additional facilities are required so that both environments maintain consistent views of the system. For more information, see *12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table*, p.623.

For the most part, the target-resident facilities work the same as their host development environment counterparts. For more information, see the appropriate chapters of the *Wind River Workbench User's Guide* and the *VxWorks Command-Line Tools User's Guide*.

This chapter describes the target-resident kernel shell, kernel object-module loader, debugging facilities, and system symbol table. It also provides an overview of the most commonly used VxWorks show routines, which are executed from the shell. In addition, it describes the WDB target agent. WDB is a target-resident, run-time facility required for connecting host tools with a VxWorks target system.

12.2 Kernel Shell

For the most part, the target-resident kernel shell works the same as the host shell (also known as WindSh—for Wind Shell).[1] The kernel shell, however, supports only the C interpreter and command interpreter (see *12.2.2 Kernel and Host Shell Differences*, p.578 for information about other differences).

For detailed information about the host shell and the shell interpreters, see the *Wind River Workbench Host Shell User's Guide*, and the online *Wind River Host Shell API Reference*.

Multiple kernel shell sessions may be run simultaneously, which allows for simultaneous access to the target from the host console and remote connections made with **telnet** or **rlogin**.

NOTE: The kernel shell operates only with **int**, **long**, **short**, **char**, **double**, or **float** data types.

12.2.1 C Interpreter and Command Interpreter

The kernel shell includes both a C interpreter and a command interpreter. Their basic differences are as follows:

- The command interpreter is designed primarily for starting, monitoring, and debugging real-time process (RTP) applications. It can also be used in conjunction with the kernel object module loader to load and unload kernel object modules. It provides a UNIX-like shell environment.

- The C interpreter is designed primarily for monitoring and debugging kernel-based code. It can be used for loading, running, and unloading object modules in conjunction with the kernel object-module loader. In addition, it provides some APIs for starting and monitoring RTP applications. The C interpreter operates on C routines.

For detailed information about the interpreters, see the *Wind River Workbench Host Shell User's Guide*.

For information about the commands supported by each interpreter, see *Interpreter Commands and References*, p.578.

[1] In versions of VxWorks prior to 6.0, the kernel shell was called the target shell. The new name reflects the fact that the target-resident shell runs in the kernel and not in a process.

For information about adding new commands to the command interpreter, and creating interpreters for the kernel shell, see *12.2.19 Adding Custom Commands to the Command Interpreter*, p.594 and *12.2.20 Creating a Custom Interpreter*, p.599.

Switching Between Interpreters

To switch between the shell's C and command interpreters, use the **cmd** command when the C interpreter is active to invoke the command interpreter, and the **C** command when the command interpreter is active to invoke the C interpreter. The following example illustrates switching from the C interpreter to the command interpreter and back again (note the difference in the shell prompt for each interpreter):

```
-> cmd
[vxWorks *]# C
->
```

You can also execute a command from the interpreter that is not active.

Interpreter Commands and References

For information about individual C interpreter routines, see the **usrLib**, **dbgLib**, and **usrShellHistLib** sections in the *VxWorks Kernel API Reference*, as well as entries for the various show routine libraries.

The **dbgLib** routines are particularly useful (for example, semaphores can be created and manipulated from the shell). Note that the kernel shell can also call any C routine that returns a data type supported by the shell (**int**, **long**, **short**, **char**, **double**, or **float**).

For information about the command interpreter commands, see the *VxWorks Kernel Shell Command Reference*.

For information about help available from the kernel shell itself, see *12.2.6 Using Kernel Shell Help*, p.585.

12.2.2 Kernel and Host Shell Differences

The major differences between the target and host shells are:

- The host and kernel shells do not provide exactly the same set of commands. The kernel shell, for example, has commands related to network, shared data, environment variables, and some other facilities that are not provided by the host shell. However, the host and kernel shells provide a very similar set of commands for their command and C interpreters.

- Each shell has its own distinct configuration parameters, as well as those that are common to both.

- Both shells include a command and a C interpreter. The host shell also provides a Tcl interpreter and a gdb interpreter. The gdb interpreter has about 40 commands and is intended for debugging processes (RTPs); and it references host file system paths.

- For the host shell to work, VxWorks must be configured with the WDB target agent component. For the kernel shell to work, VxWorks be configured with the kernel shell component, as well as the target-resident symbol tables component.

- The host shell can perform many control and information functions entirely on the host, without consuming target resources.

- The kernel shell does not require any Wind River host tool support.

- The host shell uses host system resources for most functions, so that it remains segregated from the target. This means that the host shell can operate on the target from the outside, whereas the kernel shell is part of the VxWorks kernel. For example, because the kernel shell task is created with the **taskSpawn( )** **VX_UNBREAKABLE** option, it is not possible to set breakpoints on a function executed within the kernel shell task context. Therefore, the user must create a new task, with **sp( )**, to make breakable calls. For example, from the kernel shell you must do this:

  ```
  -> b printf
  -> sp printf, "Test\n"
  ```

  Whereas from the host shell you can do this:

  ```
  -> b printf
  -> printf ("Test\n")
  ```

  Conflicts in task priority may also occur while using the kernel shell.

WARNING: Shell commands must be used in conformance with the routine prototype, or they may cause the system to hang.

- The kernel shell has its own set of terminal-control characters, unlike the host shell, which inherits its setting from the host window from which it was invoked. (See *12.2.7 Using Kernel Shell Control Characters*, p.586.)

- The kernel shell correctly interprets the tilde operator in pathnames for UNIX and Linux host systems (or remote file systems on a UNIX or Linux host accessed with ftp, rsh, NFS, and so on), whereas the host shell cannot. For example, the following command executed from the kernel shell (with the C interpreter) by user **panloki** would correctly locate the kernel module **/home/panloki/foo.o** on the host system and load it into the kernel:

  ```
  -> ld < ~/foo.o
  ```

- When the kernel shell encounters a string literal ("...") in an expression, it allocates space for the string, including the null-byte string terminator, plus some additional overhead.[2] The value of the literal is the address of the string in the newly allocated storage. For example, the following expression allocates 12-plus bytes from the target memory pool, enters the string in that memory (including the null terminator), and assigns the address of the string to **x**:

  ```
  -> x = "hello there"
  ```

  The following expression can be used to return the memory 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 memory that is never freed:

  ```
  -> printf ("hello there")
  ```

  This is because if strings were only temporarily allocated, and a string literal was passed to a routine being spawned as a task, by the time the task executed and attempted to access the string, the kernel shell would have already released (and possibly even reused) the temporary storage where the string was held.

  If the accumulation of memory used for strings has an adverse effect on performance after extended development sessions with the kernel shell, you can use the **strFree()** routine (with the C interpreter) or the equivalent **string free** command (with the command interpreter).

  The host shell also allocates memory on the target if the string is to be used there. However, it does not allocate memory on the target for commands that can be performed at the host level (such as **lkup( )**, **ld( )**, and so on).

[2] The amount of memory allocated is rounded up to the minimum allocation unit for the architecture in question, plus the amount for the header for that block of memory.

12.2.3 Configuring VxWorks With the Kernel Shell

The functionality of the kernel shell is provided by a suite of components, some of which are required, and others of which are optional.

Required Components

To use the kernel shell, you must configure VxWorks with the **INCLUDE_SHELL** component. The configuration parameters for this component are described in Table 12-1.

You must also configure VxWorks with components for symbol table support, using either the **INCLUDE_STANDALONE_SYM_TBL** or **INCLUDE_NET_SYM_TBL** component. For information about configuring VxWorks with symbol tables, see *12.4.1 Configuring VxWorks with Symbol Tables*, p.618.

Table 12-1 **INCLUDE_SHELL Configuration Parameters**

Configuration Parameter	Description
SHELL_SECURE	Access the kernel shell attached to the console through a login access.
SHELL_STACK_SIZE	Default stack size of kernel shell task.
SHELL_TASK_NAME_BASE	Default *basename* for the kernel shell tasks.
SHELL_TASK_PRIORITY	Priority of the kernel shell tasks.
SHELL_TASK_OPTIONS	Spawning options for the kernel shell tasks.
SHELL_START_AT_BOOT	The kernel shell is launched automatically at boot time on the console.
SHELL_COMPATIBLE	The kernel shell is configured to be compatible with the vxWorks 5.5 shell: one shell session, global I/O redirected, shell task options without the **VX_PRIVATE_ENV** bit.
SHELL_DEFAULT_CONFIG	The default configuration parameters for the kernel shell can be set using this string.
SHELL_FIRST_CONFIG	The configuration parameters for the initial kernel shell session can be set using this string.

Table 12-1 **INCLUDE_SHELL Configuration Parameters**

Configuration Parameter	Description
SHELL_REMOTE_CONFIG	The configuration parameters for the kernel shell sessions started for a remote connection can be set using this string.

Optional Components

Table 12-2 describes components that provide additional shell functionality.

Table 12-2 **Optional Shell Components**

Component	Description
INCLUDE_DEBUG	Debugging facilities, such as disassembly, task stack trace, setting a breakpoint, stepping, and so on.
INCLUDE_SHELL_BANNER	Display the shell banner on startup.
INCLUDE_SHELL_VI_MODE	Editing mode similar to the vi editing mode.
INCLUDE_SHELL_EMACS_MODE	Editing mode similar to the emacs editing mode.
INCLUDE_SHELL_INTERP_C	C interpreter. See *12.2.1 C Interpreter and Command Interpreter*, p.577.
INCLUDE_SHELL_INTERP_CMD	Command interpreter. See *12.2.1 C Interpreter and Command Interpreter*, p.577.
INCLUDE_STARTUP_SCRIPT	Kernel shell startup script facility.
INCLUDE_SHELL_HISTORY_FILE	Shell history commands.

Table 12-3 describes components that provide additional command interpreter functionality. They must be used with the **INCLUDE_SHELL_INTERP_CMD** component (described above in Table 12-2).

Table 12-3 **Command Interpreter Components**

Component	Description
INCLUDE_DISK_UTIL_SHELL_CMD	File system shell commands.
INCLUDE_EDR_SHELL_CMD	Error detection and reporting shell commands.
INCLUDE_TASK_SHELL_CMD	Task shell commands.
INCLUDE_DEBUG_SHELL_CMD	Debug shell commands.
INCLUDE_SYM_SHELL_CMD	Symbol shell commands.
INCLUDE_VM_SHOW_SHELL_CMD	Virtual memory show shell commands.
INCLUDE_ADR_SPACE_SHELL_CMD	Address space shell commands.
INCLUDE_SHARED_DATA_SHOW_SHELL_CMD	Shared data show shell commands.
INCLUDE_MEM_EDR_SHELL_CMD	Memory detection and reporting shell commands
INCLUDE_MEM_EDR_RTP_SHELL_CMD	Memory detection and reporting shell commands for processes (RTPs).
INCLUDE_MODULE_SHELL_CMD	Kernel loader shell command.
INCLUDE_UNLOADER_SHELL_CMD	Kernel unloader shell command.
INCLUDE_SHL_SHELL_CMD	Shared library commands for processes.
INCLUDE_RTP_SHELL_CMD	Process shell commands.
INCLUDE_RTP_SHOW_SHELL_CMD	Process show shell commands.
INCLUDE_HISTORY_FILE_SHELL_CMD	Shell history commands.

Additional components that are useful are the following:

INCLUDE_DISK_UTIL
Provides file utilities, such as **ls** and **cd** (it is required by **INCLUDE_DISK_UTIL_SHELL_CMD**).

INCLUDE_SYM_TBL_SHOW
Provides symbol table show routines, such as **lkup**.

It can also be useful to include components for the kernel object-module loader and unloader (see *12.3.1 Configuring VxWorks with the Kernel Object-Module Loader*, p.604). These components are required for the **usrLib** commands that load modules into, and unload modules from, the kernel (see *12.2.10 Loading and Unloading Kernel Object Modules*, p.587).

Note that the **BUNDLE_STANDALONE_SHELL** and **BUNDLE_NET_SHELL** component bundles are also available to provide for a standalone kernel shell or a networked kernel shell.

12.2.4 Configuring the Kernel Shell

The kernel shell can be configured statically with various VxWorks component parameter options (as part of the configuration and build of the operating stem), as well as configured dynamically from the shell terminal for a shell session.

The default configuration is defined for all shell sessions of the system with the component parameter **SHELL_DEFAULT_CONFIG**. However, the configuration for the initial shell session launched at boot time can be set differently with the **SHELL_FIRST_CONFIG** parameter, and the configuration for remote sessions (telnet or rlogin) can be set with **SHELL_REMOTE_CONFIG**.

Each of these component parameters provide various sets of shell configuration variables that can be set from the command line. These include **INTERPRETER**, **LINE_EDIT_MODE**, **VXE_PATH**, **AUTOLOGOUT**, and so on.

Some of the configuration variables are dependent on the inclusion of other VxWorks components in the operating system. For example, **RTP_CREATE_STOP** is only available if VxWorks is configured with process support and the command interpreter component (**INCLUDE_RTP** and **INCLUDE_SHELL_INTERP_CMD**).

With the C interpreter, **shConfig( )** can be used to reconfigure the shell interactively. Similarly, using the command interpreter, the shell configuration can be displayed and changed with the **set config** command.

Some useful configuration variables are:

INTERPRETER
Identify the interpreter, either **C** or **Cmd**. The default is the first interpreter registered (the C interpreter).

LINE_EDIT_MODE
Set the line edit mode, either emacs or vi. The default is the first line edit mode style registered (vi mode).

LINE_LENGTH
Set the shell line length (it cannot be changed dynamically). The default is 256 characters.

12.2.5 Starting the Kernel Shell

The kernel shell starts automatically after VxWorks boots, by default. If a console window is open over a serial connection, the shell prompt appears after the shell banner.

For information about booting VxWorks, and starting a console window, see the *Wind River Workbench User's Guide: Setting up Your Hardware*.

The shell component parameter **SHELL_START_AT_BOOT** controls if an initial shell session has to be started (**TRUE**) or not (**FALSE**). Default is **TRUE**. If set to **FALSE**, the shell session does not start. It is up to the user to start it either programmatically (from an application), from the host shell, from a telnet or rlogin shell session or from the **wtxConsole** (a host tool). Use **shellInit( )** or **shellGenericInit( )** to start a shell session.

Note that when a user calls a routine from the kernel shell, the routine is executed in the context of the shell task. So that if the routine hangs, the shell session will hang as well.

12.2.6 Using Kernel Shell Help

For either the C or the command interpreter, the **help** command displays the basic set of interpreter commands.

For information about references with detailed information on interpreter commands, see *Interpreter Commands and References*, p.578. Also see the *Wind River Workbench Host Shell User's Guide* for information about interpreter use.

12.2.7 Using Kernel Shell Control Characters

The kernel shell has its own set of terminal-control characters, unlike the host shell, which inherits its setting from the host window from which it was invoked. Table 12-4 lists the kernel 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.394).

Table 12-4 **kernel shell Terminal Control Characters**

Command	Description
CTRL+C	Aborts and restarts the shell. However, if a process is launched with the command interpreter (using **rtp exec**), the function of **CTRL+C** changes. It is used to interrupt the process.
CTRL+D	Logs out when the terminal cursor is at the beginning of a line.
CTRL+H	Deletes a character (backspace).
CTRL+Q	Resumes output.
CTRL+S	Temporarily suspends output.
CTRL+U	Deletes an entire line.
CTRL+W	If a process is launched with the command interpreter (using **rtp exec**), this key sequence suspends the process running in the foreground.
CTRL+X	Reboots (trap to the ROM monitor).
ESC	Toggles between input mode and edit mode (**vi** mode only).

The shell line-editing commands are the same as they are for the host shell. See the **ledLib** API references.

12.2.8 Kernel Shell History

The history of kernel shell activity can be recorded with the **histSave( )** and **histLoad( )** commands for the C interpreter, and the **history save** and **history load** commands for the command interpreter. The commands allow you to save the shell history to, and load it from, a file. The commands are provided by the **INCLUDE_SHELL_HISTORY_FILE** component and by the **INCLUDE_HISTORY_FILE_SHELL_CMD** component.

For more information about these commands, see the **usrShellHistLib** entry in the *VxWorks Kernel API Reference* (for the C interpreter), and the *VxWorks Kernel Shell Command Reference* (for the command interpreter).

12.2.9 Defining Kernel Shell Command Aliases

Aliases can be created for shell commands, as with a UNIX shell. They can be defined programatically using the **shellCmdAliasAdd( )** and **shellCmdAliasArrayAdd( )** routines (see *Sample Custom Commands*, p.599 for examples).

For information about creating command aliases interactively, see the *VxWorks Command-Line Tools User's Guide*.

12.2.10 Loading and Unloading Kernel Object Modules

Kernel object modules can be dynamically loaded into a running VxWorks kernel with the target-resident loader. For information about configuring VxWorks with the loader, and about its use, see *12.3 Kernel Object-Module Loader*, p.603.

NOTE: For information about working with real-time processes from the shell, see the *Wind River Workbench Host Shell User's Guide*, the *VxWorks Application Programmer's Guide: Applications and Processes*, and the online *Wind River Host Shell API Reference*.

The following is a typical load command from the shell, in which the user downloads **appl.o** using the C interpreter:

```
-> ld < /home/panloki/appl.o
```

The **ld( )** command loads an object module from a file, or from standard input into the kernel. External references in the module are resolved during loading.

Once an application module is loaded into target memory, subroutines in the module can be invoked directly from the shell, spawned as tasks, connected to an interrupt, and so on. What can be done with a routine depends on the flags used to download the object module (visibility of global symbols or visibility of all symbols).

Modules can be reloaded with **reld( )**, which unloads the previously loaded module of the same name before loading the new version. Modules can be unloaded with **unld( ).**

For more information about **ld**, see the VxWorks API reference for **usrLib**. For more information about **reld( )** and **unld( )**, see the VxWorks API reference for **unldLib**. Note that these routines are meant for use from the shell only; they cannot be used programmatically.

Undefined symbols can be avoided by loading modules in the appropriate order. Linking independent files before download can be used to avoid unresolved references if there are circular references between them, or if the number of modules is unwieldy. The static linker **ld***arch* can be used to link interdependent files, so that they can only be loaded and unloaded as a unit. (See *Statically Linking Kernel Application Modules*, p.63)

Unloading a code module releases all of the resources used when loading the module, as far as that is possible. This includes removing symbols from the target's symbol table, removing the module from the list of modules loaded in the kernel, removing the text, data, and bss segments from the kernel memory they were stored in, and freeing that memory. It does not include freeing memory or other resources (such as semaphores) that were allocated or created by the module itself while it was loaded.

12.2.11 Debugging with the Kernel Shell

The kernel shell includes task-level debugging utilities for kernel space if VxWorks has been configured with the **INCLUDE_DEBUG** component. For information about the debugging commands available, see the **dgbLib** entry in the *VxWorks Kernel API Reference* for the C interpreter; and see the *VxWorks Kernel Shell Command Reference* for the command interpreter.

Debugging SMP Systems with the Kernel Shell

The kernel shell can be used for task mode debugging of SMP systems, but it cannot be used for system mode debugging. Software breakpoints are always persistent—that is, retained in target memory (for UP systems they are not). Kernel shell debug commands are, however, not affected by persistent software breakpoints. For example, disassembling an address on which a software breakpoint is installed displays the real instruction.

Also note the following with regard to using the kernel shell to debug SMP systems:

- Breakpoint exceptions that occur while holding an ISR-callable spinlock are ignored.
- Breakpoint exceptions that occur while holding a task-only spinlock are ignored.
- Breakpoint exceptions that occur while interrupts are locked are ignored.
- The output of different tasks can be intermingled. For example, creating a task that performs a **printf( )** operation, as follows:

  ```
  -> sp printf,"Hello world."
  Task spawned: id = 0x61707478, name = Hello world.t1
  value = 1634759800 = 0x61707478 = 'x'
  ->
  ```

  The internal shell mechanism prevents intermingling of lines, but the intermingling of output cannot be prevented if other tasks or exception handlers print characters in the midst of a shell output.

For information about the SMP configuration of VxWorks, see *15. VxWorks SMP*.

12.2.12 Aborting Routines Executing from the Kernel Shell

Occasionally it is desirable to abort the shell's evaluation of a statement. For example, an invoked routine can loop excessively, suspend, or wait on a semaphore. This can happen because of errors in the arguments specified in the invocation, errors in the implementation of the routine, or oversight regarding the consequences of calling the routine. In such cases it is usually possible to abort and restart the kernel shell task. This is done by pressing the special target-shell abort character on the keyboard, **CTRL+C** by default. This causes the kernel 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 kernel shell automatically reassigns its system standard input and output streams to the original assignments they had when the kernel shell was first spawned. Thus any kernel shell redirections are canceled, and any executing shell scripts are aborted.

The abort facility works only if the following are true:

- **excTask( )** is running.
- The driver for the particular keyboard device supports it (all VxWorks-supplied drivers do).

Also, you may occasionally enter an expression that causes the kernel 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 kernel shell task, VxWorks automatically restarts the kernel 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 kernel shell.

When the kernel 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 kernel 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 kernel shell is aborted. For instance, the kernel shell might have taken a semaphore, which cannot be given automatically as part of the abort.

12.2.13 Console Login Security

Console login security can be provided for the kernel shell by adding the **INCLUDE_SECURITY** component to the VxWorks configuration. In addition, the shell's **SHELL_SECURE** component parameter must be set to **TRUE** (it is set to **FALSE** by default).

With this configuration, the shell task is not launched at startup. Instead, a login task runs on the console, waiting for the user to enter a valid login ID and password. After validation of the login, the shell task is launched for the console.

When the user logs out from the console, the shell session is terminated, and a new login task is launched.

Also see *Remote Login Security*, p.592.

12.2.14 Using a Remote Login to the Kernel Shell

Users can log into a VxWorks system with **telnet** and **rlogin** and use the kernel shell, provided that VxWorks has been configured with the appropriate components. VxWorks can also be configured with a remote-login security feature that imposes user ID and password constraints on access to the system.

Note that VxWorks does not support **rlogin** access from the VxWorks system to the host.

Remote Login With telnet and rlogin

When VxWorks is first booted, the shell's terminal is normally the system console. You can, however, use **telnet** to access the kernel shell from a host over the network if VxWorks is built with the **INCLUDE_TELNET_CLIENT** component (which can be configured with the **TELNETD_MAX_CLIENTS** parameter). This component creates the **tTelnetd** task when the system boots. It is possible to start several shells for different network connections. (Remote login is also available with the **wtxConsole** tool.)

To access the kernel shell over the network, use the telnet command with the name of the target VxWorks system. For example:

```
% telnet myVxBox
```

UNIX host systems can also use **rlogin** to access to the kernel shell from the host. VxWorks must be configured with the **INCLUDE_RLOGIN** component to create the **tRlogind** task.

To end an **rlogin** connection to the shell, you can do any of the following:

- Use the **CTRL+D** key combination.
- Use the **logout( )** command with the shell's C interpreter, or the **logout** command with the command interpreter.
- Type the tilde and period characters at the shell prompt:

```
-> ~.
```

Remote Login Security

VxWorks can be configured with a remote-login security feature that imposes user ID and password constraints on access to the system. The **INCLUDE_SECURITY** component provides this facility.

Note that **loginEncryptInstall( )** allows for use of other encryption routines (such as SHA512).

A user is then prompted for a login user name and password when accessing the VxWorks target remotely. The default login user name and password provided with the supplied system image is *target* and *password*.

The default user name and password can be changed with the **loginUserAdd( )** routine, as follows:

```
-> loginUserAdd "fred", "encrypted_password"
```

The default user name and password can be changed with **loginUserAdd( )**, which requires an encrypted password. To create an encrypted password, use the **vxencrypt** tool on the host system. The tool prompts you to enter a password, and then displays the encrypted version. The user name and password can then be changed with the **loginUserAdd( )** command with the shell's C interpreter. For example, **mysecret** is encrypted as **bee9QdRzs**, and can be used with the user name fred as follows to change the default settings:

```
-> loginUserAdd "fred", " bee9QdRzs"
```

To define a group of login names, include a list of **loginUserAdd( )** calls in a startup script and run the script after the system has been booted. Or include the **loginUserAdd( )** calls in **usrAppInit( )**; for information in this regard, see *2.6.10 Configuring VxWorks to Run Applications Automatically*, p.66.

NOTE: 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 *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

The remote-login security feature can be disabled at boot time by specifying the flag bit 0x20 (**SYSFLAG_NO_SECURITY**) in the *flags* boot parameter.

Also see *12.2.13 Console Login Security*, p.590.

12.2.15 Launching a Shell Script Programmatically

A simple way to have a script executed programmatically by an interpreter (the command interpreter for example) is as follows:

```
fdScript = open ("myScript", O_RDONLY);
shellGenericInit ("INTERPRETER=Cmd", 0, NULL, &shellTaskName, FALSE, FALSE,
fdScript, STD_OUT, STD_ERR); do
     taskDelay (sysClkRateGet ());
while (taskNameToId (shellTaskName) != ERROR); close (fdScript);
```

The do/while loop is necessary for waiting for the shell script to terminate.

12.2.16 Executing Shell Commands Programmatically

There is no **system( )** API as there is for a UNIX operating system. In order to be able to execute shell commands from an application, the same technique as described above can be used. It is not a file handle that is passed to the **shellGenericInit( )** API, but a pseudo-device slave file descriptor (see the API reference for the **ptyDrv** library for information about pseudo-devices).

The application writes the commands it wants to be executed into the master side of the pseudo-device. A pseudo-code representation of this might be as follows:

```
fdSlave = open ("system.S", O_RDWR);
fdMaster = open ("system.M", O_RDWR);
shellGenericInit ("INTERPRETER=Cmd", 0, NULL, &shellTaskName, FALSE, FALSE,
                  fdSlave, STD_OUT, STD_ERR);
taskDelay (sysClkRateGet ());
write (fdMaster, "pwd\n", 4);
close (fdMaster);
close (fdSlave);
```

12.2.17 Accessing Kernel Shell Data Programmatically

Shell data is available with the **shellDataLib** library. This allows the user to associate data values with a shell session (uniquely per shell session), and to access them at any time. This is useful to maintain default values, such as the memory dump width, the disassemble length, and so on. These data values are not accessible interactively from the shell, only programatically.

12.2.18 Using Kernel Shell Configuration Variables

Shell configuration variables are available using the **shellConfigLib** library. This allows the user to define default configurations for commands or for the shell itself. Such variables already exist for the shell (see the configuration variables **RTP_CREATE_STOP** or **LINE_EDIT_MODE**). They behave similarly to environment strings in a UNIX shell. These variables can be common to all shell sessions, or local to a shell session. They can be modified and displayed interactively by the shell user with the command **set config** or the shell routine **shConfig( )**.

12.2.19 Adding Custom Commands to the Command Interpreter

The kernel shell's command interpreter consists of a line parser and of a set of commands. It can be extended with the addition of custom commands written in C. (The host shell's command interpreter can likewise be extended, but with commands written in Tcl.)

The syntax of a command statement is standard shell command-line syntax, similar to that used with the UNIX **sh** shell or the Windows **cmd** shell. The syntax is:

command [*options*] [*arguments*]

Blank characters (such as a space or tab) are valid word separators within a statement. A blank character can be used within an argument string if it is escaped (that is, prefixed with the back-slash character) or if the argument is quoted with double quote characters. The semicolon character is used as a command separator, used for entering multiple commands in a single input line. To be used as part of an argument string, a semicolon must escaped or quoted.

The command parser splits the statement string into a command name string and a string that consists of the options and the arguments. These options and arguments are then passed to the command routine.

The command name may be a simple name (one word, such as **reboot**) or a composite name (several words, such as **task info**). Composite names are useful for creating classes of commands (commands for tasks, commands for processes, and so on).

Options do not need to follow any strict format, but the standard UNIX option format is recommended because it is handled automatically by the command parser. If the options do not follow the standard UNIX format, the command routine must parse the command strings to extract options and arguments. See *Defining a Command*, p.596 for more information.

The standard UNIX option format string is:

–character [*extra option argument*]

The special double-dash option (**--**) is used in the same way as in UNIX. That is, all elements that follow it are treated as an argument string, and not options. For example, if the command **test** that accepts option **-a**, **-b** and **-f** (the latter with an extra option argument), then the following command sets the three options, and passes the **arg** string as an argument:

```
test -a -b -f arg
```

However, the next command only sets the **-a** option. Because they follow the double-dash, the **-b**, **-f** and **arg** elements of the command are passed to the C routine of the **test** command as strings:

```
test -a -- -b -f file arg
```

The command interpreter only handles strings. As a consequence, the arguments of a command routine are strings as well. It is up to the command routine to transform the strings into numerical values if necessary.

For information about symbol access syntax, see the material on the command interpreter in the *VxWorks Command-Line Tools User's Guide*.

Creating A New Command

The command interpreter is designed to allow customers to add their own commands to the kernel shell.

Commands are stored in an internal database of the command interpreter. The information describing a command are defined by a C structure that contains:

- The command name.
- A pointer to the C routine for the command.
- The command options string (if needed).
- A short description of the command.
- A full description of the command.
- A usage synopsis.

The command descriptions and the synopsis are used by the **help** command.

A command is registered with the command interpreter database along with a topic name. The topic is used by the **help** command to display related commands. For example, to display all commands related to the memory, you would use the command **help memory**.

This section describes the conventions used for creating commands and provides information about examples of commands that can serve as models. Also see the **shellInterpCmdLib** API reference.

It may also be useful to review the other shell documentation in order to facilitate the task of writing new commands. See the **shellLib**, **shellDataLib**, and **shellConfigLib** API references, as well as the material on the command interpreter in the *VxWorks Command-Line Tools User's Guide*.

Defining a Command

A command is defined for the interpreter by a C structure composed of various strings and a function pointer, with the following elements:

```
nameStruct = {"cmdFullname",
     func,
     "opt",
     "shortDesc",
     "fullDesc",
     "%s [synopsis]"};
```

The string *cmdFullname* is the name of the command. It may be a composite name, such as **foo bar**. In this case, **foo** is the top-level command name, and **bar** is a sub-command of **foo**. The command name must be unique.

The *func* element is the name of the routine to call for that command name.

The string *opt* can be used in several different ways to define how option input is passed to the command routine:

- If *opt* is not **NULL**, it describes the possible options that the command accepts. Each option is represented as a single character (note that the parser is case sensitive). If an option takes an argument, a colon character (**:**) must be added after the option character. For example, the following means that the command accepts the **-a**, **-v**, and **-f** *arg* options:

  ```
  avf:
  ```

- If *opt* not **NULL**, and consists only of a semicolon, the option input is passed to the command routine as a single string. It is up to the routine to extract options and arguments.

- If *opt* is **NULL**, the parser splits the input line into tokens and passes them as traditional argc/argv parameters to the command routine.

Note that the command routine must be coded in a manner appropriate to how the *opt* string is used in the command-definition structure (see *Writing the Command Routine*, p.597).

The string *shortDesc* is a short description of the command. A sequence of string conversion characters (**%s**) within that string is replaced by the command name when the description is displayed. The string should not be ended by a new-line character (**\n**).

The string *fullDesc* is the full description of the command. A sequence of string conversion characters (**%s**) within that string is replaced by the command name when the description is displayed. This description should contain the explanation of the command options. The string should not be ended by newline (**\n**) character.

The string *synopsis* is the synopsis of the command. A sequence of string conversion characters (**%s**) within that string is replaced by the command name when the synopsis is displayed. The string should not be ended by a newline (**\n**) character.

The description and synopsis strings are used by the command interpreter's **help** command.

The rules for the C language command routines associated with the command structures are described in *Writing the Command Routine*, p.597. See *Sample Custom Commands*, p.599 for examples.

Writing the Command Routine

This section describes how to write the C routine for a command, including how the routine should handle command options.

The command definition structure and the command routine must be coordinated, most obviously with regard to the command routine name, but also with regard to the *opt* element of the structure defining the command:

- If the *opt* element is not equal to **NULL**, the declaration of the routine must include the **options** array:

```
int func
    (
    SHELL_OPTION options[]    /* options array */
    ...
    )
```

  In this declaration, **options** is a pointer on the argument array of the command.

- However, if the *opt* element is **NULL**, the declaration of the routine must include argc/argv elements, as follow:

```
int func
     (
     int        argc,        /* number of argument */
     char **    arcv         /* pointer on the array of arguments */
     ...
     )
```

In this declaration, *argc* is the number of arguments of the command, and *argv* is an array that contains the argument strings.

In the first case the parser populates the **options[]** array.

In the second case it splits and passes the arguments as strings using argc/argv to the routine.

When the *opt* element is used to define options, the order in which they are listed is significant, because that is the order in which they populate the **options[]** array. For example, if *opt* is **avf**:

- Option **a** is described by the first cell of the options array: **options[0]**.
- Option **v** is described by the second cell of the options array: **options[1]**.
- Option **f** is described by the third cell of the options array: **options[2]**.
- The argument of option **f** is **options[2].string**.

Each cell of the options array passed by the parser to *func* is composed of a boolean value (TRUE if the option is set, FALSE if not) and a pointer to a string (pointer to an argument), if so defined. For example, if **-a** has been defined, the value of **options[0].isSet** value is **TRUE**. Otherwise it is **FALSE**.

A boolean value indicates if it is the last cell of the array. If the option string *opt* is only a colon, the argument string of the command is passed to *func* without any processing, into the **string** field of the first element of the **options** array.

The return value of the command routine is an integer. By convention, a return value of zero means that the command has run without error. Any other value indicates an error value.

See *Defining a Command*, p.596 for information about the command-definition structure. See *Sample Custom Commands*, p.599 for examples of command structures and routines.

Registering a New Command

The shell commands have to be registered against the shell interpreter. This can be done at anytime; the shell component does not need to be initialized before commands can be registered.

A command is registered in a topic section. The topic name and the topic description must also be registered in the command interpreter database. The routine used to do so is **shellCmdTopicAdd( )**. This routine accepts two parameters: a unique topic name and a topic description. The topic name and description are displayed by the help command.

Two routines are used to register commands: **shellCmdAdd( )** adds a single command, and **shellCmdArrayAdd( )** adds an array of commands.

See *Sample Custom Commands*, p.599 for information about code that illustrates command registration.

Sample Custom Commands

For an example of custom command code, see *installDir***/vxworks-6.***x***/target/src/demo/shell/tutorialShellCmd.c**. In addition to illustrating how to implement custom commands, it shows how to create command aliases (also see *12.2.9 Defining Kernel Shell Command Aliases*, p.587).

The code can be used with Wind River Workbench as a downloadable kernel module project, or included in a kernel project. For information about using Workbench, see the *Wind River Workbench User's Guide*.

It can also be built from the command line with the command **make CPU=***cpuType*. For example:

```
make CPU=PENTIUM2
```

This resulting module can then be loaded into the kernel, using Workbench (including the host shell) or kernel shell.

The **tutorialShellCmdInit( )** routine must be called to register the commands before they can be executed, regardless of how the code is implemented.

12.2.20 Creating a Custom Interpreter

The kernel shell is designed to allow customers to add their own interpreter. Two interpreters are provided by Wind River: the C interpreter and the command interpreter. An interpreter receives the user input line from the shell, validates the input against its syntax and grammar, and performs the action specified by the input line (such as redirection, calling a VxWorks function, reading or writing memory, or any other function that an interpreter might perform).

Within the shell, an interpreter is defined by a set of static information and an interpreter context.

The static information about the interpreter is:

- an interpreter context initialization routine
- a parsing routine
- an evaluation routine
- a completion routine
- a restart routine
- an interpreter context finalization routine (to release any resources)
- an interpreter name (which must be unique)
- a default interpreter prompt

This information is set when the interpreter is registered with the shell using the **shellInterpRegister( )** routine.

The interpreter context is created the first time the interpreter is used. It is unique to each interpreter and to each shell session. When it has been created, the shell calls the interpreter context initialization routine so that the interpreter can add any private data that it requires to that context. The **pInterpParam** field of the interpreter's context structure **SHELL_INTERP_CTX** can be used for that purpose.

Each time a line is entered from terminal or read from a script, the shell calls the parsing routine of the current interpreter so that it can evaluate the line.

The arguments to the parsing routine are the interpreter context, a pointer to the input line string, and a boolean value indicating whether or not the shell session is interactive.

The job of the parsing routine is to split the line into meaningful words, according to the interpreter's syntax, and to perform the appropriate actions.

The evaluation routine is called by the shell whenever an evaluation for that interpreter is required, using the **shellInterpEvaluate( )** routine. Usually, the evaluation routine and the parsing routine can share most of their code. The evaluation routine returns an evaluation value.

The completion routine is called by the shell whenever a completion key is hit (completion keys are defined by the line editing mode of the shell; see the **ledLib** API reference). It is up to the interpreter to perform word completion according to its syntax, and to the position of the cursor in the line.

It is up to the interpreter to release its resource whenever a shell session terminates or is restarted. The *finalize* and *restart* routines specified at registration time are called by the shell for this purpose.

The restart routine is called by the shell whenever the shell task is restarted, either because it has taken an exception or it was restarted by the user. This routine is useful for releasing any resources reserved by the interpreter (memory blocks, semaphores, and so on).

The context-finalize routine is called when the shell session is terminated. It is used to free any resources allocated by the interpreter.

Note that the stream redirection characters must be handled by the interpreter itself (for example, the command interpreter's **<**, **>** and **>>** redirection characters).

The interpreter name is a string that uniquely identifies the interpreter among any others registered with the shell.

The default interpreter prompt is the string that identifies the interpreter visually. For example, the C interpreter prompt is the string:

->

The interpreter prompt may contain format conversion characters that are dynamically replaced when printed by another string. For example **%/** is replaced by the current shell session path, **%n** is replaced by the user name, and so on (see the **shellPromptLib** API reference for more information). Moreover, it is possible to add new format strings with **shellPromptFmtStrAdd( )**. For example the command-related process adds the format string **%c** to display the name of the current working memory context.

The current interpreter is defined by the shell configuration variable named **INTERPRETER** (the C macro **SHELL_CFG_INTERP** is defined for it in **shellLib.h**).

A shell user can switch to is own interpreter by setting the value of the configuration variable **INTERPRETER** to its interpreter name. This can be done either interactively or programmatically. For illustrative purposes, the following commands change the interpreter from C to command and back again at the shell command line:

```
-> shConfig "INTERPRETER=Cmd"
[vxWorks]# set config INTERPRETER=C
->
```

It is also possible to created custom commands to allow for switching to and from a custom interpreter, similar to those used to switch between the command and C interpreters (**C** is used with the command interpreter, and **cmd** with the C interpreter to switch between the two).

The following code fragment sets the command interpreter for the current session:

```
shellConfigValueSet (CURRENT_SHELL_SESSION, SHELL_CFG_INTERP, "Cmd");
```

For more information, see the **set config** command and the **shellConfigValueSet( )** routine in the **shellConfigLib** API reference.

Sample Custom Interpreter

For an example of a interpreter code, see: *installDir*/**vxworks-6.***x*/**target/src/demo/shell/shellInterpDemo.c**. The sample interpreter is called **DEMO**, and it has only six commands:

- **task** to create a task
- **list** to list the tasks
- **kill** to destroy a task
- **dump** to dump the memory contents
- **sym** to access symbol
- **C** to switch back to the C interpreter

The **DEMO** code illustrates how to create a small interpreter with a few commands, and how to make use of some of the routines exported by the shell (from the **shellDataLib** and **shellConfigLib** libraries).

To use this interpreter in the shell, you can build and download it as a downloadable kernel module. To register the commands, the initialization function **shellInterpDemoInit( )** has to be called first. You can load it, register it, and set it for the current session as follows:

```
-> ld < shellInterpDemo.o
-> shellInterpRegister (shellInterpDemoInit)
-> shConfig "INTERPRETER=DEMO"
DEMO #
```

If you choose to link the module with VxWorks instead of downloading it, you have to make the **shellInterpDemoInit( )** call in the user startup code (see *2.6.8 Linking Kernel Application Object Modules with VxWorks*, p.64 and *2.6.10 Configuring VxWorks to Run Applications Automatically*, p.66).

The code can be used with Wind River Workbench as a downloadable kernel module project, or included in a kernel project. For information about using Workbench, see the *Wind River Workbench User's Guide*.

