WIND RIVER

VxWorks®

APPLICATION PROGRAMMER'S GUIDE

6.8

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

Toll free (U.S.A.): 800-545-WIND
Telephone: 510-748-4100
Facsimile: 510-749-2010

For additional contact information, see the Wind River Web site:

www.windriver.com

For information on how to contact Customer Support, see:

www.windriver.com/support

VxWorks
Application Programmer's Guide
6.8

28 Oct 09

Contents

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. It covers the following topics:

- real-time processes (RTPs)
- RTP applications
- Static Libraries, Shared Libraries, and Plug-Ins
- C++ development
- multitasking facilities
- POSIX facilities
- memory management
- I/O system
- local file systems
- error detection and reporting

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

1.2 Related Documentation Resources

The companion volume to this book, the *VxWorks Kernel Programmer's Guide*, provides material specific to kernel features and kernel-based development.

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 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 by Example* 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 by Example* 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
Real-Time Processes

2.1 Introduction

VxWorks real-time processes (RTPs) are in many respects similar to processes in other operating systems—such as UNIX and Linux—including extensive POSIX compliance.[1] The ways in which they are created, execute applications, and terminate will be familiar to developers who understand the UNIX process model.

The VxWorks process model is, however, designed for use with real-time embedded systems. The features that support this model include system-wide scheduling of tasks (processes themselves are not scheduled), preemption of processes in kernel mode as well as user mode, process-creation in two steps to separate loading from instantiation, and loading applications in their entirety.

VxWorks real-time processes provide the means for executing applications in user mode. Each process has its own address space, which contains the executable program, the program's data, stacks for each task, the heap, and resources associated with the management of the process itself (such as memory-allocation tracking). Many processes may be present in memory at once, and each process may contain more than one task (sometimes known as a *thread* in other operating systems).

VxWorks processes can operate with two different virtual memory models: flat (the default) or overlapped (optional). With the flat virtual-memory model each

[1] VxWorks can be configured to provide POSIX PSE52 support for individual processes.

VxWorks process has its own region of virtual memory described by a unique range of addresses. This model provides advantages in execution speed, in a programming model that accommodates systems with and without an MMU, and in debugging applications. With overlapped virtual-memory model, each VxWorks process uses the same range of virtual addresses for the area in which its code (text, data, and bss segments) resides. This model provides more precise control over the virtual memory space and allows for notably faster application load time.

For information about developing RTP applications, see *3. RTP Applications*.

2.2 About Real-time Processes

A common definition of a process is "a program in execution," and VxWorks processes are no different in this respect. In fact, the life-cycle of VxWorks real-time processes is largely consistent with the POSIX process model (see *2.2.9 RTPs and POSIX*, p.11).

VxWorks processes, however, are called real-time processes (RTPs) precisely because they are designed to support the determinism required of real-time systems. They do so in the following ways:

- The VxWorks task-scheduling model is maintained. Processes are not scheduled—tasks are scheduled globally throughout the system.
- Processes can be preempted in kernel mode as well as in user mode. Every task has both a user mode and a kernel mode stack. (The VxWorks kernel is fully preemptive.)
- Processes are created without the overhead of performing a copy of the address space for the new process and then performing an *exec* operation to load the file. With VxWorks, a new address space is simply created and the file loaded.
- Process creation takes place in two phases that clearly separate instantiation of the process from loading and executing the application. The first phase is performed in the context of the task that calls **rtpSpawn( )**. The second phase is carried out by a separate task that bears the cost of loading the application text and data before executing it, and which operates at its own priority level distinct from the parent task. The parent task, which called **rtpSpawn( )**, is not impacted and does not have wait for the application to begin execution, unless it has been coded to wait.
- Processes load applications in their entirety—there is no demand paging.

All of these differences are designed to make VxWorks particularly suitable for hard real-time applications by ensuring determinism, as well as providing a common programming model for systems that run with an MMU and those that do not. As a result, there are differences between the VxWorks process model and that of server-style operating systems such as UNIX and Linux. The reasons for these differences are discussed as the relevant topic arises throughout this chapter.

2.2.1 RTPs and Scheduling

The primary way in which VxWorks processes support determinism is that they themselves are simply not scheduled. Only tasks are scheduled in VxWorks systems, using a priority-based, preemptive policy. Based on the strong preemptibility of the VxWorks kernel, this ensures that at any given time, the highest priority task in the system that is ready to run will execute, regardless of whether the task is in the kernel or in any process in the system.

By way of contrast, the scheduling policy for non-real-time systems is based on time-sharing, as well as a dynamic determination of process priority that ensures that no process is denied use of the CPU for too long, and that no process monopolizes the CPU.

VxWorks does provide an optional time-sharing capability—round-robin scheduling—but it does not interfere with priority-based preemption, and is therefore deterministic. VxWorks round-robin scheduling simply ensures that when there is more than one task with the highest priority ready to run at the same time, the CPU is shared between those tasks. No one of them, therefore, can usurp the processor until it is blocked.

For more information about VxWorks scheduling see *6.3 Task Scheduling*, p.88.

2.2.2 RTP Creation

The manner in which real-time processes are created supports the determinism required of real-time systems. The creation of an RTP takes place in two distinct phases, and the executable is loaded in its entirety when the process is created. In the first phase, the **rtpSpawn( )** call creates the process object in the system, allocates virtual and physical memory to it, and creates the initial process task (see *2.2.5 RTPs and Tasks*, p.8). In the second phase, the initial process task loads the entire executable and starts the main routine.

This approach provides for system determinism in two ways:

- First, the work of process creation is divided between the **rtpSpawn( )** task and the initial process task—each of which has its own distinct task priority. This means that the activity of loading applications does not occur at the priority, or with the CPU time, of the task requesting the creation of the new process. Therefore, the initial phase of starting a process is discrete and deterministic, regardless of the application that is going to run in it. And for the second phase, the developer can assign the task priority appropriate to the significance of the application, or to take into account necessarily indeterministic constraints on loading the application (for example, if the application is loaded from networked host system, or local disk). The application is loaded with the same task priority as the priority with which it will run. In a way, this model is analogous to asynchronous I/O, as the task that calls **rtpSpawn( )** just initiates starting the process and can concurrently perform other activities while the application is being loaded and started.
- Second, the entire application executable is loaded when the process is created, which means that the determinacy of its execution is not compromised by incremental loading during execution. This feature is obviously useful when systems are configured to start applications automatically at boot time—all executables are fully loaded and ready to execute when the system comes up.

The **rtpSpawn( )** routine has an option that provides for synchronizing for the successful loading and instantiation of the new process.

At startup time, the resources internally required for the process (such as the heap) are allocated on demand. The application's text is guaranteed to be write-protected, and the application's data readable and writable, as long as an MMU is present and the operating system is configured to manage it. While memory protection is provided by MMU-enforced partitions between processes, there is no mechanism to provide resource protection by limiting memory usage of processes to a specified amount. For more information, see *8. Memory Management*.

Note that creation of VxWorks processes involves no copying or sharing of the parent processes' page frames (copy-on-write), as is the case with some versions of UNIX and Linux. The flat virtual memory model provided by VxWorks prohibits this approach and the overlapped virtual memory model does not currently support this feature. For information about the issue of inheritance of attributes from parent processes, see *2.2.7 RTPs, Inheritance, Zombies, and Resource Reclamation*, p.9.