12.3 Kernel Object-Module Loader

The target-resident, VxWorks kernel object-module loader lets you add object modules to the kernel at run-time. This operation, called *loading*, or *downloading*, allows you to install kernel-space applications or to extend the operating system itself. (For the sake of brevity, the kernel object-module loader is also referred to simply as the kernel loader, or loader, in this section).

The downloaded code can be a set of routines, meant to be used by some other code (the equivalent of a library in other operating systems), or it can be an application, meant to be executed by a task or a set of tasks. The units of code that can be downloaded are referred to as object modules.

The ability to load individual object modules brings significant flexibility to the development process, in several different ways. The primary use of this facility during development is to unload, recompile, and reload object modules under development. The alternative is to link the developed code into the VxWorks image, to rebuild this image, and to reboot the target, every time the development code must be recompiled.

The kernel loader also enables you to dynamically extend the operating system, since once code is loaded, there is no distinction between that code and the code that was compiled into the image that booted.

Finally, you can configure the kernel loader to optionally handle memory allocation, on a per-load basis, for modules that are downloaded. This allows flexible use of the target's memory. The loader can either dynamically allocate memory for downloaded code, and free that memory when the module is unloaded; or, the caller can specify the addresses of memory that has already been allocated. This allows the user more control over the layout of code in memory. For more information, see *12.3.5 Specifying Memory Locations for Loading Objects*, p. 610.

The functionality of the kernel loader is provided by two components: the loader proper, which installs the contents of object modules in the target system's memory; and the unloader, which uninstalls object modules. In addition, the loader relies on information provided by the system symbol table.

⚠ **CAUTION:** Do not unload an object module while its tasks are running. Doing so may result in unpredictable behavior.

NOTE: The target-resident kernel object-module loader is sometimes confused with the boot loader, which is used to load the kernel image into memory. Although these two tools perform similar functions, and share some support code, they are separate entities. The boot loader loads only complete system images, and does not perform relocations. See *3. Boot Loader*.

12.3.1 Configuring VxWorks with the Kernel Object-Module Loader

By default, the kernel object-module loader is not included in VxWorks. To use the loader, you must configure VxWorks with the **INCLUDE_LOADER** component.

Adding the **INCLUDE_LOADER** component automatically includes several other components that together provide complete loader functionality. These components are:

INCLUDE_UNLOADER
Provides facilities for unloading object modules.

INCLUDE_MODULE_MANAGER
Provides facilities for managing loaded modules and obtaining information about them. For more information, see the VxWorks API reference for **moduleLib**.

INCLUDE_SYM_TBL
Provides facilities for storing and retrieving symbols. For more information, see *12.4 Kernel Symbol Tables*, p.617 and the VxWorks API reference for **symLib**.

INCLUDE_SYM_TBL_INIT
Specifies a method for initializing the system symbol table.

⚠ **CAUTION:** If you want to use the target-resident symbol tables and kernel object-module loader in addition to the host tools, you must configure VxWorks with the **INCLUDE_WDB_MDL_SYM_SYNC** component to provide host-target symbol table and module synchronization. This component is included by default when both the kernel loader and WDB agent are included in VxWorks. For more information, see *12.4.4 Using the VxWorks System Symbol Table*, p.622.

The kernel loader and unloader are discussed further in subsequent sections, and in the VxWorks API references for **loadLib** and **unldLib**.

12.3.2 Kernel Object-Module Loader API

The API routines, shell C interpreter commands, and shell command interpreter commands available for loading and unloading kernel modules are described in Table 12-5 and Table 12-6.

Note that the kernel loader routines can be called directly from the C interpreter or from code. The shell commands, however, should only be called from the shell and not from within programs.[3] In general, shell commands handle auxiliary operations, such as opening and closing a file; they also print their results and any error messages to the console.

Table 12-5 **Routines for Loading and Unloading Object Modules**

Routine	Description
loadModule( )	Loads an object module.
loadModuleAt( )	Loads an object module into a specific memory location.
unldByModuleId( )	Unloads an object module by specifying a module ID.
unldByNameAndPath( )	Unloads an object module by specifying name and path.
unldByGroup( )	Unloads an object module by specifying its group.

Table 12-6 **Shell C Interpreter Commands for Object Modules**

Command	Description
ld( )	Loads an object module into kernel memory.
reld( )	Unloads and reloads an object module (specified by filename or module ID).
unld( )	Unloads an object module (specified by filename or module ID) from kernel memory.

The use of some of these routines and commands is discussed in the following sections.

[3] In future releases, calling shell commands programmatically may not be supported.

For detailed information, see the **loadLib**, **unldLib**, and **usrLib** API references, the shell command reference, as well as *12.3.3 Summary List of Kernel Object-Module Loader Options*, p.606.

12.3.3 Summary List of Kernel Object-Module Loader Options

The kernel loader's behavior can be controlled using load flags passed to **loadLib** and **unldLib** routines. Many of these flags can be combined (using a logical **OR** operation); some are mutually exclusive. The tables in this section group these options by category.

Table 12-7 **Kernel Loader and Unloader Options for C++**

Option	Hex Value	Description
LOAD_CPLUS_XTOR_AUTO	0x1000	Call C++ constructors on loading.
LOAD_CPLUS_XTOR_MANUAL	0x2000	Do not call C++ constructors on loading.
UNLD_CPLUS_XTOR_AUTO	0x20	Call C++ destructors on unloading.
UNLD_CPLUS_XTOR_MANUAL	0x40	Do not call C++ destructors on unloading. If this option is used, be sure that the destructor routines are not used to release resources back to the system (such as memory or semaphores); or the caller may first run any static destructors by calling **cplusDtors( )**.

Table 12-8 **Kernel Loader Options for Symbol Registration**

Option	Hex Value	Description
LOAD_NO_SYMBOLS	0x2	No symbols from the module are registered in the system's symbol table. Consequently, linkage against the code module is not possible. This option is useful for deployed systems, when the module is not supposed to be used in subsequent link operations.

Table 12-8 **Kernel Loader Options for Symbol Registration** (cont'd)

Option	Hex Value	Description
LOAD_LOCAL_SYMBOLS	0x4	Only local (private) symbols from the module are registered in the system's symbol table. No linkage is possible against this code module's public symbols. This option is not very useful by it self, but is one of the base options for **LOAD_ALL_SYMBOLS**.
LOAD_GLOBAL_SYMBOLS	0x8	Only global (public) symbols from the module are registered in the system's symbol table. No linkage is possible against this code module's private symbols. This is the kernel loader's default when the **loadFlags** parameter is left as **NULL**.
LOAD_ALL_SYMBOLS	0xC	Local and global symbols from the module are registered in the system's symbol table. This option is useful for debugging.

Table 12-9 **Kernel Loader Options for Code Module Visibility**

Option	Hex Value	Description
HIDDEN_MODULE	0x10	The code module is not visible from the **moduleShow( )** routine or the host tools. This is useful on deployed systems when an automatically loaded module should not be detectable by the user. It only affects user visibility, and does not affect linking with other modules.

Table 12-10 **Kernel Unloader Options for Breakpoints and Hooks**

Option	Hex Value	Description
UNLD_KEEP_BREAKPOINTS	0x1	The breakpoints are left in place when the code module is unloaded. This is useful for debugging, as all breakpoints are otherwise removed from the system when a module is unloaded.

Table 12-10 **Kernel Unloader Options for Breakpoints and Hooks** (cont'd)

Option	Hex Value	Description
UNLD_FORCE	0x2	By default, the kernel unloader does not remove the text sections when they are used by hooks in the system. This option forces the unloader to remove the sections anyway, at the risk of unpredictable results.

Table 12-11 **Kernel Loader Options for Resolving Weak Symbols**

Option	Hex Value	Description
LOAD_WEAK_MATCH_ALL	0x20000	A weak symbol is ignored if there is an existing global symbol with the same name. If none is found, the weak symbol is registered as a global. This is the default behavior.
LOAD_WEAK_MATCH_NONE	0x10000	The loader always registers the weak symbol as a global, regardless of any existing definition. This option provides the behavior of the VxWorks 5.*x* loader.

Table 12-12 **Kernel Loader Options for Resolving Common Symbols**

Option	Hex Value	Description
LOAD_COMMON_MATCH_NONE	0x100	This option prevents any matching with already-existing symbols. Common symbols are added to the symbol table unless **LOAD_NO_SYMBOLS** is set. This is the default option. (Note that this option is only in effect for the specific load operation in which it is used; that is, it has no effect on subsequent load operations that do not use it.)
LOAD_COMMON_MATCH_USER	0x200	Seeks a matching symbol in the system symbol table, but considers only symbols in user modules, not symbols that were in the original booted image. If no matching symbol exists, this option behaves like **LOAD_COMMON_MATCH_NONE**.
LOAD_COMMON_MATCH_ALL	0x400	Seeks a matching symbol in the system symbol table, considering all symbols. If no matching symbol exists, this option behaves like **LOAD_COMMON_MATCH_NONE**.

If several matching symbols exist for the options **LOAD_COMMON_MATCH_USER** and **LOAD_COMMON_MATCH_ALL**, the symbol most recently added to the symbol table is used.

Table 12-13 **Fully-Linked Module Load Support**

Option	Hex Value	Description
LOAD_FULLY_LINKED	0x20	Provides for loading fully linked modules (that is, modules without any unresolved symbols or relocations).

Note that symbol tables are not required when VxWorks is configured with support for loading fully-linked object modules (the option is listed in Table 12-13). For more information, see **loadModuleAt( )** in the *VxWorks Kernel API Reference*.

12.3.4 Loading C++ Modules into the Kernel

For information about loading C++ modules from the shell, see *13.4 Using C++ in Signal Handlers and ISRs*, p.649. Also see *12.3.3 Summary List of Kernel Object-Module Loader Options*, p.606 for C++ kernel loader and unloader options.

12.3.5 Specifying Memory Locations for Loading Objects

By default, the kernel object-module loader allocates the memory necessary to hold a code module. It is also possible to specify where in memory any or all of the **text**, **data**, and **bss** segments of an object module should be installed using the **loadModuleAt( )** command. If an address is specified for a segment, then the caller must allocate sufficient space for the segment at that address before calling the load routine. If no addresses are specified, the kernel loader allocates one contiguous area of memory for all three of the segments.

For any segment that does not have an address specified, the loader allocates the memory (using **memPartAlloc( )** or, for aligned memory, using **memalign( )**). The base address can also be set to the value **LOAD_NO_ADDRESS**, in which case the loader replaces the **LOAD_NO_ADDRESS** value with the actual base address of the segment once the segment is installed in memory.

The basic unit of information in a relocatable ELF object file is a section. In order to minimize memory fragmentation, the loader gathers sections so that they form the logical equivalent of an ELF segment. For simplicity, these groups of sections are also referred to as segments. For more information, see *ELF Object Module Format*, p.612).

The kernel loader creates three segments: **text**, **data**, and **bss**. When gathering sections together to form segments, the sections are placed into the segments in the same order in which they occur in the ELF file. It is sometimes necessary to add extra space between sections to satisfy the alignment requirements of all of the sections. When allocating space for one or more segments, care must be taken to ensure that there is enough space to permit all of the sections to be aligned properly. (The alignment requirement of a section is given as part of the section description in the ELF format. The binary utilities **readelf***arch* and **objdump***arch* can be used to obtain the alignment information.)

In addition, the amount of padding required between sections depends on the alignment of the base address. To ensure that there will be enough space without knowing the base address in advance, allocate the block of memory so that it is aligned to the maximum alignment requirement of any section in the segment. So, for instance, if the data segment contains sections requiring 128 and 264 byte alignment, in that order, allocate memory aligned on 264 bytes.

The kernel unloader can remove the segments from wherever they were installed, so no special instructions are required to unload modules that were initially loaded at specific addresses. However, if the base address was specified in the call to the loader, then, as part of the unload, unloader does not free the memory area used to hold the segment. This allocation was performed by the caller, and the de-allocation must be as well.

12.3.6 Guidelines and Caveats for Kernel Object-Module Loader Use

The following sections describe the criteria used to load modules and issues with loading that may need to be taken into account.

Relocatable Object Files

Relocatable object files are used for modules that can be dynamically loaded into the VxWorks kernel and run. In contrast to an executable file, which is fully linked and ready to run at a specified address, a relocatable file is an object file for which **text** and **data** sections are in a transitory form, meaning that some addresses are not yet known. Relocatable object modules are generated by the compiler with **.o** extension (similar to the ones produced as an intermediate step between the application source files—**.c**, **.s**, **.cpp**— and the corresponding executable files that run in VxWorks processes).

Relocatable files are used for downloadable modules because the layout of the VxWorks image and downloaded code in memory are not available to a compiler running on a host machine. Therefore, the code handled by the target-resident kernel loader must be in relocatable form, rather than an executable. The loader itself performs some of the same tasks as a traditional linker in that it prepares the code and data of an object module for the execution environment. This includes the linkage of the module's code and data to other code and data.

Once installed in the system's memory, the entity composed of the object module's code, data, and symbols is called a code module. For information about installed code modules, see the VxWorks API reference for **moduleLib**.

ELF Object Module Format

An relocatable ELF object file is essentially composed of two categories of elements: the headers and the sections. The headers describe the sections, and the sections contain the actual **text** and **data** to be installed.

An executable ELF file is a collection of segments, which are aggregations of sections. The kernel object-module loader performs an aggregation step on the relocatable object files that is similar to the process carried out by toolchains when producing an executable ELF file. The resulting image consists of one **text** segment, one **data** segment, and one **bss** segment. (A general ELF executable file may have more than one segment of each type, but the loader uses the simpler model of at most one segment of each type.) The loader installs the following categories of sections in the system's memory:

- **text** sections that hold the application's instructions
- **data** sections that hold the application's initialized data
- **bss** sections that hold the application's un-initialized data
- read-only data sections that hold the application's constant data

Read-only data sections are placed in the text segment by the loader.

Linking and Reference Resolution

The kernel object-module loader performs some of the same tasks as a traditional linker in that it prepares the code and data of an object module for the execution environment. This includes the linkage of the module's code and data to other code and data.

The loader is unlike a traditional linker in that it does this work directly in the target system's memory, and not in producing an output file.

In addition, the loader uses routines and variables that already exist in the VxWorks system, rather than library files, to relocate the object module that it loads. The system symbol table (see *12.4.4 Using the VxWorks System Symbol Table*, p.622) is used to store the names and addresses of functions and variables already installed in the system.This has the side effect that once symbols are installed in the system symbol table, they are available for future linking by any module that is loaded. Moreover, when attempting to resolve undefined symbols in a module, the loader uses all global symbols compiled into the target image, as well as all global symbols of previously loaded modules. As part of the normal load process, all of the global symbols provided by a module are registered in the system symbol table. You can override this behavior by using the **LOAD_NO_SYMBOLS** load flag (see Table 12-8).

The system symbol table allows name clashes to occur. For example, suppose a symbol named **func** exists in the system. A second symbol named **func** is added to the system symbol table as part of a load. From this point on, all links to **func** are to the most recently loaded symbol. See also, *12.4.1 Configuring VxWorks with Symbol Tables*, p.618.

Load Sequence Requirements and Caveats

The kernel object-module loader loads code modules in a sequential manner. That is, a separate load is required for each separate code module. The user must, therefore, consider dependencies between modules and the order in which they must be loaded to link properly.

Suppose a user has two code modules named **A_module** and **B_module**, and **A_module** references symbols that are contained in **B_module**. The user may either use the host-resident linker to combine **A_module** and **B_module** into a single module, or they should load **B_module** first, and then load **A_module**.

When code modules are loaded, they are irreversibly linked to the existing environment; meaning that, once a link from a module to an external symbol is created, that link cannot be changed without unloading and reloading the module.

Therefore dependencies between modules must be taken into account when modules are loaded to ensure that references can be resolved for each new module, using either code compiled into the VxWorks image or modules that have already been loaded into the system.

Failure to do so results in incompletely resolved code, which retains references to undefined symbols at the end of the load process. For diagnostic purposes, the loader prints a list of missing symbols to the console. This code should not be executed, since the behavior when attempting to execute an improperly relocated instruction is not predictable.

Normally, if a load fails, the partially installed code is removed. However, if the only failure is that some symbols are unresolved, the code is not automatically unloaded (but the API returns **NULL** to indicate failure programmatically). This allows the user to examine the result of the failed load, and even to execute portions of the code that are known to be completely resolved. Therefore, code modules that have unresolved symbols must be removed by a separate unload command (**unld( )** with the C interpreter, or **module unload** with the command interpreter).

Note that the sequential nature of the loader means that unloading a code module which has been used to resolve another code module may leave references to code or data which are no longer available. Execution of code holding such dangling references may have unexpected results.

See *Statically Linking Kernel Application Modules*, p.63.

Resolving Common Symbols

Common symbols provide a challenge for the kernel object-module loader that is not confronted by a traditional linker. Consider the following example:

```
#include <stdio.h>

int willBeCommon;

void main (void) {}
    {
    ...
    }
```

The symbol **willBeCommon** is uninitialized, so it is technically an undefined symbol. Many compilers will generate a *common* symbol in this case.

ANSI C allows multiple object modules to define uninitialized global symbols of the same name. The linker is expected to consistently resolve the various modules references to these symbols by linking them against a unique instance of the symbol. If the different references specify different sizes, the linker should define a single symbol with the size of the largest one and link all references against it. This is not a difficult task when all of the modules are linked in the same operation, such as when executing the host-resident linker, ld*arch*.

However, when VxWorks modules are loaded sequentially, the loader can only resolve the references of the module that it is currently loading with the those of the modules that it has already loaded, regardless of what the final, full set of modules may be. The **loadLib** API functions provide three options for controlling how common symbols are linked:

- The default behavior is to treat common symbols as if there were no previous matching reference (**LOAD_COMMON_MATCH_NONE**). The result is that every loaded module has its own copy of the symbol. For example, for three loads using this option with the same common symbol, three new global symbols are created.

- Common symbols are identified with any matching symbol in the symbol table (**LOAD_COMMON_MATCH_ALL**).

- Common symbols are identified with any matching symbol that was not in the original boot image (**LOAD_COMMON_MATCH_USER**).

Note that these options only control the loader's behavior with regard to the operation in which they are used—they only affect what happens with the symbols of the module being loaded. For example, consider the case in which module A has common symbols, and module B has undefined symbols that are resolved by module A. If module A is loaded with the **LOAD_COMMON_MATCH_NONE** option, this does not prevent module B from being linked against A's symbols when B is loaded next. That is, the load flag used with module A does not prevent the loader from resolving undefined references in module B against module A.

The option to specify matching of common symbols may be set in each call using the **loadLib** API. Extreme care should be used when mixing the different possible common matching behaviors for the loader. It is much safer to pick a single matching behavior and to use it for all loads. For detailed descriptions of the matching behavior under each option, see Table 12-12.

NOTE: Note that the shell load command, **ld**, has a different mechanism for controlling how common symbols are handled and different default behavior. For details, see the reference entry for **usrLib**.

Resolving Weak Symbols

Some programming languages (such as C++) use the *weak* binding class in addition to the global and local classes. The ELF specification mandates that a weak symbol is ignored if there is an existing global symbol with the same name. This is the default behavior provided by the VxWorks 6.*x* object module loader, specified with the **LOAD_WEAK_MATCH_ALL** loader option.

Note, however, that the default behavior for VxWorks 5.*x* was to always register weak symbols as globals, regardless of any existing definitions. To replicate this behavior, use the **LOAD_WEAK_MATCH_NONE** loader option.

Stripping Symbols From Modules

Symbols can be removed from object files by *stripping* them, which is commonly done with the GNU **strip** utility. The main purpose of stripping object files is to reduce their size.

The kernel object-module loader, however, requires relocation, section, and symbol information in order to load a module. By default the **strip** utility removes the information required by the loader, and it should therefore be used with the **--strip-unneeded** option, which removes all symbols that are *not* needed for the relocation process.

Note that the **--strip-unneeded** option removes debug information.

Function Calls, Relative Branches, and Load Failures

For some architectures, function calls are performed using relative branches by default. This causes problems if the routine that is called resides further in memory than a relative branch instruction can access (which may occur if the board has a large amount of memory).

In this case, a module load fails; the kernel module loader prints an error message about relocation overflow and sets the **S_loadElfLib_RELOCATION_OFFSET_TOO_LARGE** errno (kernel shell).

To deal with this problem, compilers (both GNU and Wind River) have options to prevent the use of relative branches for function calls. See the *VxWorks Architecture Supplement* for more information.

Kernel Object Modules With SDA

The loader cannot perform small data area (SDA) relocation. If a kernel module is built with SDA, the loader will not load it, but generates the following error:

- **S_loadLib_SDA_NOT_SUPPORTED**

The Wind River Compiler (diab) assembler flag **-Xwarn-use-greg** can be used to generate the following warning if code accesses the SDA reserved registers:

```
Xwarn-use-greg=0x2004
```

The **objdump** and **readelf** tools can be used to see if there are any SDA relocations in a module. The relocation types pertaining to SDA are described in the ELF architecture ABI supplement.

The **SDA_DISABLE** makefile variable can be used to disable SDA, as follows:

```
SDA_DISABLE=TRUE
```

12.4 Kernel Symbol Tables

A symbol table is a data structure that stores information that describes the routines, variables, and constants in all modules, and any variables created from the shell. There is a symbol table library, which can be used to manipulate the two different types of kernel symbol tables: a user-created symbol table and a system symbol table, which is the most commonly used. Note that both types of symbol tables used in the kernel are entirely independent of the symbol tables used by applications running in user-space processes (RTPs).

A system symbol table is required for the kernel object-module loader. The only exception being when fully-linked object modules are loaded with the **LOAD_FULLY_LINKED** option. For more information see Table 12-13 and **loadModuleAt( )** in the *VxWorks Kernel API Reference*.

Symbol Entries

Each symbol in the table comprises these items:

name
The name is a character string derived from the name in the source code.

value
The value is usually the address of the element that the symbol refers to: either the address of a routine, or the address of a variable (that is, the address of the contents of the variable). The value is represented by a pointer.

group
The group number of the module that the symbol comes from.

symRef
The **symRef** is usually the module ID of the module that the symbol comes from.

type
The type is provides additional information about the symbol. For symbols in the system symbol table, it is one of the types defined in *installDir*/**vxworks-6.*x*/target/h/symbol.h**. For example, **SYM_UNDF**, **SYM_TEXT**, and so on. For user symbol tables, this field can be user-defined.

Symbol Updates

The symbol table is updated whenever modules are loaded into, or unloaded from, the target. You can control the precise information stored in the symbol table by using the kernel object-module loader options listed in Table 12-8.

Searching the Symbol Library

You can easily search all symbol tables for specific symbols. To search from the shell with the C interpreter, use **lkup( )**. You can also use **symShow( )** for general symbol information. For details, see the API references for these commands.

To search programmatically, use the symbol library API's, which can be used to search the symbol table by address, by name, and by type, and a function that may be used to apply a user-supplied function to every symbol in the symbol table. For details, see the **symLib** reference entry.

12.4.1 Configuring VxWorks with Symbol Tables

VxWorks can be configured with support for user symbols tables or with support for both user symbol tables and a system symbol table.

For information about user symbol tables, see *12.4.6 Creating and Using User Symbol Tables*, p. 623. For information about the system symbol table, see *12.4.4 Using the VxWorks System Symbol Table*, p. 622.

Configuration for User Symbol Tables

Configuring VxWorks with the **INCLUDE_SYM_TBL** component provides the basic symbol table library, **symLib**, and support for creating user symbol tables.

A user symbol table is created at run-time with the **symTblCreate( )** routine, with parameters for the width of the symbol table's hash table, the name clash policy and the memory partition to use. For more information, see the VxWorks API reference for **symTblCreate( )**.

Configuration for a System Symbol Table

To include information about the symbols present in the kernel—and therefore to enable the shell, kernel object-module loader, and debugging facilities to function properly—a system symbol table must be created and initialized.

To create a system symbol table, VxWorks must be configured with the **INCLUDE_SYM_TBL_INIT** component and the **INCLUDE_SYM_TBL** component (which also provides support for user symbol tables).

The **INCLUDE_SYM_TBL_INIT** component includes the configuration parameter **SYM_TBL_HASH_SIZE_LOG2** which allows you to modify the symbol table width. This parameter defines the width of the symbol table's hash table. It takes a positive value that is interpreted as a power of two. The default value for **SYM_TBL_HASH_SIZE_LOG2** is 8; and the default width of the symbol table is therefore 256. Using smaller values requires less memory, but degrades lookup performance, so the search takes longer on the average.

The system symbol table, **sysSymTbl**, is configured to allow name clashes. In the event that they occur, the most recently added symbol is the one that is returned when searching the symbol table by name.

To initialize the system symbol table (which adds VxWorks kernel symbols), VxWorks must include a component for either a symbol table that is part of the system image, or a component for a symbol table that is downloaded separately from the host system:

INCLUDE_STANDALONE_SYM_TBL
Creates a built-in system symbol table, in which both the system symbol table and the VxWorks image are contained in the same module in which the system symbol table is contained, in the VxWorks image. This type of symbol table is described in *12.4.2 Creating a Built-In System Symbol Table*, p.620.

INCLUDE_NET_SYM_TBL
Creates an separate system symbol table as a **.sym** file that is downloaded to the VxWorks system. This type of symbol table is described in *12.4.3 Creating a Loadable System Symbol Table*, p.621.

When the system symbol table is first created at system initialization time, it contains no symbols. Symbols must be added to the table at run-time. Each of these components handles the process of adding symbols differently.

12.4.2 Creating a Built-In System Symbol Table

A built-in system symbol table copies information into wrapper code, which is then compiled and linked into the kernel when the system is built.

Although using a built-in symbol table can produce a larger VxWorks image file than might otherwise be the case, it has several advantages, particularly for production systems:

- It requires less memory than using a loadable symbol table—as long as you are not otherwise using the kernel object-module loader and associated components that are required for a loadable symbol table.
- It does not require that the target have access to a host (unlike the downloadable symbol table).
- It is faster to load the single image file than loading separate files for the VxWorks image and the loadable symbol table **.sym** file because some remote operations[4] on a file take longer than the data transfer to memory.
- It is useful in deployed ROM-based systems that have no network connectivity, but require the shell as user interface.

Generating the Symbol Information

A built-in system symbol table relies on the **makeSymTbl** utility to obtain the symbol information. This utility uses the gnu utility **nm***arch* to generate information about the symbols contained in the image. Then it processes this information into the file **symTbl.c** that contains an array, **standTbl**, of type **SYMBOL** described in *Symbol Entries*, p.617. Each entry in the array has the symbol **name** and **type** fields set. The address (**value**) field is not filled in by **makeSymTbl**.

Compiling and Linking the Symbol File

The **symTbl.c** file is treated as a normal **.c** file, and is compiled and linked with the rest of the VxWorks image. As part of the normal linking process, the toolchain linker fills in the correct address for each global symbol in the array. When the build completes, the symbol information is available in the image as a global array of VxWorks symbols. After the kernel image is loaded into target memory at

[4] That use **open( )**, **seek( )**, **read( )**, and **close( )**.

system initialization, the information from the global **SYMBOL** array is used to construct the system symbol table.

The definition of the **standTbl** array can be found in the following files after the VxWorks image is built:

- *installDir***/vxworks-6.***x***/target/config/***bspName***/symTbl.c** for images built directly from a BSP directory.
- *installDir***/vxworks-6.***x***/target/proj/***projDir***/***buildDir***/symTbl.c** for images using the project facility.

12.4.3 Creating a Loadable System Symbol Table

A loadable symbol table is built into a separate object module file (**vxWorks.sym** file). This file is downloaded to the system separately from the system image, at which time the information is copied into the symbol table.

Creating the .sym File

The loadable system symbol table uses an ELF file named **vxWorks.sym** file, rather than the **symTbl.c** file. The file is created by using the **objcopy** utility to strip all sections, except the symbol information, from the final VxWorks image.

Loading the .sym File

During boot and initialization, the **vxWorks.sym** file is downloaded using the kernel object-module loader, which directly calls **loadModuleAt( )**. To download the **vxWorks.sym** file, the loader uses the current default device, which is described in *7.3.1 Filenames and the Default Device*, p.363.

To download the VxWorks image, the loader also uses the default device, as is current at the time of that download. Therefore, the default device used to download the **vxWorks.sym** file may, or may not, be the same device. This is because the default device can be set, or reset, by other initialization code that runs. This modification can happen after the VxWorks image is downloaded, but before the symbol table is downloaded.

Nevertheless, in standard VxWorks configurations, that do not include customized system initialization code, the default device at the time of the download of the **vxWorks.sym**, is usually set to one of the network devices, and using either **rsh** or **ftp** as the protocol.

12.4.4 Using the VxWorks System Symbol Table

Once it is initialized, the VxWorks system symbol table includes a complete list of the names and addresses of all global symbols in the compiled image that is booted. This information is needed on the target to enable the full functionality of the target tools libraries.

The target tools maintain the system symbol table with up-to-date name and address information for all of the code statically compiled into the system or dynamically downloaded. (The **LOAD_NO_SYMBOLS** option can be used to *hide* loaded modules, so that their symbols do not appear in the system symbol table; see Table 12-9).

Symbols are dynamically added to, and removed from, the system symbol table when:

- modules are loaded and unloaded
- variables are dynamically created from the shell
- the wdb agent synchronizes symbol information with the host (see *12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table*, p.623)

The exact dependencies between the system symbol table and the other target tools are as follows:

- **Kernel Object-Module Loader**: The kernel loader requires the system symbol table. The system symbol table does not require the presence of the loader.
- **Debugging Facilities**: The target-based symbolic debugging facilities and user commands such as **i** and **tt**, rely on the system symbol table to provide information about entry points of tasks, symbolic contents of call stacks, and so on.
- **Kernel Shell**: The kernel shell does not strictly require the system symbol table, but its functionality is greatly limited without it. The kernel shell requires the system symbol table to provide the ability to run functions using their symbolic names. The kernel shell uses the system symbol table to execute shell commands, to call system routines, and to edit global variables. The kernel shell also includes the library **usrLib**, which contains the commands **i**, **ti**, **sp**, **period**, and **bootChange**.

- **WDB Target Agent**: The WDB target agent adds symbols to the system symbol table as part of the symbol synchronization with the host.

If the facilities provided by the symbol table library are needed for user (non-operating system) code, another symbol table should be created and manipulated using the symbol library. See *12.4.6 Creating and Using User Symbol Tables*, p.623.

NOTE: If you choose to use both the host-resident and target-resident tools at the same time, use the synchronization method to ensure that both the host and target resident tools share the same list of symbols.

12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table

If both host tools and target tools are going to be used with a target system, the modules list and symbol table maintained on the host system must be synchronized with the modules list and symbol table maintained on the target. This ensures that the host and target tools share the same list of symbols.

The host tools maintain their own modules list and symbol table—the target server modules list and symbol table— on the host. In this chapter it is referred to as the host modules list and symbol table.

Module list and symbol table synchronization is provided automatically when VxWorks is configured with the WDB target agent and the kernel object-module loader (**INCLUDE_WDB** and **INCLUDE_LOADER**). To remove this feature, you need only remove the **INCLUDE_WDB_MDL_SYM_SYNC** component.

Note that the modules and symbols synchronization will only work if the WDB agent is in task mode. If the WDB agent is in system mode, the modules and symbols added from both the host and the target will not be synchronized.

For information about WDB, see *12.6 WDB Target Agent*, p.626.

12.4.6 Creating and Using User Symbol Tables

Although it is possible for user code in the kernel to manipulate symbols in the system's symbol table, this is not a recommended practice. Addition and removal of symbols to and from the symbol table should only be carried out by operating system libraries. Any other use of the system symbol table may interfere with the proper operation of the operating system; and even simply introducing additional symbols could have an adverse and unpredictable effect on linking any modules that are subsequently downloaded.

Therefore, user-defined symbols should not be added programmatically to the system symbol table. Instead, when user code in kernel space requires a symbol table for its own purposes, a user symbol table should be created. For more information, see the VxWorks API reference for **symLib**.

12.5 Show Routines

VxWorks includes system information routines that can be invoked from the shell's C interpreter. They should not be used programmatically.

The show routines 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 include the appropriate component when you configure VxWorks. When you invoke them, their output is sent to the standard output device. Table 12-14 lists common system show routines:

Table 12-14 **Show Routines**

Call	Description	Component
envShow( )	Displays the environment for a given task on **stdout**.	**INCLUDE_TASK_SHOW**
memPartShow( )	Shows the partition blocks and statistics.	**INCLUDE_MEM_SHOW**
memShow( )	System memory show routine.	**INCLUDE_MEM_SHOW**
moduleShow( )	Prints information for all loaded modules, or an individual module.	**INCLUDE_MODULE_MANAGER**
msgQShow( )	Message queue show utility (for both POSIX and native VxWorks message queues).	**INCLUDE_POSIX_MQ_SHOW** **INCLUDE_MSG_Q_SHOW**

Table 12-14 **Show Routines** (cont'd)

Call	Description	Component
semShow( )	Semaphore show utility (for both POSIX and native VxWorks semaphores).	**INCLUDE_SEM_SHOW**, **INCLUDE_POSIX_SEM_SHOW**
show( )	Generic object show utility. The **show( )** routine does not work with modules or symbol tables; see **moduleShow( )** and **symshow( )**.	**INCLUDE_SHOW_ROUTINES**
stdioShow( )	Standard I/O file pointer show utility.	**INCLUDE_STDIO_SHOW**
symShow( )	Prints symbol table information.	**INCLUDE_SYM_TBL_SHOW**
taskSwitchHookShow( )	Shows the list of task switch routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskCreateHookShow( )	Shows the list of task create routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskDeleteHookShow( )	Shows the list of task delete routines.	**INCLUDE_TASK_HOOKS_SHOW**
taskShow( )	Displays the contents of a task control block.	**INCLUDE_TASK_SHOW**
wdShow( )	Watchdog show utility.	**INCLUDE_WATCHDOGS_SHOW**

Table 12-15 **Network Show Routines**

Call	Description
ifShow( )	Display the attached network interfaces.
inetstatShow( )	Display all active connections for IP sockets.
ipstatShow( )	Display IP statistics.
netPoolShow( )	Show pool statistics.
netStackDataPoolShow( )	Show network stack data pool statistics.
netStackSysPoolShow( )	Show network stack system pool statistics.
mbufShow( )	Report **mbuf** statistics.

Table 12-15 **Network Show Routines** (cont'd)

Call	Description
netShowInit( )	Initialize network show routines.
arpShow( )	Display entries in the system ARP table.
arptabShow( )	Display the known ARP entries.
routestatShow( )	Display routing statistics.
routeShow( )	Display host and network routing tables.
hostShow( )	Display the host table.
mRouteShow( )	Print the entries of the routing table.

12.6 WDB Target Agent

The VxWorks WDB target agent is a target-resident, run-time facility that is required for connecting host tools to a VxWorks target system. It is not required (or generally useful) for deployed systems, nor is it required for development using the kernel shell (see *12.2 Kernel Shell*, p.577). The facility is also referred to as the target agent, the WDB agent, or simply as WDB. The acronym stands for Wind DeBug.

The WDB agent carries out requests transmitted from host-based debugging tools and replies with the results. The WDB agent contains a compact implementation of UDP/IP, and a proprietary RPC messaging protocol called WDB. The WDB (Wind DeBug) protocol specifies how the target server (on the host) communicates with the target agent (on the target). The protocol includes a compact programming language called Gopher, which permits on-the-fly extension by supporting programmable investigation of the target system.

The WDB protocol provides a core minimum of the services necessary to respond to requests from the host tools. These protocol requests include memory transactions, breakpoint/event notification services, virtual I/O support, tasking control, and real-time process control. The WDB protocol uses the Sun Microsystems specification for External Data Representation (XDR) for data transfer.

WDB can be configured for system mode debugging, task mode debugging, or both (switching between the two modes under the control of host tools). In task mode, WDB runs as a kernel task. In system mode, WDB operates independently of the kernel, and the kernel is under WDB's control. With system mode, WDB can be started before VxWorks is running, which can be particularly useful in the early stages of porting a BSP to a new board. (See *Debugging Mode Options*, p.632 and *12.6.6 Starting the WDB Target Agent Before the VxWorks Kernel*, p.640).

The WDB agent's interface to communications drivers avoids the run-time I/O system, so that the WDB agent remains independent of the run-time OS. Drivers for the WDB agent are low-level drivers that provide both interrupt-driven and polling-mode operation. Polling mode is required to support system-level control of the target.

The WDB agent synthesizes the target-control strategies of task-level and system-wide debugging. The agent can execute in either mode and switch dynamically between them, provided the appropriate drivers are present in the Board Support Package (BSP). This permits debugging of any aspect of an embedded application whether it is a task, an interrupt service routine, or the kernel itself.

NOTE: If both host tools and target tools are going to be used with a target system, the modules list and symbol table maintained on the host system must be synchronized with the modules list and symbol table maintained on the target. This ensures that the host and target tools share the same list of symbols. See the discussion of the **INCLUDE_WDB_MDL_SYM_SYNC** component in *Additional Options*, p.634, and *12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table*, p.623.

12.6.1 Configuring VxWorks with the WDB Target Agent

WDB target agent functionality is provided by a suite of components, some of which are optional, and others of which provide support for alternate modes of connection. By default VxWorks is configured with a pipe connection for the VxWorks simulator, and an Enhanced Network Driver (END) connection for all hardware targets.

The **INCLUDE_WDB** component provides the basic target agent facilities. It allows you to connect a target server, get basic information about the target, and load modules.

Basic WDB Configuration

The configuration parameters for the basic **INCLUDE_WDB** component are:

WDB_COMM_PORT
The UDP port used by the WDB agent to connect to the host (the default is 0x4321). This is also the default port used by the target server. If you must change this port, be sure to update the port information in Wind River Workbench.

WDB_STACK_SIZE
The stack size of the WDB agent.

WDB_POOL_SIZE
The size of the memory pool used by the WDB agent.

To configure the WDB agent, you must also choose the appropriate connection-type component, one or more debugging modes, one initialization component, and any other options you may need. For information about reducing the size of the agent, see *12.6.4 Scaling the WDB Target Agent*, p.639.

Host-Target Communication Options

The WDB components required for different types of host-target connections are described in Table 12-16. VxWorks should be configured with only one WDB communication component.

Table 12-16 **WDB Connection Components**

Component	Description
INCLUDE_WDB_COMM_END	The WDB enhanced network driver (END) connection component. The END driver supports both system and task mode debugging. This component is the default.
INCLUDE_WDB_COMM_NETWORK	The WDB UDP/IP network connection component. This communication type only supports task mode debugging.
INCLUDE_WDB_PROXY INCLUDE_ WDB_PROXY_TIPC	The WDB TIPC components required for the gateway system on a TIPC network.

Table 12-16 **WDB Connection Components** (cont'd)

Component	Description
INCLUDE_WDB_COMM_TIPC	The WDB TIPC component for other targets (non-gateway) on a TIPC network.
INCLUDE_WDB_COMM_SERIAL	The WDB serial connection component. Useful when no network connections are available.
INCLUDE_WDB_COMM_VTMD	The WDB visionICE II or visionProbe II emulator connection component. This communication link is useful when debugging hardware bring up.
INCLUDE_WDB_COMM_PIPE	The WDB simulator pipe connection component—used only for the VxWorks simulator.
INCLUDE_WDB_COMM_CUSTOM	A WDB custom connection component, created by the user (see *12.6.7 Creating a Custom WDB Communication Component*, p.642).

⚠ **WARNING:** Both VxWorks and the host target connection must be configured for the same type of host-target communication facilities. For example, if a serial connection is going to be used, then VxWorks must be configured with **INCLUDE_WDB_COMM_SERIAL** and the host target server must be configured with the **wdbserial** back end. For more information about target connection configuration, see the *Wind River Workbench User's Guide: New Target Server Connections*.

Enhanced Network Driver Connection Configuration

The configuration parameters for the **INCLUDE_WDB_COMM_END** component are:

WDB_MTU
The maximum transfer unit (MTU). The default MTU is 1500 bytes.

WDB_END_DEVICE_NAME
By default, this parameter is set to NULL and the END driver used by the WDB agent is the one specified with a VxWorks boot loader device parameter. If you want to use a different device, set this parameter to the name of the device (for example, **dc**).

WDB_END_DEVICE_UNIT
If **WDB_END_DEVICE_NAME** is specified, set this parameter to the unit number of the END device you want to use.

Network Connection Configuration

The sole configuration parameter for the **INCLUDE_WDB_COMM_NETWORK** component is **WDB_MTU**, which defines the MTU for a UDP/IP network connection.

TIPC Network Connection Configuration

The configuration parameters for the **INCLUDE_WDB_COMM_TIPC** component are:

WDB_TIPC_PORT_TYPE
The TIPC port type. The default is 70.

WDB_TIPC_PORT_INSTANCE
The TIPC port instance. The default is 71.

Note that the **INCLUDE_WDB_COMM_TIPC** component is used for the targets on a TIPC network that you want to connect to with host tools, but not for the target that serves as a gateway between the host system's TCP/IP network and the targets' TIPC network. See *12.6.3 Using the WDB Target Agent with a TIPC Network*, p.638 for more information.

For more information about TIPC itself, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*.

Serial Connection Configuration

The configuration parameters for the **INCLUDE_WDB_COMM_SERIAL** component are:

WDB_MTU
The MTU for a serial connection.

WDB_TTY_BAUD
The bps rate of the serial channel. The default is 9600 bps. For better performance, use the highest line speed available, which is often 38400 bps. Values higher than 34800 may not provide satisfactory performance. Try a slower speed if you suspect data loss.

WDB_TTY_CHANNEL
The channel number. Use 0 if you have only one serial port on the target. Use 1 (the default) if you want to keep the VxWorks console on the first serial port.[5]

If your target has a single serial channel, you can use the target server virtual console to share the channel between the console and the target agent. You must configure your system with the **CONSOLE_TTY** parameter set to **NONE** and the **WDB_TTY_CHANNEL** parameter set to 0.

When multiplexing the virtual console with WDB communications, excessive output to the console may lead to target server connection failures. The following may help resolve this problem:

- Decrease the amount of data being transmitted to the virtual console from your application.
- Increase the time-out period for the target server.
- Increase the baud rate of the target agent and the target server connection.

INCLUDE_WDB_TTY_TEST
When set to **TRUE**, this parameter causes words **WDB READY** to be displayed on the WDB serial port on startup. By default, this parameter is set to **TRUE**.

WDB_TTY_ECHO
When set to **TRUE**, all characters received by the WDB agent are echoed on the serial port. As a side effect, echoing stops the boot process until a target server is attached. By default, this parameter is set to **FALSE**.

[5] VxWorks serial channels are numbered starting at 0. Thus Channel 1 corresponds to the second serial port if the board's ports are labeled starting at 1. If your board has only one serial port, you must change **WDB_TTY_CHANNEL** to 0 (zero).

visionICE or visionProbe Connection Configuration

The sole configuration parameter for the **INCLUDE_WDB_COMM_VTMD** component is **TMD_DEFAULT_POLL_DELAY**, which specifies the clock tick interval for polling data on the target.

Pipe Connection Configuration

The sole configuration parameter for the **INCLUDE_WDB_COMM_PIPE** component (for the VxWorks simulator only) is **WDB_MTU**, which defines the MTU for a pipe connection.

Debugging Mode Options

WDB provides two debugging mode options by default: system mode and task mode. With system mode, the entire system is stopped when a breakpoint is hit. This allows you to set breakpoints anywhere, including ISRs. Note that for SMP systems, software breakpoints are persistent—that is they are retained in target memory (for UP systems they are not).

With task mode, a task or group of tasks is stopped when a breakpoint is set, but an exception or an interrupt does not stop if it hits a breakpoint. When the WDB agent is configured for task mode, the **tWdbTask** task is used to handle all WDB requests from the host.

You can include support for both modes, which allows tools such as the host shell or the debugger to dynamically switch from one mode to the other.

For information about WDB behavior with SMP systems, see *12.6.2 WDB Target Agent and VxWorks SMP*, p.637. For information about the SMP configuration of VxWorks, see *15. VxWorks SMP*.

System Mode Debugging Configuration

The **INCLUDE_WDB_SYS** component provides support for system mode debugging. Note that this mode is only supported when the communication type has a polling mode for reading the device, which is not the case with the network component **INCLUDE_WDB_COMM_NETWORK**.

The configuration parameters for the **INCLUDE_WDB_SYS** component are:

WDB_SPAWN_OPTS
The task options flag used by tasks spawned in system mode.

WDB_SPAWN_PRI
The task priority used by tasks spawned in system mode.

WDB_SPAWN_STACK_SIZE
The stack size used by tasks spawned by the WDB target agent.

Task Mode Debugging Configuration

The **INCLUDE_WDB_TASK** component provides support for task mode. The configuration parameters are:

WDB_MAX_RESTARTS
The maximum number of times an agent can be restarted when it gets an error (the default is 5).

WDB_RESTART_TIME
The delay (in seconds) before restarting the WDB agent task when it gets an error (the default is 10 seconds).

WDB_TASK_OPTIONS
The options parameter of the WDB task.

WDB_TASK_PRIORITY
The priority of the WDB task. The default priority is 3.

WDB_SPAWN_STACK_SIZE
The stack size used by tasks spawned by the WDB target agent.

Process Management Options

The **INCLUDE_WDB_RTP** component provides support for real time process (RTP) operations (creation, deletion) and notifications (creation, deletion). This component is automatically included if the system supports real time processes (**INCLUDE_RTP**) and task debugging mode (**INCLUDE_WDB_TASK**).

The **INCLUDE_WDB_RTP_BP** component provides support for real time process debugging. It allows use of process-wide breakpoints. This component is automatically included when the system supports real time processes (**INCLUDE_RTP**) and task breakpoints (**INCLUDE_WDB_TASK_BP**).

The **INCLUDE_WDB_RTP_CONTROL** component allows the debugger to configure a process or kernel task such that its child processes are stopped at creation; that is, they do not start automatically when they are spawned. By default, processes can be spawned with an option that causes them to stop before they run, but child processes to not inherit this characteristic.

Initialization Options

WDB can be configured to start either before or after kernel initialization. By default, WDB is started after the kernel has been initialized.

The **INCLUDE_WDB_POST_KERNEL_INIT** component causes WDB to be started once kernel has fully been initialized. The **INCLUDE_WDB_PRE_KERNEL_INIT** component causes WDB to be started before kernel initialization has completed.

If WDB starts before kernel initialization, it is possible to perform early system debugging. However, because the kernel has not been initialized when WDB starts, task debugging is not supported in this mode. In addition, the END connection cannot be used with this mode because the network has not been initialized when WDB starts. Also see *12.6.6 Starting the WDB Target Agent Before the VxWorks Kernel*, p.640.

When WDB starts after kernel initialization, all WDB features are fully supported. It is, of course, not possible to debug kernel initialization activity.

Additional Options

The following components provide additional optional functions. You can include or exclude them based on your requirements.

The **INCLUDE_WDB_BANNER** component displays the WDB banner on the console.

The **INCLUDE_WDB_BP** component provides support for breakpoints in the WDB agent itself. This component is needed if you want to debug the target from a host tool. The configuration parameter for this component is **WDB_BP_MAX**, which specifies the maximum number of breakpoints allocated on the target at startup. When this number of breakpoints is reached, it is still possible to allocate space for new breakpoints in task mode. In system mode, however, it is not possible to set additional breakpoints once the limit has been reached.

The **INCLUDE_WDB_BP_SYNC** component provides a breakpoint synchronization mechanism between host tools and target system. If this component is included in the VxWorks configuration, host tools are notified of any breakpoint creations and deletions that are made from the kernel shell. The component is automatically included when debug is provided for both the kernel shell (with **INCLUDE_DEBUG**) and the host tools (with **INCLUDE_WDB_TASK_BP**).

The **INCLUDE_WDB_CTXT** component provides support for context operations: creation, deletion, suspension, resumption. A context can be a task, a real time process, or the system itself.

The **INCLUDE_WDB_DIRECT_CALL** component allows you to call functions in the WDB agent context directly.

The **INCLUDE_WDB_EVENTPOINTS** component adds support for eventpoints. An eventpoint can be a breakpoint, an eventpoint on context creation, or an eventpoint on context deletion. This component is the core component for all eventpoint types. Each time an eventpoint is hit, the corresponding event is sent to the target server.

The **INCLUDE_WDB_EVENTS** component adds support for asynchronous events. Asynchronous events are sent from the target to target server, to notify host tools about event activity on the target; for example, if a breakpoint has been hit, an exception occurred, or a context (task or process) has started or exited. The component is required (and is automatically included) when using breakpoints, exception notification, context start/exit notification, and so on.

The **INCLUDE_WDB_EXC_NOTIFY** component adds support for exception notification. When an exception occurs on the target, the appropriate event is sent to the target server.

The **INCLUDE_WDB_EXIT_NOTIFY** component adds support for context deletion notification. To be notified of a context exit, an eventpoint of type **WDB_CTX_EXIT** must be set. Tools set this eventpoint when they must be notified. This component supports notification for task and real time process contexts.

The **INCLUDE_WDB_FUNC_CALL** component handles function calls by spawning tasks to run the functions. This service is only available in task mode.

The **INCLUDE_WDB_GOPHER** component provides support for the Gopher information gathering language. It is used by many host tools and cannot be removed from a system that uses other WDB options. The configuration parameters for this component are:

- **WDB_GOPHER_TAPE_LEN**, which defines the length of one gopher tape. Gopher tapes are used to record and upload data processed by the gopher. The default tape length is 1400 words, each of which are 32 bits wide.

- **WDB_GOPHER_TAPE_NB**, which defines the maximal number of gopher tapes that can be dynamically allocated. At startup, only one gopher tape is available. As needed, more tapes can be allocated. Dynamic allocation of tapes is only available in task mode. The default number of tapes is 10.

The **INCLUDE_WDB_MEM** component provides support for reading from, and writing to, target memory.

The **INCLUDE_WDB_REG** component provides support for reading from, and writing to, registers. The **WDB_REGS_SIZE** configuration parameter defines the size of an internal memory buffer used by the WDB agent to store coprocessor registers (to allow access to the registers in system mode).

The **INCLUDE_WDB_START_NOTIFY** component provides support for context creation notification. To be notified of a context exit, an eventpoint of type **WDB_CTX_START** must be set. Tools set this eventpoint when they must be notified. This component supports task and real time process contexts.

The **INCLUDE_WDB_TASK_BP** component provides support for breakpoints in task debugging mode. This component is automatically included when WDB breakpoints (**INCLUDE_WDB_BP**) and task debugging mode (**INCLUDE_WDB_TASK**) are included in the system.

The **INCLUDE_WDB_TASK_HOOKS** component initializes task hooks needed to support task debugging mode. It is automatically included when task debugging mode (**INCLUDE_WDB_TASK**) is included, and should never be removed manually.

The **INCLUDE_WDB_TASK_REG** component provides support for task register operations (read and write). It is automatically included when WDB supports register operations (**INCLUDE_WDB_REG**) and task debugging mode (**INCLUDE_WDB_TASK**).

The **INCLUDE_WDB_TSFS** component adds support for a virtual file system enabled by the WDB protocol, the Target Server File System (see *8.9 Target Server File System: TSFS*, p.518). This component is automatically included when the **INCLUDE_WVUPLOAD_TSFSSOCK** component is included for System Viewer upload support.

The **INCLUDE_WDB_USER_EVENT** component handles user defined events. For more information about user events, see the VxWorks **wdbUserEvtLib** API reference.

The **INCLUDE_WDB_VIO** component provides a driver for a virtual I/O (VIO) access.

The **INCLUDE_WDB_VIO_LIB** component handles VIO access (**read( )** and **write( )**). It requires the VIO driver component and the events component.

The **INCLUDE_WDB_MDL_SYM_SYNC** component handles module and symbol synchronization between the target and the target server. It is required only if both the WDB agent (**INCLUDE_WDB**) and the kernel object-module loader

(**INCLUDE_LOADER**) are available on the target and you are using the host-based loader (through the debugger, for example). The component synchronizes the records of modules and symbols that are kept by the host and kernel loaders. For more information, see the VxWorks **wdbMdlSymSyncLib** API reference and *12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table*, p.623.

12.6.2 WDB Target Agent and VxWorks SMP

This section describes the behavior of the WDB target agent when used with an SMP configuration of VxWorks. For information about the SMP configuration of VxWorks, see *15. VxWorks SMP*.

Task Mode Debugging

With the task mode debugging agent:

- Breakpoint exceptions that occur while holding an ISR-callable spinlock are ignored.
- Breakpoint exceptions that occur while holding a task-only spinlock are ignored.
- Breakpoint exceptions that occur while interrupts are locked are ignored.

System Mode Debugging

With the system mode debugging agent:

- When the system is stopped either by a breakpoint or an exception, all CPUs of the target are stopped.
- Breakpoint cannot be set on a specific CPU, they are always system wide.
- The maximum number of hardware breakpoints that can be installed on a SMP system is limited to the maximum number of hardware breakpoints of one core.
- On a stop request (for example, a **sysSuspend( )** call from the host shell), all processors of the system stop. However, if one processor of the system is executing a portion of code protected against interrupts (for example with an **intCpuLock( )** call), then it will stop only when the interrupts are re-enabled.

12.6.3 Using the WDB Target Agent with a TIPC Network

Wind River host tools can be used to debug VxWorks targets on a TIPC network that do not have direct access to the host by way of TCP/IP or a serial line. In order to do so, however, one of the VxWorks targets on the TIPC network must serve as a gateway system.

A gateway must have access to both the host's TCP/IP network and the targets' TIPC network, and it must run a *target agent proxy* server. The proxy server supports both networking protocols and provides the link between the host target server and the WDB agent on the target system, thus allowing for *remote* debugging of the other VxWorks targets on the TIPC network. The proxy server can support multiple connections between the host system and different VxWorks targets.

Note that with TIPC, WDB system mode debugging is not supported over TIPC (see *Debugging Mode Options*, p.632).

For information about TIPC, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*.

Target System Configuration

The VxWorks gateway target and the other VxWorks targets on the TIPC network (to which the host tools attach) must each be configured with different WDB components:

- The gateway target must be configured with the **INCLUDE_WDB_PROXY** and **INCLUDE_WDB_PROXY_TIPC** components (as well as with both TIPC and TCP/IP support).
- Any other targets to which host tools will attach must be configured with the basic **INCLUDE_WDB** component and the **INCLUDE_WDB_COMM_TIPC** component (as well as with TIPC support).

When the **INCLUDE_WDB_COMM_TIPC** component is included, WDB system mode is excluded from the configuration, as it is not supported with TIPC communication.

For information about the configuration parameters for these components, see *Basic WDB Configuration*, p.628 and *TIPC Network Connection Configuration*, p.630.

Establishing a Host-Target Connection

To establish a connection between the host and the targets on the TIPC network, first boot the gateway and other targets.

Wind River Workbench provides options for connecting with a target running a WDB agent proxy. See the *Wind River Workbench User's Guide* for more information.

From the command line, the syntax for starting a target server connection with a target running the WDB agent proxy is as follows:

```
tgtsvr -V -B wdbproxy -tipc -tgt targetTipcAddress -tipcpt tipcPortType -tipcpi
tipcPortInstance  wdbProxyIpAddress/name
```

In this command:

- *targetTipcAddress* is the TIPC address of the target to which you want to connect.
- *tipcPortType* is the TIPC port type used for the WDB connection (the default is 70).
- *tipcPortInstance* is the TIPC port instance used for the WDB connection (the default is 71).
- *wdbProxyIpAddress/name* is the IP address or target name of the gateway target that is running the WDB proxy agent.

12.6.4 Scaling the WDB Target Agent

In a memory-constrained system, you may wish to create a smaller target agent. To reduce its size, you can remove the optional facilities listed in Table 12-17. They are otherwise included by default.

Table 12-17 **Optional WDB Agent Components**

Component	Description
INCLUDE_WDB_BANNER	Prints a banner to console after the agent is initialized.
INCLUDE_WDB_VIO	Provides the VxWorks driver for accessing virtual I/O.
INCLUDE_WDB_USER_EVENT	Provides the ability to send user events to the host.

You can also reduce the maximum number of WDB breakpoints with the **WDB_BP_MAX** parameter of the **INCLUDE_WDB_BP** component. If you are using a serial connection, you can also set the **INCLUDE_WDB_TTY_TEST** parameter to **FALSE**.

If you are using a communication path that supports both system and task mode agents, then by default both agents are started. Since each agent consumes target memory (for example, each agent has a separate execution stack), you may wish to exclude one of the agents from the target system. You can configure the target to use only a task-mode or only a system-mode agent with the **INCLUDE_WDB_TASK** or **INCLUDE_WDB_SYS** options.

12.6.5 WDB Target Agent and Exceptions

If an application or BSP uses **excHookAdd( )** or signal handlers to handle exceptions, WDB does not notify the host tools of the exceptions handled by those facilities. Host tool notification can be suppressed for all other exceptions by removing the **INCLUDE_WDB_EXC_NOTIFY** component from the VxWorks configuration.

If the WDB task (**tWdbTask)** takes an exception, it is restarted after a (configurable) delay. The connection between the target server and the target agent is down during the delay period. The length of the delay can be set with the **WDB_RESTART_TIME** parameter. Note that the WDB task is started in the kernel only if WDB is set to run in task mode.

12.6.6 Starting the WDB Target Agent Before the VxWorks Kernel

By default, the WDB target agent is initialized near the end of the VxWorks initialization sequence. This is because the default configuration calls for the agent to run in task mode and to use the network for communication; thus, WDB is initialized after the kernel and the network.

In some cases—such as during BSP development—you may want to start the agent before the kernel, and initialize the kernel under the control of the host tools.

VxWorks Configuration

To be able to start WDB before the kernel, reconfigure VxWorks as follows:

1. Choose a communication path that can support a system-mode agent; a raw serial connection. (The END communication path cannot be used as it requires that the system be started before it is initialized.)

2. Select *only* the **INCLUDE_WDB_SYS** component—and *not* the task mode component.

   By default, the task mode starts two agents: a system-mode agent and a task-mode agent. Both agents begin executing at the same time, but the task-mode agent requires the kernel to be running.

3. Remove the **INCLUDE_WDB_BANNER** component. For some architectures, calling this component before kernel is initialized may hang the target.

4. Add the **INCLUDE_WDB_PRE_KERNEL_INIT** component and remove the **INCLUDE_WDB_POST_KERNEL_INIT** component. (See *Initialization Options*, p.634.)

   This causes the project code generator to make the **usrWdbInit( )** call earlier in the initialization sequence. It will be called from **usrInit( )** just before the kernel is started.[6]

Run-time Operation

When the host target server has connected to the system-mode WDB target agent, you can resume the system to start the kernel under the agent's control.

After connecting to the target agent, set a breakpoint in **usrRoot( )**, then continue the system. The routine **kernelInit( )** starts the multi-tasking kernel with **usrRoot( )** as the entry point for the first task. Before **kernelInit( )** is called, interrupts are still locked. By the time **usrRoot( )** is called, interrupts are unlocked.

Errors before reaching the breakpoint in **usrRoot( )** are most often caused by a stray interrupt: check that you have initialized the hardware properly in the BSP **sysHwInit( )** routine. Once **sysHwInit( )** is working properly, you no longer need to start the agent before the kernel.

[6] The code generator for **prjConfig.c** is based on the component descriptor language, which specifies when components are initialized. The component descriptor files are searched in a specified order, with the project directory being last, and overriding the default definitions in the generic descriptor files. For more information, see *2.8.2 CDF Precedence and CDF Installation*, p.75.

NOTE: If you use a serial connection when you start WDB before the kernel, you must modify the SIO driver so that it can properly deal with interrupts and the order of system initialization in this context. See the *VxWorks Device Driver Developer's Guide: Additional Drivers* for detailed information.

CAUTION: When the agent is started before the kernel, there is no way for the host to get the agent's attention until a breakpoint occurs. This is because only system mode is supported and the WDB communication channel is set to work in polled mode only. On the other hand, the host does not really need to get the agent's attention: you can set breakpoints in **usrRoot( )** to verify that VxWorks can get through this routine. Once **usrRoot( )** is working, you can start the agent after the kernel (that is, within **usrRoot( )**), after which the polling task is spawned normally.

12.6.7 Creating a Custom WDB Communication Component

To create a custom communication component:

1. Write a WDB packet driver. The template file *installDir***/vxworks-6.***x***/target/src/drv/wdb/wdbTemplatePktDrv.c** can be used as a starting point.

2. Create a configlette file in *installDir***/vxworks-6.***x***/target/config/comps/src** that contains the routine **wdbCommDevInit( )** to initialize the driver. You can base it on one of the WDB communication path configlettes in this directory (**wdbEnd.c**, **wdbSerial.c**, and so on).

3. Create a component descriptor file (CDF) for the custom component called **01wdbCommCustom.cdf** in the directory *installDir***/vxworks-6.***x***/target/config/comps/vxWorks**. The file must identify the driver module, the configlette, and any special parameters.

For information about creating custom components, see *2.8 Custom VxWorks Components and CDFs*, p.67, including *Defining a Component*, p.68.

The custom communication component can then be used by configuring VxWorks with **WDB_COMM_CUSTOM**.

12.7 Common Problems

This section lists frequently encountered problems that can occur when using the target tools.

Kernel Shell Debugging Never Hits a Breakpoint

I set a breakpoint on a function I called from the kernel shell, but the breakpoint is not being hit. Why not?

Explanation

The kernel shell task runs with the **VX_UNBREAKABLE** option. Functions that are called directly from the kernel shell command prompt, are executed within the context of the kernel shell task. Therefore, breakpoints set within the directly called function will not be hit.

Solution

Instead of running the function directly, use **taskSpawn( )** with the function as the entry point, or the shell's C interpreter **sp( )** command.

Insufficient Memory

The kernel object-module loader reports insufficient memory to load a module; however, checking available memory indicates the amount of available memory to be sufficient. What is happening and how do I fix it?

Explanation

The kernel loader calls the device drivers through a VxWorks' transparent mechanism for file management, which makes calls to **open**, **close**, and **ioctl**. If you use the kernel 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. This is because, depending on the device that is actually being used for the load, the calls initiate very different operations.

For some devices, the I/O library makes a copy of the file in target memory. Loading a file that is mounted over a device using such a 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. This copy is then used to carry out various **seek** and **read** operations. Therefore, using these drivers requires sufficient memory available to hold two copies of the file to be downloaded, as well as a small amount of memory for the overhead required or the load operation.

Also consider that loading a module sometimes requires additional space, as the sections have to be aligned in memory (whereas the toolchain may compact them all in the object file to save space). See *12.3.5 Specifying Memory Locations for Loading Objects*, p.610.

Solution

Download the file using a different device. 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).

"Relocation Does Not Fit" Error Message

When downloading, the following type of error message occurs:

```
Relocation value does not fit in 26 bits (offset: 0x10, type: 1)
```

What does this error mean and what should I do?

Explanation

Some architectures have instructions that use less than 32 bits to reference a nearby position in memory. Using these instructions can be more efficient than always using 32 bits to refer to nearby places in memory.

The problem arises when the compiler has produced such a reference to something that lies farther away in memory than the range that can be accessed with the reduced number of bits. For instance, if a call to **printf( )** is encoded with one of these instructions, the load may succeed if the object code is loaded near the kernel code, but fail if the object code is loaded farther away from the kernel image.

For additional information, see *Function Calls, Relative Branches, and Load Failures*, p.616.

Solution

T he loader prints the relocation type and offset as part of the error message to facilitate diagnostics. The offset and types can be retrieved with the **readelf -r** command. Relocation types are described in the ELF architecture supplement.

Recompile the object file using **-Xcode-absolute-far** for the Wind River compilers, and for GNU compilers, the appropriate *long call* option, **-mlongcall** (for PPC architecture). See the *VxWorks Architecture Supplement* for the appropriate options.

Missing Symbols

Symbols in code modules downloaded from the host do not appear from the kernel shell, and vice versa. Symbols created from the host shell are not visible from the kernel shell, or symbols created from the kernel shell are not visible from the host shell. Why is this happening, and how can I get them to appear?

Explanation

The symbol synchronization mechanism must be enabled separately on the host and target.

Solution

Check to see if the module and symbol synchronization is enabled for the target server as well as compiled into the image. For more information, see *12.4.5 Synchronizing Host and Kernel Modules List and Symbol Table*, p.623.

Kernel Object-Module Loader is Using Too Much Memory

Including the kernel loader causes the amount of available memory to be much smaller. How can I get more memory?

Explanation

Including the kernel loader causes the system symbol table to be included. This symbol table contains information about every global symbol in the compiled VxWorks image.

Using the kernel loader takes additional memory away from your application—most significantly for the target-resident symbol table required by the kernel loader.

Solution

Use the host tools rather than the target tools and remove all target tools from your VxWorks image.

Symbol Table Unavailable

The system symbol table failed to download onto my target. How can I use the kernel shell to debug the problem, since I cannot call functions by name?

Solution

Use addresses of functions and data, rather than using the symbolic names. The addresses can be obtained from the VxWorks image on the host, using the **nm***arch* utility.

The following is an example from a UNIX host:

```
> nmarch vxWorks | grep memShow
0018b1e8 T memShow
0018b1ac T memShowInit
```

Use this information to call the function by address from the kernel shell. (The parentheses are mandatory when calling by address.)

```
-> 0x0018b1e8 ()

 status   bytes    blocks   avg block  max block
------ --------- -------- ---------- ----------
current
  free   14973336     20      748666     12658120
  alloc  14201864    16163       878        -

cumulative
  alloc  21197888   142523       148        -
  value = 0 = 0x0
```

For modules that are relocated, use **nm** on the module to get the function address (which is the offset within the module's text segment) then add to that value the starting address of the text segment of the module when it was loaded in memory.

13
C++ Development

13.1 Introduction

This chapter provides information about C++ development for VxWorks using the Wind River and GNU toolchains.

⚠ **WARNING:** Wind River Compiler C++ and GNU C++ binary files are not compatible.

NOTE: This chapter provides information about facilities available in the VxWorks kernel. For information about facilities available to real-time processes, see the corresponding chapter in the *VxWorks Application Programmer's Guide*.

13.2 Configuring VxWorks for C++

By default, VxWorks includes only minimal C++ support. You can add C++ functionality by including any or all of the following components:

INCLUDE_CTORS_DTORS
(included in default kernels)
Ensures that compiler-generated initialization functions, including initializers for C++ static objects, are called at kernel start up time.

INCLUDE_CPLUS
Includes basic support for C++ applications. Typically this is used in conjunction with **INCLUDE_CPLUS_LANG**.

INCLUDE_CPLUS_LANG
Includes support for C++ language features such as **new**, **delete**, and exception handling.

INCLUDE_CPLUS_IOSTREAMS
Includes all library functionality.

INCLUDE_CPLUS_DEMANGLER
Includes the C++ demangler, which is useful if you are using the kernel shell loader because it provides for returning demangled symbol names to kernel shell symbol table queries. This component is added by default if both the **INCLUDE_CPLUS** and **INCLUDE_SYM_TBL** components are included in VxWorks.

13.3 C++ Code Requirements

Any VxWorks task that uses C++ must be spawned with the **VX_FP_TASK** option. By default, tasks spawned from host tools (such as the Wind Shell) automatically have **VX_FP_TASK** enabled.

WARNING: Failure to use the **VX_FP_TASK** option when spawning a task that uses C++ can result in hard-to-debug, unpredictable floating-point register corruption at run-time.

If you reference a (non-overloaded, global) C++ symbol from your C code you must give it C linkage by prototyping it using **extern "C"**:

```
#ifdef __cplusplus
extern "C" void myEntryPoint ();
#else
void myEntryPoint ();
#endif
```

You can also use this syntax to make C symbols accessible to C++ code. VxWorks C symbols are automatically available to C++ because the VxWorks header files use this mechanism for declarations.

Each compiler has its own C++ libraries and C++ headers (such as **iostream** and **new**). The C++ headers are located in the compiler installation directory rather than in *installDir***/vxworks-6.***x***/target/h**. No special flags are required to enable the compilers to find these headers.

NOTE: In releases prior to VxWorks 5.5, Wind River recommended the use of the flag **-nostdinc**. This flag *should not* be used with the current release since it prevents the compilers from finding headers such as **stddef.h**.

13.4 Using C++ in Signal Handlers and ISRs

Special care must be taken when using C++ code in signal handlers and ISRs. For information in this regard, see *4.18.5 Signal Handlers*, p.236 and *4.20.3 Writing and Debugging ISRs*, p.244.

13.5 Downloadable Kernel Modules in C++

C++ code that is downloaded into the VxWorks kernel should be linked into a single downloadable object module. It must also be *munched*, and any COMDAT or linkonce sections collapsed. VxWorks provides several strategies available for calling static constructors and destructors in downloadable modules.

13.5.1 Use a Single C++ Module

The VxWorks loader only supports C++ modules that are *self-contained*. A self-contained C++ module is one that does not use classes from other C++ modules, and whose classes are not used by other C++ modules. In particular, a module must either contain its own copy of the standard library, or not use the C++ standard library at all.

To produce self-contained modules, all C++ object files that are to be downloaded should be linked into a single downloadable object module.

Unloading a C++ module that is not self-contained may result in dangling references from objects created in other modules to data structures in the unloaded module. Specifically, this can happen if the iostreams portion of the standard library is initialized from a module that is later unloaded. In that case, any further use of iostreams may fail with a kernel exception (accessing an invalid address).

WARNING: C++ object files must be linked into one downloadable kernel module.

For information about the kernel loader, see *12.3 Kernel Object-Module Loader*, p.603.

13.5.2 Munching a C++ Application Module

Before a C++ module can be downloaded to the VxWorks kernel, it must undergo an additional host processing step, which for historical reasons, is called *munching*. Munching performs the following tasks:

- Initializes support for static objects.
- Ensures that the C++ run-time support calls the correct constructors and destructors in the correct order for all static objects.
- For the Wind River Compiler, collapses any COMDAT sections automatically; for the GNU compiler, collapses any linkonce automatically.

Munching must be performed after compilation and before download.

Munching Examples

For each toolchain, the following examples compile a C++ application source file, **hello.cpp**, run munch on the **.o**, compile the generated **ctdt.c** file, and link the application with **ctdt.o** to generate a downloadable module, **hello.out**.

Using the Wind River Toolchain

1. Compile the source code:

```
$ dcc -tPPC604FH:vxworks61  -Xlocal-data-area-static-only -XO  \
-IinstallDir/vxworks-6.1/target/h -DCPU=PPC32 -DTOOL_FAMILY=diab
-DTOOL=diab  \ -D_WRS_KERNEL  -c hello.cpp
```

2. Munch the object file:

```
$ ddump -Ng hello.o | tclsh  \
installDir/vxworks-6.1/host/resource/hutils/tcl/munch.tcl -c ppc > ctdt.c
```

3. Compile the munch output:

```
$ dcc -tPPC604FH:vxworks61  -Xlocal-data-area-static-only -XO  \
-IinstallDir/vxworks-6.1/target/h -DCPU=PPC32 -DTOOL_FAMILY=diab
-DTOOL=diab  \ -D_WRS_KERNEL  -c ctdt.c
```

4. Link the original object file with the munched object file to create a downloadable module:

```
$ dld -tPPC604FH:vxworks61 -X -r4 -o hello.out hello.o ctdt.o
```

NOTE: The **-r4** option collapses any COMDAT sections contained in the input files.

Using the GNU Toolchain

1. Compile the source code:

```
ccppc -mcpu=604 -mstrict-align  -O2 -fno-builtin  \
-IinstallDir/vxworks-6.1/target/h \
-DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu -c hello.cpp
```

2. Munch the object file:

```
nmppc hello.o | wtxtcl installDir/vxworks-6.1/host/src/hutils/munch.tcl \
-c ppc > ctdt.c
```

3. Compile the munch output:

```
ccppc -mcpu=604 -mstrict-align  -fdollars-in-identifiers -O2 \
-fno-builtin -IinstallDir/vxworks-6.1/target/h \
-DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu  -c ctdt.c
```

4. Link the original object file with the munched object file to create a downloadable module:

```
ccppc -r -nostdlib -Wl,-X \
-T installDir/vxworks-6.1/target/h/tool/gnu/ldscripts/link.OUT \
-o hello.out hello.o ctdt.o
```

NOTE: The VxWorks kernel object module loader does not support *linkonce* sections directly. Instead, the linkonce sections must be merged and collapsed into standard **text** and **data** sections before loading. The GNU **-T** option collapses any linkonce sections contained in the input files.

Using a Generic Makefile Rule

If you use the VxWorks makefile definitions, you can write a simple munching rule which (with appropriate definitions of CPU and TOOL) works across all architectures for both GNU and Wind River Compiler toolchains.

```
CPU     = PPC604
TOOL    = gnu

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp

default : hello.out

%.o : %.cpp
    $(CXX) $(C++FLAGS) -c $<

%.out : %.o
    $(NM) $*.o | $(MUNCH) > ctdt.c
    $(CC) $(CFLAGS) $(OPTION_DOLLAR_SYMBOLS) -c ctdt.c
    $(LD_PARTIAL) $(LD_PARTIAL_LAST_FLAGS) -o $@ $*.o ctdt.o
```

After munching, downloading, and linking, the static constructors and destructors are called. This step is described next.

13.5.3 Calling Static Constructors and Destructors Interactively

The kernel loader provides both manual and automatic options for calling static constructors and destructors.

Automatic invocation is the default strategy. Static constructors are executed just after the module is downloaded to the target and before the module loader returns to its caller. Static destructors are executed just prior to unloading the module.

Manual invocation means that the user must call static constructors explicitly, after downloading the module, but before running the application. It also requires the user to call static destructors explicitly, after the task finishes running, but before unloading the module.

Static constructors are called by invoking **cplusCtors( )**. Static destructors are called by invoking **cplusDtors( )**. These routines take an individual module name as an argument. However, you can also invoke all of the static constructors or destructors that are currently loaded into a system by calling these routines without an argument.

CAUTION: When using the manual invocation method, constructors for each module must only be run once.

You can change the strategy for calling static constructors and destructors at run-time with the **cplusXtorSet( )** routine. To report on the current strategy, call **cplusStratShow( )**.

For more information on the routines mentioned in this section, see the API entries in the online reference manuals.

Also see *12.3.3 Summary List of Kernel Object-Module Loader Options*, p.606 for information about the C++ loader and unloader options.

13.6 C++ Compiler Differences

The Wind River C++ Compiler uses the Edison Design Group (EDG) C++ front end. It fully complies with the ANSI C++ Standard. For complete documentation on the Wind River Compiler and associated tools, see the *Wind River C/C++ Compiler User's Guide*.

The GNU compiler supports most of the language features described in the ANSI C++ Standard. For complete documentation on the GNU compiler and on the associated tools, see the *GNU ToolKit User's Guide*.

WARNING: Wind River Compiler C++ and GNU C++ binary files are not compatible.

The following sections briefly describe the differences in compiler support for template instantiation and run-time type information.

13.6.1 Template Instantiation

In C, every function and variable used by a program must be defined in exactly one place (more precisely one *translation unit*). However, in C++ there are entities which have no clear point of definition but for which a definition is nevertheless required. These include template specializations (specific instances of a generic template; for example, **std::vector** *int*), out-of-line bodies for inline functions, and virtual function tables for classes without a non-inline virtual function. For such entities a source code definition typically appears in a header file and is included in multiple translation units.

To handle this situation, both the Wind River Compiler and the GNU compiler generate a definition in every file that needs it and put each such definition in its own section. The Wind River compiler uses *COMDAT* sections for this purpose, while the GNU compiler uses *linkonce* sections. In each case the linker removes duplicate sections, with the effect that the final executable contains exactly one copy of each needed entity.

NOTE: Only the WRC linker can be used to process files containing COMDAT sections. Similarly only the GNU linker can be used on files containing linkonce sections. Furthermore the VxWorks target and host loaders are not able to process COMDAT and linkonce sections. A fully linked VxWorks image will not contain any COMDAT or linkonce sections. However intermediate object files compiled from C++ code may contain such sections. To build a downloadable C++ module, or a file that can be processed by any linker, you must perform an intermediate link step using the **-r5** option (WRC) or specifying the **link.OUT** linker script (GCC). See 10.4.1 Munching C++ Application Modules for full details. (Note that while the the **-r5** and **-r4** options—the latter referred to elsewhere in this chapter—both collapse COMDAT files, their overall purpose is different, and their use is mutually exclusive in a single linker command.)

It is highly recommended that you use the default settings for template instantiation, since these combine ease-of-use with minimal code size. However it is possible to change the template instantiation algorithm; see the compiler documentation for details.

Wind River Compiler

The Wind River Compiler C++ options controlling multiple instantiation of templates are:

-Xcomdat

This option is the default. When templates are instantiated implicitly, the generated **code** or **data** section are marked as **comdat**. The linker then collapses identical instances marked as such, into a single instance in memory.

CAUTION: If code is going to be used as downloadable kernel modules, the **-r4** option must be used to collapse any COMDAT sections contained in the input files. See *13.5.2 Munching a C++ Application Module*, p.650.

-Xcomdat-off

Generate template instantiations and **inline** functions as static entities in the resulting object file. Can result in multiple instances of static member-function or class variables.

For greater control of template instantiation, the **-Ximplicit-templates-off** option tells the compiler to instantiate templates only where explicitly called for in source code; for example:

```
template class A<int>;    // Instantiate A<int> and all member functions.
template int f1(int);     // Instantiate function int f1{int).
```

GNU Compiler

The GNU C++ compiler options controlling multiple instantiation of templates are:

-fimplicit-templates

This option is the default. Template instantiations and out-of-line copies of **inline** functions are put into special *linkonce* sections. Duplicate sections are merged by the linker, so that each instantiated template appears only once in the output file.

CAUTION: The VxWorks dynamic loader does not support *linkonce* sections directly. Instead, the linkonce sections must be merged and collapsed into standard **text** and **data** sections before loading. This is done with a special link step described in *13.5.2 Munching a C++ Application Module*, p.650.

-fno-implicit-templates

This is the option for explicit instantiation. Using this strategy explicitly instantiates any templates that you require.

13.6.2 Run-Time Type Information

Both compilers support Run-time Type Information (RTTI), and the feature is enabled by default. This feature adds a small overhead to any C++ program containing classes with virtual functions.

For the Wind River Compiler, the RTTI language feature can be disabled with the **-Xrtti-off** flag.

For the GNU compiler, the RTTI language feature can be disabled with the **-fno-rtti** flag.

13.7 Namespaces

Both the Wind River and GNU C++ compilers supports namespaces. You can use namespaces for your own code, according to the C++ standard.

The C++ standard also defines names from system header files in a *namespace* called **std**. The standard requires that you specify which names in a standard header file you will be using.

The following code is technically invalid under the latest standard, and will not work with this release. It compiled with a previous release of the GNU compiler, but will not compile under the current releases of either the Wind River or GNU C++ compilers:

```
#include <iostream.h>
int main()
    {
        cout << "Hello, world!" << endl;
    }
```

The following examples provide three correct alternatives illustrating how the C++ standard would now represent this code. The examples compile with either the Wind River or the GNU C++ compiler:

```
// Example 1
    #include <iostream>
    int main()
        {
            std::cout << "Hello, world!" << std::endl;
        }
```

```
// Example 2
    #include <iostream>
    using std::cout;
    using std::endl;
    int main()
        {
            cout << "Hello, world!" << endl;
        }

// Example 3
    #include <iostream>
    using namespace std;
    int main()
        {
            cout << "Hello, world!" << endl;
        }
```

13.8 C++ Demo Example

The **factory** demo example demonstrates various C++ features in the kernel, including the Standard Template Library, user-defined templates, run-time type information, and exception handling. This demo is located in *installDir*/**vxworks-6.***x***/target/usr/apps/samples/cplusplus/factory**.

To create, compile, build, and run this demo program you can use either Workbench or the command line, as shown below.