For information about what operations are possible on a process in each phase of its instantiation, see the VxWorks API reference for **rtpLib**. Also see *3.3.7 Using Hook Routines*, p.37.

VxWorks processes can be started in the following ways:

- interactively from the kernel shell
- interactively from the host shell and debugger
- automatically at boot time, using a startup facility
- programmatically from applications or the kernel

Form more information in this regard, see *3.6 Executing RTP Applications*, p.40.

2.2.3 RTP Termination

Processes are terminated under the following circumstances:

- When the last task in the process exits.
- If any task in the process calls **exit( )**, regardless of whether or not other tasks are running in the process.
- If the process' **main( )** routine returns.

  This is because **exit( )** is called implicitly when **main( )** returns. An application in which **main( )** spawns tasks can be written to avoid this behavior—and to allow its other tasks to continue operation—by including a **taskExit( )** call as the last statement in **main( )**. See *3.3 Developing RTP Applications*, p.30.
- If the **kill( )** routine is used to terminate the process.
- If **rtpDelete( )** is called on the process—from a program, a kernel module, the C interpreter of the shell, or from Workbench. Or if the **rtp delete** command is used from the shell's command interpreter.
- If a process takes an exception during its execution.

> This default behavior can be changed for debugging purposes. When the error detection and reporting facilities are included in the system, and they are set to debug mode, processes are not terminated when an exception occurs.

Note that if a process fails while a shell is running, a message is printed to the shell console. Error messages can be recorded with the VxWorks error detection and reporting facilities (see *11. Error Detection and Reporting*).

For information about attribute inheritance and what happens to a process' resources when it terminates, see *2.2.7 RTPs, Inheritance, Zombies, and Resource Reclamation*, p.9.

2.2.4 **RTPs and Memory**

Each process has its own address space, which contains the executable program, the program's data, stacks for each task, the heap, and resources associated with the management of the process itself (such as local heap management). Many processes may be present in memory at once.

Virtual Memory Models

VxWorks processes can operate with two different virtual memory models: flat (the default) or overlapped (optional).

The flat virtual-memory model provides advantages in execution speed, in a programming model that accommodates systems with and without an MMU, and in debugging applications. In this model each VxWorks process has its own region of virtual memory described by a unique range of addresses.

The overlapped virtual-memory model provides more precise control over the virtual memory space and allows for notably faster application load time. With this model, each VxWorks process uses the same range of virtual addresses for the area where its code (text, data, and bss segments) resides. The overlapped virtual-memory model will not work unless VxWorks is configured with MMU support and the MMU is turned on.

The two virtual memory models are mutually exclusive, and are described in more detail in *2.5 About VxWorks RTP Virtual Memory Models*, p.16.

Memory Protection

Each process is protected from any other process that is running on the system, whenever the target system has an MMU, and MMU support has been configured into VxWorks. Operations involving the code, data, and memory of a process are accessible only to code executing in that process. It is possible, therefore, to run several instances of the same application in separate processes without any undesired side effects occurring between them. The name and symbol spaces of the kernel and processes are isolated.

As processes run a fully linked image without external references, a process cannot call a routine in another process, or a kernel routine that is not exported as a system call—whether or not the MMU is enabled. However, if the MMU is not enabled, a process can read and write memory external to its own address space, and could cause the system to malfunction.

2.2.5 RTPs and Tasks

VxWorks can run many processes at once, and any number of processes can run the same application executable. That is, many instances of an application can be run concurrently.

For general information about tasks, see *6. Multitasking*.

Numbers of Tasks and RTPs

Each process can execute one or more tasks. When a process is created, the system spawns a single task to initiate execution of the application. The application may then spawn additional tasks to perform various functions. There is no limit to the number of tasks in a process, other than that imposed by the amount of available memory. Similarly, there is no limit to the number of processes in the system—but only for architectures that *do not* have (or do not use) a hardware mechanism that manages concurrent address spaces (this mechanism is usually known as an address space identifier, or ASID). For target architectures that do use ASIDs or equivalent mechanisms, the number of processes is limited to that of the ASID (usually 255). For more information, see the *VxWorks Architecture Supplement*.

Initial Task in an RTP

When a process is created, an initial task is spawned to begin execution of the application. The name of the process's initial task is based on the name of the executable file, with the following modifications:

- The letter **i** is prefixed.
- The first letter of the filename capitalized.
- The filename extension is removed.

For example, when **foobar.vxe** is run, the name of the initial task is **iFoobar**.

The initial task provides the execution context for the program's **main( )** routine, which it then calls. The application itself may then spawn additional tasks.

RTP Tasks and Memory

Task creation includes allocation of space for the task's stack from process memory. As needed, memory is automatically added to the process as tasks are created from the kernel free memory pool.

Heap management routines are available in user-level libraries for tasks in processes. These libraries provide the various ANSI APIs such as **malloc( )** and **free( )**. The kernel provides a pool of memory for each process in user space for these routines to manage.

Providing heap management in user space provides for speed and improved performance because the application does not incur the overhead of a system call for memory during its execution. However, if the heap is exhausted the system automatically allocates more memory for the process (by default), in which case a system call is made. Environment variables control whether or not the heap grows (see *8.3 Heap and Memory Partition Management*, p.208).

2.2.6 RTPs and Inter-Process Communication

While the address space of each process is invisible to tasks running in other processes, tasks can communicate across process boundaries through the use of various IPC mechanisms (including *public* semaphores, *public* message queues, and message channels) and shared data memory regions. See *6.8 Intertask and Interprocess Communication*, p.107 and *3.5 Creating and Using Shared Data Regions*, p.38 for more information.

2.2.7 RTPs, Inheritance, Zombies, and Resource Reclamation

VxWorks has a process hierarchy made up of parent/child relationships. Any process spawned from the kernel (whether programmatically, from the shell or other development tool, or by an automated startup facility) is a child of the kernel. Any process spawned by another process is a child of that process. As in human societies, these relationships are critical with regard to what characteristics children inherit from their parents, and what happens when a parent or child dies.

Inheritance

VxWorks processes inherit certain attributes of their parent. The child process inherits the file descriptors (stdin, stdout and stderr) of its parent process—which means that they can access the same files (if they are open), and signal masks. If the child process is started by the kernel, however, then the child process inherits only the three standard file descriptors. Environment variables are not inherited, but the parent can pass its environment, or a sub-set of it, to the child process (for information in this regard, see *2.2.8 RTPs and Environment Variables*, p.10).

While the signal mask is not actually a property of a process as such—it is a property of a task—the signal mask for the initial task in the process is inherited from the task that spawned it (that is, the task that called the **rtpSpawn( )** routine). If the kernel created the initial task, then the signal mask is zero, and all signals are unblocked.

The **getppid( )** routine returns the parent process ID. If the parent is the kernel, or the parent is dead, it returns NULL.

Zombie Processes

By default, when a process is terminated, and its parent is not the kernel, it becomes a zombie process.[2]

In order to respond to a **SIGCHLD** signal (which is generated whenever a child process is stopped or exits, or a stopped process is started) and get the exit status of the child process, the parent process must call **wait( )** or **waitpid( )** before the child exits or is stopped. In this case the parent is blocked waiting. Alternatively, the parent can set a signal handler for **SIGCHLD** and call **wait( )** or **waitpid( )** in the

[2] A zombie process is a "process that has terminated and that is deleted when its exit status has been reported to another process which is waiting for that process to terminate." (The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition.)

signal handler. In this case the parent is not blocked. After the parent process receives the exit status of a child process, the zombie entity is deleted automatically.

The default behavior with regard to zombies can be modified in the following ways:

- By leaving the parent process unaware of the child process' termination, and not creating a zombie. This is accomplished by having the parent process ignore the **SIGCHLD** signal. To do so, the parent process make a **sigaction( )** call that sets the **SIGCHLD** signal handler to **SIG_IGN**.
- By not transforming a terminating child process into a zombie when it exits. This is accomplished having the parent process make a **sigaction( )** call that sets the **sa_flag** to **SA_NOCLDWAIT**.

Resource Reclamation

When a process terminates, all resources owned by the process (objects, data, and so on) are returned to the system. The resources used internally for managing the process are released, as are all resources owned by the process. All information about that process is eliminated from the system (with the exception of any temporary zombie process information). Resource reclamation ensures that all resources that are not in use are immediately returned to the system and available for other uses.

Note, however, that there are exceptions to this general rule:

- Public objects—which may be referenced by tasks running in other processes that continue to run—must be explicitly deleted.
- Socket objects can persist for some time after a process is terminated. They are reclaimed only when they are closed, which is driven by the nature of the TCP/IP state machine. Some sockets must remain open until timeout is reached.
- File descriptors are reclaimed only when all references to them are closed. This can occur implicitly when all child processes—which inherit the descriptors from the parent process—terminate. It can also happen explicitly when all applications with references to the file descriptors close them.

For information about object ownership, and about public and private objects, see *6.9 Inter-Process Communication With Public Objects*, p.107.

2.2.8 RTPs and Environment Variables

By default, a process is created without environment variables. In a manner consistent with the POSIX standard, all tasks in a process share the same environment variables—unlike kernel tasks, which each have their own set of environment variables.

Setting Environment Variables From Outside a Process

While a process is created without environment variables by default, they can be set from outside the process in the following ways:

- If the new process is created by a kernel task, the contents of the kernel task's environment array can be duplicated in the application's environment array. The the **envGet( )** routine is used to get the kernel task's environment, which is then used in the **rtpSpawn( )** call.
- If the new process is created by a process, the child process can be passed the parent's environment if the environment array is used in the **rtpSpawn( )** call.
- If the new process is created from the kernel shell—using either **rtp exec** command or **rtpSp( )** routine—then all of the shell's environment is passed to the new process (the process' **envp** is set using the shell's environment variables). This makes it simple to set environment variables specifically for a process by first using **putenv( )** to set the variable in the shell's environment before creating the process. (For example, this method can be used to set the **LD_LIBRARY_PATH** variable for the runtime locations of shared libraries; see *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64.)

For more information, see the **rtpSpawn( )** API reference and *3.3.1 RTP Application Structure*, p.31 for details.

Setting Environment Variables From Within a Process

A task in a process (or in an application library) can create, reset, and remove environment variables in a process. The **getenv( )** routine can be used to get the environment variables, and the **setenv( )** and **unsetenv( )** routines to change or remove them. The environment array can also be manipulated directly—however, Wind River recommends that you do not do so, as this bypasses the thread-safe implementation of **getenv( )**, **setenv( )** and **putenv( )** in the RTP environment.

2.2.9 RTPs and POSIX

The overall behavior of the application environment provided by the real-time process model is close to the POSIX 1003.1 standard, while maintaining the embedded and real-time characteristics of the VxWorks operating system. The key areas of deviation from the standard are that VxWorks does not provide the following:

- process creation with **fork( )** and **exec( )**
- memory-mapped files
- file ownership and file permissions

For information about POSIX support, see *7. POSIX Facilities*.

POSIX PSE52 Support

VxWorks can be configured to provide POSIX PSE52 support (for individual processes, as defined by the profile). For detailed information, see *2.3 Configuring VxWorks For Real-time Processes*, p.12, *7.1 Introduction*, p.146, *7.2 Configuring VxWorks with POSIX Facilities*, p.147, and *10.4.3 HRFS and POSIX PSE52*, p.251.

2.3 Configuring VxWorks For Real-time Processes

The VxWorks operating system is configured and built independently of any applications that it might execute. To support RTP applications, VxWorks need only be configured with the appropriate components for real-time processes and any other facilities required by the application (for example, message queues). This independence of operating system from applications allows for development of a variety of systems, using differing applications, that are based on a single VxWorks configuration. That is, a single variant of VxWorks can be combined with different sets of applications to create different systems. The operating system does not need to be *aware* of what applications it will run before it is configured and built, as long as its configuration includes the components required to support the applications in question.

RTP applications can either be stored separately or bundled with VxWorks in an additional build step that combines the operating system and applications into a single system image (using the ROMFS file system). For information in this regard, see *2.3.3 Additional Component Options*, p.13 and *3.7 Bundling RTP Applications in a System using ROMFS*, p.50.

2.3.1 Basic RTP Support

In order to run RTP applications on a hardware target, VxWorks must be configured with the **INCLUDE_RTP** component. Doing so automatically includes other components required for RTP support.

Note that many of the components described in this chapter provide configuration parameters. While not all are discussed in this chapter, they can be reviewed and managed with the kernel configuration facilities (either Workbench or the **vxprj** command-line tool).

Information about alternative configurations that support specific RTP technologies are covered in the context of the general topics themselves. For example:

- *Configiuation With Process Support and Without an MMU*, p.16
- *Setting Configuration Parameters for the RTP Code Region*, p.24
- *4.4 Configuring VxWorks for Shared Libraries and Plug-ins*, p.56.

NOTE: The default VxWorks configuration for hardware targets does not include support for running applications in real-time processes (RTPs). VxWorks must be re-configured and rebuilt to provide these process facilities. The default configuration of the VxWorks simulator does, however, include full support for running RTP applications.

The reason that the default configuration of VxWorks (for hardware targets) does not include process support, is that it facilitates migration of VxWorks 5.5 kernel-based applications to VxWorks 6.*x* by providing functionally the same basic set of kernel components, and nothing more.

VxWorks 6.*x* systems can be created with kernel-based applications and without any process-based applications, or with a combination of the two. Kernel applications, however, cannot be provided the same level of protection as process-based applications. When applications run in kernel space, both the kernel and those applications are subject to any misbehavior on the part application code. For more information about kernel-based applications, see the *VxWorks Kernel Programmer's Guide: Kernel*.

2.3.2 MMU Support for RTPs

If a system is configured with **INCLUDE_RTP** the VxWorks components required for MMU memory protection are included by default—except for the MIPS architecture.

CAUTION: For the MIPS architecture, MMU support is not provided by default. In contrast to other architectures, this support is not added automatically when VxWorks is configured with **INCLUDE_RTP**. For MIPS, VxWorks must be configured with a mapped kernel by adding the **INCLUDE_MAPPED_KERNEL** component. For more information in this regard, see the *VxWorks Architecture Supplement*.

To create a system with processes, but without MMU support, the MMU components must be removed from the VxWorks configuration after the **INCLUDE_RTP** component has been added (for information in this regard see *2.4 Using RTPs Without MMU Support*, p.15).