For the **factory** demo, your kernel must include the following components in VxWorks:

- **INCLUDE_CPLUS**
- **INCLUDE_CPLUS_LANG**
- **INCLUDE_CPLUS_IOSTREAMS**

In addition, for GNU only, include the following components:

- **INCLUDE_CPLUS_STRING**
- **INCLUDE_CPLUS_STRING_IO**

To build **factory** from the command line, simply copy the **factory** sources to the BSP directory, as shown below:

```
cd installDir/vxworks-6.1/target/config/bspDir
cp installDir/vxworks-6.1/target/src/demo/cplusplus/factory/factory.* .
```

Then, to build a bootable image containing the **factory** example, run make as shown below:

```
make ADDED_MODULES=factory.o
```

and boot the target.

To build a downloadable image containing the **factory** example, run make as shown below:

```
make factory.out
```

Then, from the WindSh, load the **factory** module, as shown below:

```
ld < factory.out
```

Finally, to run the **factory** demo example, type at the shell:

```
-> testFactory
```

Full documentation on what you should expect to see is provided in the source code comments for the demo program.

PART II
Multiprocessing Technologies

14
Overview of Multiprocessing Technologies

14.1 Introduction

VxWorks provides various multiprocessor technologies, for asymmetric multiprocessing (AMP) and symmetric multiprocessing (SMP) systems. These include VxWorks SMP (an optional product), shared memory objects (VxMP), distributed shared memory (DSHM), TIPC over DSHM, and message channels.

14.2 VxWorks SMP

VxWorks SMP is a configuration of VxWorks designed for symmetric multiprocessing (SMP). It provides the same distinguishing RTOS characteristics of performance and determinism as the uniprocessor (UP) configuration. The differences between the SMP and UP configurations are limited, and strictly related to support for multiprocessing. For detailed information, see *15. VxWorks SMP*.

14.3 Shared Memory Objects

VxMP is a VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running in separate instances of VxWorks. For detailed information, see *16. Shared-Memory Objects: VxMP*.

14.4 Distributed Shared Memory

The VxWorks distributed shared memory (DSHM) facility is a middleware subsystem that allows multiple services to communicate over different types of buses that support shared-memory communication. For detailed information, see *17. Distributed Shared Memory: DSHM*.

14.5 TIPC Over Distributed Shared Memory

For information about TIPC, and about using TIPC with DSHM, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*.

14.6 Message Channels

Message channels are a socket-based facility that provides for inter-task communication within a memory boundary, between memory boundaries (kernel and processes), between nodes (processors) in a multi-node cluster, and between between multiple clusters. For detailed information, see *18. Message Channels*.

15
VxWorks SMP
Optional Product

15.1 Introduction

VxWorks SMP is a configuration of VxWorks designed for symmetric multiprocessing (SMP). It provides the same distinguishing RTOS characteristics of performance and determinism as the uniprocessor (UP) configuration. The differences between the SMP and UP configurations are limited, and strictly related to support for multiprocessing.

This chapter describes the features provided by VxWorks to support symmetric multiprocessing. It discusses the features that are unique to the SMP configuration, as well as the differences in operating system facilities and programming practices used for the UP configuration and the SMP configuration. It also provides guidelines for migrating UP code to SMP code. In this chapter, the terms *VxWorks SMP* and *VxWorks UP* is used to identify the uniprocessor and symmetric multiprocessing configurations of VxWorks, respectively.

For information about features that are common to both the VxWorks SMP and VxWorks UP configurations—such as multitasking, I/O, file systems, and so on—see *Part ICore Technologies*, p.1.

NOTE: SMP support for VxWorks is available as an optional product. However, default SMP system images for the simulators (including the WDB target agent, kernel shell, object module loader, and so on) are provided with the standard VxWorks installation as an introduction to the SMP product. For location information, see *Default VxWorks SMP Images*, p.674.

15.2 Technology Overview

Multiprocessing systems include two or more processors in a single system. Symmetric multiprocessing (SMP) is a variant of multiprocessing technology in which one instance of an operating system controls all processors, and in which memory is shared. SMP differs from asymmetric multiprocessing (AMP) in that an AMP system has a separate instance of an operating system executing on each processor (and each instance may or may not be the same type of operating system).

15.2.1 Terminology

The terms *CPU* and *processor* are often used interchangeably in computer documentation. However, it is useful to distinguish between the two for hardware that supports SMP. In this guide, particularly in the context of VxWorks SMP, the terms are used as follows:

CPU
A single processing entity capable of executing program instructions and processing data (also referred to as a *core*, as in *multicore*).

processor
A silicon unit that contains one or more CPUs.

multiprocessor
A single hardware system with two or more processors.

uniprocessor
A silicon unit that contains a single CPU.

For example, a dual-core MPC8641D would be described as a processor with two CPUs. A quad-core Broadcom 1480 would be described as a processor with four CPUs.

Uniprocessor code may not always execute properly on an SMP system, and code that has been adapted to execute properly on an SMP system may still not make optimal use of symmetric multiprocessing. The following terms are therefore used to clarify the state of code in relation to SMP:

SMP-ready
Runs correctly on an SMP operating system, although it may not make use of more than one CPU (that is, does not take full advantage of concurrent execution for better performance).

SMP-optimized
Runs correctly on an SMP operating system, uses more than one CPU, and takes sufficient advantage of multitasking and concurrent execution to provide performance gains over a uniprocessor implementation.

15.2.2 VxWorks SMP Operating System Features

With few exceptions, the SMP and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few routines. There is binary compatibility for both kernel and RTP applications between UP and SMP configurations of VxWorks (for the same VxWorks release), as long as the applications are based on the subset of APIs used by VxWorks SMP. A few uniprocessor APIs are not suitable for an SMP system, and they are therefore not provided. Similarly, SMP-specific APIs are not relevant to a uniprocessor system—but default to appropriate uniprocessor behaviors (such as task spinlocks defaulting to task locking), or have no effect.

VxWorks SMP is designed for symmetric target hardware. That is, each CPU has equivalent access to all memory and all devices. VxWorks SMP can therefore run on targets with multiple single-core processors or with multicore processors, as long as they provide a uniform memory access (UMA) architecture with hardware-managed cache-coherency.

This section provides a brief overview of areas in which VxWorks offers alternate or additional features designed for symmetric multiprocessing. The topics are covered in detail later in this chapter.

Multitasking

SMP changes the conventional uniprocessor paradigm of priority-based preemptive multitasking programming, because it allows true concurrent execution of tasks and handling of interrupts. This is possible because multiple tasks can run on multiple CPUs, while being controlled by a single instance of an operating system.

Uniprocessor multitasking environments are often described as ones in which multiple tasks can run *at the same time*, but the reality is that the CPU only executes one task at a time, switching from one task to the another based on the characteristics of the scheduler and the arrival of interrupts. In an SMP system concurrent execution is a fact and not an illusion.

Scheduling

VxWorks SMP provides a priority-based preemptive scheduler, like VxWorks UP. In both VxWorks UP and VxWorks SMP, tasks are scheduled—and real-time processes (RTPs) are not. However, the VxWorks SMP scheduler is different from the uniprocessor scheduler in that it also manages the concurrent execution of tasks on different CPUs.

Mutual Exclusion

Because SMP systems allow for truly concurrent execution, the uniprocessor mechanisms for disabling (masking) interrupts and for suspending task preemption in order to protect critical regions are inappropriate for—and not available in—an SMP operating system. Enforcing interrupt masking or suspending task preemption across all CPUs would defeat the advantages of truly concurrent execution and drag multiprocessing performance down towards the level of a uniprocessor system.

VxWorks SMP therefore provides specialized mechanisms for mutual exclusion between tasks and interrupts executing and being received (respectively) simultaneously on different CPUs. In place of uniprocessor task and interrupt locking routines—such as **taskLock( )** and **intLock( )**—VxWorks SMP provides spinlocks, atomic memory operations, and CPU-specific mutual exclusion facilities.

CPU Affinity

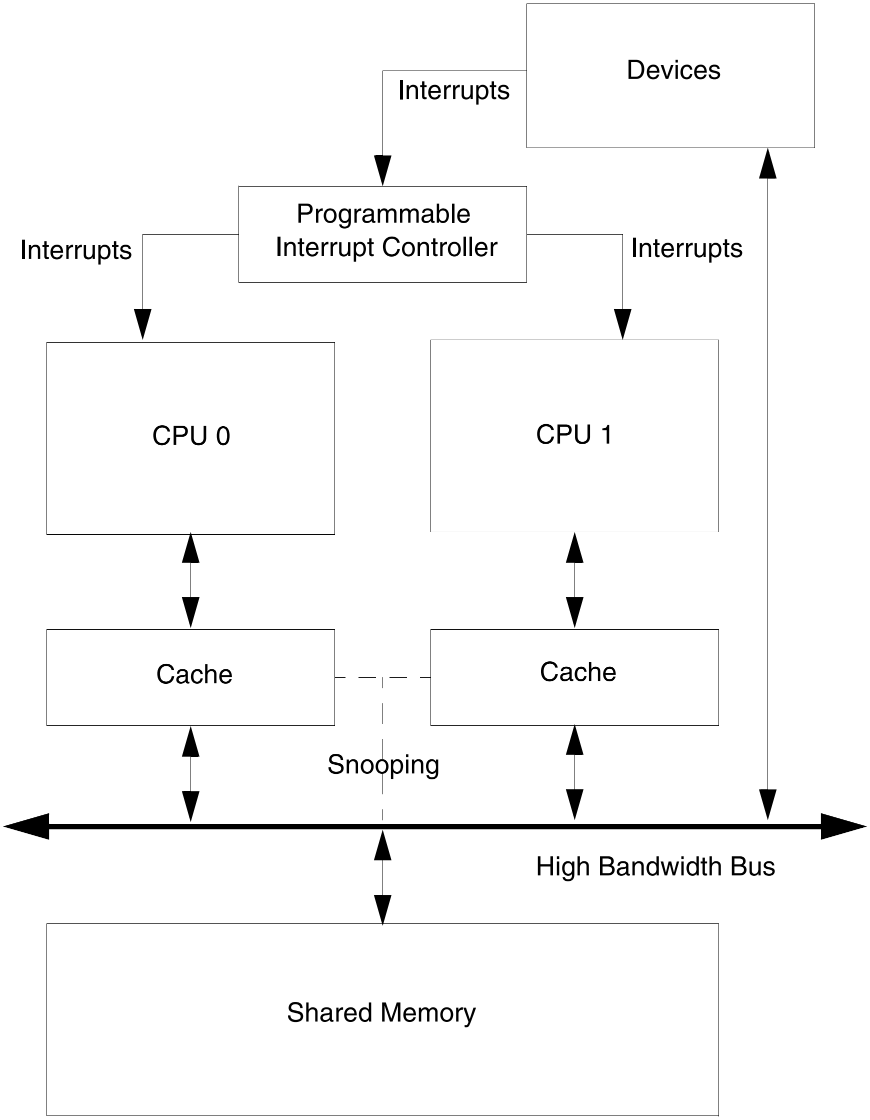
By default, any task can run on any of the CPUs in the system (which generally provides the best load balancing) and interrupts are routed to CPU 0 (the bootstrap CPU). There are instances, however, in which it is useful to assign specific tasks or interrupts to a specific CPU. VxWorks SMP provides this capability, which is referred to a as *CPU affinity*.

15.2.3 VxWorks SMP Hardware

The hardware required for use with VxWorks SMP must consist of symmetric multiprocessors—either multicore processors or hardware systems with multiple single CPUs. The processors must be identical, all memory must be shared between the CPUs (none may be local to a CPU), and all devices must be equally accessible from all CPUs.That is, targets for VxWorks SMP must adhere to the uniform memory access (UMA) architecture.

Figure 15-1 illustrates the typical target hardware for a dual CPU SMP system.

Figure 15-1 **SMP Hardware**



Regardless of the number of CPUs (typically 2, 4 or 8) in an SMP system, the important characteristics are the same:

- Each CPU accesses the very same physical memory subsystem; there is no memory local to a CPU. This means it is irrelevant which CPU executes code.
- Each CPU has its own memory management unit that allows concurrent execution of tasks with different virtual memory contexts. For example, CPU 0 can execute a task in RTP 1 while CPU 1 executes a task in RTP 2.
- Each CPU has access to all devices. Interrupts from these devices can be routed to any one of the CPUs through an a programmable interrupt controller. This means that it is irrelevant which CPU executes interrupt service routines (ISRs) when handling interrupts.
- Tasks and ISRs can be synchronized across CPUs and mutual exclusion enforced by using spinlocks.
- Bus snooping logic ensures the data caches between CPUs are always coherent. This means that the operating system does not normally need to perform special data cache operations order to maintain coherent caches. However, this implies that only memory access attributes that allow bus snooping are used in the system. Restrictions in terms of memory access modes allowed in an SMP system, if any, are specific to a hardware architecture.

15.2.4 Comparison of VxWorks SMP and AMP

The features of VxWorks SMP may be highlighted by comparison with the way VxWorks is used in asymmetric multiprocessing (AMP), using the same target hardware in both cases. VxWorks AMP technologies include VxMP, TIPC (over shared memory), and distributed shared memory (DSHM). The relationship between CPUs and basic uses of memory in SMP and AMP systems are illustrated in Figure 15-2 and Figure 15-3.

Figure 15-2 **SMP System**

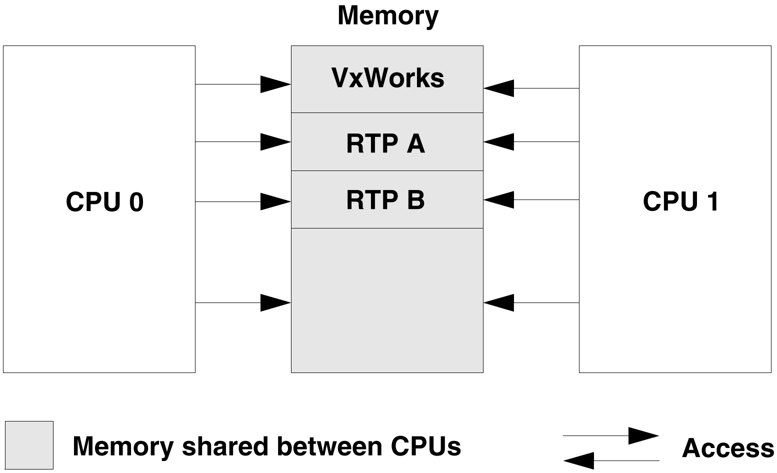
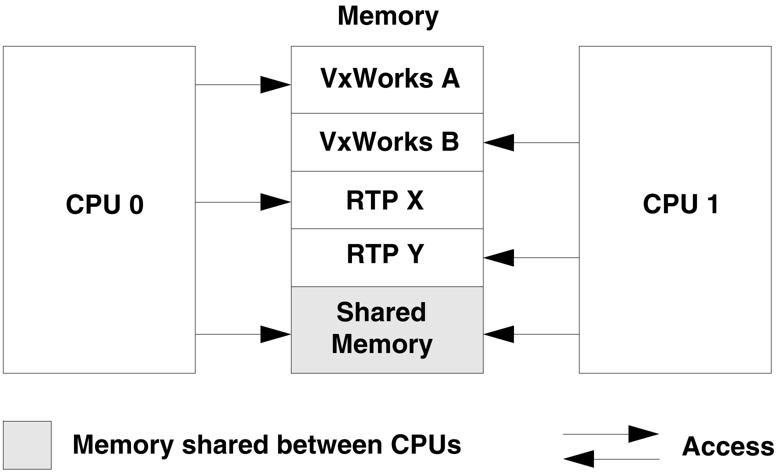


Figure 15-3 **AMP System**



In an SMP configuration the entire physical memory space is shared between the CPUs. This memory space is used to store a single VxWorks SMP image (text, data, bss, heap). It is also used to store any real-time processes (RTPs) that are created during the lifetime of the system. Because both CPUs can potentially read from, write to and execute any memory location, any kernel task or user (RTP) task can be executed by either CPU.

In an AMP configuration there is one copy of the VxWorks image in memory for *each* CPU. Each operating system image can only be accessed by the CPU to which it belongs. It is therefore impossible for CPU 1 to execute kernel tasks residing in VxWorks CPU 0's memory, or the reverse. The same situation applies for RTPs. An RTP can only be accessed and executed by the instance of VxWorks from which it was started.

In an AMP system some memory is shared, but typically the sharing is restricted to reading and writing data. For example, for passing messages between two instances of VxWorks. Hardware resources are mostly divided between instances of the operating system, so that coordination between CPUs is only required when accessing shared memory.

With an SMP system, both memory and devices are shared between CPUs, which requires coordination within the operating system to prevent concurrent access to shared resources.

15.3 VxWorks SMP Configuration and Build

VxWorks SMP can be configured and built using the standard Wind River Workbench and **vxprj** facilities. VxWorks SMP images *cannot* be created with the legacy BSP directory configuration and build method (using **config.h** and **make**).

Workbench provides a VxWorks Image Project (VIP) option for creating VxWorks SMP projects. The **vxprj** provides the **-smp** option for configuring and building VxWorks SMP from the command line. For example:

```
vxprj create -smp hpcNet8641 diab
```

⚠ **CAUTION:** VxWorks SMP does not support MMU-less configurations.

⚠ **CAUTION:** Boot loaders for VxWorks SMP must not be built with the SMP build option—neither with Workbench nor with **vxprj**. For more information about boot loaders for VxWorks SMP, see *15.4 Booting VxWorks SMP*, p.676.

Default VxWorks SMP Images

Default VxWorks SMP images are provided in project directories parallel to those for VxWorks UP images. For example, for the hpcNet8641 BSP, the directories are as follows:

- *installDir*/**vxworks-6.***x*/**target/proj/hpcNet8641_diab_smp**
- *installDir*/**vxworks-6.***x*/**target/proj/hpcNet8641_gnu_smp**
- *installDir*/**vxworks-6.***x*/**target/proj/hpcNet8641_diab**
- *installDir*/**vxworks-6.***x*/**target/proj/hpcNet8641_gnu**

Debug Versions of Spinlock Components

The **INCLUDE_SPINLOCK_DEBUG** component provides versions of spinlocks that are useful for debugging SMP applications. By default the standard **INCLUDE_SPINLOCK** component is included in VxWorks SMP; if **INCLUDE_SPINLOCK_DEBUG** is included, it is removed automatically. For more information, see *Debug Versions of Spinlocks*, p.681.

CPU Configuration Parameters

There are several configuration parameters that are specific to VxWorks SMP, which are provided by the **INCLUDE_KERNEL** component. These parameters are as follows:

VX_SMP_NUM_CPUS
Defines the number of CPUs that should be enabled for VxWorks SMP. The maximum number of CPUs for each architecture is as follows: ARM = 4, IA32 = 8, MIPS = 16, PowerPC = 8, VxWorks Simulator = 32.

ENABLE_ALL_CPUS
Enables all CPUs that have been configured for the system's use with **VX_SMP_NUM_CPUS**. The default is **TRUE**, in which case VxWorks boots with all CPUs enabled and running. The parameter can be set to **FALSE** for debugging purposes, in which case only CPU 0 (the bootstrap CPU) will be enabled by the VxWorks initialization code. The **kernelCpuEnable( )** routine can then be used to enable a specific CPU once the system has booted.

VX_ENABLE_CPU_TIMEOUT

The time-out value (in seconds) for the period during which additional cores may be enabled. When **kernelCpuEnable( )** is called, it waits for the time defined by **VX_ENABLE_CPU_TIMEOUT** for the additional core to come up. If **ENABLE_ALL_CPUS** is set to **TRUE**, the value of **VX_ENABLE_CPU_TIMEOUT** is used as the time-out period for enabling all CPUs.

Idle Task Configuration Parameters

The **INCLUDE_PROTECT_IDLE_TASK_STACK** component provides a set of parameters for configuring the exception stack for the CPU idle task. For information about idle tasks, see *CPU Idle Tasks*, p.678.

IDLE_TASK_EXCEPTION_STACK_SIZE
Size (in bytes) of the idle tasks' exception stacks.

IDLE_TASK_EXC_STACK_OVERFLOW_SIZE
Size (in bytes) of the overflow protection area adjacent to the idle task's exception stack.

IDLE_TASK_EXC_STACK_UNDERFLOW_SIZE.
Size (in bytes) of the underflow protection area adjacent to the idle task's exception stack.

15.4 Booting VxWorks SMP

Booting VxWorks SMP is essentially the same operation as booting VxWorks UP. The boot loader is simply responsible for booting the *bootstrap CPU* (referred to as CPU 0). The boot loader has no knowledge of any other CPUs. Once the VxWorks SMP image is loaded on CPU 0 and started, that instance of the operating system enables the other CPUs in the system.

VxWorks SMP can, however, be configured so that the image loaded onto CPU 0 does not automatically enable the other CPUs, which can then be enabled interactively or programmatically. For more information in this regard, see *ENABLE_ALL_CPUS*, p.675).

⚠ **CAUTION:** Boot loaders for VxWorks SMP must not be built with the SMP build option—neither with the SMP selection for a Workbench VxWorks Image Project (VIP), nor with the **-smp** option for vxprj. Boot loaders built with the SMP build option will not function properly.

For detailed information about VxWorks boot loaders, see *3. Boot Loader*.

15.5 Programming for VxWorks SMP

Programming for VxWorks SMP and VxWorks UP is in many respects the same.

With few exceptions, the SMP and uniprocessor (UP) configurations of VxWorks share the same API—the difference amounts to only a few routines. There is binary compatibility for both kernel and RTP applications between UP and SMP configurations of VxWorks (for the same VxWorks release), as long as the applications are based on the subset of APIs used by VxWorks SMP. A few uniprocessor APIs are not suitable for an SMP system, and they are therefore not provided. Similarly, SMP-specific APIs are not relevant to a uniprocessor system—but default to appropriate uniprocessor behaviors (such as task spinlocks defaulting to task locking), or have no effect.

However, because of the nature of SMP systems, SMP programming requires special attention to the mechanisms of mutual exclusion, and to design considerations that allow for full exploitation of the capabilities of a multiprocessing system. Also note that VxWorks SMP maintains an *idle task* for each CPU, and that idle tasks must not be interfered with.

SMP and Mutual Exclusion

The use of mutual exclusion facilities is one of the critical differences between uniprocessor and SMP programming. While some facilities are the same for VxWorks UP and VxWorks SMP, others are necessarily different. In addition, reliance on implicit synchronization techniques—such as relying on task priority instead of explicit locking—do not work in an SMP system (for more information on this topic, see *15.15.4 Implicit Synchronization of Tasks*, p.706).

Unlike uniprocessor systems, SMP systems allow for truly concurrent execution, in which multiple tasks may execute, and multiple interrupts may be received and serviced, all at the same time. In most cases, the same mechanisms—semaphores, message queues, and so on—can be used in both uniprocessor and SMP systems for mutual exclusion and coordination of tasks (see *4.8 Intertask and Interprocess Communication*, p.193).

However, the specialized uniprocessor mechanisms for disabling (masking) interrupts and for suspending task preemption in order to protect critical regions are inappropriate for—and not available in—an SMP system. This is because they would defeat the advantages of truly concurrent execution by enforcing masking or preemption across all CPUs, and thus drag a multiprocessing system down towards the performance level of uniprocessor system.

The most basic differences for SMP programming therefore have to do with the mechanisms available for mutual exclusion between tasks and interrupts executing and being received (respectively) on different CPUs. In place of uniprocessor task and interrupt locking routines—such as **taskLock( )** and **intLock( )**—VxWorks SMP provides the following facilities:

- spinlocks for tasks and ISRs
- CPU-specific mutual exclusion for tasks and ISRs
- atomic memory operations
- memory barriers

As with the uniprocessor mechanisms used for protecting critical regions, spinlocks and CPU-specific mutual exclusion facilities should only used when they are guaranteed to be in effect for very short periods of time. The appropriate use of these facilities is critical to making an application *SMP-ready* (see *15.2.1 Terminology*, p.667).

Note that both spinlocks and semaphores provide full memory barriers (in addition to the memory barrier macros themselves).

For more information about these topics, see *15.6 Spinlocks for Mutual Exclusion and Synchronization*, p.679, *15.7 CPU-Specific Mutual Exclusion*, p.685, *15.8 Memory Barriers*, p.687, and *15.9 Atomic Memory Operations*, p.690.

CPU Affinity for Interrupts and Tasks

By default, any task can run on any of the CPUs in the system (which generally provides the best load balancing) and interrupts are routed to CPU 0. There are cases, however, in which it is useful to assign tasks or interrupts to a specific CPU. VxWorks SMP provides this capability, which is referred to a as *CPU affinity*.

For more information about interrupt and CPU affinity, see *15.10 CPU Affinity*, p.691.

CPU Idle Tasks

VxWorks SMP includes a per-CPU idle task that does not exist in VxWorks UP. The idle task has the lowest priority in the system, below the range permitted for application use (for more information, see *4.3.1 Task Priorities*, p.166). Idle tasks make an SMP system more efficient by providing task context when a CPU enters and exits an idle state.

The existence of idle tasks does not affect the ability of a CPU to go to sleep (when power management is enabled) if there is no work to perform. Do not perform any operations that affect the execution of an idle task.

The **kernelIsCpuIdle( )** and **kernelIsSystemIdle( )** routines provide information about whether a specific CPU is executing an idle task, or whether all CPUs are executing idle tasks (respectively).

For information about configuration options for idle tasks, see *15.3 VxWorks SMP Configuration and Build*, p.674.

WARNING: Do not suspend, stop, change the priority, attempt a task trace, or any similar operations on an idle task. Deleting, suspending, or stopping an idle task causes the system to crash due to an exception in the scheduler. Changing the priority of an idle task to a higher priority puts the CPU into a low power mode prematurely. That is, simply do not use the task ID (**tid**) of an idle task as a parameter to any VxWorks routine except **taskShow( )**.

RTP Applications

As in VxWorks UP systems, RTP (user mode) applications have a more limited set of mutual exclusion and synchronization mechanisms available to them than kernel code or kernel applications. In VxWorks SMP, they can make use of semaphores and atomic operations, but not spinlocks, memory barriers, or CPU-specific mutual exclusion mechanisms. In addition, the **semExchange( )** routine provides for an atomic give and exchange of semaphores.

Optimization for SMP

Using the appropriate facilities in the appropriate manner alone allows an application to execute properly on an SMP system, but does not necessarily take full advantage of the multiprocessing capabilities of the hardware. In order to do so, the design of the application must be geared to exploiting the advantages offered by SMP.

For information in this regard, see *15.13 Optimizing SMP Performance*, p.699.

15.6 Spinlocks for Mutual Exclusion and Synchronization

Spinlocks provide a facility for short-term mutual exclusion and synchronization in an SMP system. Spinlocks must be explicitly taken and released. While semaphores can also be used for mutual exclusion and synchronization, spinlocks are designed for use in situations comparable to those in which **taskLock( )** and **intLock( )** are used in VxWorks UP. Semaphores should be used in an SMP system for the same purposes as in a uniprocessor system. (Note that both spinlocks and semaphores provide full memory barriers.)

One of the unique characteristics of VxWorks spinlocks is that they are implemented with algorithms that ensure that they are *fair*, meaning that they are deterministic in the time between the request and take, and they operate in as close to FIFO order as possible. (For information on the different types of spinlocks, see *Types of Spinlocks*, p.680.)

For information about why uniprocessor mechanisms are not supported on VxWorks SMP for interrupt locking and suspension of task preemption, and SMP alternatives, see *15.15.5 Synchronization and Mutual Exclusion Facilities*, p.707, *Interrupt Locking: intLock( ) and intUnlock( )*, p.710, *Task Locking: taskLock( ) and* *taskUnlock( )*, p.711, and *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p.711.

NOTE: Spinlocks are not available to RTP (user-mode) applications.

Spinlocks as Full Memory Barriers

VxWorks spinlocks operate as full memory barriers between acquisition and release. A full memory barrier forces both read and write memory access operations to be performed in strict order. The process of updating data structures is therefore fully completed between the time a spinlock is acquired and released.

Types of Spinlocks

VxWorks SMP provides two types of spinlocks:

- *ISR-callable spinlocks*, which are used to address contention between ISRs—or between a task and other tasks and ISRs. They disable (mask) interrupts on the local CPU. When called by tasks they suspend task preemption on the local CPU as well.

- *Task-only spinlocks*, which are used to address contention between tasks (and not ISRs). They suspend task preemption on the local CPU.

The *local CPU* is the one on which the spinlock call is performed. For detailed information about spinlocks, see *15.6.1 ISR-Callable Spinlocks*, p.682 and *15.6.2 Task-Only Spinlocks*, p.682.

Spinlock Behavior and Usage Guidelines

Unlike the behavior associated with semaphores, a task that attempts to take a spinlock that is already held by another task does not pend; instead it continues executing, simply spinning in a tight loop waiting for the spinlock to be freed.

The terms *spinning* and *busy waiting*—which are both used to describe this activity—provide insight into both the advantages and disadvantages of spinlocks. Because a task (or ISR) continues execution while attempting to take a spinlock, the overhead of rescheduling and context switching can be avoided (which is not the case with a semaphore). On the other hand, spinning does no useful work, and ties up one or more of the CPUs.

Spinlocks should therefore only be used when they are likely to be efficient; that is, when they are going to be held for very short periods of time (as with **taskLock( )** and **intLock( )** in a uniprocessor system). If a spinlock is held for a long period of time, the drawbacks are similar to **intLock( )** and **taskLock( )** being held for a long time in VxWorks UP—increased interrupt and task latency.

Acquisition of a spinlock on one CPU does not affect the processing of interrupts or scheduling of tasks on other CPUs. Tasks cannot be deleted while they hold a spinlock.

For detailed cautionary information about spinlock use, see *15.6.3 Caveats With Regard to Spinlock Use*, p.683 and *15.6.4 Routines Restricted by Spinlock Use*, p.683.

Debug Versions of Spinlocks

The debug version of VxWorks spinlocks (provided with the **INCLUDE_SPINLOCK_DEBUG** component) is designed for use while developing applications that use spinlocks. It allows for catching violations of guidelines for appropriate spinlock use (for information in this regard, see *15.6.3 Caveats With Regard to Spinlock Use*, p.683).

The following is a list of checks that are performed by the debug version of spinlocks:

- **task-only take with spinLockTaskTake( )**
  - Calling from an ISR context results in an error.
  - Recursive taking of a spinlock results in an error.
  - Nested taking of spinlocks results in an error.
- **task-only give with spinLockTaskGive( )**
  - Calling from an ISR context results in an error.
  - Attempting to give up a spinlock without first acquiring a spinlock results in an error.
- **ISR-only take with spinLockIsrTake( )**
  - Recursive taking of a spinlock results in an error.
  - Nested taking of spinlocks result in an error.
- **ISR-only give with spinLockIsrGive( )**
  - Attempting to give up a spinlock without first acquiring a spinlock results in an error.

Errors are handled by the error detection and reporting facility, in the form of a fatal kernel error with an appropriate error string (for information about the error detection and reporting facility, see *11. Error Detection and Reporting*).

15.6.1 ISR-Callable Spinlocks

Spinlocks that are used to address contention between ISRs—or between a task and other tasks and ISRs—are referred to as *ISR-callable spinlocks*.

These spinlocks can be acquired by both tasks and ISRs. They disable (mask) interrupts on the local CPU, which prevents the caller from being preempted while it holds the spinlock (which could otherwise lead to a livelock). If a task acquires an ISR-callable spinlock, task preemption is also suspended on the local CPU while that task holds the spinlock. This allows the task to execute the critical section that the spinlock is protecting. Interrupts and tasks on other CPUs are not affected. The routines used for ISR-callable spinlocks are listed in Table 15-1.

For VxWorks UP, ISR-callable spinlocks are implemented with the same behavior as the interrupt locking routines **intLock( )** and **intUnlock( )**.

Table 15-1 **ISR-Callable Spinlock Routines**

Routine	Description
spinLockIsrInit( )	Initializes an ISR-callable spinlock.
spinLockIsrTake( )	Acquires an ISR-callable spinlock.
spinLockIsrGive( )	Relinquishes ownership of an ISR-callable spinlock.

15.6.2 Task-Only Spinlocks

Spinlocks that are used to address contention between tasks alone (and not ISRs) are called *task-only spinlocks*. These spinlocks disable task preemption on the local CPU while the caller holds the lock (which could otherwise lead to a livelock situation). This prevents the caller from being preempted by other tasks and allows it to execute the critical section that the lock is protecting. Interrupts are not disabled and task preemption on other CPUs is not affected. The routines used for task-only spinlocks are listed in Table 15-2.

For VxWorks UP, task-only spinlocks are implemented with the same behavior as the task locking routines **taskLock( )** and **taskUnlock( )**.

Table 15-2 **Task-Only Spinlock Routines**

Routine	Description
spinLockTaskInit( )	Initializes a task-only spinlock.
spinLockTaskTake( )	Acquires a task-only spinlock.
spinLockTaskGive( )	Relinquishes ownership of a task-only spinlock.

15.6.3 Caveats With Regard to Spinlock Use

Because of the concurrency of execution inherent in SMP systems, spinlocks should be used with care. The following prescriptions should be adhered to avoid problems with spinlocks:

- A spinlock should only be held for a short and deterministic period of time.
- A task or ISR must not take more than one spinlock at a time. Livelocks may result when an entity that already holds a spinlock takes another spinlock. Livelocks are similar to the deadlocks that occur with semaphore use. With spinlocks, however, the entity does not pend or block; it spins without ever acquiring the spinlock and the CPU appears to be hung. Because interrupts are masked or task preemption is disabled, the state cannot be remedied.
- A task or ISR must not take a spinlock that it already holds. That is, recursive takes of a spinlock should not be made. A livelock will occur.
- In order to prevent a task or ISR from entering a kernel critical region while it already holds a spinlock—and cause the system to enter a livelock state—a task or ISR must not call specified routines while it holds a spinlock. The VxWorks SMP kernel itself uses spinlocks to protect its critical regions. For information about these routines, see *15.6.4 Routines Restricted by Spinlock Use*, p.683.

A debug version of spinlocks can be used to catch these problems. For information, see *Debug Versions of Spinlocks*, p.681.

15.6.4 Routines Restricted by Spinlock Use

Certain routines should not be called while the calling entity (task or ISR) holds a spinlock. This restriction serves to prevent a task or ISR from entering a kernel critical region while it already holds a spinlock—and cause the system to enter a livelock state (for more information, see *15.6.3 Caveats With Regard to Spinlock Use*, p.683). The routine restriction also apply to **intCpuLock( )** (for more information about this routine see *15.7.1 CPU-Specific Mutual Exclusion for Interrupts*, **p.685**). This restriction applies because the kernel requires interrupts to be enabled to implement its multi-CPU scheduling algorithm.

It is outside the scope of this document to list all the VxWorks spinlock restricted routines. However, generally speaking these are routines related to the creation, destruction and manipulation of kernel objects (semaphores, tasks, message queues, and so on) as well as any routine that can cause a scheduling event.

While the restriction imposed by spinlock use may seem to be a hindrance, it really should not be. Spinlocks are meant for very fast synchronization between processors. Holding a spinlock and attempting to perform notable amounts of work, including calling into the kernel, results in poor performance on an SMP system, because either task preemption or interrupts, or both, are disabled when a CPU owns a spinlock.

Table 16 identifies some of the routines restricted by spinlock and CPU lock use.

Table 16 **Routines Restricted by Spinlock and CPU Lock Use**

Library	Routines
taskLib	**taskExit( )**, **taskDelete( )**, **taskDeleteForce( )**, **taskInitExcStk( )**, **taskUnsafe( )**, **exit( )**, **taskSuspend( )**, **taskResume( )**, **taskPrioritySet( )**, **taskDelay( )**, **taskStackAllot( )**, **taskRestart( )**, **taskCpuLock( )**, **taskCpuUnlock( )**, **taskCreateLibInit( )**, **taskCreate( )**, **taskActivate( )**, **taskCpuAffinitySet( )**, **taskCpuAffinityGet( )**, **taskSpawn( )**, **taskInit( )**
msgQLib	**msgQCreate( )**, **msgQDelete( )**, **msgQSend( )**, **msgQReceive( )**, **msgQInitialize( )**, **msgQNumMsgs( )**,
msgQEvLib	**msgQEvStart( )**, **msgQEvStop( )**
semLib	**semTake( )**, **semGive( )**, **semFlush( )**, **semDelete( )**
semBLib	**semBInitialize( )**, **semBCreate( )**
semCLib	**semCInitialize( )**, **semCCreate( )**
semMLib	**semMInitialize( )**, **semMGiveForce( )**, **semMCreate( )**
semEvLib	**semEvStart( )**, **semEvStop( )**

Table 16 **Routines Restricted by Spinlock and CPU Lock Use**

Library	Routines
wdLib	**wdCreate( )**, **wdDelete( )**, **wdinitialise( )**, **wdStart( )**, **wdCancel( )**
kernelLib	**kernelTimeSlice( )**, **kernelCpuEnable( )**
intLib	**intDisconnect( )**
intArchLib	**intConnect( )**, **intHandlerCreate( )**, **intVecTableWriteProtect( )**
eventLib	**eventSend( )**, **eventReceive( )**

15.7 CPU-Specific Mutual Exclusion

VxWorks SMP provides facilities for CPU-specific mutual exclusion, that is for mutual exclusion operations whose scope is entirely restricted to the CPU on which the call is made (the *local* CPU). These facilities are designed to facilitate porting uniprocessor code to an SMP system.

15.7.1 CPU-Specific Mutual Exclusion for Interrupts

CPU-specific mutual exclusion for interrupts allows for disabling (masking) interrupts on the CPU on which the calling task or ISR is running. For example if task A, running on CPU 0, performs a local CPU interrupt lock operation, no interrupts can be processed by CPU 0 until the lock is released by task A.

Execution of interrupts on other CPUs in the SMP system is not affected. In order to be an effective means of mutual exclusion, therefore, all tasks and ISRs that should participate in the mutual exclusion scenario should have CPU affinity set for the local CPU (for information, see *15.10.1 Task CPU Affinity*, p.691).

Note that some routines should not be used if the calling task or ISR has locked interrupts on the local CPU—similar to the case of holding spinlocks (see *15.6.3 Caveats With Regard to Spinlock Use*, p.683). The restricted routines are described in *15.6.4 Routines Restricted by Spinlock Use*, p.683.

The routines listed in Table 15-3 are used for disabling and enabling interrupts on the local CPU.

Note that in a uniprocessor system they default to the behavior of **intLock( )** and **intUnlock( )**.

Table 15-3 **CPU-Specific Mutual Exclusion Routines for Interrupts**

Routine	Description
intCpuLock( )	Disables interrupts on the CPU on which the calling task or ISR is running.
intCpuUnlock( )	Enables interrupts on the CPU on which the calling task or ISR is running.

For more information about these routines, see the **intLib** entry in the VxWorks API references.

15.7.2 CPU-Specific Mutual Exclusion for Tasks

CPU-specific mutual exclusion for tasks allows for suspending task preemption on the CPU on which the calling task is running. That is, it provides for local CPU task locking, and effectively prevents any other task from running on the local CPU. For example, task A running on CPU 0 can perform a local CPU task lock operation so that no other task can run on CPU 0 until it releases the lock or makes a blocking call.

The calling task is also prevented from migrating to another CPU until the lock is released.

Execution on other CPUs in the SMP system is not affected. In order to be an effective means of mutual exclusion, therefore, all tasks that should participate in the mutual exclusion scenario should have CPU affinity set for the local CPU (for information, see *15.10.1 Task CPU Affinity*, p.691).

The routines listed in Table 15-4 are used for suspending and resuming task preemption on the local CPU.

Note that in a uniprocessor system they default to the behavior of **taskLock( )** and **taskUnlock( )**.

Table 15-4 **CPU-Specific Mutual Exclusion Routines for Tasks**

Routine	Description
taskCpuLock( )	Disables task preemption for the CPU on which the calling task is running.
taskCpuUnlock( )	Enables context task switching on the CPU on which the calling task is running.

For more information about these routines, see the **taskLib** entry in the VxWorks API references.

15.8 Memory Barriers

In modern multiprocessing architectures, individual CPUs can reorder both read and write operations in order to improve overall system efficiency. From the perspective of a single CPU in the system, this reordering is completely transparent because the CPU ensures that any read operation gets data that was previously written, regardless of the order in which the read and write operations are actually committed to system memory. The reordering occurs in the background, and is never visible to the programmer.