2.3.3 Additional Component Options

The following components provide useful facilities for both development and deployed systems:

- **INCLUDE_ROMFS** for the ROMFS file system.
- **INCLUDE_RTP_APPL_USER**, **INCLUDE_RTP_APPL_INIT_STRING**, **INCLUDE_RTP_APPL_INIT_BOOTLINE**, and **INCLUDE_RTP_APPL_INIT_CMD_SHELL_SCRIPT** for various ways of automatically starting applications at boot time.
- **INCLUDE_SHARED_DATA** for shared data regions.
- **INCLUDE_SHL** for shared libraries.

- **INCLUDE_RTP_HOOKS** for the programmatic hook facility, which allows for registering kernel routines that are to be executed at various points in a process' life-cycle.
- **INCLUDE_POSIX_PTHREAD_SCHEDULER** and **INCLUDE_POSIX_CLOCK** for POSIX thread support. This replaces the traditional VxWorks scheduler with a scheduler handling user threads in a manner consistent with POSIX.1. VxWorks tasks as well as kernel pthreads are handled as usual. Note that the **INCLUDE_POSIX_PTHREAD_SCHEDULER** is required for using pthreads in processes. For more information, see *7.15 POSIX and VxWorks Scheduling*, p.171.
- **INCLUDE_PROTECT_TASK_STACK** for stack protection. For deployed systems this component may be omitted to save on memory usage. See *6.4.5 Task Stack*, p.96 for more information.

The following components provide facilities used primarily in development systems, although they can be useful in deployed systems as well:

- The various **INCLUDE_SHELL_***feature* components for the kernel shell, which, although not required for applications and processes, are needed for running applications from the command line, executing shell scripts, and on-target debugging.
- The **INCLUDE_WDB** component for using the host tools.
- Either the **INCLUDE_NET_SYM_TBL** or the **INCLUDE_STANDALONE_SYM_TBL** component, which specify whether symbols for the shell are loaded or built-in.
- The **INCLUDE_DISK_UTIL** and **INCLUDE_RTP_SHOW** components, which include useful shell routines.

For information about the kernel shell, symbol tables, and show routines, see the *VxWorks Kernel Programmer's Guide: Target Tools*. For information about the host shell, see the *Wind River Workbench Host Shell User's Guide*.

Component Bundles

The VxWorks configuration facilities provide component bundles to simplify the configuration process for commonly used sets of operating system facilities. The following component bundles are provided for process support:

- **BUNDLE_RTP_DEPLOY** is designed for deployed systems (final products), and is composed of **INCLUDE_RTP**, **INCLUDE_RTP_APPL**, **INCLUDE_RTP_HOOKS**, **INCLUDE_SHARED_DATA**, and the **BUNDLE_SHL** components.
- **BUNDLE_RTP_DEVELOP** is designed for the development environment, and is composed of **BUNDLE_RTP_DEPLOY**, **INCLUDE_RTP_SHELL_CMD**, **INCLUDE_RTP_SHOW**, **INCLUDE_SHARED_DATA_SHOW**, **INCLUDE_SHL_SHOW**, **INCLUDE_RTP_SHOW_SHELL_CMD**, **INCLUDE_SHL_SHELL_CMD**, components.
- **BUNDLE_RTP_POSIX_PSE52** provides POSIX PSE52 support for individual processes (for more information see *7.2 Configuring VxWorks with POSIX Facilities*, p.147). It can be used with either **BUNDLE_RTP_DEPLOY** or **BUNDLE_RTP_DEVELOP**.

2.3.4 Configuration and Build Facilities

For information about configuring and building VxWorks, see the *Wind River Workbench by Example* guide and the *VxWorks Command-Line Tools User's Guide*.

Note that the VxWorks simulator includes all of the basic components required for processes by default.

2.4 Using RTPs 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 (which does support RTPs), see the *VxWorks Kernel Programmer's Guide: 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*.

Configuation With Process Support and 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.

2.5 About VxWorks RTP Virtual Memory Models

VxWorks can be configured to use either a flat or an overlapped virtual memory model for RTPs. By default, it uses the flat virtual memory model.

2.5.1 Flat RTP Virtual Memory Model

The flat virtual memory model is the default and requires no specific configuration. With this model each VxWorks process has its own region of virtual memory—processes do not overlap in virtual memory. The flat virtual-memory map provides the following advantages:

- Speed—context switching is fast.
- Ease of debugging—the addresses for each process are unique.
- A flexible programming model that provides the same process-model regardless of MMU support. VxWorks' application memory model allows for running the same applications with and without an MMU. Hard real-time determinism can be facilitated by using the same programming model, and by disabling the MMU.

  Systems can be developed and debugged on targets with an MMU, and then shipped on hardware that does not have one, or has one that is not enabled for deployment. The advantages of being able to do so include facilitating debugging in development, lower cost of shipped units, as well as footprint and performance advantages of targets without an MMU, or with one that is not enabled. (For information in this regard, see *2.4 Using RTPs Without MMU Support*, p.15.)

With the flat virtual memory model, however, executable files must be relocated, which means that their loading time is slower than with the overlapped model.

In addition, the flat model does not allow for selection of the specific virtual memory area into which the RTP application's text, data, and bss segments are installed. Instead a best-fit algorithm automatically selects the most appropriate area of free virtual memory, based on the memory's current usage, where those segments are to be loaded.

2.5.2 Overlapped RTP Virtual Memory Model

VxWorks can be configured to use an overlapped virtual memory model instead of the flat model. With the overlapped model all VxWorks RTP applications share a common range of virtual memory addresses into which the applications' text, data, and bss segments are installed, and all applications share the same execution address.

In addition to holding the RTP application's text, data, and bss segments, the overlapped area of virtual memory is automatically used for any elements that are private to a process, such as its heap and task stacks (as long as there is enough room; otherwise outside this area). The overlapped area is not, however, used for elements that are designed to be accessible to other RTPs, such as shared data regions and shared libraries.

To take advantage of the overlapped model, RTP applications must be built as absolutely-linked executables (For information in this regard, see *Building Absolutely-Linked RTP Executables*, p.26.). They share the same execution address, which is within the overlapped area of virtual memory. For simplicity sake, the execution address should be defined to be at the base of this area. When they are loaded, the relocation step is therefore unnecessary.

Note that if several instances of the same RTP application are started on a system, they occupy exactly the same ranges of virtual memory addresses, but each instance still has physical memory allocated for its text, and data, and bss segments. In other words, there is no sharing of the text segment between multiple instances of an application.

The overlapped virtual memory model provides the following advantages:

- Faster loading time when applications are built as absolutely-linked executables. The relocation phase is skipped when absolutely-linked executables applications are loaded because they are all installed at the same virtual memory execution address. The improvement in loading times is on the order of 10 to 50 percent. Note that the improvement in loading time is dependent on the number of relocation entries in the file.

- More precise control of the system's virtual memory. The user selects the area in virtual memory into which the RTP application's text, data, and bss segments are installed. With the flat memory model, on the other hand, a best-fit algorithm automatically selects the most appropriate area of free virtual memory, based on the memory's current usage, where those segments are to be loaded.

- More efficient usage of the system's virtual memory by reducing the possibility of memory fragmentation.

- The memory footprint of absolutely-linked RTP executables stored in ROMFS is about only half that of relocatable executables once they (the absolutely-linked executables) are stripped of their symbols and relocation information. For information about using ROMFS and stripping executables, see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50 and *Stripping Absolutely-Linked RTP Executables*, p.27.

The overlapped virtual memory model does, however, require that the system make use of an MMU.

⚠ **CAUTION:** For architectures that do not have (or use) a hardware mechanisms that handles concurrent address spaces (usually known as an address space identifier, or ASID), the overlapped virtual memory model also makes a system susceptible to performance degradation. This is basically because each time a RTP's task is switched out and another kernel or RTP task is switched in the entire cache must be flushed and reloaded with the new task's context. With the ASID mechanism this flushing is not (or less often) necessary because the cache is indexed using the ASID. The hardware therefore knows when such a flushing is required or not.

VxWorks Overlapped Virtual Memory Model and Other Operating Systems

The VxWorks overlapped virtual memory model is similar to implementations for other operating systems (such as UNIX), but differs in the following ways:

- For VxWorks the user-mode virtual address space is not predefined. For Windows, on the other hand, the operating system ensures that 2 GB (or 3 GB with some configuration changes) are reserved for a process. For Solaris, almost the entirety of the 4 GB address space is usable by a process. For Linux almost 3 GB of the address space is reserved for a process. A VxWorks process uses whatever is available and this can be different from target to target (see *User Regions of Virtual Memory*, p.19).

- The text, data, and bss segments of VxWorks applications are allocated together in memory. In other operating systems (such as UNIX) there is usually a separate allocation for each. This is significant only in the context of defining the size of the RTP code region so that it is large enough to hold the sum of the text, data, and bss segments, and not simply the text segment (see *RTP Code Region in Virtual Memory*, p.20).

- For Unix and Windows the concept of overlapped virtual address space for a process covers the whole 4 GB of the address range and the address space organization is set in advance. In particular shared libraries are mapped differently in each process but usually go in a pre-defined and reserved area of the address space. In VxWorks only the RTP code region and the private elements of an RTP are set in address ranges that are overlapped. The shared libraries and shared data regions are not overlapped and appear at the same virtual address in all processes; furthermore they can use any place in the user regions except for the area covered by the RTP code region (this cannot be controlled by the user). A process that does not use shared libraries still has a portion of its address space unavailable if any other process uses shared libraries and or shared data regions.

2.6 Using the Overlapped RTP Virtual Memory Model

Using the VxWorks overlapped virtual memory model requires an understanding of *user regions* and how one of those regions is used for the area of overlapped virtual memory called the *RTP code region*. This background information is provided in *2.6.1 About User Regions and the RTP Code Region*, p.19.

The process of implementing the overlapped memory model with an RTP code region involves getting information about the user regions that are available in a system, selecting the RTP code region based on the available user regions and RTP application requirements, and configuring VxWorks accordingly. These steps are described in *2.6.2 Configuring VxWorks for Overlapped RTP Virtual Memory*, p.21.

In order for RTP applications make use of the advantages of overlapped virtual memory, they, must be built as absolutely-linked executables. The compiler options required to do so are described in *2.6.3 Using RTP Applications With Overlapped RTP Virtual Memory*, p.26 (in addition to information about optimization by stripping and application execution).

2.6.1 About User Regions and the RTP Code Region

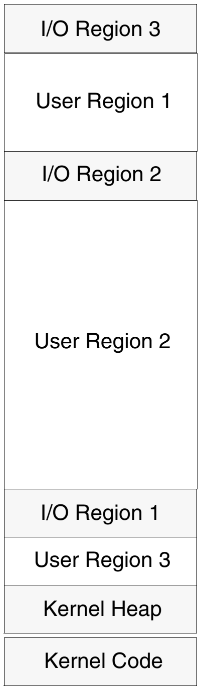
The VxWorks overlapped virtual memory model depends on using one of the available *user regions* of virtual memory for the area of overlapped virtual memory called an *RTP code region*.

User Regions of Virtual Memory

The virtual address space of a process in VxWorks does not correspond to a contiguous range of address from 0 to 4 GB. It is generally composed of several discontinuous blocks of virtual memory.

The blocks of virtual memory that are available for RTPs (that is, not used by the kernel) are referred to as *user regions*. User regions are used for the RTP applications' text, data, and bss segments, as well as for their heaps and task stacks. In addition, user regions are used for shared data regions, shared libraries, and so on. Figure 2-1 illustrates an example of virtual memory layout in VxWorks.

Figure 2-1 **Virtual Memory and User Regions**



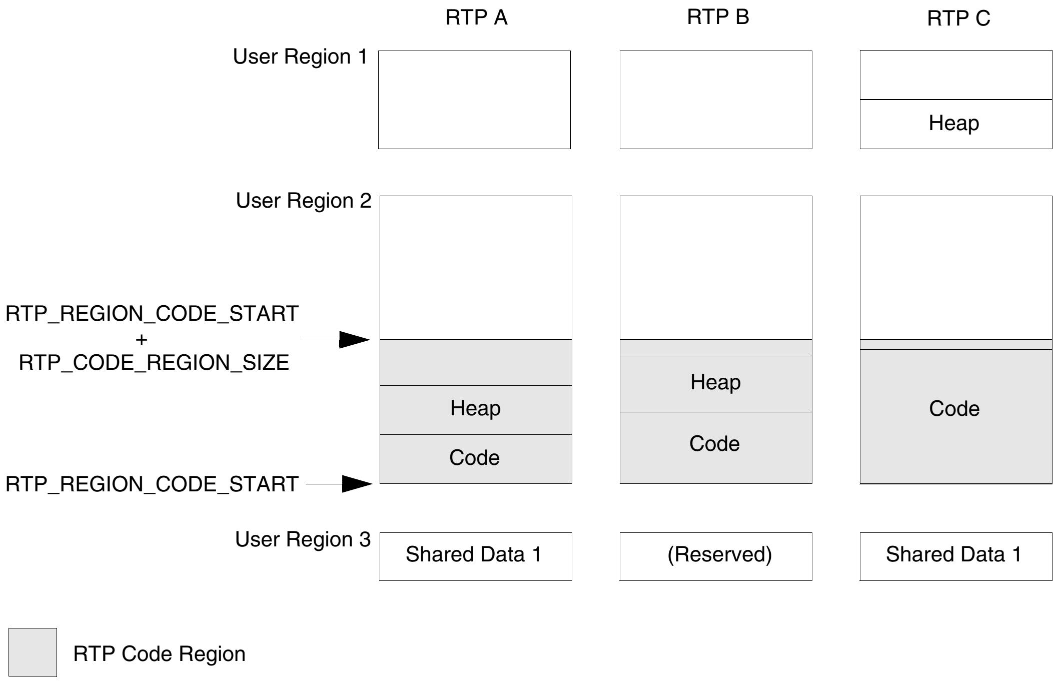
RTP Code Region in Virtual Memory

Only one continuous area of the virtual address space of a process can be used to overlap RTP application code, and this area must correspond (fully or in part) to one of the user regions available in the target system's memory. The user region that is selected for the overlap is referred to as the *RTP code region*. The base address and size of the RTP code region are defined by the user when the system is configured.

RTP Code Region Example

Figure 2-2 illustrates the virtual memory layout of a system running three applications, with the same three user regions as in Figure 2-1.