In an multiprocessor system, an individual CPU can execute a series of write operations to memory, and these write operations can be queued between the CPU and system memory. The CPU is allowed to commit these queued operations to system memory in any order, regardless of the order in which the operations arrive in the CPU's *write queue*. Similarly, a CPU is free to issue more than one read operation in parallel, whether as the result of speculative execution, or because the program has requested more than one independent read operation.

Because of this reordering, two tasks that share data should never assume that the order in which an operation is performed on one CPU will be the same as the order in which the operations are written to or read from memory. A classic example of this ordering problem involves two CPUs, in which one CPU prepares an item of work to be performed, and then sets a boolean flag to announce the availability of the work unit to a second CPU that is waiting for it. The code in this case would look like the following:

```
/* CPU 0 - announce the availability of work */

pWork = &work_item;  /* store pointer to work item to be performed */
workAvailable = 1;

/* CPU 1 - wait for work to be performed */

while (!workAvailable);
doWork (pWork);    /* error - pWork might not be visible to this CPU yet */
```

It is very likely that the **pWork** pointer used by CPU 1 will contain incorrect data because CPU 0 reorders its write operations to system memory, which causes CPU 1 to observe the change to the **workAvailable** variable before the value of the **pWork** variable has been updated. In a case like this, the likely result is a system crash due to de-referencing an invalid pointer.

To solve the memory ordering problem, VxWorks provides a set of memory barrier operations. The sole purpose of memory barrier operations is to provide a way to guarantee the ordering of operations between cooperating CPUs. Memory barriers fall into three general classes:

- read memory barrier
- write memory barrier
- full (read/write) memory barrier

NOTE: VxWorks SMP provides a set of synchronization primitives to protect access to shared resources. These primitives include semaphores, message queues, and spinlocks. These primitives include full memory barrier functionality. Additional memory barrier operations are not required with these facilities to protect shared resources.

NOTE: Memory barriers are not available to RTP (user-mode) applications.

Read Memory Barrier

The **VX_MEM_BARRIER_R( )** macro provides a read memory barrier. **VX_MEM_BARRIER_R( )** enforces ordering between all of the read operations that have occurred prior to the barrier, and all of the read operations that occur after the barrier. Without this barrier, a CPU is free to reorder its pending read operations in any way that does not affect program correctness from a uniprocessor perspective. For example, a CPU is free to reorder the following independent reads:

```
a = *pAvalue;    /* read may occur _after_ read of *pBvalue */
b = *pBvalue;    /* read may occur _before read of *pAValue */
```

By inserting a memory barrier between the read operations, you can guarantee that the reads occur in the appropriate order:

```
a = *pAvalue;                    /* will occur before read of *pBvalue */
VX_MEM_BARRIER_R();
b = *pBvalue;                    /* will occur after read of *pAvalue */
```

While **VX_MEM_BARRIER_R( )** can ensure that the read operations occur in the correct order, this guarantee is not helpful unless the writer of the shared data also ensures that the writes of the shared data also occur in the correct order. For this reason, the **VX_MEM_BARRIER_R( )** and **VX_MEM_BARRIER_W( )** macros should always be used together.

Write Memory Barrier

The **VX_MEM_BARRIER_W( )** macro provides a write memory barrier. **VX_MEM_BARRIER_W( )** enforces the ordering between all of the write operations that have occurred prior to the barrier, and all of the write operations that occur after the barrier. The following code fragment is taken from a preceding example, but modified to take advantage of a memory barrier:

```
pWork = &work_item;
VX_MEM_BARRIER_W();
workAvailable = 1;
```

Inserting a barrier between the update of ***pWork** and the update of **workAvailable** ensures that the value of **workAvailable** in system memory is updated after the value of **pWork** has been updated in system memory. Note that **VX_MEM_BARRIER_W( )** does not actually force the writing of these values to system memory. Instead, it merely enforces the order in which these values are written. Note that **VX_MEM_BARRIER_W( )** should always be used with **VX_MEM_BARRIER_R( )** or **VX_MEM_BARRIER_RW( )**.

Read/Write Memory Barrier

The **VX_MEM_BARRIER_RW( )** macro provides a read/write memory barrier. This is also referred to as a *full fence* memory barrier. **VX_MEM_BARRIER_RW( )** combines the effects of both the **VX_MEM_BARRIER_R( )** and **VX_MEM_BARRIER_W( )** operations into a single primitive. On some systems, **VX_MEM_BARRIER_RW( )** may be substantially more expensive than either **VX_MEM_BARRIER_R( )** or **VX_MEM_BARRIER_W( )**. Unless both read and write ordering is required, Wind River does not recommend the use of **VX_MEM_BARRIER_RW( )**.

15.9 Atomic Memory Operations

Atomic operations make use of CPU support for atomically accessing memory. They combine a set of (architecture-specific) operations into what is effectively a single operation that cannot be interrupted by any other operation on the memory location in question. Atomic operations thereby provide mutual exclusion for a simple set of operations (such as incrementing and decrementing variables).

Atomic operations can be useful as a simpler alternative to spinlocks, such as for updating a single data element. For example, you can update the *next* pointer in a singly-linked list from NULL to non-NULL (without interrupts locked) using an atomic operation, which allows you to create lock-less algorithms.

Because the atomic operations are performed on a memory location supplied by the caller, users must ensure the location has memory access attributes and an alignment that allows atomic memory access—otherwise an access exception will occur. Restrictions, if any, are specific to the CPU architecture. For more information, see *15.15.9 Memory-Access Attributes*, p.714.

The **vxAtomicLib** library provides a number of routines that perform atomic operations. They are described in Table 15-5.

Atomic operation routines are available in user space (for RTP applications) as well as in the kernel.

Table 15-5 **Atomic Memory Operation Routines**

Routine	Description
vxAtomicAdd( )	Adds two values atomically.
vxAtomicSub( )	Subtracts one value from another atomically.
vxAtomicInc( )	Increments a value atomically.
vxAtomicDec( )	Decrements a value atomically.
vxAtomicOr( )	Performs a bitwise OR operation on two values atomically.
vxAtomicXor( )	Performs a bitwise XOR operation on two values atomically.
vxAtomicAnd( )	Performs a bitwise AND operation on two values atomically.

Table 15-5 **Atomic Memory Operation Routines**

Routine	Description
vxAtomicNand( )	Performs a bitwise NAND operation on two values atomically.
vxAtomicSet( )	Sets one value to another atomically.
vxAtomicClear( )	Clears a value atomically.
vxCas( )	Performs an atomic compare-and-swap of two values atomically.

15.10 CPU Affinity

VxWorks SMP provides facilities for *CPU affinity*; that is, for assigning specific interrupts or tasks to specific CPUs.

15.10.1 Task CPU Affinity

VxWorks SMP provides the ability to assign tasks to a specific CPU, after which the scheduler ensures the tasks are only executed on that CPU. This assignment is referred to as *task CPU affinity*.

While the default SMP operation in which any task can run on any CPU often provides the best overall load balancing, there are cases in which assigning a specific set of tasks to a specific CPU can be useful. For example, if a CPU is dedicated to signal processing and does no other work, the cache remains filled with the code and data required for that activity. This saves the cost of moving to another CPU—which is incurred even within single piece of silicon, as the L1 cache is bound to a single CPU, and the L1 must be refilled with new text and data if the task migrates to a different CPU.

Another example is a case in which profiling an application reveals that some of its tasks are frequently contending for the same spinlock, and a fair amount of execution time is wasted waiting for a spinlock to become available. Overall performance could be improved by setting task CPU affinity such that all tasks involved in spinlock contention run on the same CPU. This would free up more time other CPUs for other tasks.

Task CPU affinity can be set in the following manner:

- A task can set its own CPU affinity or the CPU affinity of another task by calling **taskCpuAffinitySet( )**.
- A newly created task inherits the CPU affinity (if any) of the parent task. A task created or initialized by any of the following routines inherits the CPU affinity of the calling task: **taskSpawn( )**, **taskCreate( )**, **taskInit( )**, **taskOpen( )**, and **taskInitExcStk( )**.

  The creating task's CPU affinity is not inherited, however, when the task that is created is an RTPs initial task. For example, if a task invokes **rtpSpawn( )**, the initialization task of the resulting RTP does not inherit the CPU affinity of the caller.

The **taskLib** library provides routines for managing task CPU affinity. They are described in Table 15-6.

Table 15-6 **Task CPU Affinity Routines**

Routine	Description
taskCpuAffinitySet( )	Sets the CPU affinity for a task.
taskCpuAffinityGet( )	Returns the CPU affinity for a task.

The routine **taskCpuAffinitySet( )** takes a CPU set variable (of type **cpuset_t**) to identify the CPU to which the task should be assigned. Similarly, the **taskCpuAffinityGet( )** routine takes a pointer to a **cpuset_t** variable for the purpose of recording the CPU affinity for a given task.

In both cases the **CPUSET_ZERO( )** macro must be used to clear the **cpuset_t** variable before the call is made. For **taskCpuAffinitySet( )**, the **CPUSET_SET( )** macro must be used after **CPUSET_ZERO( )** and before the routine itself is called.

To remove task CPU affinity, use the **CPUSET_ZERO( )** macro to clear the **cpuset_t** variable, and then make the **taskCpuAffinitySet( )** call again.

For more information about using these routines and macros see *Task CPU Affinity Examples*, p.693 and *CPU Set Variables and Macros*, p.696

RTP Tasks and CPU Affinity

By default, real-time process (RTP) tasks inherit the CPU affinity setting of the task that created the RTP. If the parent task has no specific CPU affinity (that is, it can execute on any available CPU and may migrate from one CPU to the other during its lifetime), then the RTP's tasks have no specific CPU affinity either. If the parent task has its affinity set to a given CPU, then by default, the RTP tasks inherit this affinity and execute only on the same CPU as the parent task.

The **RTP_CPU_AFFINITY_NONE** option for **rtpSpawn( )** can be used to create an RTP in which tasks have no CPU affinity, despite the fact that the RTP's parent task may have itself have had one.

Task CPU Affinity Examples

The following sample code illustrates the sequence to set the affinity of a newly created task to CPU 1.

```
STATUS affinitySetExample (void)
   {
   cpuset_t affinity;
   int tid;

   /* Create the task but only activate it after setting its affinity */
   tid = taskCreate ("myCpu1Task", 100, 0, 5000, printf,
                     (int) "myCpu1Task executed on CPU 1 !", 0, 0, 0,
                     0, 0, 0, 0, 0, 0);

   if (tid == NULL)
       return (ERROR);

   /* Clear the affinity CPU set and set index for CPU 1 */
   CPUSET_ZERO (affinity);
   CPUSET_SET  (affinity, 1);

   if (taskCpuAffinitySet (tid, affinity) == ERROR)
       {
       /* Either CPUs are not enabled or we are in UP mode */
       taskDelete (tid);
       return (ERROR);
       }

   /* Now let the task run on CPU 1 */
   taskActivate (tid);

   return (OK);
   }
```

The next example shows how a task can remove its affinity to a CPU:

```
{
cpuset_t affinity;

CPUSET_ZERO (affinity);

taskCpuAffinitySet (0, affinity);
}
```

15.10.2 Interrupt CPU Affinity

SMP hardware requires programmable interrupt controller devices (for more information see *15.2.3 VxWorks SMP Hardware*, p.669). VxWorks SMP makes use of this hardware to allow assignment interrupts to a specific CPU. By default, interrupts are routed to the bootstrap CPU (CPU 0).

Interrupt CPU affinity can be useful for load balancing (for example, if there is too much total interrupt traffic for one CPU to handle). It can also be used as an aid in migrating code from VxWorks UP (for more information, see *Interrupt Locking: intLock( ) and intUnlock( )*, p.710).

Runtime assignment of interrupts to a specific CPU occurs at boot time, when the system reads interrupt configuration information from the BSP. The interrupt controller then receives a command directing that a given interrupt be routed to a specific CPU. For information about the mechanism involved, see the *VxWorks Device Driver Developer's Guide*.

15.11 CPU Information and Management

VxWorks SMP provides several routines and macros for getting and manipulating information about CPUs, as well as for managing their operation.

CPU Information and Management Routines

The **kernelLib** and **vxCpuLib** libraries provide routines for getting information about, and for managing, CPUs. They are described in Table 15-7 and Table 15-8.

Table 15-7 **kernelLib CPU Routines**

Routine	Description
kernelIsCpuIdle( )	Returns **TRUE** if the specified CPU is idle.
kernelIsSystemIdle( )	Returns **TRUE** if all enabled CPUs are idle
kernelCpuEnable( )	Enables the CPU with the specified index.

The **kernelCpuEnable( )** routine allows you to enable a specific CPU. Once a CPU is enabled, it starts dispatching tasks as directed by the scheduler. All CPUs are enabled by default, but the **ENABLE_ALL_CPUS** component parameter can be used to boot VxWorks SMP with just CPU 0 enabled (for more information see *ENABLE_ALL_CPUS*, p.675). Then, **kernelCpuEnable( )** can be used to selectively enable individual CPUs.

Table 15-8 **vxCpuLib CPU Routines**

Routine	Description
vxCpuConfiguredGet( )	Returns the number of CPUs that have been statically configured into the system with the BSP.
vxCpuEnabledGet( )	Returns the set of CPUs that are enabled in the system.
vxCpuIndexGet( )	Returns the *mapped* CPU number as provided by the operating system.
vxCpuIdGet( )	Returns the true CPU number, as defined by the architecture variant (and not the operating system).

The **vxCpuConfiguredGet( )** routine returns the *number* of CPUs configured into a VxWorks SMP system with the BSP, which may not be the same as the number of CPUs available in the hardware platform.

The **vxCpuEnabledGet( )** routine returns the *set* of CPUs enabled (running) on the system, of which the total may be different from the number of CPUs configured into the system with the BSP, or available in the hardware platform. As noted above, all CPUs are enabled by default, but the **ENABLE_ALL_CPUS** configuration parameter can be set so that VxWorks SMP boots with just CPU 0 enabled (for more information see *ENABLE_ALL_CPUS*, p.675). Then the **kernelCpuEnable( )** routine can be used to selectively enable individual CPUs.

The return type used by **vxCpuEnabledGet( )** to identify a CPU set is **cpuset_t**. Note that the **CPUSET_ZERO( )** macro must be used to clear the **cpuset_t** variable before the **vxCpuEnabledGet( )** call is made. For information about the **cpuset_t** variable type, and the macros used to manipulate **cpuset_t** variables, see *CPU Set Variables and Macros*, p.696.

The routine **vxCpuIndexGet( )** returns the index of the calling task's CPU. The index of a CPU is a number between 0 and *N*-1, where *N* is the number of CPUs configured into the SMP system. Note, however, that tasks can migrate from one CPU to another (by default), so there is no guarantee that the index returned by **vxCpuIndexGet( )** identifies the CPU on which the task is running after the return—unless the calling task is prevented from migrating to another CPU with **taskCpuLock( )** or **intCpuLock( )**.

CPU Set Variables and Macros

VxWorks SMP provides a *CPU set* variable type, and CPU set macros for manipulating variables defined by that type. The variable and macros must be used in conjunction with various routines—such as **taskCpuAffinitySet( )**—for getting information about CPUs and managing their use.

The **cpuset_t** variable type is used for identifying the CPUs that have been configured into a VxWorks SMP system with the target BSP, which may be a subset of the CPUs in the hardware platform.

Each bit in a **cpuset_t** variable corresponds to a specific CPU, or *CPU index*, with the first bit representing CPU 0 (the bootstrap CPU). The first bit corresponds to index 0, the second to 1, the third to 2, and so on (regardless of the physical location of the CPUs in the hardware).

As an example, for an eight CPU hardware system, for which the BSP configures four CPUs for VxWorks SMP, the **CPUSET_ZERO( )** macro would clear all the bits in a **cpuset_t** variable, and then a call to **vxCpuIndexGet( )** would set the first four.

CPU set macros *must* be used to set and unset CPU indices (change the bits of **cpuset_t** variables). These macros are described in *Table 15-9CPU Set Macros*, p.697. In order to use these macros, include the **cpuset.h** header file.

⚠ **CAUTION:** Do not manipulate **cpuset_t** type variables directly. Use CPU set macros.

Table 15-9 **CPU Set Macros**

Macro	Description
CPUSET_SET( )	Sets a specific CPU index (one specific **cpuset_t** variable bit).
CPUSET_SETALL( )	Sets CPU indices (all **cpuset_t** variable bits) for all CPUs that are configured into the system.
CPUSET_SETALL_BUT_SELF( )	Sets indices (**cpuset_t** variable bits) for all CPUs that are configured into the system, except for the index of the CPU on which the macro is called.
CPUSET_CLR( )	Clears a specific CPU index (one specific **cpuset_t** variable bit).
CPUSET_ZERO( )	Clears all CPU indices (all **cpuset_t** variable bits).
CPUSET_ISSET( )	Returns **TRUE** if the specified index (**cpuset_t** variable bit) is set in the **cpuset_t** variable.
CPUSET_ISZERO( )	Returns **TRUE** if the no indices (**cpuset_t** variable bits) are set in the **cpuset_t** variable.
CPUSET_ATOMICSET( )	Atomically sets a specific CPU index (one specific **cpuset_t** variable bit).
CPUSET_ATOMICCLR( )	Atomically clears a specific CPU index (one specific **cpuset_t** variable bit).

For an example of how CPU set macros are used, see *15.10.1 Task CPU Affinity*, p.691. For more information about the macros, see the entry for **cpuset** in the VxWorks API references.

15.12 Debugging SMP Code

Debugging and system monitoring tools such as System Viewer, and the Workbench debugger provide support for debugging SMP code.

For information about debugging problems related to spinlock use, see *Debug Versions of Spinlocks*, p.681.

For information about target tool behavior with VxWorks SMP, see *12. Target Tools*.

As appropriate, debugging facilities provide CPU-specific information. For example, **checkStack( )** displays the interrupt stack of all CPUs. The output from the kernel shell looks like the following:

```
-> checkStack
  NAME        ENTRY        TID       SIZE   CUR  HIGH  MARGIN
------------ ------------ ---------- ----- ----- ----- ------
tJobTask     0x60056a50   0x603ce228 24576   104   832  23744
(Exception Stack)                    12072     0     0  12072
tExcTask     0x60055c60   0x601a3b30 24576   184   520  24056
(Exception Stack)                    12216     0     0  12216
tLogTask     logTask      0x603d7c38 24576   228   564  24012
(Exception Stack)                    12216     0     0  12216
tNbioLog     0x60057720   0x603d2010 24576   188   524  24052
(Exception Stack)                    12072     0     0  12072
tShell0      shellTask    0x6051ce90 81920  6780  9344  72576
(Exception Stack)                    12072     0     0  12072
tWdbTask     wdbTask      0x603c7b88 24576   196   532  24044
(Exception Stack)                    12072     0     0  12072
tAioIoTask1  aioIoTask    0x60461740 40960   116   452  40508
(Exception Stack)                    12216     0     0  12216
tAioIoTask0  aioIoTask    0x60461a60 40960   116   452  40508
(Exception Stack)                    12216     0     0  12216
tNet0        ipcomNetTask 0x60473af0 24576   156   652  23924
(Exception Stack)                    12216     0     0  12216
ipcom_syslog 0x60108350   0x60485558 16384   352  1092  15292
(Exception Stack)                    12216     0     0  12216
ipnetd       0x6010c630   0x604a58c8 24576   128  2012  22564
(Exception Stack)                    12216     0     0  12216
tAioWait     aioWaitTask  0x60451020 40960   244   880  40080
(Exception Stack)                    12216     0     0  12216
tIdleTask0   idleTaskEntr 0x60389a20 12288   396   900  11388
(Exception Stack)                    12200     0   772  11428
tIdleTask1   idleTaskEntr 0x603a2000 12288   396   900  11388
(Exception Stack)                    12200     0  1156  11044
INTERRUPT CPU 0                      57344     0  1796  55548
INTERRUPT CPU 1                      57344     0  1272  56072
value = 2 = 0x2
->
```

And **spy( )** reports the number of ticks spent in kernel, interrupt, idle, and task code for each CPU. The output looks like the following:

```
-> spy
value = 1634761696 = 0x61707be0
->
    NAME         ENTRY       TID      PRI   total % (ticks)  delta % (ticks)
------------ ------------ ----------  ---   ---------------  ---------------
tJobTask     0x60056ae0   0x603d2010    0     0% (       0)    0% (       0)
tExcTask     0x60055cf0   0x601a3b30    0     0% (       0)    0% (       0)
tLogTask     logTask      0x603d7c38    0     0% (       0)    0% (       0)
tNbioLog     0x600577b0   0x603db110    0     0% (       0)    0% (       0)
tShell0      shellTask    0x6051cec8    1     0% (       0)    0% (       0)
tWdbTask     wdbTask      0x603c7840    3     0% (       0)    0% (       0)
tSpyTask     spyComTask   0x61707be0    5     0% (       0)    0% (       0)
tAioIoTask1  aioIoTask    0x60443888   50     0% (       0)    0% (       0)
tAioIoTask0  aioIoTask    0x60443c88   50     0% (       0)    0% (       0)
tNet0        ipcomNetTask 0x60485020   50     0% (       0)    0% (       0)
ipcom_syslog 0x60109060   0x60485c78   50     0% (       0)    0% (       0)
ipnetd       0x6010d340   0x603bf0c8   50     0% (       0)    0% (       0)
tAioWait     aioWaitTask  0x60443590   51     0% (       0)    0% (       0)
tIdleTask0   idleTaskEntr 0x60389a38  287   100% (    1000)  100% (     500)
tIdleTask1   idleTaskEntr 0x603a2000  287   100% (    1000)  100% (     500)
KERNEL                                        0% (       0)    0% (       0)
INTERRUPT                                     0% (       0)    0% (       0)
TOTAL                                       200% (    1000)  200% (     500)

CPU      KERNEL        INTERRUPT          IDLE           TASK       TOTAL
--- --------------- --------------- --------------- --------------- -----
0      0% (       0)   0% (       0) 100% (    1000)   0% (       0)  100%
1      0% (       0)   0% (       0) 100% (    1000)   0% (       0)  100%
```

Note that while **timexLib** can avoid precision errors by auto-calibrating itself and doing several calls of the functions being monitored, it suffers from the lack of scheduling management during the calls. The tasks can move between CPUs while the measurements take place. Depending on how often this occurs, this is likely to have an impact the precision of the measurement.

15.13 Optimizing SMP Performance

The purpose of SMP systems is to increase performance. Simply making code *SMP-ready* does not necessarily exploit the performance potential available with multiple CPUs. Additional work is necessary to make code *SMP-optimized*. (For definitions of these terms, see *15.2.1 Terminology*, p.667.)

The performance improvement of an SMP algorithm is almost completely dependent on the amount of parallelism in the algorithm and the quality of the multi-threaded implementation. Some algorithms are highly parallel in nature, and take good advantage of multiple CPUs. A good example is an image compressor, which can compress separate bands of data independently on separate threads of execution. Since contention is low, the utilization of the CPU can be very high, resulting in good SMP performance.

With a poorly-designed SMP algorithm, on the other hand, the cost of synchronizing two threads of execution can completely negate the benefits of using more than one CPU. Similarly, if you have a data-dependent algorithm and both CPUs are competing for the same data, the system bus can be swamped with competing bus transactions, slowing the throughput to the point that the CPUs are data-starved, and overall throughput suffers.

In the worst case, SMP will actually slow down an algorithm, resulting in worse performance than on a uniprocessor system. In the best case, taking advantage of the fact that there is twice as much L1 cache in a dual-processor system, might allow algorithms to run twice as fast, simply because the working set of the algorithm fits better in the twice-as-large cache. These types of algorithms are, however, fairly rare.

Threading

Threading involves turning a single-thread application into a multi-threaded one by replicating tasks. A typical example involves a worker task that fetches work from a queue that is being filled by another task or an ISR. Assuming the bottleneck in the application is the worker task, performance can be increased by replicating the worker task. Threading is not a new concept—it was introduced when multitasking operating systems were created. However, in a uniprocessor system threading only increases throughput of the application when its threads are subject to wait periods. That is, when one thread waits for a resource, the scheduler can dispatch another thread, and so on. In cases where the bottleneck is the CPU itself, threading cannot help performance. For example, compute intensive applications typically do not benefit from threading on a uniprocessor system. However, this is not the case on a SMP system. Because of the presence of additional CPUs, threading increases performance particularly when the bottleneck is the CPU.

Using Spinlocks

Spinlocks affect interrupt and task-preemption latency and should therefore be used sparingly and only for very short periods of time. For more information, see *15.6 Spinlocks for Mutual Exclusion and Synchronization*, p.679.

Using Floating Point and Other Coprocessors

For reasons of efficiency, coprocessor task creation options (such as **VP_FP_TASK**) should be used carefully—that is, only with tasks that will actually make use of coprocessors. When a task is created with a coprocessor option, the state of the coprocessor is saved and restored with each context switch, which is unnecessary overhead if the coprocessor is never used. VxWorks SMP does not support the VxWorks UP option of *lazy state save-and-restore*, because tasks are not guaranteed to resume execution on the same CPU on which they were last scheduled to run.

Using vmBaseLib

The **vmBaseLib** library is the VxWorks MMU management library that allows kernel applications and drivers to manage the MMU. An important task of an SMP operating system is to ensure the coherency of the translation look aside buffers (TLBs) of the MMU contained in each CPU. Some CPUs, like the MPC8641D, have hardware that ensures TLBs are always coherent. Other CPUs, such as the BCM1480 and Intel Dual Core Xeon LV, do not have this capability. In these cases the operating system is responsible for propagating MMU events that affect TLB coherency to all CPUs in the system.

While not all events require propagation—it is generally limited to events that modify an existing page mapping such as with **vmStateSet( )—the** propagation that must be performed has a negative impact on some VxWorks SMP **vmBaseLib** routines. To reduce the negative impact on your system's performance, minimize the number of calls to **vmStateSet( )**, and so on. For example, if a region with special settings is needed from time to time during system operation, it is better to set it up once during startup, and then reuse it as needed, rather than creating and destroying a region for each use.

Interrupt and Task CPU Affinity

For some applications and systems, assigning specific interrupts or specific tasks to designated CPUs can provide performance advantages. For more information, see *15.10.2 Interrupt CPU Affinity*, p.694 and *15.10.1 Task CPU Affinity*, p.691.

15.14 Sample Programs

Sample programs are provided for VxWorks SMP, which demonstrates VxWorks SMP features and performance.

The following applications illustrate I/O intensive activity and system-call intensive activity:

philDemo
Dijkstra's *Dining Philosophers Problem*.

smpLockDemo
Demonstrates VxWorks SMP’s synchronization mechanism for sharing data across multiple CPUs.

The next set of demos illustrate computation-intense activity:

primesDemo
Prime number computation.

rawPerf
Calculation of *pi* using floating-point arithmetic.

The demo applications can be linked to a VxWorks SMP image by configuring VxWorks with the **INCLUDE_SMP_DEMO** component. The source code is provided in *installDir***/vxworks-6.***x***/target/src/demo/smp**, if VxWorks source code has been installed. For more information about the demos, refer to their entries in the *VxWorks Kernel API Reference*.

15.15 Migrating Code to VxWorks SMP

The key issue to consider in migrating code to an SMP system is that symmetric multiprocessing allows for concurrent execution of tasks with other tasks, of tasks with ISRs, and of ISRs with other ISRs.

Concurrent execution in SMP requires the use of different facilities for mutual exclusion and synchronization, it precludes some routines that are available for a uniprocessor system, and it makes the practice of relying on implicit synchronization of tasks dangerous (if not disastrous). For example, the concurrent execution of several of the highest priority tasks in a system can uncover unprotected race conditions that were hidden in a uniprocessor system.

In addition, multiple CPUs introduce complexities with regard to objects that are *global* in a uniprocessor system, but must be *CPU-specific* in an SMP system.

Migrating code from VxWorks UP to VxWorks SMP necessarily involves several steps between uniprocessor code and hardware to SMP code and hardware.

The migration process also involves using different multitasking facilities, different BSP support, and so on. Some parts of migration activity involve replacing a uniprocessor technology with an SMP one—such as replacing **taskLock( )** with **spinLockTaskTake( )**—while others involve changing the use of features that have different behaviors in the VxWorks UP and VxWorks SMP (for example, some **vmBaseLib** routines).

This section provides an overview of the migration process, a summary of the operating system facilities that need to be taken into account in migration, and more detailed information about individual migration issues. It does not provide a completely self-contained discussion of migration to SMP. It is necessarily a supplement to the preceding material in this chapter, which provides information about the core features of VxWorks SMP. Incorporation of these features naturally forms the basis for migrating code from VxWorks UP to VxWorks SMP. The material following the discussion of general issues—*15.15.1 Code Migration Path*, p.703 and *15.15.2 Overview of Migration Issues* , p.705—therefore covers some of the less tidy aspects of migration.

15.15.1 Code Migration Path

This section describes the migration model and the recommended path for migrating applications from VxWorks UP to VxWorks SMP.

Wind River recommends that you approach migrating code designed for an earlier version of VxWorks UP to the current version of VxWorks SMP with the following steps:

Step 1: Migrate Uniprocessor Code from Previous to Current Version of VxWorks

Migrating the VxWorks UP 6.*previous* code to a VxWorks UP 6.*current* system involves eliminating or replacing any routines and coding practices that are unsupported or incompatible with VxWorks SMP. For an overview of what must be replaced or changed, see *15.15.2 Overview of Migration Issues* , p.705. Note that this step might include migrating from uniprocessor hardware to a single processor on SMP hardware; that is, the hardware used for the end-point of the first and second migration steps shown in Figure 15-4 may therefore be the same.

Step 2: Migrate Code from Current VxWorks UP to VxWorks SMP

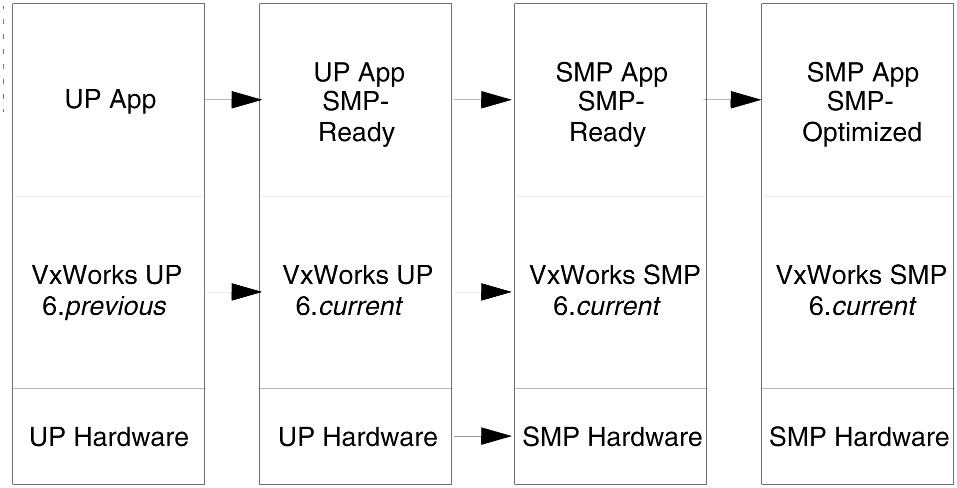
Migrating the code from the VxWorks UP 6.*current* system to a VxWorks SMP 6.*current* system involves correcting any concurrent execution bugs (such as contention issues and deadlocks) that appear in an SMP environment. This step includes migrating to multiprocessor use of SMP hardware.

Step 3: Optimize Code for SMP Performance

Optimizing the code on VxWorks SMP allows it to make the fullest use of symmetric multiprocessing. For more information about this topic, see *15.13 Optimizing SMP Performance*, p.699.

For definitions SMP-ready and SMP-optimized see *15.2.1 Terminology*, p.667. Figure 15-4 illustrates the recommended application migration path.

Figure 15-4 **VxWorks SMP Migration**



15.15.2 Overview of Migration Issues

Table 15-10 provides an overview of the uniprocessor features or programming practices that are incompatible with (or unsupported by) VxWorks SMP, the appropriate SMP alternatives, and references to these topics. All code designed for VxWorks UP should be examined carefully with regard to these issues as part of the migration process.

Table 15-10 **VxWorks SMP Migration Issues**

Incompatible Uniprocessor Features and Practices	**SMP Features and Practices**	**Reference**
Coding practices relying on implicit synchronization.	Use of explicit synchronization facilities, such as semaphores and spinlocks.	*15.15.4 Implicit Synchronization of Tasks*, p.706
Various **cacheLib** routines.	Revise use of routines.	*cacheLib Restrictions*, p.708
Various **vmBaseLib** routines.	Restrict use of routines.	*vmBaseLib Restrictions*, p.709
taskLock( ), **intLock( )**	**spinLockLib**, **taskCpuLock( )**, **intCpuLock( )**, atomic operators	*15.15.5 Synchronization and Mutual Exclusion Facilities*, p.707 and *Task Locking: taskLock( ) and taskUnlock( )*, p.711
taskRtpLock( ), **taskRtpUnlock( )**	semaphores, atomic operators	*15.15.5 Synchronization and Mutual Exclusion Facilities*, p.707 and *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p.711
task variables, **taskVarLib** routines	**__thread** storage class	*Task Variable Management: taskVarLib* , p.712
tlsLib routines	**__thread** storage class	*Task Local Storage: tlsLib*, p.712
Accessing global variables that are *CPU-specific* variables or inaccessible in SMP.	Replace with *CPU-specific* variable routines and practices.	*15.15.8 SMP CPU-Specific Variables and Uniprocessor Global Variables*, p.712

Table 15-10 **VxWorks SMP Migration Issues**

Incompatible Uniprocessor Features and Practices	SMP Features and Practices	Reference
Memory-access attributes unsuited for SMP memory coherency.	Review calls that directly manipulate coherency protocols and caching modes. Adhere to restrictions as documented in architecture supplements.	*15.15.9 Memory-Access Attributes*, p.714
Drivers that are not VxBus compliant.	VxBus-compliant drivers.	*15.15.10 Drivers and BSPs*, p.715
Uniprocessor BSP.	SMP BSP.	*15.15.10 Drivers and BSPs*, p.715
Uniprocessor boot loader.	Boot loader that supports VxWorks SMP.	

Also note that the custom scheduler framework is not supported for VxWorks SMP.

15.15.3 RTP Applications and SMP

As in VxWorks UP systems, RTP (user mode) applications have a more limited set of mutual exclusion and synchronization mechanisms available to them than kernel code or kernel applications. In VxWorks SMP, they can make use of semaphores and atomic operations, but not spinlocks, memory barriers, or CPU-specific mutual exclusion mechanisms. In addition, the **semExchange( )** routine provides for an atomic give and exchange of semaphores.

15.15.4 Implicit Synchronization of Tasks

VxWorks is a multitasking operating system, and VxWorks and its applications are re-entrant; therefore migrating to a system in which tasks run concurrently is not normally a problem as long as tasks are explicitly synchronized. For example, Task A giving a semaphore to Task B to allow it to run is a form of explicit synchronization. On the other hand, implicit synchronization techniques—such as those that rely on task priority—cannot be relied on in VxWorks SMP. For example, if high priority Task A spawns low priority Task B, expecting that Task

B will not run until Task A releases the CPU is an invalid assumption on an SMP system.

Implicit synchronization based on task priority is not easy to detect. Careful review of all code that causes a task to become ready to run would be a useful approach. For example, review code that uses the following types of routines:

- **Routines That Create Tasks**

  The **taskSpawn( )**, **rtpSpawn( )**, and other routines create a new task. In an SMP system, the new task may have already started running on another CPU by the time the procedure call returns—regardless of the relative priority of the new task compared to the creating task. If the creator and the created tasks interact using semaphores, message queues or other objects, these objects *must* be created or initialized before creating the new task.

- **Routines that Unpend a Waiting Task**

  The **semGive( )**, **msgQSend( )**, **eventSend( )** and other routines can unpend a waiting task, which may begin running before the procedure call returns—even though the waiting task has a lower priority than the calling task.

For example, in a VxWorks UP system, a task can protect a critical section of code by using **intLock( )**, which prevents all interrupts from being processed and thereby prevents ISRs from entering the critical section. The ISR does not use explicit mutual exclusion when accessing the critical section because the task cannot be running when the ISR is running on a uniprocessor system. (This is likely to be a common occurrence in drivers where a portion of a driver runs in an ISR and queues up work for a task to perform.) The assumption that tasks do not run when ISRs do is simply not true in an SMP system. Therefore ISRs must use explicit mutual exclusion in cases such as the one described above. The preferred mechanism is the ISR-callable spinlock as described in *15.6 Spinlocks for Mutual Exclusion and Synchronization*, p.679.

15.15.5 Synchronization and Mutual Exclusion Facilities

Because of concurrent execution on an SMP system, there are necessarily differences in facilities available for explicit synchronization and mutual exclusion for VxWorks UP and VxWorks SMP.

Semaphores are appropriate for both environments, but uniprocessor interrupt and task locking mechanisms are not appropriate and not available for SMP—spinlocks and other mechanisms should be used instead.

In VxWorks SMP it is, for example, possible for a task to be running at the same time that an ISR is executing. This is not possible in VxWorks UP, and therefore requires changes to the way mutual exclusion between a task and an ISR is done. A common synchronization method between an ISR and a task in VxWorks is the binary semaphore. This mechanism works equally well in VxWorks SMP, and therefore code that uses binary semaphores in this manner need not be modified for VxWorks SMP—provided the ISR is running with interrupts enabled when it calls **semGive( )**. This is also true of other messaging and synchronization routines, such as message queues and VxWorks events. Note, however, that when an ISR wakes up a task (by giving a binary semaphore, sending a VxWorks event, sending a message to a message queue, etc.), the awakened task may start running *immediately* on another CPU.

For more information about uniprocessor synchronization mechanisms and the SMP alternatives, see *15.15.7 Unsupported Uniprocessor Routines and SMP Alternatives* , p.709.

15.15.6 VxWorks SMP Variants of Uniprocessor Routines

While the routines provided in VxWorks UP and VxWorks SMP are largely the same, there are a few that have different behaviors in the VxWorks SMP due to the requirements of multiprocessor systems, and their use has restrictions.

cacheLib Restrictions

The VxWorks UP cache routines are designed around a uniprocessor system. Enabling and disabling the caches, invalidating, flushing or clearing elements of the cache all have a CPU-specific nature, as they refer to the local CPU's cache. In an SMP system, this CPU-specific nature is less meaningful. The systems that are supported by VxWorks SMP all provide hardware cache coherency, both between the individual CPUs in the SMP system and between the memory subsystem and the device address space. Given these characteristics, the cache restrictions and behavior modifications described below apply to VxWorks SMP.

cacheEnable( ) and cacheDisable( )

The only way for the hardware cache coherency to be effective is to have the caches turned on at all times. VxWorks SMP therefore turns on the caches of each CPU as it is enabled, and never allows them to be disabled. Calling **cacheEnable( )** in VxWorks SMP always returns **OK**. Calling **cacheDisable( )** in VxWorks SMP always returns **ERROR**, with errno set to **S_cacheLib_FUNCTION_UNSUPPORTED**.

cacheClear( ), cacheFlush( ), and cacheInvalidate( )

Because of the hardware cache coherency of SMP-capable platforms, these routines are not necessary. If these functions are called in VxWorks SMP, they perform no function (are NOOPs) and simply return **OK**.

cacheLock( ) and cacheUnlock( )

These routines are not supported in VxWorks SMP. If they are called, they return **ERROR** and set errno to **S_cacheLib_FUNCTION_UNSUPPORTED**

vmBaseLib Restrictions

VxWorks SMP does not provide APIs for changing memory page attributes. On an SMP system it is essential that the RAM regions that are shared between the CPUs never be allowed to get out of coherency with one another. If a single page in system RAM were to have its attributes changed so that it no longer correctly participates in the hardware coherency protocol, any operating system use of that page (for spinlocks, shared data structures, and so on) would be at risk of unpredictable behavior. This unpredictable behavior might even occur long after the offending change in the page attributes has occurred. This type of problem would be extremely difficult to debug, because of the underlying assumption that the hardware coherency in SMP simply works.

vmBaseStateSet( ) and vmStateSet( )

These routines are called to modify the attributes of a single page of virtual memory. In an SMP system, the caching attributes of a page cannot be modified. Attempting to do so causes these routines to return **ERROR** with errno set to **S_vmLib_BAD_STATE_PARAM**.