Figure 2-2 **Virtual Memory Layout With RTP Code Region**



VxWorks has been configured to use the largest of the three user regions depicted in Figure 2-1 (user region 2) for the RTP code region because the others were too small for the code (text, data, and bss segments) of the largest application (RTP C). As the heap for RTP C would not fit in the RTP code region, the best-fit algorithm automatically placed it in User Region 1 in order to leave a larger area for other purposes in User Region 2 (for example, it might be used for a large shared data region).

Note that the size of the RTP Code Region is defined to be slightly larger than the size of the text, data, and bss segments of that application.

The system illustrated in Figure 2-2 includes a shared data region used by RTP A and RTP C, which map the region into their memory context. The location of shared data regions is determined automatically at runtime on the basis of a best-fit algorithm when they are created (for information shared data regions, see *3.5 Creating and Using Shared Data Regions*, p.38). Note that RTP B is not allowed to make use of the virtual addresses in the shared data region, even though the application does not make use of it.

For information about the configuration parameters that define location and size of the RTP code region (**RTP_CODE_REGION_START** and **RTP_CODE_REGION_SIZE**), see *Setting Configuration Parameters for the RTP Code Region*, p.24.

2.6.2 Configuring VxWorks for Overlapped RTP Virtual Memory

By default VxWorks is not configured for overlapped RTP virtual memory and does not have an RTP code region defined. In order to use the overlapped RTP virtual memory model, you must determine which user region is suitable for your applications, and then reconfigure and rebuild VxWorks. This process involves the following basic steps:

1. Boot an instance of VxWorks that has been configured with RTP support and the *flat* virtual memory model, and get information about the available user regions.
2. Identify the RTP code region (size and base address), based on the available user regions and the requirements of your RTP applications.
3. Configure VxWorks for the overlapped memory model and specifying the RTP code region in that configuration. Then rebuild the system.

These steps are described in detail in the following sections.

Getting Information About User Regions

Before you can configure VxWorks with an RTP code region, you must determine what range of virtual memory is available for this purpose. To do so, boot an instance of VxWorks that has been configured with RTP support (and which is by default configured for the flat memory model), and get a listing of the user regions that are available on the target.

⚠ **CAUTION:** For the MIPS architecture, in order to get correct information about user regions, the system must be configured with MMU support. In contrast to other architectures, this support is not provided automatically when VxWorks is configured with **INCLUDE_RTP**. For MIPS, VxWorks must be configured with a mapped kernel by adding the **INCLUDE_MAPPED_KERNEL** component. For more information in this regard, see the *VxWorks Architecture Supplement*.

The **adrSpaceShow( )** kernel shell command (in verbose mode) can be used to list user regions. For example:

```
-> adrSpaceShow 1

RAM Physical Address Space Info:
-------------------------------
        Allocation unit size:           0x1000
        Total number of units:          16384       (67108864 bytes)
        Number of allocated units:      12150       (49766400 bytes)
        Largest contiguous free block:  17342464
        Number of free units:           4234        (17342464 bytes)
               1 block(s) of 0x0108a000 bytes  (0x02f70000-0x03ff9fff)

User Region (RTP/SL/SD) Virtual Space Info:
-------------------------------------------
        Allocation unit size:           0x1000
        Total number of units:          851968      (3489660928 bytes)
        Number of allocated units:      0           (0 bytes)
        Largest contiguous free block:  3221225472
        Number of free units:           851968      (3489660928 bytes)
               1 block(s) of 0xf000000 bytes  (0x10000000-0x1effffff)
               1 block(s) of 0xc0000000 bytes  (0x30000000-0xefffffff)

Kernel Region Virtual Space Info:
---------------------------------
        Allocation unit size:           0x1000
        Total number of units:          196608     (805306368 bytes)
               1 block(s) of 0x04000000 bytes  (0x00000000-0x03ffffff)
               1 block(s) of 0x0c000000 bytes  (0x04000000-0x0fffffff)
               1 block(s) of 0x08000000 bytes  (0x20000000-0x27ffffff)
               1 block(s) of 0x08000000 bytes  (0x28000000-0x2fffffff)
               1 block(s) of 0x04000000 bytes  (0xf0000000-0xf3ffffff)
               1 block(s) of 0x01000000 bytes  (0xf4000000-0xf4ffffff)
               1 block(s) of 0x05000000 bytes  (0xf5000000-0xf9ffffff)
               1 block(s) of 0x01010000 bytes  (0xfa000000-0xfb00ffff)
               1 block(s) of 0x00fe0000 bytes  (0xfb010000-0xfbfeffff)
               1 block(s) of 0x00050000 bytes  (0xfbff0000-0xfc03ffff)
               1 block(s) of 0x00fc0000 bytes  (0xfc040000-0xfcffffff)
               1 block(s) of 0x01810000 bytes  (0xfd000000-0xfe80ffff)
               1 block(s) of 0x00770000 bytes  (0xfe810000-0xfef7ffff)
               1 block(s) of 0x00010000 bytes  (0xfef80000-0xfef8ffff)
               1 block(s) of 0x00060000 bytes  (0xfef90000-0xfefeffff)
               1 block(s) of 0x00010000 bytes  (0xfeff0000-0xfeffffff)
               1 block(s) of 0x01000000 bytes  (0xff000000-0xffffffff)
value = 0 = 0x0
```

This output shows that the following two user regions are available on this target:

- from 0x10000000 to 0x1effffff
- from 0x30000000 to 0xefffffff

Either region can be used for the RTP code region, depending on the requirements of the system.

Identifying the RTP Code Region

The following guidelines should be used when defining the RTP code region:

- The RTP code region should be slightly larger than the combined size of the text, data, and bss segments of the largest RTP application that will run on the system.
- The RTP code region must fit in one user region—it cannot span multiple user regions.
- In selecting the user region to use for the RTP code region, take into consideration number and size of the shared data regions, shared libraries and other public mappings—such as those created by **mmap( )**—that are going to be used in the system.
- For simplicity sake, set the base address of the RTP code region to the base address of a user region.

⚠ **CAUTION:** If the overlapped virtual memory model is selected, but the base address and size are not defined, or if the size is too small, absolutely-linked executables will be relocated. If the executables are stripped, however, they cannot be relocated and the RTP launch will fail.

RTP Code Region Size

The RTP code region should be slightly larger than the combined size of the text, data, and bss segments of the largest RTP application that will run on the system. Allowing a bit of extra room allows for a moderate increase in the size of the applications that you will run on the system. The **readelf***arch* tool (for example, **readelfppc**) can be used to determine the size of the text, data, and bss segments.

In addition, the page alignments of the text and data segments must be taken into account for (the bss is already aligned by virtues of its being included in the data segment's size). As a basic guideline for the alignment, use **readelf***arch* **-l** (for example, **readelfppc**), and round up to one page for each of the segment's sizes displayed in the **MemSiz** fields.

For example, for a text segment with a **MemSiz** of 0x16c38, the round up value would be 0x17000; for a data segment with a **MemSiz** field (do not use the **FileSiz** field, which is much smaller as it does not account for the bss area) of 0x012a8, the round up value would be 0x02000. The sum of the rounded values for the two segments would then be 0x19000.

While it may be tempting to select the largest user region for the RTP code region in order to accommodate the largest possible RTP application that the system might run, this may not leave enough room in the other user regions to accommodate all of the shared data regions, shared libraries, or other public mappings required by the system.

User Region Choice

The RTP code region cannot span user regions. It must fit in one user region.

The RTP code region and public mappings are mutually exclusive because the RTP code region is intended to receive the text, data, and bss segments of absolutely-linked executables that cannot be relocated. However, public mappings appear at the same address in all the RTPs that may want to use them (by design). Since the location of the public mappings in virtual memory is not controlled by the applications themselves, and since VxWorks applies a best-fit algorithm when allocating virtual memory for a public mappings, there would be a risk of blocking out a range of virtual addresses at a location meant to be used by an absolutely-linked application's text, data, and bss segments.

RTP Code Region Base Address Choice

For simplicity sake, set the base address of the RTP code region to the base address of a user region. If the RTP code region is set elsewhere in the user region, make sure that its base address is page aligned. The page size for the target architecture can be checked via the **vmPageSizeGet( )** routine, which can be called directly from the target shell.

RTP Code Region Size

When determining the size of the RTP code region, make sure that it is page-aligned. The page size for the target architecture can be checked via the **vmPageSizeGet( )** routine, which can be called directly from the target shell.

Setting Configuration Parameters for the RTP Code Region

By default, VxWorks is configured for the flat virtual memory model. In order to use the overlapped memory model, the configuration parameters listed below must be set appropriately. They serve to select the overlapped virtual memory model itself, and to define the RTP code region in virtual memory.

⚠ **CAUTION:** The overlapped virtual memory model requires MMU support, which is provided by default configurations of VxWorks—except for the MIPS architecture. For MIPS, MMU protection requires a mapped kernel, which is provided by **INCLUDE_MAPPED_KERNEL** component. This component is not included by default and must be added to the kernel configuration. For more information in this regard, see the *VxWorks Architecture Supplement*.

RTP Code Region Component Parameters

For information about determining the virtual memory addresses that you need for setting the **RTP_CODE_REGION_START** and **RTP_CODE_REGION_SIZE** parameters, see *2.6 Using the Overlapped RTP Virtual Memory Model*, p.19, and specifically *2.6.1 About User Regions and the RTP Code Region*, p.19.

RTP_OVERLAPPED_ADDRESS_SPACE

Set to **TRUE** to change the virtual memory model from flat to overlapped. By default it is set to **FALSE**. The following parameters have no effect unless this one is set to **TRUE**.

RTP_CODE_REGION_START

Identifies the virtual memory address within a user region where the RTP code region will start. That is, the base address of the memory area into which the text, data, and bss segments of RTP applications will be installed. For simplicity sake, set the base address of the RTP code region to the base address of a user region. If the RTP code region is set elsewhere in the user region, make sure that its base address is page aligned. For more information, see *Identifying the RTP Code Region*, p.23.

Note that the VxWorks kernel is usually located in low memory addresses except when required by the architecture (MIPS for instance). In that case, the user mode virtual address space appears higher in the address range. This is significant, as changing the configuration of the kernel (adding components or devices) may have an impact on the base addresses of the available user regions and may therefore impact the base address and size of the RTP code region.

RTP_CODE_REGION_SIZE

The size (in bytes) of the virtual memory area into which the text, data, and bss segments of RTP applications will be installed. Select a size that is slightly larger than the combined size of the text, data, and bss segments of the largest RTP application that will be run on the system. Make sure that the size is page-aligned. For more information, see *Identifying the RTP Code Region*, p.23.

An RTP code region is defined for a system when **RTP_OVERLAPPED_ADDRESS_SPACE** is set to **TRUE** and both **RTP_CODE_REGION_START** and **RTP_CODE_REGION_SIZE** are set to values other than zero.

⚠ **CAUTION:** If the overlapped virtual memory model is selected (by setting **RTP_OVERLAPPED_ADDRESS_SPACE** to **TRUE**), then the user should set the **RTP_CODE_REGION_START** and **RTP_CODE_REGION_SIZE** parameters appropriately. If they are not defined, or if the size of the region is too small, absolutely-linked executables will be relocated to another user region if there is sufficient space. If not, the RTP launch will fail. In addition, if the executables are stripped, they cannot be relocated regardless of the availability of space in other user regions, and the RTP launch will fail.

Setting RTP Code Region Parameters for Multiple Projects

Once the settings for the **RTP_OVERLAPPED_ADDRESS_SPACE**, **RTP_CODE_REGION_START** and **RTP_CODE_REGION_SIZE** parameters have been determined, configuration for multiple projects can be automated by creating a custom component description file (CDF) file containing those settings. Using a CDF file means that the parameters do not have to be set manually each time a project is created (either with Workbench or **vxprj**). If the file is copied into a project directory before it is built, it applies the settings to that project. If it is copied into the BSP directory before the project is created, it applies to all projects subsequently based on that BSP.

The contents of the local CDF file (called, for example, **00rtpCodeRegion.cdf**) would look like this:

```
Parameter RTP_OVERLAPPED_ADDRESS_SPACE {
    DEFAULT    TRUE
}

Parameter RTP_CODE_REGION_START {
    DEFAULT    0xffc00000
}

Parameter RTP_CODE_REGION_SIZE {
    DEFAULT    0x100000
}
```

For information about CDF files, file naming conventions, precedence of files, and so on, see the *VxWorks Kernel Programmer's Guide: Kernel Customization*.

2.6.3 Using RTP Applications With Overlapped RTP Virtual Memory

In order to take advantage of the efficiencies provided by the overlapped virtual memory model, RTP applications must be built as absolutely-linked executables. They are started in the same manner as relocatable RTP executables, and can also be run on systems configured with the default flat virtual memory model, but will be relocated. Absolutely-linked executables can stripped of their symbols to reduce their footprint, but they would fail to run on a system configured with the flat virtual memory model.

Building Absolutely-Linked RTP Executables

In order to take advantage of the efficiencies provided by the overlapped virtual memory model, RTP applications must be built as absolutely-linked executables. This is done by defining the link address with a special linker option. The option can be used directly with the Wind River or GNU toolchain, or indirectly with a Wind River Workbench GUI option or a command-line **make** macro. For simplicity sake, it is useful to use the same link address for all executables (with consideration of each of them fitting within the RTP code region, of course).

Selecting the Link Address

While technically the link address of the RTP executable could be anywhere in the RTP code region (provided the address is low enough to leave room for the applications text, data, and bss segments), Wind River recommends that it be set to base address of the RTP code region itself (that is, the address specified with the **RTP_CODE_REGION_START** configuration parameter; see *Setting Configuration Parameters for the RTP Code Region*, p.24).

Linker Options

The following linker options must be used to generate executable with a pre-determined link address (the base address of the text segment):

- Wind River Compiler (diab): **-Wl,-Bt**
- GNU compiler: **-Wl,--defsym,__wrs_rtp_base**

These linker options are automatically invoked when the default makefile rules of the VxWorks cross-development environment are used.

The link address of the data segment cannot be specified separately. By design the data segment of an application immediately follows the text segment, with consideration for page alignment constraints. Both segments are installed together in one block of allocated memory.

Note that Wind River Workbench provides GUI options for defining the base address and the command-line build environment provides a make macro for doing the same.