15.15.7 Unsupported Uniprocessor Routines and SMP Alternatives

Some of the routines available in VxWorks UP are not supported in VxWorks SMP because their functionality is at odds with truely concurrent execution of tasks and ISRs, or because they would degrade performance to an unacceptible extent. SMP alternatives provide comparable functionality that is designed for symmetric multiprocessing.

Interrupt Locking: intLock( ) and intUnlock( )

In VxWorks UP, the **intLock( )** routine is used by a task or ISR to prevent VxWorks from processing interrupts. The typical use of this routine is to guarantee mutually exclusive access to a critical section of code between tasks, between tasks and ISRs, or between ISRs (as with *nested ISRs*—when ISR can be preempted by an ISR of higher priority).

This mechanism would be inappropriate for a multiprocessor system, and VxWorks SMP provides the following alternatives for interrupt locking:

- If interrupt locking is used to make a simple pseudo-atomic operation on a piece of memory, atomic operations may be a suitable alternative.
- If interrupt locking is used as a mutual exclusion mechanism between tasks only, semaphores or task-only spinlocks are a suitable replacements. Spinlock acquisition and release operations are faster than semaphore operations, so they would be suitable to protect a short critical section that needs to be fast. Semaphores are suitable for longer critical sections.
- If interrupt locking is used as a mutual exclusion mechanism between tasks and ISRs, or between ISRs, ISR-callable spinlocks are a suitable replacement.
- If interrupt locking is used as a mutual exclusion mechanism between tasks only, **taskCpuLock( )** can be used instead as long as all tasks taking part in the mutual exclusion scenario have the same CPU affinity. This alternative should not be used in custom extensions to the operating system other than as a temporary measure when migrating code the from VxWorks UP to VxWorks SMP.
- If interrupt locking is used as a mutual exclusion mechanism between tasks, between tasks and ISRs, or between ISRs, then **intCpuLock( )** can be used as long as all tasks and ISRs taking part in the mutual exclusion scenario have the same CPU affinity. This alternative should not to be used in custom extensions to the operating system other than as a temporary measure when migrating the code from VxWorks UP to VxWorks SMP.

Note that for VxWorks SMP, ISR-callable spinlocks are implemented with the same behavior as the interrupt locking routines **intLock( )** and **intUnlock( )**.

For information about SMP mutual exclusion facilities, see *15.6 Spinlocks for Mutual Exclusion and Synchronization*, p.679, *15.7 CPU-Specific Mutual Exclusion*, p.685, and *15.9 Atomic Memory Operations*, p.690.

Task Locking: taskLock( ) and taskUnlock( )

In VxWorks UP, task locking routines are used by a task to prevent the scheduling of any other task in the system, until it calls the corresponding unlock routine. The typical use of these routines is to guarantee mutually exclusive access to a critical section of code.

With VxWorks UP, the kernel routine **taskLock( )** is used to lock out all other tasks in the system by suspending task preemption (also see *Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )*, p.711). This mechanism would be inappropriate for a multiprocessor system, and VxWorks SMP provides the following alternatives:

- Semaphores.
- Atomic operations.
- task-only spinlocks. Spinlock acquisition and release operations are faster than semaphore operations (the other alternative in this case) so they would be suitable to protect a short critical section that needs to be fast.
- The **taskCpuLock( )** routines for situations where all tasks taking part in the task-locking scenario have the same CPU affinity. This alternative should not be used in custom extensions to the operating system other than as a temporary measure when migrating the code from VxWorks UP to VxWorks SMP.

Note that for VxWorks UP, task-only spinlocks are implemented with the same behavior as the task locking routines **taskLock( )** and **taskUnlock( )**.

For information about SMP mutual exclusion facilities, see *15.6 Spinlocks for Mutual Exclusion and Synchronization*, p.679, *15.7 CPU-Specific Mutual Exclusion*, p.685, and *15.9 Atomic Memory Operations*, p.690.

Task Locking in RTPs: taskRtpLock( ) and taskRtpUnlock( )

The **taskRtpLock( )** routine is used in RTP applications to prevent scheduling of any other tasks in the process of the calling task. As with **taskLock( ),** the **taskRtpLock( )** routine is not appropriate for an SMP system (also see *Task Locking: taskLock( ) and taskUnlock( )*, p.711).

The **taskRtpLock( )** routine is provided with VxWorks SMP, but it generates a fatal error when called, and the process terminates. Semaphores or atomic operators should be used instead.

Task Variable Management: taskVarLib

The VxWorks UP task variable facility provided by **taskVarLib** is not compatible with an SMP environment, as more than one task using the same task variable location could be executing concurrently. Therefore task variables and the **taskVarAdd( )** and **taskVarDelete( )** routines are not available in VxWorks SMP. The **__thread** storage class should be used instead. For more information, see *4.7.3 Task-Specific Variables*, p.190.

Task Local Storage: tlsLib

The VxWorks UP task local storage routines provided by **tlsLib** for user-mode (RTP) applications are not compatible with an SMP environment, as more than one task using the same task variable location could be executing concurrently. The **tlsLib** routines are as follows:

- **tlsKeyCreate( )**
- **tlsValueGet( )**
- **tlsValueSet( )**
- **tlsValueOfTaskGet( )**
- **tlsValueOfTaskSet( )**

The **__thread** storage class should be used instead. For more information, see *4.7.3 Task-Specific Variables*, p.190.

15.15.8 SMP CPU-Specific Variables and Uniprocessor Global Variables

Some objects that are global in a uniprocessor system (such as errno) are CPU-specific entities in VxWorks SMP, and others are inaccessible or non-existent in VxWorks SMP.

⚠ **CAUTION:** Wind River recommends that you do not manipulate any CPU-specific or global variables directly. Using the appropriate API is recommended to prevent unpredictable behavior and to ensure compatibility with future versions of VxWorks.

SMP Per-CPU Variables

The SMP CPU-specific variables that can be accessed indirectly with appropriate routine are as follows:

- **errno**
- **taskIdCurrent**
- **intCnt**
- **isrIdCurrent**

errno

From a programming perspective errno behaves like a global variable that contains the error value of the currently running task or ISR. VxWorks SMP has mechanism that allows it to manage errno as a CPU-specific variable in a transparent manner.

Wind River recommends that you use **errnoLib** routines to work with errno. However, you may access the errno variable directly from C and C++ code that includes **errno.h**. Do not access the errno variable directly from assembly code.

⚠ **CAUTION:** Do not access errno directly from assembly code. Do not access errno directly from C or C++ code that does not include **errno.h**.

taskIdCurrent

The uniprocessor **taskIdCurrent** global variable (declared in **taskLib.h**) does not exist in VxWorks, because of concurrent execution on multiple CPUs. Any uniprocessor code that reads **taskIdCurrent** should make calls to **taskIdSelf( )** instead.

intCnt

In an SMP system, specific interrupts are dedicated to a specific CPU. The **intCnt** variable is used to track the number of nested interrupts that exist on a specific CPU. Code that references this variable should be changed to use the **intCount( )** routine instead.

isrIdCurrent

The **isrIdCurrent** variable is used to identify the ISR executing on the specific CPU. This global is only available if the **INCLUDE_ISR_OBJECTS** component is included in VxWorks.Code that accesses **isrIdCurrent** must be changed to use the **isrIdSelf( )** routine instead.

Uniprocessor-Only Global Variables

The VxWorks UP variables that do not exist in VxWorks SMP—or that must not be accessed by user code in any way—are as follows:

- **vxIntStackBase**
- **vxIntStackEnd**
- **kernelIsIdle**
- **windPwrOffCpuState**

vxIntStackBase

The **vxIntStackBase** variable identifies base of the interrupt stack used for processing interrupts. For VxWorks SMP, each CPU has a **vxIntStackBase** to process interrupts since interrupts may be processed by multiple CPUs simultaneously. There is no routine for accessing this variable and it must not be accessed by user code.

vxIntStackEnd

The **vxIntStackEnd** variable identifies the end of the interrupt stack for each CPU. There is no routine for accessing this variable and it must not be accessed by user code.

kernelIsIdle

In VxWorks UP the **kernelIsIdle** variable indicates whether or not the system is idle. This variable does not exist in VxWorks SMP. *CPU Information and Management Routines*, p.694 describes routines that can be used instead.

windPwrOffCpuState

The **windPwrOffCpuState** variable identifies power management state on the specific CPU. There is no routine for accessing this variable and it must not be accessed by user code.

15.15.9 Memory-Access Attributes

In an SMP system memory coherency is required to ensure that each CPU sees the same memory contents. Depending on the CPU architecture, some memory access attributes may not be suitable for a system where memory coherency is required. For information in this regard, see the *VxWorks Architecture Supplement*.

15.15.10 **Drivers and BSPs**

Both drivers and BSPs developed for VxWorks SMP must adhere to the programming practices described throughout this chapter. Drivers must also conform to the VxBus driver model. BSPs, in addition to providing support for VxBus, must provide facilities different from the VxWorks UP for reboot handling, CPU enumeration, interrupt routing and assignment, and so on. For more information, see *VxWorks Device Driver Developer's Guide* and *VxWorks BSP Developer's Guide*.

16
Shared-Memory Objects: VxMP

16.1 Introduction

VxMP is a VxWorks component that provides shared-memory objects dedicated to high-speed synchronization and communication between tasks running in separate instances of VxWorks.

Shared-memory objects are a class of system objects that can be accessed by tasks running on different processors. The object's data structures 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 the following types 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 going through an unnecessary network stack.
- The shared memory can reside either in dual-ported RAM or on a separate memory board.

VxMP consists of the following facilities: a name database (**smNameLib**), task synchronization and resource tracking with semaphores (**semSmLib**), messaging with message queues (**msgQSmLib**) to build a custom protocol, and a shared-memory allocator (**smMemLib**).

NOTE: VxMP can only be used in kernel space. It cannot be used in user space (real-time processes).

NOTE: VxMP is currently not supported for SMP systems.

16.2 Using Shared-Memory Objects

VxMP provides creation APIs specifically for shared memory objects—shared semaphores, message queues, and memory partitions. As with standard kernel objects, an object ID is returned when an object is created, and the ID is subsequently used to identify that object. For any given shared-memory object, the ID is global to the VxMP system. That is, the same ID is used to access that object, regardless of the CPU from which it is accessed.

After a shared-memory object is created, kernel tasks can operate on it with the same routines used for the corresponding local object. Shared semaphores, shared message queues, and shared-memory partitions have the same syntax and interface as their local counterparts, which allows routines such as **semGive( )**, **semTake( )**, **msgQSend( )**, **msgQReceive( )**, **memPartAlloc( )**, and **memPartFree( )** to operate on both types of objects.

Kernel tasks running on different CPUs can provide and obtain the object ID of shared memory objects in a variety of ways, including shared message queues and data structures in shared memory. The most convenient method, however, is by using the VxMP name database to publish and access the object ID.

After the shared-memory facilities are initialized at run-time, all processors are treated alike. Kernel 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]

There are few restrictions on shared-memory object use (they cannot, for example, be used at interrupt level), and they are easily portable between uniprocessor and multiprocessor systems, which can be advantageous in the development process.

Note that throughout the remainder of this chapter, system objects under discussion refer to shared objects unless otherwise indicated.

16.2.1 Multiprocessor-Uniprocessor Portability

VxMP provides a transparent interface that makes it easy to execute code using shared-memory objects on both a multiprocessor system and a uniprocessor system.

Only the object creation routines are different for shared-memory objects. After creation, the same routines as used for operations on local objects can be used for the shared memory objects. This allows an application to run in either a uniprocessor or a multiprocessor environment with only minor changes to system configuration, initialization, and object creation.

Using shared-memory objects on a uniprocessor system 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.

16.2.2 Multiprocessing and Byte Ordering

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.

[1] Do not confuse this type of priority with the CPU priorities associated with VMEbus access.

Note, however, on systems where the processors have different byte ordering, you must call the **ntohl** and **htonl** macros to byte-swap the *application* shared data that is passed with message queues and so on. VxMP handles the endianness of all system data structures and IDs internally. Names are byte-streams (strings) so they are not subject to endianness issues. The IDs returned by the name database are converted internally and the ID obtained by the user has the correct endianness.

16.2.3 Restrictions on Shared Memory Object Use

Shared-memory objects are only available to kernel tasks. 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 on which 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; see *16.5.7 Dual-Port or External Memory*, p.748. If applications are creating more than the specified maximum number of objects, it is possible to run out of memory. For more information in this regard, see *16.7 Troubleshooting*, p.752.

16.2.4 Publishing Objects With the Name Database

The VxMP *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 *publish* 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 making an object's ID known to other nodes (such as with message queues, by being written to a shared memory block at a pre-determined offset, and so on), the name database is the most convenient method for doing so—it is simpler and it allows any node access to the information at will.

Typically, the kernel task that creates an object also publishes 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 string **myObj** in the name database.

Table 16-1 **Name Service Routines**

Routine	Description
smNameAdd( )	Adds a name to the name database.
smNameRemove( )	Removes a name from the name database.
smNameFind( )	Finds a shared symbol by name.
smNameFindByValue( )	Finds a shared symbol by value.
smNameShow( )	Displays the name database to the standard output device.[a]

[a] Automatically included if **INCLUDE_SM_OBJ** is selected.

This same technique can be used to publish a shared-memory address. For example, task **t1** on CPU 0 allocates a portion 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 the string **mySharedMem**.

Tasks on different processors can use an agreed-upon name to get a newly created object's value. See Table 16-1 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. An ID can simply be retrieved on a per-processor basis, if it is stored in a global variable (for example). However, it is generally a good practice to retrieve IDs on a per-task basis.

The name database service routines automatically convert to or from network-byte order; do not call **htonl( )** or **ntohl( )** C macros explicitly for values provided by the name database. These C macros must, however be used on *application* shared data that is passed between processors with different byte orders using message queues and so on. For more information in this regard, see *16.2.2 Multiprocessing and Byte Ordering*, p.719.

The object types listed in Table 16-2 are defined in **smNameLib.h**.

Table 16-2 **Shared-Memory Object Types**

Constant	Hex Value	Description
T_SM_SEM_B	0	Shared binary semaphore.
T_SM_SEM_C	1	Shared counting semaphore.
T_SM_MSG_Q	2	Shared message queue.
T_SM_PART_ID	3	Shared memory partition ID.
T_SM_BLOCK	4	Shared memory block.

The following example shows the name database as displayed by **smNameShow( )**, which is automatically included if VxWorks is configured with the **INCLUDE_SM_OBJ** component. The parameter to **smNameShow( )** specifies the level of information displayed; in this case, 1 indicates that all information is shown. For additional information, see the **smNameShow( )** API reference.

```
-> 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
```

16.2.5 Shared Semaphores

Like local semaphores, *shared semaphores* provide synchronization by means of atomic updates of semaphore state information. See *4. Multitasking* and the API reference for **semLib** for a complete discussion of semaphores. Shared semaphores can be given and taken by tasks executing in the kernel 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 announces its ID to other nodes. This can be done, for example, 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 publishes the semaphore's ID by adding it to the database and assigning the name **myMutexSem**. Task **t2** looks up the string **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.

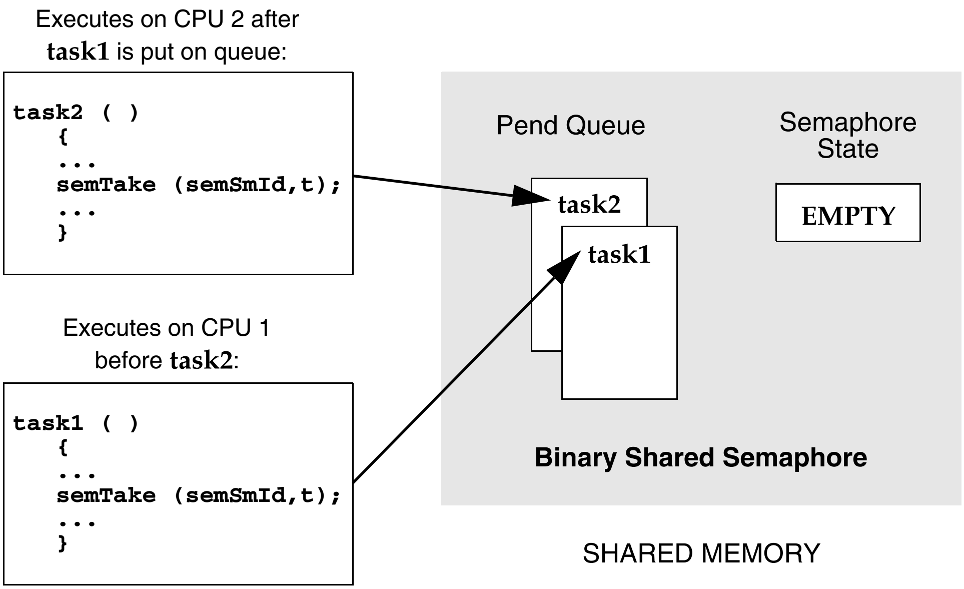
For example, there are two tasks, **t1** on CPU 0 and **t2** on CPU 1. Both **t1** and **t2** are monitoring robotic arms. The robotic arm that is controlled by **t1** is passing a physical object to the robotic arm controlled by **t2**. Task **t2** moves the arm into position but must then wait until **t1** indicates that it is ready for **t2** to take the object. Task **t1** creates the shared semaphore and publishes the semaphore's ID by adding it to the database and assigning the name **objReadySem**. Task **t2** looks up the string **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.

Table 16-3 **Shared Semaphore Create Routines**

Create Routine	Description
semBSmCreate( )	Creates a shared binary semaphore.
semCSmCreate( )	Creates a shared counting semaphore.

There are two types of shared semaphores, binary and counting. Shared semaphores have their own create routines and return a **SEM_ID**. Table 16-3 lists the create routines. All other semaphore routines, except **semDelete( )**, operate transparently on the created shared semaphore.

Figure 16-1 **Shared Semaphore Queues**



```
task2 ( )
    {
    ...
    semTake (semSmId,t);
    ...
    }
```

```
task1 ( )
    {
    ...
    semTake (semSmId,t);
    ...
    }
```

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. If it is not, an error is generated, and **errno** is set to **S_msgQLib_INVALID_QUEUE_TYPE**.

  Figure 16-1 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 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. If they are, an error is generated, and **errno** is set to **S_intLib_NOT_ISR_CALLABLE**.

- Shared semaphores cannot be deleted. Attempts to delete a shared semaphore return **ERROR** and set **errno** to **S_smObjLib_NO_OBJECT_DESTROY**.

Use **semInfo( )** to get the shared task control block of tasks pended on a shared semaphore. Use **semShow( )** to display the status of the shared semaphore and a list of pended tasks. (VxWorks must be configured with the **INCLUDE_SEM_SHOW** component.)

The following example displays detailed information on the shared semaphore, using the variable **mySmSemaphoreId**, which holds the ID. The level of information is specified 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 16-1 **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);
    }
```

16.2.6 Shared Message Queues

Shared message queues are FIFO queues used by kernel 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 kernel tasks running on different CPUs. See *4. Multitasking* and the API reference for **msgQLib** for a complete discussion of message queues.

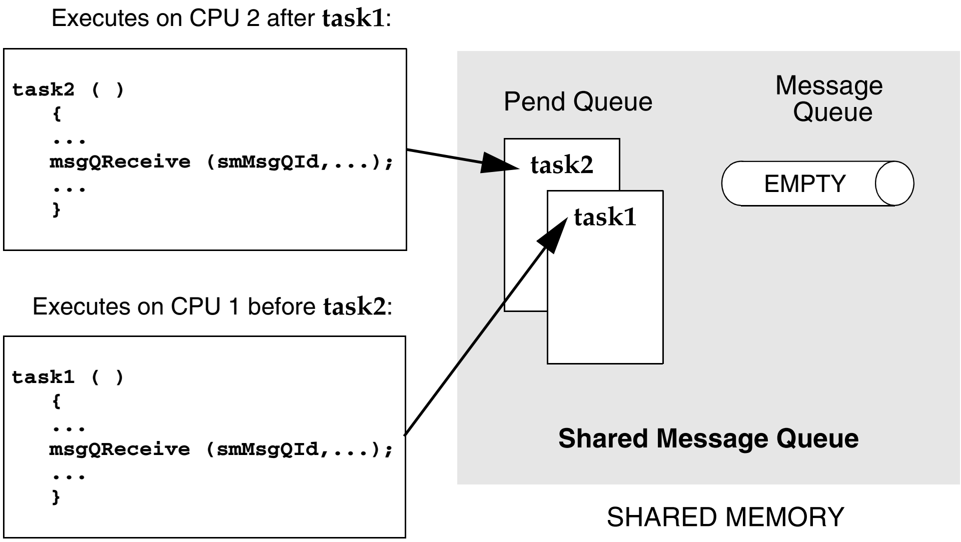
To use a shared message queue, a task creates the message queue and publishes its ID. A task that wants to send or receive a message with this message queue first gets the message queue's ID. It then uses this ID to access the message queue.

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 publishes 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 string **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 publishes 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 kernel 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.

Figure 16-2 **Shared Message Queues**



```
task2 ( )
   {
   ...
   msgQReceive (smMsgQId,...);
   ...
   }
```

```
task1 ( )
   {
   ...
   msgQReceive (smMsgQId,...);
   ...
   }
```

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. If it is not, an error is generated and **errno** is set to **S_msgQLib_INVALID_QUEUE_TYPE**.

  Figure 16-2 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.) If they are, an error is generated, and **errno** is set to **S_intLib_NOT_ISR_CALLABLE**.

- 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.

To display the status of the shared message queue as well as a list of tasks pended on the queue, call **msgQShow( )** (VxWorks must be configured with the **INCLUDE_MSG_Q_SHOW** component.) 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 16-2 **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 C 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);
    }
```

16.2.7 Shared-Memory Allocator

The *shared-memory allocator* allows kernel tasks on different CPUs to allocate and release variable size portions 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 announces its ID to other nodes. The most convenient way 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 published 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 published 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 string **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 the string **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 16-4 lists routines for manipulating the shared-memory system partition.

Table 16-4 **Shared-Memory System Partition Routines**

Call	Description
smMemMalloc( )	Allocates a block of shared system memory.
smMemCalloc( )	Allocates a block of shared system memory for an array.
smMemRealloc( )	Resizes a block of shared system memory.
smMemFree( )	Frees a block of shared system memory.
smMemShow( )	Displays usage statistics of the shared-memory system partition on the standard output device. This routine is automatically included if VxWorks is configured with the **INCLUDE_SM_OBJ** component.
smMemOptionsSet( )	Sets the debugging options for the shared-memory system partition.
smMemAddToPool( )	Adds memory to the shared-memory system pool.
smMemFindMax( )	Finds 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 announced 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 API reference for **smObjLib**.

Example 16-3 **Shared-Memory System Partition**

The following code example uses memory from the shared-memory system partition to share data between kernel 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 16-4 **User-Created Partition**

This example is similar to Example 16-3, 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 API references 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.

16.3 System Requirements

The system requirements for VxMP include the following:

- shared memory visible to all CPUs
- test-and-set (TAS) operations across CPUs
- inter-CPU notification facilities
- a maximum of 20 CPUs (10 is the default)

Shared Memory Visibility

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 local address for all CPUs. For information about the shared memory anchor, see *16.5.4 Shared-Memory Anchor*, p.745.

Test-and-Set Cycle

All CPUs in the system must support indivisible test-and-set operations across the (VME) bus. The indivisible test-and-set operation is used by the spinlock mechanism to gain exclusive access to internal shared data structures. Because all the boards must support hardware test-and-set, the parameter **SM_TAS_TYPE** must be set to **SM_TAS_HARD**.

⚠ **CAUTION:** Boards that make use of VxMP must support hardware test-and-set (indivisible read-modify-write cycle). PowerPC is an exception; see the *VxWorks Architecture Supplement*.

Inter-CPU Notification

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 it generally results in a significant performance penalty. For information about configuration for different CPU-notification facilities, see *16.5.3 Mailbox Interrupts and Bus Interrupts*, p.745

Maximum Number of CPUs

The maximum number of CPUs that can use shared-memory objects is 20 (CPUs numbered 0 through 19; the default is 10). For information about configuring VxMP for the number of CPUs in a system, see *16.5.1 Maximum Number of CPUs*, p.744.

16.4 Performance Considerations

The performance of a multi-processor system using shared memory objects can be affected by the operation of VxMP spinlocks and well as by the use of both VxMP and shared-memory network facilities in the same system.

Spinlock Operation

The performance of a system using shared memory objects can be affected by the operation of spinlocks, which are used internally for cross-processor synchronization. The spinlocks may need to be tuned for proper operation, and interrupt latency is increased while spinlocks are held. However, spinlocks are used only for very short periods of time to protect critical regions (in a manner similar to the use of interrupt locking on uniprocessor systems.

Internal shared-memory object data structures are protected against concurrent access by a spinlock mechanism. The spinlock mechanism operates as a loop in which an attempt is made to gain exclusive access to a resource (in this case an internal data structure). An indivisible hardware test-and-set operation is used for this mutual exclusion. If the first attempt to take the lock fails, multiple attempts are made.

For the duration of the spinlock, interrupts are disabled to avoid the possibility of a task being preempted while holding the spinlock. 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.

For more information about and spinlocks and performance tuning, see *16.7 Troubleshooting*, p.752.

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. Both facilities make use of the same shared memory region, the same shared-memory anchor, the same interrupt, some data structures, their traffic goes over the same bus, and so on. While their software does not effectively interact, using them together can result in reduced performance.

If the two facilities are used together, the shared-memory anchor must be configured in the same way for each. The shared-memory anchor is a location accessible to all CPUs on the system. It 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. For information about using the shared memory anchor with shared-memory objects, see *16.5.4 Shared-Memory Anchor*, p.745.

For information about shared-memory network, see the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

[2] Also known as the *backplane network*.

16.5 Configuring VxWorks for Shared Memory Objects

To include VxMP shared-memory objects in VxWorks, configure the operating system with the **INCLUDE_SM_OBJ** component.

The configuration of VxMP itself involves the following elements:

- number of CPUs—all nodes
- cache coherency—on a node-by-node basis
- mailbox and bus interrupts—on a node-by-node basis
- shared memory anchor—all nodes
- shared memory region—master node only
- number of shared memory objects—master node only
- dual-port or external memory—master node only

Detailed descriptions of each of these elements, as well as examples of system configurations, are provided in the following sections.

16.5.1 Maximum Number of CPUs

The maximum number of CPUs that can use shared-memory objects is 20 (CPUs numbered 0 through 19). This limitation is imposed by the VMEbus hardware itself. The practical maximum is usually a smaller number that depends on the CPU, bus bandwidth, and application. The number is set with the **SM_CPUS_MAX** configuration parameter of the **INCLUDE_SM_COMMON** component. By default it is set to 10. Note that if the number is set higher than the number of boards that are actually going to be used, it will waste memory.

16.5.2 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 runtime error message, make sure that the **INCLUDE_CACHE_ENABLE** component is *not* included in the VxWorks configuration:

```
usrSmObjInit - cache coherent buffer not available. Giving up.
```

Additional configuration is 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 **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.) For more information, see *6.9.1 Configuring Virtual Memory Management*, p.344.

⚠ **CAUTION:** For the MC68K in general, if the MMU is off, data caching must be turned off globally; see the API reference for **cacheLib**.

16.5.3 Mailbox Interrupts and Bus Interrupts

Two types of interrupts are supported for inter-CPU communication: mailbox interrupts and bus interrupts. The interrupt type is specified with the **INCLUDE_SM_COMMON** component parameter **SM_INT_TYPE**. 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 it 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 with the **SM_INT_TYPE**, **SM_INT_ARG1**, **SM_INT_ARG2**, and **SM_INT_ARG3** parameters.

16.5.4 Shared-Memory Anchor

One of the most important aspects of configuring shared-memory objects is defining the address of the shared-memory anchor.

Determining the address is specific to the target architecture. For example, PowerPC leaves space at 0x4100 in its memory map to store the anchor on the master. Most of the other architectures do the same thing at 0x600. This is defined in VxWorks architecture-dependent header files, which ensure that the kernel reserves memory at that location.

On PowerPC, assume for a dual-ported system that the VMEbus is seen at address 0x10000 on the master, and 0x20000 on a slave. The master can either see the anchor at 0x4100 or 0x14100, and the slave sees it at 0x24100.

The shared-memory anchor is a location accessible to all CPUs on the system. 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.

If the default value for the shared-memory anchor address is modified, the anchor must be on a 256-byte boundary.

The address of the shared memory anchor can be defined statically with the **SM_ANCHOR_ADRS** configuration parameter, or dynamically with the **sm** boot loader parameter (**sm=***anchorAddress*). If the boot loader parameter is defined, it takes precedence over the static configuration.

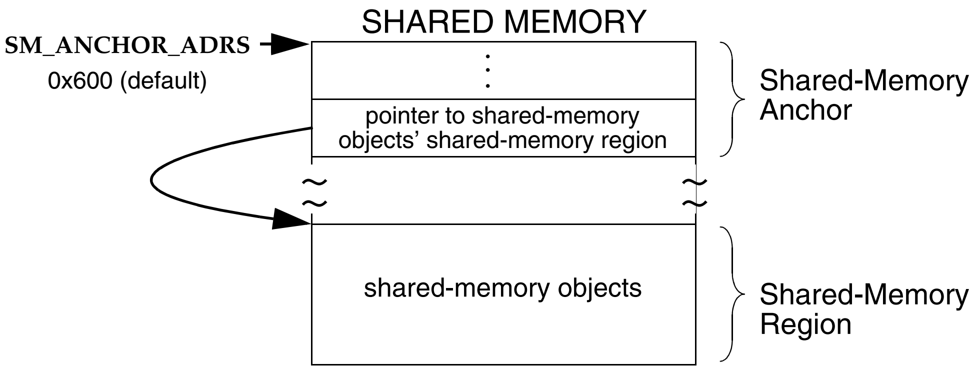
NOTE: The shared memory anchor is used by both VxMP and the shared-memory network driver (if both are included in the system). For information about using VxMP and the shared memory network driver at the same time, see *Shared-Memory Objects and Shared-Memory Network Driver*, p.743.

16.5.5 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 CPU, but it can also be located on a separate memory card. The shared-memory region address is defined as an offset from the shared-memory anchor address (**SM_ANCHOR_ADRS**), as illustrated in Figure 16-3. The default is 0x600 for all architectures but PowerPC, which is 0x4100.

Figure 16-3 **Shared-Memory Layout**



16.5.6 Numbers of Shared Memory Objects

The configuration parameters of the **INCLUDE_SM_OBJ** component are used to set the maximum number of different types of objects. These constants are described in Table 16-5.

Table 16-5 **Configuration Constants for Shared-Memory Objects**

Configuration Parameter	Default Value	Description
SM_OBJ_MAX_TASK	40	Maximum number of tasks using shared-memory objects.
SM_OBJ_MAX_SEM	60	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.

If the size of the objects created exceeds the shared-memory region, an error message is displayed on CPU 0 during initialization.

At runtime, VxMP sets aside memory for the configured number of objects, and then uses what is left over for the shared memory system partition.

16.5.7 Dual-Port or External Memory

The key distinction between configuration for dual-port and for external memory is as follows:

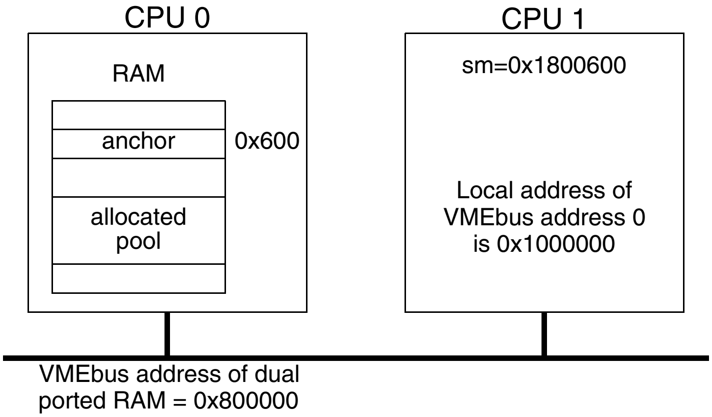
- For dual-port memory, the **SM_OFF_BOARD** parameter is set to **FALSE** for the master CPU and to **TRUE** for all slave CPUs.
- For external memory, the **SM_OFF_BOARD** parameter is set to **TRUE** for all CPUs (master and slave CPUs).

The following sections describe configurations for each type of memory use.

Configuration for Dual-Port Memory

The configuration illustrated in Figure 16-4 uses the shared memory in the master CPU's dual-ported RAM.

Figure 16-4 **Dual-Ported Memory**


CPU 0
RAM
anchor
0x600
allocated pool
CPU 1
sm=0x1800600
Local address of VMEbus address 0 is 0x1000000
VMEbus address of dual ported RAM = 0x800000

In this example, the settings for the master (CPU 0) are as follows: the **SM_OFF_BOARD** parameter is **FALSE** and **SM_ANCHOR_ADRS** is 0x600 (the value is specific to the processor architecture), **SM_OBJ_MEM_ADRS** is set to **NONE**, because on-board memory is used (it is dynamically allocated at runtime); **SM_OBJ_MEM_SIZE** is set to 0x20000.

NOTE: When using dual-ported memory, the shared memory can be allocated from the master's kernel heap by setting **SM_OBJ_MEM_ADRS** to **NONE**. Note, however, the following caveats in this regard:

- The entire kernel heap must be mapped onto the shared bus address space (that is, the VMEbus address space), since the memory can be allocated from anywhere within it. The memory space of the shared bus mapped on the slaves must also be large enough to see the whole heap.
- Mapping the entire kernel heap might mean mapping the entire kernel or the entire local RAM of the master. In this case, there is the potential risk of a malfunctioning remote target overwriting critical kernel text and data structures on the master.

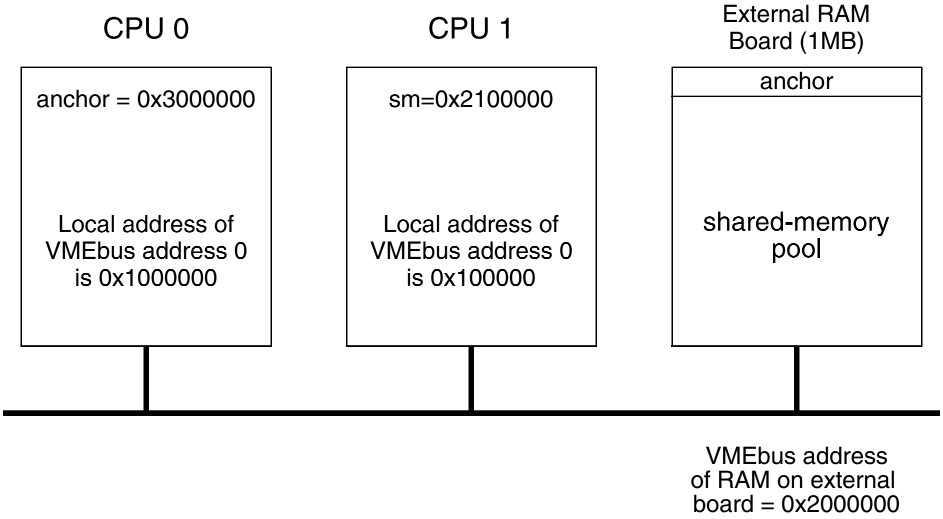
Wind River recommends the alternative of assigning a static address to **SM_OBJ_MEM_ADRS** and mapping only **SM_OBJ_MEM_SIZE** bytes of local RAM onto the shared bus memory space.

For the slave (CPU 0) in this example, 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.

Configuration for External Memory

In the configuration illustrated in Figure 16-5, the shared memory is on a separate memory board.

Figure 16-5 **External Memory Board**


CPU 0
CPU 1
External RAM Board (1MB)
anchor = 0x3000000
Local address of VMEbus address 0 is 0x1000000
sm=0x2100000
Local address of VMEbus address 0 is 0x100000
anchor
shared-memory pool
VMEbus address of RAM on external board = 0x2000000

For the master (CPU 0) in this example, the **SM_OFF_BOARD** parameter 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 (CPU 1), **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).

16.5.8 Configuration Example

This section describes the configuration settings for a multiprocessor system with three CPUs and dual-ported memory.

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. On CPU 0, the shared-memory pool is configured to be on-board. This memory is allocated from the processor's system memory. On CPU 1 and CPU 2, the shared-memory pool is configured to be off-board. Table 16-6 shows the parameter values set for the **INCLUDE_SM_OBJ** and **INCLUDE_SM_COMMON** components. Note that for the slave CPUs, the value of **SM_OBJ_MEM_SIZE** is not actually used.

Table 16-6 **Configuration Settings for Three CPU System**

CPU	Configuration Parameter	Value
Master (**CPU 0**)	**SM_OBJ_MAX_TASK**	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	**FALSE**
	SM_MEM_ADRS	NONE
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000
Slaves (**CPU 1**, **CPU 2**)	**SM_OBJ_MAX_TASK**	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	**TRUE**
	SM_ANCHOR_ADRS	(char *) 0xfb800000
	SM_MEM_ADRS	**SM_ANCHOR_ADRS**
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000

16.6 Displaying Information About Shared Memory Objects

The routine **smObjShow( )** can be used to display the current number of used shared-memory objects and other statistics, as follows:

```
-> smObjShow
value = 0 = 0x0
```

The **smObjShow( )** routine is automatically included if VxWorks is configured with the **INCLUDE_SM_OBJ** component.

The output of **smObjShow( )** 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
```

⚠ **CAUTION:** 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.

16.7 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.

16.7.1 Configuration Problems

Use the following list to confirm that your system is properly configured:

- Be sure to verify that VxWorks is configured with the **INCLUDE_SM_OBJ** component for each processor using VxMP.

- Be sure the anchor address specified is the address seen by the CPU. This can be defined statically (with the **SM_ANCHOR_ADRS** configuration parameter), or at boot time (with the **sm** boot loader parameter) 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.

- If applications create 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; see Table 16-5 for a list of the applicable configuration parameters. 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 and the size of the overall shared-memory region, **SM_OBJ_MEM_SIZE**. See *16.5 Configuring VxWorks for Shared Memory Objects*, p.744 for a discussion of configuration parameters.

- Operating time for the spinlock cycle can vary 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 the **SM_OBJ_MAX_TRIES** parameter), **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 spinlock prevents proper functioning of shared-memory objects. In most cases, this is due to problems with the shared-memory configuration (see above).

16.7.2 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 spinlock has been reached. By default, message printing is enabled.
- 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 spinlock on a particular CPU. A high value can indicate that an application might run into problems due to contention for shared-memory resources. For information about **smObjShow( )**, see *16.6 Displaying Information About Shared Memory Objects*, p.752 and the API reference for the routine.
- 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 *Wind River Network Stack for VxWorks 6 Programmer's Guide*.)

  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.

17
Distributed Shared Memory: DSHM

17.1 Introduction

The VxWorks distributed shared memory (DSHM) facility is a middleware subsystem that allows multiple services to communicate over different types of buses that support shared-memory communication. DSHM provides two main features for the services that make use of distributed shared memory: messaging over shared memory, and allocation of shared memory resources to services for their use in writing custom data. DSHM currently provides optional support for TIPC for communication over shared memory media.

Custom services can be developed for use with DSHM, and custom DSHM hardware interfaces can be developed for hardware that is not supported currently by Wind River. This chapter provides information about how to pursue both of these development activities.

17.2 Technology Overview

On VxWorks the term *shared memory* refers to a variety of software and hardware implementations, in which each node in a multi-node cluster has access to a shared memory region in addition to its own private memory. (For the purposes of discussion in this chapter, the term *node* refers to a single instance of an operating system).

In contrast to DSHM, the legacy Wind River shared memory technology is implemented with a *master-slave* model, in which the memory that is shared between nodes is located exclusively on a master node or external *dumb* board. That is, one centralized block of memory is shared, and one node is responsible for its initialization and management. The master node is involved to a certain extent in *all transactions* when the shared memory is local to its board, since remote nodes must read from its local memory. (For information about legacy systems, see *Wind River TIPC for VxWorks 6 Programmer's Guide* for information about TIPC over shared memory, and *16. Shared-Memory Objects: VxMP* for information about VxMP shared memory objects.)

The architecture of DSHM ensures superior reliability, determinism, and performance in comparison with the more conventional master-slave model. To do so, DSHM makes use of memory that is truly distributed across all nodes participating in a system. This distribution of memory can be accomplished in a variety of ways, including the following:

- Each node in a system has a pool of dedicated memory that is shared on the common bus, and all nodes can write to each of their peers' memory pools.
- A dumb board in a VME chassis provides a dedicated memory pool for each node.
- A multi-core board with one physical RAM DIMM provides a virtual distribution of memory; each OS instance allocates part of it as its own shared memory pool.

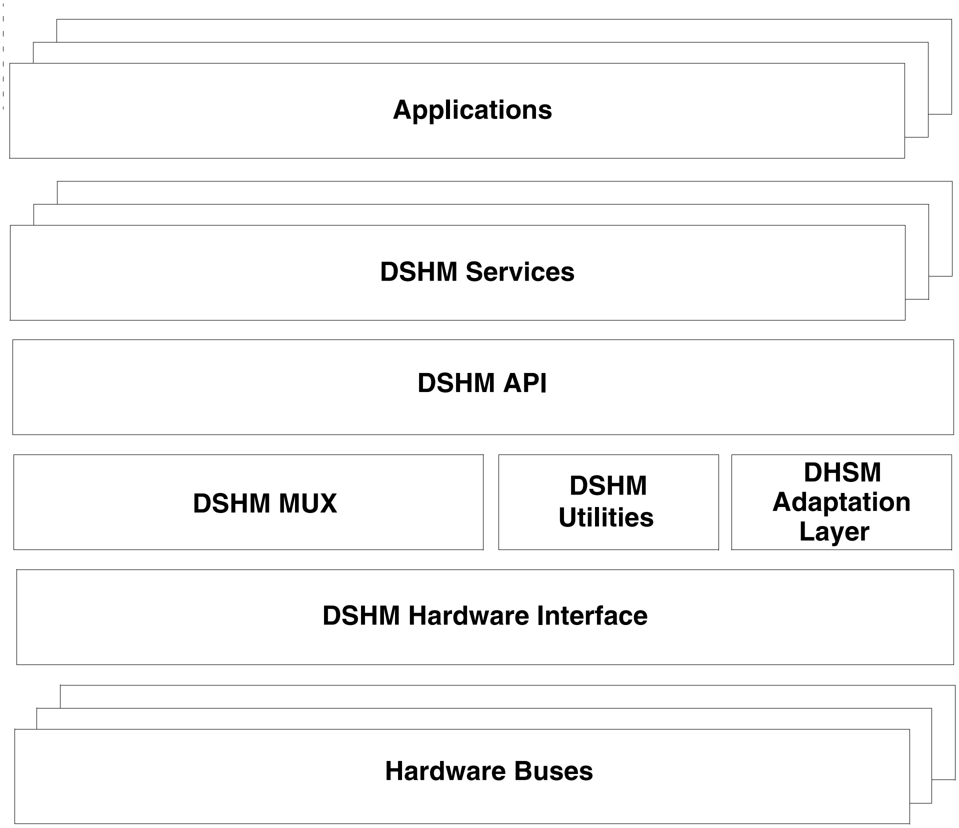
DSHM provides both a messaging service and facilities for allocation of shared memory resources to services for writing custom data messaging service. The messaging service is also used for passing data itself, such as network packets. Currently the messaging service is only supported over shared memory, although the DSHM API allows for expansion to other types of messaging.

The messaging protocol provides the means for sending simple messages between nodes, such as notification that a peer has come up or gone down, that a packet has been sent, or that a remote node has thrown a synchronization event. A message can be used like a semaphore. For example, Node 1 fills a buffer on Node 2, and then sends a message signalling the arrival of the data. Then it pends, waiting for the buffer to be available again. When the buffer is available, Node 2 then sends a message signalling this event.

17.2.1 Architecture

Figure 17-1 illustrates the architecture of the VxWorks distributed shared memory facility. This architecture is designed for a system in which memory is distributed across all nodes in the system.

Figure 17-1 **Distributed Shared Memory Architecture**



The DSHM elements illustrated in Figure 17-1 are as follows:

- Services make use of DSHM facilities (messaging over shared memory and allocation of shared memory resources for writing custom data) to support applications. The DSHM management service performs node and resource management activities.
- The DSHM API layer is the interface to services.
- The DSHM MUX (multiplexor) layer is the heart of the DSHM system. It provides the send and receive APIs, service query and access routines, and hardware query and access functionality. The MUX allows for multiple hardware elements and services to register with DSHM. It routes outgoing messages to the correct hardware bus on which they are transmitted, and routes incoming messages to the correct service for processing.
- The DSHM utilities are used internally by the DSHM implementation but are available to service and hardware interface writers.
- The DSHM adaptation layer provides macros and functions to OS-agnostic code that needs to make OS-specific calls.
- The hardware interface implements hardware-dependent features.

DSHM Management Service and Custom Services

Services interface directly with the DSHM facility. They are generally hidden from the user. For example, the TIPC bearer accessed by way of the socket API interacting with the TIPC stack.

The DSHM *management service* runs on each node in the system and handles the messages that deal with nodes appearing and disappearing, management of resources, and so on. One instance of the service runs on each node in the system as service number zero. Each is registered as a service when a hardware interface is ready to handle incoming messages.

Custom services can written based on the DSHM APIs. For information in this regard, see *17.4 Developing Custom Services*, p.767.

Hardware Interface

The hardware interface can implement hardware-dependent features, such as hardware messaging functionality provided by modern hardware (for example, RapidIO messaging). It currently implements shared memory messaging for cases in which hardware-assisted messaging is not available. For information about developing a hardware interface, see *17.5 Developing a Hardware Interface*, p.777.

17.2.2 Communication Model

Figure 17-2 illustrates the flow of messages and data using distributed shared memory in a two-node system using a TIPC bearer. In this example, the DSHM service knows that when it receives a message, it is intended for the TIPC bearer. The DSHM MUX routes the message to that service. Then, if the message is a **DSHM_SVC_NET_SEND** message, the TIPC bearer delivers the DSHM buffer to the TIPC stack (the message could also be a management message for the DSHM bearer, that does not go to TIPC). This type of message is associated with a buffer in shared memory, with contents that are identified as a TIPC message. Only the contents of the buffer are delivered to the TIPC stack. From that point on, DSHM does not have anything to do with the buffer. TIPC takes care of delivering it to the application, if it is application data (the contents could be a TIPC management message).

By way of analogy, you can think of DSHM as replacing the ethernet portion, the physical layer, of a network. DSHM performs a role similar to an ethernet device when it receives data. An ethernet device receives data in ethernet frame format. The software managing the device receives the physical data in a buffer in memory that it provided to the device. It then takes ownership of that buffer and replaces it in the device with another one (if available). Then, that buffer may be formatted, and then given to whomever it was sent to. With VxWorks END devices, networking MUX routes it to the correct protocol stack, which then routes it to the correct socket, which delivers it to the correct application.

Broadcasting

There are two ways of accomplishing broadcasting of data packets with DSHM: true broadcast and replicast.

True broadcast

The true broadcast implementation uses the broadcasting facility of the underlying hardware interface implementation, coupled with the fact that the local shared memory can also be read by remote nodes, on top of being written. The idea is to have a certain memory pool that is written by the broadcasting node, where the data packet is copied to. The node then sends a broadcast message of a certain type that specifies that there is a broadcast data packet to be read from the broadcasting node. This is sent using the broadcast address, in essence making use of the hardware interface's capability to send broadcast messages efficiently. When a remote node receives the message, it reads the data packet from the broadcasting node's shared memory. Some kind of mutual exclusion between the nodes might have to be used to prevent reusing (rewriting) the packet before all remote nodes have read it.

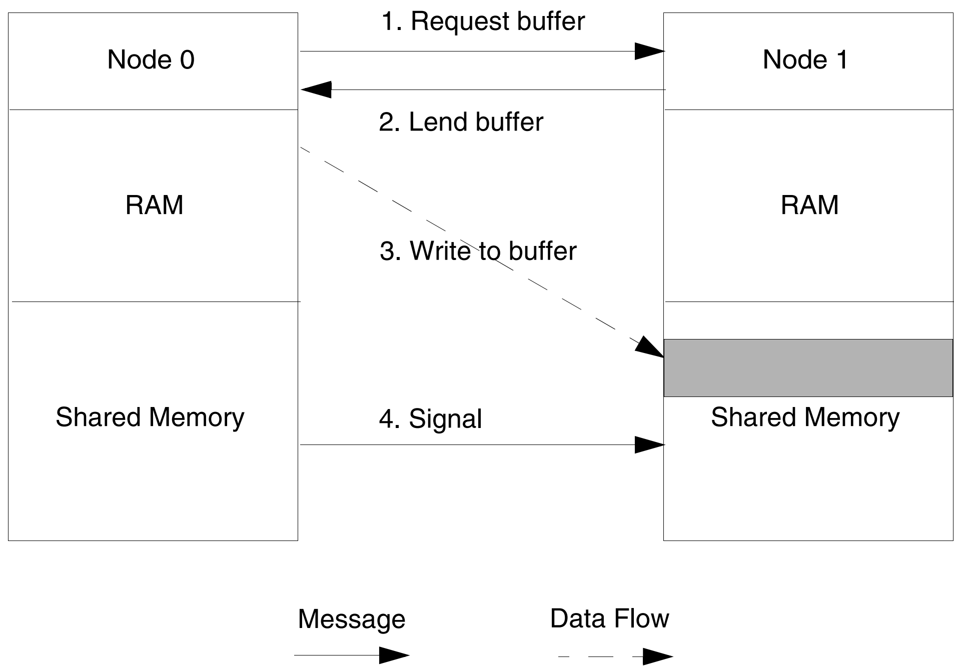
This method of broadcasting should be used whenever possible since it puts less burden on the broadcasting node, in effect sharing it on all receiving nodes. However, it might be impossible to use in certain cases. The TIPC DSHM bearer, when using shared memory messaging where a send operation can fail because of a full message queue, is an example of this. If the message sending operation cannot fail on a particular hardware interface implementation, it can be used.

Replicast

Replicasting is based on the concept of putting the burden of sending the broadcast packet on the sender. The broadcasting node in effect has to obtain a buffer for every node the broadcast is destined to, and copy the data into them. Then, a unicast message has to be send to each one of them. It can be the same type as a regular message signalling the arrival of a data packet since they are in essence the exact same thing.

The TIPC bearer uses this type of broadcasting when running over the default messaging implementation over shared memory, to be able to regulate the amount of messages that are sent to each node and thus ensure that sending a message, in a sane system, will always succeed. This allows for better flow control.

Figure 17-2 **Distributed Shared Memory Send Operation**



Send Operation

A send operations, as illustrated in Figure 17-2, takes place as follows:

1. Node 0 requests a buffer used for writing from Node 1.

   ```
   DSHM_BUILD(msg, DSHM_VNIC, 0, 1, DSHM_VNIC_REQUEST);
   dshmMuxMsgSend (hwID, msg, 0, 0);
   ```

   The **DSHM_BUILD** call fills the message header with the appropriate values The arguments (defined in **dshm.h**) are as follows: message (an array of type **char**; the size of which is defined in **dshm.h**), service number, source, destination, and type of message.

   Then Node 0 waits for buffer.

2. Node 1 allocates a buffer and informs Node 0 that it is available.

   ```
   DSHM_BUILD(msg, DSHM_VNIC, 1, 0, DSHM_VNIC_ALLOC);
   DSHM_DAT32_SET(1, pBuffer);
   dshmMuxMsgSend (hwID, msg, 0, 0);
   ```

3. Node 0 writes to the buffer.

   ```
   pBuffer = DSHM_DAT32_GET(msg, 1);
   bcopy (pPacket, pBuffer, sizeof(pPacket);
   ```

4. Node 0 sends message to Node 1 telling it that there telling it that there is data in the buffer.

   ```
   DSHM_BUILD(msg, DSHM_VNIC, 0, 1, DSHM_VNIC_SEND);
   DSHM_DAT32_SET(1, pBuffer);
   ```

   Then remote node reads the message and passes the packet to the stack.

For information about the C macro functions used in this example, see *17.4.2 DSHM Messaging Protocols and Macro Functions*, p.769.

Broadcast Operation

A broadcast operation involves the following steps (custom APIs required):

1. Node 2 obtains a buffer from its stack.
2. Node 2 writes to the buffer in local shared memory.
3. Node 2 broadcasts a message to remote nodes.
4. Node 0 and Node 1 receive the message, and obtain a broadcast tag, read the remote share memory to get the data, and the buffer is passed up the stack.

   A broadcast tag is part of the message. It is is a 32-bit value (**DAT32**), and is the second 32-bit value in the message (at index 1, which explains the 1 in the call).

For information about the macros and APIs used in this example, see *17.4.2 DSHM Messaging Protocols and Macro Functions*, p.769 and *17.4.3 DSHM Service APIs*, p.770.

17.3 Configuring VxWorks for DSHM

VxWorks can be configured and built with DSHM using the standard Wind River Workbench and **vxprj** facilities.

Workbench VxWorks Image Projects (VIPs) provide an asymmetric multiprocessing (AMP) option for creating projects with DSHM. The **vxprj** provides the **-amp** option for configuring and building VxWorks with DSHM from the command line. For example:

```
vxprj create -amp hpcNet8641 diab
```

CAUTION: Boot loaders must not be built with the AMP build option—neither with Workbench nor with **vxprj**. For more information about boot loaders, see *3. Boot Loader*.

17.3.1 Components and Parameters

DSHM componentization is divided into multiple areas, some mandatory, some removable, and some hardware dependant.

Core support

These are needed for correct functionality in a VxWorks system. They should be pulled in automatically by any other component that depends on DSHM.

INCLUDE_DSHM

BSP support enabling. This component is only present in BSPs that support DSHM. This is because DSHM needs some specific hardware to run, namely the possibility of having shared memory across VxWorks instances.

MUX. Allows multiple services to use the messaging system over the same medium, such as the shared memory between two cores on a multi-core chip. It also allows the usage of the same DSHM API by multiple concurrent media (for example, if there was a multi-core AMP system on a VME blade).

The **INCLUDE_DSHM** parameters are as follows:

- Maximum instances of hw buses (**DSHM_MAX_HW**). Unless there are two concurrent buses in your system, this should always be 1.
- Maximum number of services per bus (**DSHM_MAX_SERVICES**). The default is 2: one for the management service, and one for a user service, which might be one provided by Wind River, such as the TIPC bearer. If you intend to run more than one service concurrently, increase this value.

INCLUDE_DSHM_ADAPT

VxWorks adaptation layer. DSHM is meant to be portable to other operating systems. This component is the adaptation layer for VxWorks.

Distributed Multi-Processing Messaging

This is the selection of the messaging type on a particular bus if more than one is available. Currently, only shared memory messaging (**INCLUDE_DSHM_MSG_SM**) is available. It should always be selected.

Peer-to-Peer Drivers

These are the drivers that implement the shared memory messaging and housekeeping of the shared memory used for data passing. They are different for each BSP that provides support for DSHM. However, some parts are shared between implementations.

Note that the driver implementation is a VxBus implementation. VxBus will always get added to the VxWorks image if it is not already selected.

INCLUDE_DSHM_BUS

DSHM virtual bus. This should always be selected. It provides the framework for drivers.

INCLUDE_DSHM_BUS_PLB

DSHM virtual bus on PLB. The current implementation all are for a Processor-local-bus type bus, such as the one on a multi-core chip. Even the VxWorks simulator implementation follows that model, as if the VxWorks simulator instances would share a local bus. This must be selected. It is also where most of the parameters live. The parameters are as follows:

DSHM_BUS_PLB_NODE

Address of local node. This is the unique address on the shared bus. The current drivers are able to find their own address at runtime, using the processor number assigned at boot time with the boot line. Use -1 for this, or another number to force a specific one for a specific image. The address must be less than the next parameter.

DSHM_BUS_PBL_MAXNODES

Maximum number of nodes. There can be no more nodes than this value in the system. Note that all nodes must agree on that value so that it works as intended.

DSHM_BUS_PLB_NENTRIES

Number of entries in the shared memory. Each message sent over DSHM is sent asynchronously and takes up one entry in the message queue. When the queue is full, if another message is sent, an error code is returned to the sender. For implementation reasons, the real number of concurrent messages is actually one less than this number.

DSHM_BUS_PLB_NRETRIES

Number of retries. When trying to send a message, the internals will actually retry sending in the case where the queue is full. If you do not want that to happen, you can set it to 0. If you would like more retries, pick a higher number. It can help getting less sending errors. WARNING: This is a ‘busy’ retry, in effect hogging the CPU.

DSHM_BUS_PLB_RMW

Read-modify-write routine. This is per-bus type. It should be left alone.

DSHM_BUS_PLB_POOLSIZE

Shared memory pool size. If you decide to share more or less memory on this node, adjust this number accordingly.

DSHM_BUS_PLB_ENTRY_SIZE
Currently unsupported.

Virtual bus controller and peer drivers are BSP-specific, but have to be selected for DSHM to work properly. They cannot be put in as defaults since each different hardware implementation component has a different name. The components are as follows:

- wrSbc8561 and hpcNet8641: **INCLUDE_DSHM_BUS_CTLR_8641** and **INCLUDE_DSHM_VX8641**
- Any VxWorks simulator: **INCLUDE_DSHM_BUS_CTLR_SIM** and **INCLUDE_DSHM_VXSIM**
- Any sibyte board (sb1250/sb1480): **INCLUDE_DSHM_BUS_CTLR_SIBYTE** and **INCLUDE_DSHM_VXSIBYTE**

Interface Debug Aid

INCLUDE_DSHM_DEBUG
Provides a debugging aid that allows for multiple levels of debugging output, selectable at runtime. You can choose an initial level (**DSHM_INIT_DBG_LVL**). By default, it is **OFF (**no message is printed). See **dshm/debug/dshmDebug.h** for more information on usage.

Services

INCLUDE_DSHM_SVC_MNG
Services provided by Wind River. The node manager must be present. The TIPC bearer is a special case and lives under the TIPC component directory. For information about the TIPC bearer, see the *Wind River TIPC Programmer's Guide*.

Utilities

These are utilities provided for service and hardware interface writers. They are used internally in the TIPC bearer and the hardware interface implementations provided by Wind River. They are pulled in when needed.

17.4 Developing Custom Services

DSHM provides macros and APIs that can be used to develop custom services. Custom services may, for example, be developed to provide the following sorts of features:

- Faster data transfers—without going through TIPC or IP—by using a ring of data buffers in shared memory. When a number of buffers are ready for reading by the peer, the local node sends a message with a payload describing how many buffers should be read. The peer sends a message back when the buffers are read and ready to be filled again, also with a message payload specifying how many buffers (to keep the amount of messages sent to a minimum). Each side keeps a local view of the state of the system. This implementation would be less flexible than TIPC or IP, but would also have considerably less overhead, and potentially faster rates of data transfer. A simplified version of this type of feature is provided in *17.4.4 Service Code Example*, p.771.
- Synchronization between tasks across nodes by sending a message to a remote node when a resource is available. This can simply be implemented by having local tasks pending on a local semaphore that is then given when the message arrives. This is similar to VxMP semaphores, but more constrained, as this is strictly peer-to-peer communication.
- Control and command messages that do not need any data transfer, for a specific application. The service could support a small payload, instant feedback as to whether the message got through or not, and so on.

Services provide functionality over a specific bus type, and make calls to the DSHM APIs to interface with the bus (to obtain a shared memory, to send messages, and so on).

Services should also register callback routines for events such as a node joining or departing from a system. When an instance of a DSHM hardware interface discovers a node, it calls **dshmSvcNodeJoin( )**, which calls callbacks installed by all services that need to be notified.

When discovering a node, the hardware interface propagates that information to all services registered on that bus. This allows services to take actions such as initialization and allocation of data structures used for that particular node. The services have their *join* callback invoked at that point. The callbacks are described *17.5.2 Callbacks*, p.778.

If a node is declared gone or dead, the hardware interface instance calls **dshmSvcNodeLeave( )**, which similarly calls callbacks installed by all services that need to be notified. If a service on the local node decides to quit, it can call **dshmMuxSvcWithdraw( )**, which calls a callback registered by the service to do cleanup.The service can then call **dshmMuxSvcWithdrawComplete( )** when it is satisfied that the cleanup is completed.

Callbacks can be used to take care of allocating shared memory pools, network buffers, and so on. The custom service writer provides the desired functionality.

17.4.1 Service Numbers

All DSHM services are identified internally by unique service numbers. Wind River reserves zero for the DSHM management service (for information about the service, see *DSHM Management Service and Custom Services*, p.758).

For the greatest efficiency, use the smallest service numbers possible, since they are used directly as indices into arrays. Service number should also be implemented as configurable parameters, in the event that there is a conflict with another software provider's usage. The numbers should also be documented if the software is provided to a third party.

The maximum number of services is defined with the **DSHM_MAX_SERVICES** parameter of the **INCLUDE_DSHM** component. This parameter should be set to one more than the number of services that will be supported (that is N-1), because the service number zero is reserved for the DSHM management service. If that number is 3, for example) each bus can have 2 (plus the management service) services each, which can be totally different.

17.4.2 DSHM Messaging Protocols and Macro Functions

Each custom service must provide its own message types for its own protocol. DSHM provides a set of C macro functions to facilitate building messages. The macros can, for example, be used to build the message header, access each one of the per-message type parameters, and so on. The messaging macros are as follows:

- **DSHM_DAT8_SET( )**
- **DSHM_DAT16_SET( )**
- **DSHM_DAT32_SET( )**
- **DSHM_DAT_GET( )**
- **DSHM_DAT8_GET( )**
- **DSHM_DAT16_GET( )**
- **DSHM_DAT32_GET( )**
- **DSHM_SVC_GET( )**
- **DSHM_SRC_GET( )**
- **DSHM_DST_GET( )**
- **DSHM_TYP_GET( )**

The macros are defined in *installDir***/vxworks-6.***x***/target/h/dshm/dshm.h**.

The following macros builds a correctly formatted message in the **msg** parameter using the four other parameters:

```
#define DSHM_BUILD(msg, svc, dest, src, type)
```

The next macros accesses the data of a specified size at a specified offset:

```
#define DSHM_DAT[8|16|32]_[GET|SET](msg, offset)
```

The offset units depends on the width of data to be set or retrieved. For example, the following call retrieves the second byte in the message:

```
DSHM_DAT8_GET(msg, 1)
```

But the following macro retrieves the second word:

```
DSHM_DAT32_GET(msg, 1)
```

If a message body is comprised of one word, one byte, one byte, one half-word and one word, the following would retrieve each, respectively:

```
DSHM_DAT32_GET(msg, 0)
DSHM_DAT8_GET(msg, 4)
DSHM_DAT8_GET(msg, 5)
DSHM_DAT16_GET(msg, 3)
DSHM_DAT32_GET(msg, 2)
```

The following macros used for obtaining a pointer to a message, and for casting a pointer to a DSHM message pointer:

```
DSHM(variable_name);
DSHM_TYPE(ptr_name);
DSHM_CAST(ptr);
```

For an example of how the macros are used, see *17.4.4 Service Code Example*, p.771.

17.4.3 DSHM Service APIs

The APIs described in Table 17-1 are provided by **dshmMuxLib** for use by custom services.

Table 17-1 **DSHM Service Routines**

Routine	Description
dshmMuxHwGet( )	Obtain an hardware registration handle based on name.
dshmMuxHwNodesNumGet( )	Obtain the maximum number of nodes on a hardware bus.
dshmMuxHwRmwGet( )	Obtain the atomic Read-Modify-Write routine on this bus.
dshmMuxHwOffToAddr( )	Translate a shared memory offset to a local address.
dshmMuxHwAddrToOff( )	Translate a local address to a shared memory offset.
dshmMuxHwLocalAddrGet( )	Obtain address of the local node.
dshmMuxSvcRegister( )	Register a service with the MUX.
dshmMuxSvcObjGet( )	Retrieve the reference to a service object.
dshmMuxSvcWithdraw( )	Remove service from MUX.
dshmMuxMsgSend( )	Transmit a message.

Table 17-1 **DSHM Service Routines**

Routine	Description
dshmMuxMsgRecv( )	Receive a message.
dshmMuxMemAlloc( )	Allocate shared memory from a specific hardware.

Once a service is up and running, the bulk of the API calls that are used would be **dshmMuxMsgSend( )** and **dshmMuxMsgRecv( )**, which are called on a one-to-one basis with the number of messages directed to the local node, as well as **dshmMuxSvcObjGet( )** and **dshmMuxSvcObjRelease( )** for obtaining the object when sending or receiving. If data buffers are exchanged and are dynamic—as with a network driver service—**dshmMuxHwAddrToOff( )** and **dshmMuxHwOffToAddr( )** would also be used for converting buffer pointer to shared memory offsets and back. The remainder of the APIs would be used infrequently, primarily for housekeeping functions.

17.4.4 Service Code Example

The code provided below illustrates using messages for synchronization.

```
/* dshmTestDemoSync.c - Demo code: synchronization between nodes */

#include <taskLib.h>
#include <sysLib.h>
#include <stdio.h>
#include <stdlib.h>
#include <dshm/dshm.h>
#include <dshm/dshmMuxLib.h>

/*
DESCRIPTION

This is an example service, where two nodes each have one task that depends
on
the remote node having finished some operation. DSHM messages are used as
synchronization events across nodes.
*/
```

```
/* the service object */

typedef struct _svc_obj
    {
    DSHM_SVC_OBJ    obj;        /* base object */
    struct
        {
        SEM_ID        sync;    /* binary sem: recv'd sync message */
    uint16_t    addr;    /* remote node address */
    uint16_t    pad;
        } remote;        /* only support one remote peer */
    } svc_obj;

/* the service type */

#define SVC_TYPE    0x02        /* service type */

/* message types */

#define MSG_SYNC    0x11    /* normal message */
#define MSG_JOIN    0x22    /* join */

/* seconds */

#define SECONDS(x)  (x * sysClkRateGet())

/* callback prototypes */

static STATUS rx
    (
    svc_obj * const pObj,   /* the service object */
    DSHM(msg)               /* message received */
    );
static STATUS join
    (
    svc_obj * const pObj,   /* the service object */
    const uint_t addr       /* address of node joining */
    );
static void leave
    (
    svc_obj * const pObj,   /* the service object */
    const uint_t addr       /* address of node leaving */
    );

/* service hooks structure */
static const DSHM_SVC_HOOKS svchooks =
    {
    (STATUS(*)(void * const, DSHM_TYPE()))  rx,
    (STATUS(*)(void * const, const uint_t)) join,
    (void(*)(void * const, const uint_t))   leave,
    (void(*)(void * const))                 NULL
    };
```

```
/* worker task */

static void worker (SEM_ID sync, uint16_t hw, uint16_t svc, uint16_t remote);

/* service init */

/***************************************************************************
**
 *
 * dshmTestMsgStart - start test service
 *
 */

void dshmTestDemoSync
    (
    const char * const pHwName  /* hw interface name */
    )
    {
    svc_obj *pObj;           /* the service object */
    int hw;                  /* hw registration number */
    DSHM(msg);               /* the messages */

    if ((NULL == pHwName) || (-1 == (hw = dshmMuxHwGet (pHwName))))
        {
        printf ("FAILED: no such hw registered\n");
        return;
        }

    pObj = malloc (sizeof (svc_obj));
    if (!pObj)
        {
        printf ("FAILED: out-of-memory\n");
        return;
        }

    memset ((char *)pObj, 0, sizeof (svc_obj));
    pObj->obj.hw = hw;
    pObj->obj.svc = SVC_TYPE;
    pObj->remote.addr = DSHM_ADDR_INVALID;
    pObj->remote.sync = NULL;

    if (dshmMuxSvcRegister (hw, SVC_TYPE, pObj, &svchooks) == ERROR)
        {
        printf ("FAILED: cannot register service\n");
        return;
        }
    /* from here on, cannot use pObj directly */

    /* broadcast to remote nodes that we're ready */
    DSHM_BUILD (msg, SVC_TYPE, 0, DSHM_ADDR_BCAST, MSG_JOIN);
    dshmMuxMsgSend (hw, msg, 0, 0);
    }
```

```
/* service callbacks */

/***************************************************************************
**
 *
 * rx - invoked when receiving a message
 *
 * This service handles the following types of messages:
 * - remote has joined: perform needed initialization.
 * - remote sync: signal worker task that it can resume its work.
 */

static STATUS rx
    (
    svc_obj * const pObj,   /* the service object */
    DSHM(msg)               /* message received */
    )
    {
    uint16_t src;

    src = DSHM_SRC_GET (msg);
    switch (DSHM_TYP_GET(msg))
        {
        case MSG_JOIN:
        /* only acknowledge if not ourselves */
        if (src != dshmMuxHwLocalAddrGet (pObj->obj.hw))
        {
        join (pObj, src);
        }
            break;
        case MSG_SYNC:
        /* signal worker task that remote finished its work */
        semGive (pObj->remote.sync);
            break;
        default:
            /* discard */
            break;
        }

    dshmMuxSvcObjRelease (pObj->obj.hw, pObj->obj.svc);
    return 0;
    }

/***************************************************************************
**
 *
 * join - initialize service interaction with a remote node
 *
 * This service expects one remote node to interact with. If no node has
 * previously joined the service, this routine will create the
synchronization
 * semaphore, record the remote node's address and spawn the worker task that
 * waits for synchronization events.
 */
```

```
static STATUS join
    (
    svc_obj * const pObj,   /* the service object */
    const uint_t addr       /* address of node joining */
    )
    {
    DSHM(msg);    /* reply message */
    int tid;    /* worker task id */

    if (pObj->remote.addr != DSHM_ADDR_INVALID)
    {
    return ERROR;    /* remote node already registered */
    }

    pObj->remote.sync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    if (!pObj->remote.sync)
        {
        printf ("FAILED: out-of-memory\n");
        return ERROR;
        }
    pObj->remote.addr = addr;

    printf ("join called\n");

    /* reply to remote node, telling it we're here */
    DSHM_BUILD(msg, pObj->obj.svc, 0, addr, MSG_JOIN);
    dshmMuxMsgSend (pObj->obj.hw, msg, 0, 0);

    tid = taskSpawn ("worker", 100, 0, 0x1000, (FUNCPTR)worker,
                        (int)pObj->remote.sync, (int)pObj->obj.hw,
                    (int)pObj->obj.svc, (int)pObj->remote.addr,
                        0,0,0,0,0,0);

    if (tid == ERROR)
        {
        logMsg ("Could not spawn worker task\n",
                    0,0,0,0,0,0);
        pObj->remote.addr = DSHM_ADDR_INVALID;
        semDelete (pObj->remote.sync);
        return ERROR;
        }

    return OK;
    }

/***************************************************************************
**
 *
 * leave - invoked when a remote node disappears
 *
 * This routine cleans up the node-specific service data when a remote node
 * disappears.
 */
```

```
static void leave
    (
    svc_obj * const pObj,   /* the service object */
    const uint_t addr       /* address of node leaving */
    )
    {
    if (pObj->remote.addr == addr)
    {
    semDelete (pObj->remote.sync);
    pObj->remote.addr = DSHM_ADDR_INVALID;
    }
    printf ("leave called\n");
    }


/***************************************************************************
**
 *
 * worker - worker task that waits on sync events from remote node
 *
 * This routine simulates work that needs synchronization from a remote node.
 * It pends on a semaphore that is given when the remote peer sends a sync
 * event, signalling the local node that it a condition needed by it has been
 * met so that it can resume work.
 */

static void worker
    (
    SEM_ID sync,    /* synchronization semaphore */
    uint16_t hw,    /* hardware bus unique identifier */
    uint16_t svc,    /* service unique identifier */
    uint16_t remote    /* address of remote node on bus */
    )
    {
    logMsg ("worker task: interacting on svc %d, hw %d with node %d\n",
        svc, hw, remote, 0,0,0);

    FOREVER
    {
    DSHM(msg);

    /* placeholder: do real work */
    taskDelay (SECONDS(2));

    /* signal remote task we're done with our part */

    DSHM_BUILD(msg, svc, 0, remote, MSG_SYNC);
    dshmMuxMsgSend (hw, msg, 0, 0);

    /* wait for sync event from remote node */
    if (semTake (sync, WAIT_FOREVER) == ERROR)
        {
        logMsg ("Semaphore deleted, remote task must have quit\n",
                        0,0,0,0,0,0);
        break;
        }
    logMsg ("Got sync event, resuming...\n",
                        0,0,0,0,0,0);
    }
    }
```

17.5 Developing a Hardware Interface

This section describes how to develop an interface for hardware that is not currently supported by Wind River. A DSHM hardware interface must include functionality for a bus controller device driver and a peer node device driver (for remote node support). DSHM drivers must conform to the VxBus device driver model (for information about VxBus, see the *VxWorks Device Driver Developer's Guide*).

⚠ **CAUTION:** The current implementations of hardware support and services require that shared memory always be present. Systems in which memory can be swapped out and cause exceptions when accessed are not currently supported.

A peer (remote) node is seen as a device by DSHM. Peer node device drivers provide the means for accessing the distributed shared memory on a peer nodes, and for signalling peer nodes with inter-processor interrupts.

Most of the functionality required by DSHM for the bus controller device driver and peer node device driver can be provided in driver code, although some functionality may require support in the BSP (for example, triggering interrupts on the sb1250).

DSHM supports multiple concurrent buses that provide shared memory functionality on the same node. As the DSHM MUX takes care of such combinations, the BSP does not have to.

The interfaces provided by DSHM conform to the VxBus driver model. These interfaces see the local node as a virtual controller and peer nodes as virtual devices sitting on the virtual controller. The code can therefore be reused for different BSPs that support the same devices. Furthermore, common base drivers exist for similar implementations, such as multicore devices.

Currently, implementations of DSHM hardware interfaces are provided for the following multicore devices: MIPS sb1250 and sb1480, and PowerPC hpcNet8641.

Most of the code is shared between them. These BSPs can be used as starting points to implement support for similar BSPs. They should be a good starting point for implementing support for different buses as well, not only for the local bus used by multi-core devices, but particularly for bus controller drivers that are designed to use shared memory as the messaging interface (including VMEbus).

Note that it is especially important that the local node should also be seen as a remote node on the bus interface (to be able to participate in broadcasts for example). If a service needs to treat the local node differently, the service should be written so as to address this requirement—and not the bus controller driver.

17.5.1 Driver Initialization

Bus controller device drivers differ primarily in their *xxx***Init2( )** routine. The major differences due to the location of the anchor for shared memory messaging and the location of the pool of shared memory. In addition, some parameters (such as shared memory) can be defined statically for some systems, but not for others; it depends on the configuration of the physical shared memory.

The peer node device drivers also mostly differ in their the *xxx***Init2( )** routines, and for mostly the same reasons as the bus controller device driver. An instance of a peer driver is a peer device. There is a peer device for each node in the DSHM system, on a bus. Peer devices must be able to find the remote shared memory data structures in order to message the remote node that it is responsible for.

Also, peer drivers can differ in the means of interrupting the remote nodes, and so on. For example, the hpcNet8641 uses the EPIC interrupt controller to send an inter-processor interrupt to remote nodes, while the SB1250 uses a different mechanism.

VxBus has three initialization phases. During the first phase, the kernel memory allocator is not initialized yet, so **malloc( )** cannot be used to get memory from the kernel heap. If a device driver needs to be initialized during the first phase—and it needs to dynamically allocate memory—VxBus has its own memory allocator that can be used (which is more limited and manages a very small amount of memory).

17.5.2 Callbacks

Each bus controller device driver must provide a set of callbacks that are invoked by the DSHM MUX. The required types of callbacks are as follows:

Allocate

Shared memory allocation callback. A pool of shared memory on the local target is managed by the bus controller driver. The memory can be reserved for each service by passing a empty-size pool to the bus controller. If a non-empty pool is passed, the interface can allocate from it. The **dshmMem** library provides allocation and de-allocation routines that can be used by the bus interface as its allocation and free callbacks. The interface writer is also free to implement custom callbacks.

Free

Free shared memory callback. The counterpart to the allocation callback.

Transmit

Message transmission callback. This routine sends one message to the destination provided.

Broadcast

Method for broadcasting a message to all nodes on the bus, including the local node.

Test-and-Set

Test-and-set primitive provided by the bus controller driver. An atomic operation that checks if a zero is stored at a memory location and replaces it with a non-zero value, if so. Returns TRUE if so, FALSE if not.

Clear

The opposite of test-and-set. Can be set to NULL if not needed (such as for multi-core devices). Some bus controllers need a special implementation (some VMEbus bus controller chips need it).

Offset-To-Address

Callback that converts a shared memory offset from the start of the address that the shared memory is *visible*, to a local pointer to that same location.

Address-To-Offset

Callback that converts a pointer to a local shared memory address into an offset from the start of the address at which the shared memory is *visible*. This allows passing values that can be converted to pointers on the remote nodes, in case they see the shared memory at a different address.

The offset-to-address and address-to-offset routines should always be used when passing addresses (of buffers) between nodes, unless the service is only meant to be used in situations in which the shared memory address is the same on all nodes.

Local Address

Obtain the address of the local node on the common bus. This is the address used for messaging.

Virtual Memory Allocator

Used if the bus controller driver can allocate a virtual address range dynamically. Unused in Wind River implementations. Can be left NULL.

Fast Copy

Used if the bus controller driver provides something better than **bcopy( )** for writing to shared memory. Can be left NULL.

Prototypes for Callbacks

```
void*(*alloc)       /* memory allocation */
    (
    DSHM_PEER_ID,
    int * const,
    const int
    );
STATUS(*free)       /* memory deallocation */
    (
    DSHM_PEER_ID,
    void * const
    );
STATUS(*tx)         /* regular transmit */
    (
    DSHM_PEER_ID,
    DSHM_TYPE()
    );
STATUS(*bcast)      /* broadcast on hw bus */
    (
    DSHM_PEER_ID,
    DSHM_TYPE()
    );

DSHM_TAS tas;       /* test-and-set routine */
DSHM_TAS_CLEAR clear;   /* test-and-set clear routine */

void*(*offToAddr)   /* sm offset to local address */
    (
    DSHM_PEER_ID,
    const uint32_t
    );
uint32_t(*addrToOff)/* local address to sm offset */
    (
    DSHM_PEER_ID,
    const void * const
    );
uint16_t(*localAddr)/* obtain address of local node */
    (
    DSHM_PEER_ID
    );
VIRT_ADDR(*vmAlloc) /* address range allocation (manager only) */
    (
    DSHM_PEER_ID,
    const int
    );
void(*fastcopy)     /* fast bcopy, dma, page swap, etc */
    (
    const char *,
    char *,
    int
    );
```

17.5.3 Registering with the DSHM MUX

When the bus controller device is ready to be accessed by services, it must announce its presence. To do so, it calls **dshmMuxHwRegister( )**. This allows services to register for that interface, and messages to be sent on it. At this point, remote nodes are not visible to the DSHM hardware interface yet.

When a remote node subsequently is discovered, the bus controller device should call **dshmMuxSvcJoin( )** so that services running are notified of the nodes appearance. Service may require notification so they can allocate resources for the managing their view of the remote node. These resources may be either local or in shared memory (for example, buffers for incoming data from the peer node). The bus controller device should therefore also call **dshmMuxSvcLeave( )** when a remote node disappear, to let the service know of that status change as well.

However, a service can be started at any time, so when a service registers with the DSHM MUX, it should broadcast its presence to the remote nodes, most likely by sending a service-specific **JOIN** message. For more information about writing services, see *17.4 Developing Custom Services*, p.767.

17.5.4 Messaging Support

DSHM attempts to provide immediate feedback about whether a message has reached the intended peer node recipient. In order to do so, the bus controller driver must accomplish one of the following with its message transmission facility:

- Feedback is immediately available from the messaging system if the message cannot be delivered. This could be the result of a full queue, for example. The shared memory messaging facility follows this model.
- The messaging system is loss-less. This meaning that the send operation might report that the message has been sent even if it has not got through to the receiver yet, but there is some level of support in the messaging system to keep retrying until the message goes through.

Note that a message getting through does not imply anything more than the fact that it has arrived at its destination. It does not mean that there is any certainty of it being processed by the service on the remote node that received it. Services should, of course, be designed to handle all incoming messages. Failure to handle a message should only occur if something is wrong on the remote node (the node does not respond, it is rebooting, and so on).

DSHM expects messages to be delivered in the same order in which they were sent. For example, if a **JOIN** message is sent before a **REQUEST** message, and the sender expects the **JOIN** to be processed first, since if not, the **REQUEST** message will be ignored. They must arrive in order.

Sending a message must also double as a memory barrier. If a buffer in shared memory is filled, and then a message sent to indicate that the buffer is ready to read, the sending of the message should not occur before the data is actually written to memory. Services should not have to create the barrier themselves before signaling remote nodes. DSHM provides the **DSHM_MEM_BARRIER( )** macro, which is defined in *installDir***/vxworks-6.***x***/target/h/dshm/adapt/types.h**.

Shared Memory Messaging Support

The messaging system provided with this release of DSHM is based on shared memory. It does not require any special hardware support except for providing shared memory between nodes, and a way of sending interrupts to peer (remote) nodes. The shared memory itself must be fully coherent between nodes. Coherency is required both for messaging and for the shared memory pool used by services. It can be achieved by using a non-cached region of memory, by using a snooping mechanism on a cached region, and so on.

Note that certain instructions used for inter-process synchronization may require specific cache modes. For example, on PowerPC architectures, the ll/sc primitives used to implement atomic operations require the cache to be in a certain mode; otherwise they cause an exception. Consult your hardware architecture documentation in this regard.

Each node using shared memory messaging must provide a small data structure at a well-known address, accessible by all nodes in the system, called the **anchor**, which is defined in the BSP. This data structure's size is determined as follow:

$$(12 + n * 4) \text{ bytes}$$

where n is the max number of nodes allowed (for information about configuring the maximum number of nodes, see *17.3 Configuring VxWorks for DSHM*, p.763).

Most of the data in the anchor is for discovery and keep alive signal. It also provides the location of the rest of the shared memory provided by the node.

As noted earlier, the action of sending a message must be protected by a memory barrier as well (with the **DSHM_MEM_BARRIER( )** macro). With the shared-memory messaging implementation, this is achieved through a spinlock mechanism that keeps messaging structures coherent when accessed concurrently by multiple nodes.

Since this messaging mechanism relies on a portion of shared memory reserved in advance (of a fixed size), the number of messages, that can be sent by the messaging node and that have not yet been processed by the receiving node, is limited. If the sender fills the message input on the remote node, it will try re-sending a limited number of times (configurable) before giving up and returning an error to the caller. Note that the sender is *busy-waiting* (spinning in a loop) while repeatedly trying to send.

To avoid busy-waiting while sending a message, special consideration should be given to the implementation services.

Services that are used over a bus controller device driver that uses shared memory messaging should try to throttle their messages to prevent sending more than are allowed. This should be combined with a configurable setting for the number of concurrent messages to achieve the desired effect.

For example, consider a system with three nodes. Each node provides two buffers for each of its peer to write to. Each of its peer does the same. So, the local node has two buffers on each peer to write to, and two buffers of its own, per-peer, from which it reads.

The service keeps a local view of the status of the remote buffers. In this service, a message is sent every time a buffer is filled to let the remote node that it should read it. A message is also sent when a local buffer is read as well, to signal the remote node that it can use it again. In this case, the concurrent messages that can have been sent by one node to the other are as follows:

- Two messages that tell the remote node to read the buffers (the service should stop sending when the buffers are full, so it won't send a next message until it got a reply from the remote node telling it can reuse the first buffer)
- Two messages that tell the remote node that it can reuse its two buffers (the local node only sends these after the remote node has sent it messages telling it to read the buffers, and we know it is limited to two of them at once).

So if this service is the only one running in this system, four message entries should be sufficient.

The number of required messages entries must, however, be set to one more than the number calculated. This is due to the implementation of the messaging system, which uses a ring and must have one empty entry. In addition, for performance considerations, the number of entries in the ring must be a power-of-two. If a number that is not a power-of-two parameter is specified, it is automatically rounded down to the nearest power-of-two.

Shared Memory Lock-Less Discovery

Bus controller device drivers using shared memory messaging must use a method other than sending a join (**JOIN**) message to remote peers when a node comes up. The reason behind this has to do with rebooting nodes and re-initialization of shared memory data structures. This method is comprised of two types of messages, reset (**RST**) and acknowledge (**ACK**). The following diagram illustrates the state machine of the discovery mechanism, for each peer relative to the remote node.

The first letter of the state is the state of the current node and the second is the state of the remote peer, as viewed by the local node. The **R** is *reset*, **U** is *unknown*, and **A** is *received acknowledgment*. The state transitions show the event received/action taken. The AA designation is the *fully-up connection* state.

17.5.5 Management Service

There is a generic management service that exists in the VxWorks adaptation layer component. It can be registered with a hardware interface by calling **dshmMngSvcInstall( )** (see **dshmBusCtlrSim.c** under **target/src/dshm/drivers** for an example of usage). To be of any use, the driver must provide a **dshmBusCtlrMethodNodeReadyId** method. Look in **dshmBusCtlrPlb.c** for an example (that file is the base driver on top of which **dshmBusCtlrSim.c** is built). **dshmBusCtlrPlbNodeReady** is registered as a method in the method table.

This service handles two types of messages currently: **DSHM_TYP_MNG_JOIN** and **DSHM_TYP_MNG_QUIT**, although the second is a placeholder in the VxWorks adaptation layer that simply acknowledge the message and logs a console message. The implementation is in **target/src/dshm/service/dshmSvcMng.c**, and can be used as a starting point for a more involved implementation if needed. The service is very minimal at this point since all the current hardware interface use another mechanism for detecting topology changes.

One of the main goals of this service is to propagate the information that a node has either appeared or disappeared to the services, which are the modules that provide the bulk of the functionality that is of interest to end users. Services that register a *join* and a *leave* callback receive events those events when the hardware interface sees a change in the nodes topology.

17.5.6 DSHM Hardware Interface APIs

The routines described in Table 17-1 are for use by a hardware interface.

Table 17-2 **DSHM Hardware Interface Routines**

Routine	Description
dshmMuxHwRegister( )	Register a hardware bus with the MUX.
dshmMuxSvcNodeJoin( )	Call *join* callbacks from registered services.
dshmMuxSvcNodeLeave( )	Call *leave* callbacks from registered services.

18
Message Channels

18.1 **Introduction**

Message channels are a socket-based facility that provides for inter-task communication within a memory boundary, between memory boundaries (kernel and processes), between nodes (processors) in a multi-node cluster, and between between multiple clusters. In addition to providing a superior alternative to TCP for multi-node intercommunication, message channels provide a useful alternative to message queues for exchange data between two tasks on a single node.

Message channels provide a connection-oriented messaging mechanism. Tasks exchange information in the form of messages that can be of variable size and format. They can be passed back and forth in full duplex mode once the connection is established.

The implementation of message channels for multi-node communication is based on TIPC, which provides faster throughput than TCP. The implementation of message channels for single-node, inter-process communication is somewhat slower than message queues, but provides notable advantages (including portability of applications for multi-node message channel use; for more information see *18.8 Comparison of Message Channels and Message Queues*, p.806).

Message channel communication can take place between tasks running in the kernel and tasks running in processes (RTPs) on a single node, as well as between multiple nodes, regardless of the memory context in which the tasks are running. For example, message channels can be used to communicate between:

- a task in the kernel of one node and a task in a process on another node
- a task in a process on one node and a task in a process on another node
- a task in the kernel and a task in a process on a single node
- a task in one process and a task in another process on a single node

and so on.

The scope of message channel communication can be configured to limit server access to:

- one memory space on a node (either the kernel or one process)
- all memory spaces on a node (the kernel and all processes)
- a cluster of nodes in a system (including all memory spaces in each node)

Message channels can also be used for multi-node communication between multiple clusters.

18.2 Message Channel Facilities

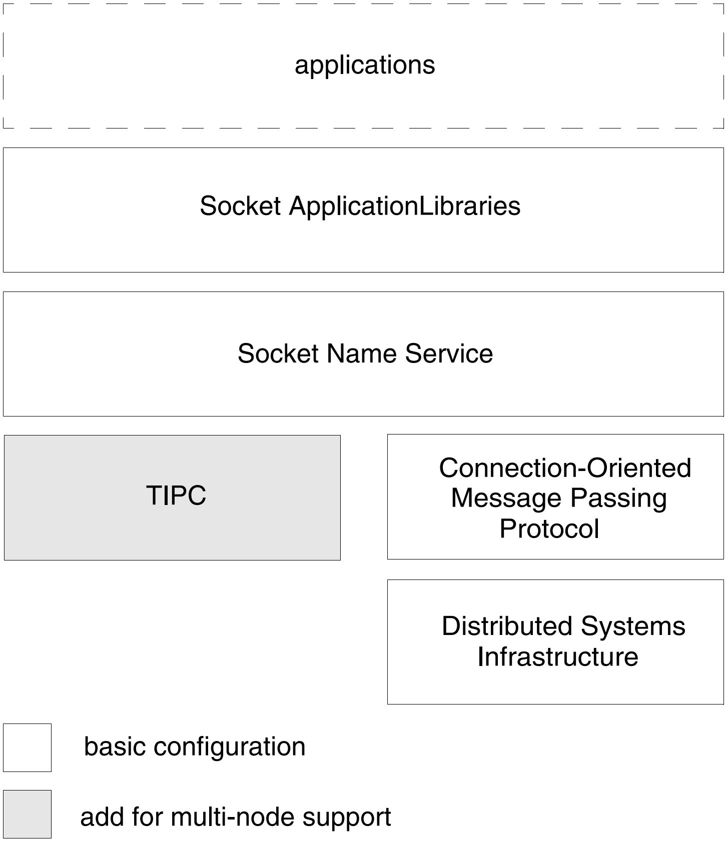
The message channel technology consists of the following basic facilities:

- Transparent Inter-Process Communication (TIPC) for multi-node communication. See *18.3 Multi-Node Communication with TIPC*, p.791.
- Connection-Oriented Message Passing (COMP) protocol with the DSI socket back end for single node communication. See *18.4 Single-Node Communication with COMP and DSI*, p.791. Note that COMP is also provides services for multi-node communication.
- Socket Name Service (SNS), which provides location and interface transparency for message channel communication between tasks on a single node, and maintains communication between nodes for multi-node message channel communications. In addition, it controls the scope of message channel communication (to two memory spaces, a node, a cluster of nodes, or a zone). See *18.5 Socket Name Service*, p.795.
- Socket Application Libraries (SAL), which provide APIs for using message channels in both client and server applications, as well as the mechanism for registering the tasks that are using a message channel with a Socket Name Service. See *18.6 Socket Application Libraries*, p.799.

Client and server applications that make use of SAL can be developed as either kernel applications or RTP applications (or both).

Figure 18-1 illustrates the architecture of the message channel facilities.

Figure 18-1 **Message Channel Architecture**


applications
Socket ApplicationLibraries
Socket Name Service
TIPC
Connection-Oriented Message Passing Protocol
Distributed Systems Infrastructure
basic configuration
add for multi-node support

The basic configuration of VxWorks with support for message channels includes SAL, SNS, COMP, and DSI—which provides support for single node use. TIPC must be added to the basic configuration for multi-node use. TIPC can also be added to provide connection-less socket types on a single node (for use outside of message channels).

For detailed information about configuration, see *18.7 Configuring VxWorks for Message Channels*, p.803. For information about TIPC, see the *Wind River TIPC Programmer's Guide*.

18.3 Multi-Node Communication with TIPC

The underlying transport mechanism for multi-node message channels is based on the Transparent Inter-Process Communication (TIPC) protocol, which provides a fast method for transferring messages across node boundaries in a cluster environment, or across cluster boundaries in a zone environment. TIPC can also be used within a single node.

TIPC is designed for use with the standard socket API. For connection-oriented messaging, the socket type associated with message channels is the **SOCK_SEQPACKET**. TIPC also supports **SOCK_RDM**, **SOCK_DGRAM**, and **SOCK_STREAM**, but those socket types are not supported by the message channel facility as only **SOCK_SEQPACKET** is supported by COMP. This simplifies migration between single-node and multi-node applications of message channels.

The TIPC protocol is connection-based, like other stream-based protocols such as TCP, but it carries variable-sized messages, like datagram-based protocols such as UDP. In providing cluster and node based communications, TIPC sockets are available in the **AF_TIPC** domain. TIPC provides several means of identifying end points that are handled transparently through the SNS name server. For more information about TIPC, see the *Wind River TIPC for VxWorks 6 Programmer's Guide*.

18.4 Single-Node Communication with COMP and DSI

The Connection-Oriented Message Passing protocol (COMP) provides services for multi-node as well as the protocol for single-node communication. The underlying transport mechanism for single-node message channels is based on the COMP protocol, which provides a fast method for transferring messages across memory boundaries on a single node.

COMP, using the **AF_LOCAL** family, is designed for use with the standard socket API. Because it provides connection-oriented messaging, the socket type associated with message channels is the **SOCK_SEQPACKET**. The protocol is connection-based, like other stream-based protocols such as TCP, but it carries variable-sized messages, like datagram-based protocols such as UDP.

While COMP provides for standard socket support, it has no dependency on TCP/IP networking facilities, which can be left out of a system if the facilities are not otherwise needed.

In providing single-node local communications, COMP sockets are available as part of the **AF_LOCAL** domain. Although this domain is traditionally related to the UNIX file system, in VxWorks the addressing is completely independent of any file system. Like UNIX sockets, COMP uses a string to define the address, and it has a structure similar to a file path name, but this is the extent of the similarity in this regard. The address is simply a logical representation of the end-point.

The transfer of data in message channels is based on an internal buffer management implementation that allows for deterministic memory allocation, which reduces the amount of copies needed to transfer the data whenever possible. Only one copy is needed for the internal transfer; the data coming from the user is directly moved into the receiver buffer space. Another copy is required to submit and retrieve the data to and from the channel.

COMP supports the standard socket options, such as **SO_SNDBUF** or **SO_RECVBUF** and **SO_SNDTIMEO** and **SO_RCVTIME**. For information about the socket options, see the socket API references. For information about how COMP uses them, see *installDir***/vxworks-6.***x***/target/src/dsi/backend/dsiSockLib.c**.

Express Messaging

Express messaging is also available for sending and receiving a message. An express message is placed on a special queue on the sending side and placed at the front of the normal queue at the receiving end. This allows for urgent messages to be sent and received with a higher priority than the normal messages. In order to send an express message, the *flags* parameter of the standard **send( )** routine must have the **MSG_EXP** bit set. (Also see the **socket send( )** API reference).

Show Routines

Because COMP is based on the standard socket API, traditional network show routines can be used, such as **netstat( )**. In addition, information on local sockets can be retrieved with the **unstatShow( )** routine (for more information, see the VxWorks API reference entry).

18.4.1 COMP Socket Support with DSI

The COMP socket functional interface is provided by the Distributed Systems Infrastructure (DSI) back end.

The DSI back end is a simplified version of the BSD socket back-end. It is designed for optimized communications when both end points are in a single node. The DSI back end provides the set of implementations of the standard socket functions for the COMP protocol specific calls. The traditional network protocols in VxWorks, such as TCP and UDP, use the BSD Internet Domain Socket back end and are described in the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

The DSI back end requires its own system and data memory pools, which are used to handle the creation of sockets and the data transfers between two endpoints. The pools are similar to those required for the network stack. In addition, the pools are configured so as to enhance performance for the local transfers. The system pool provides COMP with the memory it needs for its internal structures and data types. The data pool provides COMP with the memory it needs for receiving data. Because COMP is local, data transfer has been optimized so that data are put directly in the receiver's packet queue.

Both the DSI back end and DSI memory pools complement the BSD equivalent. Therefore, both BSD and DSI sockets can coexist in the system. They do not depend on each other, so that they can be added or removed, as needed.

COMP uses **netBufLib** to manage its internal system and data memory pools. For detailed information on how buffers are configured, see the coverage of the similar technology, **netBufPool**, in the *Wind River Network Stack for VxWorks 6 Programmer's Guide*.

These pools are created automatically by the **INCLUDE_DSI_POOL** component. The DSI parameters listed in Table 18-1 are used for memory pool configuration. These parameters are used when **usrNetDsiPoolConfig( )** routine is called, which happens automatically when the system boots. The **dsiSysPoolShow( )** and **dsiDataPoolShow( )** can be used to display related information (see the VxWorks API reference for **dsiSockLib**).

Table 18-1 **INCLUDE_DSI_POOL Component Parameters**

Parameter	Default Value
DSI_NUM_SOCKETS	200
DSI_DATA_32	50
DSI_DATA_64	100
DSI_DATA_128	200
DSI_DATA_256	40
DSI_DATA_512	40
DSI_DATA_1K	10
DSI_DATA_2K	10
DSI_DATA_4K	10
DSI_DATA_8K	10
DSI_DATA_16K	4
DSI_DATA_32K	0
DSI_DATA_64K	0

The DSI pool is configured more strictly and more efficiently than the core network pool since it is more contained, fewer scenarios are possible, and everything is known in advance (as there is only the one node involved). The **DSI_NUM_SOCKETS** parameter controls the size of the system pool. It controls the number of clusters needed to fit a socket, for each family and each protocol supported by the back end. Currently, only the **AF_LOCAL** address family is supported by COMP.

The clusters allocated in the back end are of these sizes:

- **aligned sizeof (struct socket)**
- **aligned sizeof (struct uncompcb)**
- **aligned sizeof (struct sockaddr_un)**

One cluster of size 328 and of size 36 are needed for each socket that is created since currently, the COMP protocol is always linked to a DSI socket. Only one cluster of **sizeof (struct sockaddr_un)** is required, therefore the size of the system pool is basically determined by: (**DSI_NUM_SOCKETS** * (328 + 36) + 108)). Using these sizes prevents any loss of space since they are the actual sizes needed.

All other parameters for the DSI pool are used to calculate the size of clusters in the data pool, and at the same time, the size of the pool itself. The data pool is used as packet holders during the transmissions between two sockets, between the time the data is copied from the sender's buffer to the receiver's buffer. Each of them represent a cluster size from 32 bytes to 64 kilobytes and the number of allocated clusters of that specific size.

To set reasonable values for the parameters in this component, you must know how much memory your deployed application will require. There is no simple formula that you can use to anticipate memory usage. Your only real option is to determine memory usage empirically. This means running your application under control of the debugger, pausing the application at critical points in its execution, and monitoring the state of the memory pool. You will need to perform these tests under both stressed and unstressed conditions.

18.5 Socket Name Service

A Socket Name Service (SNS) allows a server application to associate a service name with a collection of listening sockets, as well as to limit the visibility of the service name to a restricted (but not arbitrary) set of clients.

Both Socket Application Library (SAL) client and server routines make use of an SNS server to establish a connection to a specified service without the client having to be aware of the address of the server's listening sockets, or the exact interface type being utilized (see *18.6 Socket Application Libraries*, p.799). This provides both location transparency and interface transparency. Such transparency makes it possible to design client and server applications that can operate efficiently without requiring any knowledge of the system's topology.

An SNS server is a simple database that provides an easy mapping of service names and their associated sockets. The service name has this URL format:

`[SNS:]`*service_name*`[@`*scope*`]`

The **[SNS:]** prefix is the only prefix accepted, and it can be omitted. The scope can have the following values: **private**, **node**, **cluster**, or **system**. These values designate an access scope for limiting access to the same single memory space (the kernel or a process), the same node (the kernel and all processes on that node), a set of nodes, or the entire system, respectively. A server can be accessed by clients within the scope that is defined when the server is created with the **salCreate( )** routine (see *18.6.1 SAL Server Library*, p.800).

NOTE: The SNS server creates a COMP socket for its own use for local communication. It has the socket address of 0x0405. All of the SAL routines send messages to the SNS server at this socket address.

SNS provides a resource reclamation mechanism for servers created within processes. If a process dies before **salDelete( )** has been called on a SAL server, SNS will be notified and will remove the entry from the database. Note, however, that this mechanism is not available for tasks in the kernel. If a task in the kernel terminates before **salDelete( )** is called, the service name is not automatically removed from SNS. In order to avoid stale entries that may prevent new services with the same name from being created, the **salRemove( )** routine should be used.

The SNS server can be configured to run in either kernel or user space (as a kernel application or an RTP application, respectively). For more information, see *Running SNS as a Process*, p.804).

A node should not be configured with more than one SNS server. The server starts at boot time, and is named **tSnsServer** if it is running in the kernel, or **iSnsServer** if it is running as a process. For a multi-node system, a monitoring task is automatically spawned to maintain a list of all the SNS servers in the zone (system). The monitoring task is named **tDsalMonitor**, and it runs in the kernel.

The **snsShow( )** command allows a user to verify that SAL-based services are correctly registered with the SNS server from the shell (see *18.5.2 snsShow( ) Example*, p.797).

For more information, see theVxWorks API reference for **snsLib**.

18.5.1 Multi-Node Socket Name Service

For a multi-node system, each node in the system must be configured with the Socket Name Service (SNS). Note that VxWorks SNS components for multi-node use are different from those used on single node systems (see *18.7 Configuring VxWorks for Message Channels*, p.803).

When a distributed SNS server starts on a node at boot time, it uses a TIPC bind operation to publish a TIPC port name. This is visible to all other nodes in the zone. The other existing SNS servers then register the node in their tables of SNS servers.

A separate monitoring task (called **tDsalMonitor**) is started on each node at boot time, which uses the TIPC subscription feature to detect topology-change events such as a new SNS server coming online, or an existing SNS server leaving the zone (system).

Note that if the TIPC networking layer does not start up properly at boot time, the distributed SAL system will not initialize itself correctly with TIPC, and the SNS server will work strictly in local mode. The SNS server does not check for a working TIPC layer after the system boots, so that it will not detect the layer if it is subsequently started manually, and the SNS server will continue to run in local mode.

When a new node appears, each SNS server sends a command to that node requesting a full listing of all sockets that are remotely accessible. The SNS server on the new node sends a list of sockets that can be reached remotely.

Each time a new socket is created with **salCreate( )** on a node that has a server scope greater than **node**, this information is sent to all known SNS servers in the cluster. All SNS servers are thereby kept up to date with relevant information. Similarly, when a socket is deleted using the **salRemove( )** function, this information is sent to all known SNS servers in the cluster. The addition and removal of sockets is an infrequent occurrence in most anticipated uses and should be of minimal impact on network traffic and on the performance of the node.

When the **tDsalMonitor** task detects that an SNS server has been withdrawn from the system, the local SNS server purges all entries related to the node that is no longer a part of the distributed SNS system.

Note that only information on accessible sockets is transmitted to remote SNS servers. While it is acceptable to create an **AF_LOCAL** socket with **cluster** scope, this socket will use the COMP protocol which can only be accessed locally. SNS servers on remote nodes will not be informed of the existence of this socket.

On a local node, if a socket name exists in the SNS database in both the **AF_LOCAL** and **AF_TIPC** families, when a connection is made to that name using **salOpen( )**, the **AF_LOCAL** socket will be used.

18.5.2 snsShow( ) Example

The **snsShow( )** shell command provides information about all sockets that are accessible from the local node, whether the sockets are local or remote. The command is provided by the VxWorks **INCLUDE_SNS_SHOW** component.

The following examples illustrate **snsShow( )** output from three different nodes in a system.

From Node <1.1.22>

```
            NAME            SCOPE FAMILY  TYPE   PROTO          ADDR
-------------------------- ----- ------ ------- ----- -------------------
astronaut_display          clust LOCAL  SEQPKT  0     /comp/socket/0x5
                                 TIPC   SEQPKT  0     <1.1.22>,1086717967
ground_control_timestamp   clust TIPC   SEQPKT  0   * <1.1.25>,1086717965
ground_control_weblog      systm TIPC   SEQPKT  0   * <1.1.25>,1086717961
heartbeat_private          priv  LOCAL  SEQPKT  0     /comp/socket/0x4
                                 TIPC   SEQPKT  0     <1.1.22>,1086717966
local_temperature          node  LOCAL  SEQPKT  0     /comp/socket/0x2
newsfeed                   clust TIPC   SEQPKT  0   * <1.1.50>,1086717962
rocket_diagnostic_port     clust TIPC   SEQPKT  0     <1.1.22>,1086717964
rocket_propellant_fuel_level_interface
           ----            clust TIPC   SEQPKT  0     <1.1.22>,1086717960
spacestation_docking_port  clust TIPC   SEQPKT  0   * <1.1.55>,1086717963
```

From Node <1.1.25>

```
            NAME            SCOPE FAMILY  TYPE   PROTO          ADDR
-------------------------- ----- ------ ------- ----- -------------------
astronaut_display          clust TIPC   SEQPKT  0   * <1.1.22>,1086717967
ground_control_timestamp   clust LOCAL  SEQPKT  0     /comp/socket/0x3
                                 TIPC   SEQPKT  0     <1.1.25>,1086717965
ground_control_weblog      systm TIPC   SEQPKT  0     <1.1.25>,1086717961
local_billboard            node  LOCAL  SEQPKT  0     /comp/socket/0x2
                                 TIPC   SEQPKT  0     <1.1.25>,1086717964
newsfeed                   clust TIPC   SEQPKT  0   * <1.1.50>,1086717962
rocket_diagnostic_port     clust TIPC   SEQPKT  0   * <1.1.22>,1086717964
rocket_propellant_fuel_level_interface
           ----            clust TIPC   SEQPKT  0   * <1.1.22>,1086717960
spacestation_docking_port  clust TIPC   SEQPKT  0   * <1.1.55>,1086717963
```

From Node <1.1.55>

```
            NAME            SCOPE FAMILY  TYPE   PROTO          ADDR
-------------------------- ----- ------ ------- ----- -------------------
astronaut_display          clust TIPC   SEQPKT  0   * <1.1.22>,1086717967
ground_control_timestamp   clust TIPC   SEQPKT  0   * <1.1.25>,1086717965
ground_control_weblog      systm TIPC   SEQPKT  0   * <1.1.25>,1086717961
newsfeed                   clust TIPC   SEQPKT  0   * <1.1.50>,1086717962
rocket_diagnostic_port     clust TIPC   SEQPKT  0   * <1.1.22>,1086717964
rocket_propellant_fuel_level_interface
           ----            clust TIPC   SEQPKT  0   * <1.1.22>,1086717960
spacestation_docking_port  clust LOCAL  SEQPKT  0     /comp/socket/0x2
                                 TIPC   SEQPKT  0     <1.1.55>,1086717963
```

The output of the **snsShow( )** command is fairly self-explanatory. The first field is the name of the socket. If the name is longer than the space allocated in the output, the entire name is printed and the other information is presented on the next line with the name field containing several dashes.

The scope values are **priv** for private, **node** for node, **clust** for cluster, and **systm** for system.

The family types can be **TIPC** for **AF_TIPC** or **LOCAL** for **AF_LOCAL**.

The socket type can be **SEQPKT** for **SOCK_SEQPACKET**, **RDM**.

The protocol field displays a numeric value and a location indicator. The numeric value is reserved for future use, and currently only zero is displayed. The final character in the field indicates whether the socket was created on a remote or local node, with an asterisk (*) designating remote.

The address field indicates the address of the socket. All addresses of the form **/comp/socket** belong to the **AF_LOCAL** family. All addresses of the form **<x.y.z>,refID** belong to the **AF_TIPC** family. The TIPC address gives the TIPC **portID** which consists of the **nodeID** and the unique reference number.

18.6 Socket Application Libraries

The Socket Application Libraries (SAL) simplify creation of both server and client applications by providing routines to facilitate use of the sockets API.

SAL also provides an infrastructure for the development of location-transparent and interface-transparent applications. By allowing SAL to handle the basic housekeeping associated with a socket-based application, developers can focus on the application-specific portions of their designs. Developers are free to use the complete range of SAL capabilities in their applications, or just the subset that suits their needs; they can even bypass SAL entirely and develop a socket-based application using nothing but custom software. The SAL client and server APIs can be used in both kernel and user space.

Several VxWorks components are available to provide SAL support in different memory spaces, for single or multi-node systems, and so on (see *18.7 Configuring VxWorks for Message Channels*, p.803).

SAL-based applications can also utilize the Socket Name Service (SNS), which allows a client application to establish communication with a server application without having to know the socket addresses used by the server (see *18.5 Socket Name Service*, p.795).

CAUTION: SAL applications should not use any of the following as part of a name:

```
* ? @ : # / < > % | [ ] { } , \\ \ ' & ; = + $
```

In addition, these should not be used when attempting to find a service:

```
@ : # / < > % | [ ] { } , \\ \ ' & ; = + $
```

18.6.1 SAL Server Library

The SAL server routines provide the infrastructure for implementing a socket-based server application. The SAL server allows a server application to provide service to any number of client applications. A server application normally utilizes a single SAL server in its main task, but is free to spawn additional tasks to handle the processing for individual clients if parallel processing of client requests is required. The SAL server library is made of the following routines:

salCreate( )
Creates a named socket-based server.

salDelete( )
Deletes a named socket-based server.

salServerRtnSet( )
Configures the processing routine with the SAL server.

salRun( )
Activates a socket-based server.

salRemove( )
Removes a service from the SNS by name.

A server application typically calls **salCreate( )** to configure a SAL server with one or more sockets that are then automatically registered with SNS under a specified service identifier. The number of sockets created depends on which address families, socket types, and socket protocols are specified by the server application. **AF_LOCAL** and **AF_TIPC** sockets are supported.

If the address family specified is **AF_UNSPEC**, the system attempts to create sockets in all of the supported address families (**AF_LOCAL** and **AF_TIPC**). The socket addresses used for the server's sockets are selected automatically, and cannot be specified by the server application with **salCreate( )**.

A server can be accessed by clients within the scope that is defined when the server is created with the **salCreate( )** routine.

The scope is identified as part of the first parameter, with one the following values: **private**, **node**, **cluster**, or **system**. These values designate an access scope for limiting access to the same task (kernel or process), the same node (the kernel and all processes on that node), and a set of nodes, respectively. The second parameter identifies the protocol (with 1 being COMP and 33 being TIPC; 0 is used for all supported families). The third parameter identifies the socket type.

For example, the following call would create a socket named **foo** with **cluster** scope, with the COMP protocol:

```
salCreate("foo@cluster",1,5)
```

⚠ **CAUTION:** A COMP (single node) socket can be created with cluster or system scope, but this setting has no effect in a multi-node system. That is, in a multi-node system, SNS will not transmit this information to other nodes because a COMP socket is only available on the node on which it was created.

NOTE: It is possible to create both a COMP socket and a TIPC socket with the same name. Only the TIPC socket information is sent to other nodes in a multi-node system (assuming the scope is set appropriately).

Once created, a SAL server must be configured with one or more processing routines before it is activated. These routines can be configured by calling **salServerRtnSet( )**.

Once the server is ready, **salRun( )** is called to start the server activities. The **salRun( )** routine never returns unless there is an error or one of the server processing routines requests it. You must call **salDelete( )** to delete the server and its sockets regardless of whether or not the routine has terminated. This is accomplished with **salDelete( )**. This routine can be called only by tasks in the process (or the kernel) where the server was created. In order for tasks outside the process to remove a service name from SNS, **salRemove( )** must be used. The **salRemove( )** routine does not close sockets, nor does it delete the server. It only deletes the SNS entry, and therefore access to any potential clients.

For more information, including sample service code, see the VxWorks API reference for the **salServer** library.

18.6.2 SAL Client Library

The SAL client library provides a simple means for implementing a socket-based client application. The data structures and routines provided by SAL allow the application to easily communicate with socket-based server applications that are registered with the Socket Name Service (see *18.5 Socket Name Service*, p.795). Additional routines can be used to communicate with server applications that are not registered with the SNS. The SAL client library is made of the following routines:

salOpen( )
Establishes communication with a named socket-based server.

salSocketFind( )
Finds sockets for a named socket-based server.

salNameFind( )
Finds services with the specified name.

salCall( )
Invokes a socket-based server.

A client application typically calls **salOpen( )** to create a client socket and connect it to the named server application. The client application can then communicate with the server by passing the socket descriptor to standard socket API routines, such as **send( )** and **recv( )**.

As an alternative, the client application can perform a **send( )** and **recv( )** as a single operation using **salCall( )**. When the client application no longer needs to communicate with a server it calls the standard socket **close( )** routine to close the socket to the server.

A client socket can be shared between two or more tasks. In this case, however, special care must be taken to ensure that a reply returned by the server application is handled by the correct task.

The **salNameFind( )** and **salSocketFind( )** routines facilitate the search of the server and provide more flexibility for the client application.

The **salNameFind( )** routine provides a lookup mechanism for services based on pattern matching, which can be used with (multiple) wild cards to locate similar names. For example, if the names are **foo**, **foo2**, and **foobar**, then a search using **foo*** would return them all. The scope of the search can also be specified. For example, a client might want to find any server up to a given scope, or only within a given scope. In the former case the **upto_** prefix can be added to the scope specification. For example, **upto_node** defines a search that look for services in all processes and in the kernel in a node.

Once a service is found, the **salSocketFind( )** routine can be used to return the proper socket ID. This can be useful if the service has multiple sockets, and the client requires use of a specific one. This routine can also be used with wild cards, in which case the first matching server socket is returned.

For more information, including sample client code, see the VxWorks API reference for the **salClient** library.

18.7 Configuring VxWorks for Message Channels

To provide the basic set of message channel facilities in a system, VxWorks must be configured with COMP, DSI, SAL, and SNS components. Selection of SNS components determines whether the system supports single-node or multi-node communication.

COMP, DSI, and SAL Components

The required COMP, DSI, and SAL components are as follows:

- **INCLUDE_UN_COMP**
- **INCLUDE_DSI_POOL**
- **INCLUDE_DSI_SOCKET**
- **INCLUDE_SAL_SERVER**
- **INCLUDE_SAL_CLIENT**

Note that **INCLUDE_UN_COMP** is required for both single and multi-node systems, as it provides support for communication between SAL and SNS. While COMP provides for standard socket support, it has no dependency on TCP/IP networking facilities, which can be left out of a system if they are not otherwise needed.

SNS Component Options

In addition to the COMP, DSI, and SAL components, one of the four following components listed below is required for SNS support.

Single Node Options

- **INCLUDE_SNS** to run SNS as a kernel daemon, for single node communication.
- **INCLUDE_SNS_RTP** to start SNS as a process automatically at boot time, for single node communication.

Multi-Node Options

- **INCLUDE_SNS_MP** to run SNS as a kernel daemon, supporting distributed named sockets for multi-node communication.
- **INCLUDE_SNS_MP_RTP** to start SNS as a process (RTP) automatically at boot time, supporting distributed named sockets for multi-node communication. Additional system configuration is required to run SNS as a process; for information in this regard, see *Running SNS as a Process*, p.804.

Note that including a distributed SNS server automatically includes TIPC.

Running SNS as a Process

In order to run SNS as a process (RTP), the developer must also build the server, add it to ROMFS, configure VxWorks with ROMFS support, and then rebuild the entire system:

a. Build *installDir***/vxworks-6.***x***/target/usr/apps/dsi/snsd/snsd.c** (using the makefile in the same directory) to create **snsServer.vxe**.

b. Copy **snsServer.vxe** to the ROMFS directory (creating the directory first, if necessary.

   The **INCLUDE_SNS_RTP** and **INCLUDE_SNS_MP_RTP** components must know the location of the server in order to start it at boot time. They expect to find the server in the ROMFS directory. If you wish to store the server somewhere else (in another file system to reduce the VxWorks image size, for example) use the **SNS_PATHNAME** parameter to identify the location.

c. Configure VxWorks with the ROMFS component.

d. Rebuild VxWorks.

These steps can also be performed with Wind River Workbench (see the *Wind River Workbench User's Guide*). For information about ROMFS, see *8.8 Read-Only Memory File System: ROMFS*, p.516.

SNS Configuration Parameters

The following SNS component parameters can usually be used without modification:

SNS_LISTEN_BACKLOG
This parameter defines the number of outstanding service requests that the SNS server can track on the socket that it uses to service SNS requests from SAL routines. The default value is 5. The value may be increased if some SAL requests are not processed on a busy system.

SNS_DISTRIBUTED_SERVER_TYPE and **SNS_DISTRIBUTED_SERVER_INSTANCE**
These parameters are used in the multi-node configuration of SNS servers to define the TIPC port name that all SNS servers use. The default is type 51 and instance 51 in the TIPC name tables. If this type and instance conflict with other usages in the network, they can be changed to values that are unique for the network. Note that it is recommended to use a type of 50 or above (types 0 through 7 are reserved by TIPC).

⚠ **CAUTION:** It is recommended that you do not change the default values of the **SNS_PRIORITY** and **SNS_STACK_SIZE** parameters. The default for **SNS_PRIORITY** is 50 and the default for **SNS_STACK_SIZE** is 20000.

Show Routines

The show routines related to COMP can be included by adding the **INCLUDE_UN_COMP_SHOW** component. The **snsShow( )**routine is included with the **INCLUDE_SNS_SHOW** component. In order to use **netstat( )** the network show routines must be included. Note that this will force the inclusion of networking components.

For information about processes and applications, see *VxWorks Application Programmer's Guide: Applications and Processes*.

18.8 Comparison of Message Channels and Message Queues

In addition to providing a superior alternative to TCP for multi-node intercommunication, message channels provide an alternative to message queues for exchanging data between two tasks on a single node. Both message channels and message queues allow multiple tasks to send and receive from the same channel. The main differences between these facilities are:

- Message channels can be used to communicate between nodes in a multi-node cluster, but message queues cannot.

- Implementation of message channels in a single-node system facilitates porting an application to a multi-node system.

- Message channels are slower than message queues because of the overhead of the socket back end. While message channels are notably slower in a single memory space (such as within the kernel), they are only slightly slower when communicating across a memory barrier (such as between processes). Note that message channels are much faster than TCP for inter-node communication. Message channels are connection-oriented while message queues are not. There is no way to establish a connection between two tasks with message queues. In a connection-oriented communication, the two end-points are aware of each other, and if one leaves the other eventually finds out. By way of analogy, a connection-oriented communication is like a telephone call, whereas a connection-less communication is like sending a letter. Both models are valid, and the requirements of the application should determine their use.

  Each message queue is unidirectional. In order to establish a bidirectional communication, two queues are needed, one for each end-point (see Figure 4-13). Each message channel is bidirectional and data can be sent from both end-points at any time. That is, each message channel provides connection-oriented full-duplex communication.

- The messages communicated by message channels can be of variable size, whereas those communicated by message queues have a maximum size that is defined when the queue is created. Message channels therefore allow for a better utilization of system resources by using exactly what is needed for the message, and nothing more.

- Message queues have a fixed capacity. Only a pre-defined number of messages can be in a queue at any one time. Message channels, on the other hand, have a flexible capacity. There is no limit to the number of messages that a message channel can handle.

- Message channels provide location transparency. An endpoint can be referred to by a name, that is by a simple string of characters (but a specific address can also be used). Message queues only provide location transparency for inter-process communication when they are created as public objects.
- Message channels provide a simple interface for implementing a client/server paradigm. A location transparent connection can be established by using two simple calls, one for the client and one for the server. Message queues do not provide support for client/server applications.
- Message channels use the standard socket interface and support the **select( )** routine; message queues do not.
- Message channels cannot be used with VxWorks events; message queues can.
- Message queues can be used within an ISR, albeit only the **msgQsend( )** routine. No message channel routines can be used within an ISR.
- Message queues are based entirely on a proprietary API and are therefore more difficult to port to a different operating systems than message channels, which are based primarily on the standard socket API.

Message channels are better suited to applications that are based on a client/server paradigm and for which location transparency is important.

Index