RTP_LINK_ADDR Make Macro

The **RTP_LINK_ADDR** make macro can be used to set the link address for absolutely-linked executables when using the VxWorks build environment from the command line. For example, as in executing the following command from *installDir*/**vxworks-6.***x***/target/usr/apps/sample/helloworld**:

```
% make CPU=PENTIUM4 TOOL=gnu RTP_LINK_ADDR=0xe0000000
```

Stripping Absolutely-Linked RTP Executables

Absolutely-linked RTP executables are generated for execution at a predetermined address. They do not, therefore, need symbolic information and relocation information during the load phase. This information can be stripped out of the executable using the **strip***arch* utility with the **-s** option. The resulting file is notably smaller (on average 30%-50%). This make its footprint in ROMFS noticeably smaller (if the executables are stored in ROMFS, this can reduce overall system footprint), as well as making its load time somewhat shorter.

Note, however, that stripping symbols and relocation sections from an absolutely-linked RTP executable means that it cannot be loaded if the predetermined execution address cannot be granted by the system. This may occur under the following circumstances:

- The RTP code region is too small for the text, data, and bss segments of the executable.
- The executable file has been generated for an execution address that does not correspond to the system's current RTP code region.
- The absolutely-linked executable file is used on a system configured for the flat virtual memory model.

Note that it may be useful to leave the symbolic and relocation information in executable file situations for situations in which the execution environment may change. For example, if a deployed system is updated with a new configuration of VxWorks for which the existing applications’ execution addresses are no longer valid (but the applications cannot be updated at the same) the applications suffer the cost of relocation, but still execute. If, however, the applications had been stripped, would be unusable.

Executing Absolutely-Linked RTP Executables

Absolutely-linked executables can be started in the same ways as relocatable executables (for information in this regard, see *3.6 Executing RTP Applications*, p. 40). The load time of absolutely-linked executables, however, is noticeably shorter because they are not relocated. Their text, data, and bss segments are installed according to the link address defined when they are compiled.

RTP executables can be executed on different target boards of the same architecture as long as the VxWorks images running on those boards provide the features that the applications require. However, if the RTP code regions are not at the same location and of the same size, the RTP executables text, data, and bss segments are relocated. This would typically happen if the RTP code region cannot accommodate the segments, for any of the following reasons:

- The size of the region is not sufficient.
- The base address of the executable is too close to the top address of the RTP code region (which prevents the segments from fitting in the remaining space).
- The executable's base address does not corresponds to an address within a user region.

Note that if the base address of the executable is completely outside of the RTP code region but still corresponds to a user region then the executable is *not* relocated, and is installed outside of the RTP code region. The side-effect of this situation is that this may reduce the memory areas available to public mappings (in particular shared data regions and shared libraries).

Executing Relocatable RTP Executables

Relocatable executables are supported when the RTP address space is overlapped. They are relocated as usual, and their text and data segments are installed in the RTP code region providing that the RTP code region is big enough to accommodate them. If the RTP code region is too small for the segments, they are installed elsewhere, providing again that there is enough room available in the remaining areas of the user regions.

3
RTP Applications

3.1 Introduction

Real-time process (RTP) applications are user-mode applications similar to those used with other operating systems, such as UNIX and Linux. This chapter provides information about writing RTP application code, using shared data regions, executing applications, and so on. For information about multitasking, I/O, file system, and other features of VxWorks that are available to RTP applications, see the respective chapters in this guide.

Before you begin developing RTP applications, you should understand the behavior of RTP applications in execution—that is, as *processes*. For information about RTP scheduling, creation and termination, memory, tasks and so on, see *2. Real-Time Processes*.

For information about using Workbench and the command-line build environment for developing RTP applications, see the *Wind River Workbench by Example* guide and the *VxWorks Command-Line Tools User's Guide*, respectively.

For information about developing kernel-mode applications (which execute in VxWorks kernel space) see the *VxWorks Kernel Programmer's Guide: Kernel Applications*.

3.2 Configuring VxWorks For RTP Applications

RTP applications require VxWorks kernel support. For information about configuring VxWorks for RTPs, see *2.3 Configuring VxWorks For Real-time Processes*, p. 12.

⚠ **CAUTION:** Because RTP applications are built independently of the operating system, 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. It is, therefore, important for application code to check for errors indicating that kernel facilities are not available and to respond appropriately. For more information, see *3.3.6 Checking for Required Kernel Support*, p. 37.

3.3 Developing RTP Applications

Real-time process (RTP) applications have a simple structural requirement that is common to C programs on other operating systems—they must include a **main( )** routine. VxWorks provides C and C++ libraries for application development, and the kernel provides services for user-mode applications by way of system calls.

RTP applications are built independently of the VxWorks operating system, using cross-development tools on the host system. When an application is built, user code is linked to the required VxWorks application API libraries, and a single ELF executable is produced. By convention, VxWorks RTP executables are named with a **.vxe** file-name extension. The extension draws on the **vx** in VxWorks and the **e** in executable to indicate the nature of the file.

Applications are created as either fully-linked or partially linked executables using cross-development tools on a host system. (Partially-linked executables are used with shared libraries.) They can be either relocatable or absolutely-linked objects depending on the virtual memory model in use on the VxWorks system. By default RTP applications are created as relocatable executables so as to be usable with either the flat virtual memory model or the overlapped virtual memory model. They can also be generated absolutely-linked to fully take advantage of the overlapped virtual memory model (for information about RTP virtual memory models, see *2.5 About VxWorks RTP Virtual Memory Models*, p. 16).

During development, processes can be spawned to execute applications from the VxWorks shell or various host tools. Applications can also be started programmatically, and systems can be configured to start applications automatically at boot time for deployed systems. For systems with multiple applications, not all must be started at boot time. They can be started later by other applications, or interactively by users. Developers can also implement their own application startup managers.

A VxWorks application can be loaded from any file system for which the kernel has support (NFS, ftp, and so on). RTP executables can be stored on disks, in RAM, flash, or ROM. They can be stored on the target or anywhere else that is accessible over a network connection.

In addition, applications can be bundled into a single image with the operating system using the ROMFS file system (see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50). The ROMFS technology is particularly useful for deployed systems. It allows developers to bundle application executables with the VxWorks image into a single system image. Unlike other operating systems, no root file system (on NFS or diskette, for example) is required to hold application binaries, configuration files, and so on.

RTP Applications With Shared Libraries and Plug-Ins

For information about the special requirements of applications that use shared libraries and plug-ins, see *4.8.9 Developing RTP Applications That Use Shared Libraries*, p.67 and *4.9.3 Developing RTP Applications That Use Plug-Ins*, p.73.

RTP Applications for the Overlapped Virtual Memory Model

Relocatable RTP application executables (the default) can be run on a system that has been configured for the overlapped virtual memory model. However, to take advantage of this model, RTP applications must be built as absolutely-linked executables. For more information, see *2.6.3 Using RTP Applications With Overlapped RTP Virtual Memory*, p.26

RTP Applications for UP and SMP Configurations of VxWorks

RTP applications can be used for both the uniprocessor (UP) and symmetric multiprocessing (SMP) configurations of VxWorks. They must, however, only use the subset of APIs provided by VxWorks SMP and be compiled specifically for the system in question (SMP or UP).

Among other things, this means that the RTP application must do the following in order to run on both VxWorks UP and VxWorks SMP systems:

- Use semaphores or another mechanism supported for SMP instead of **taskRtpLock( )**.
- Use the **__thread** storage class instead of **tlsLib** routines.

For more information about using the SMP configuration of VxWorks, and about migrating applications from VxWorks UP to VxWorks SMP, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

Migrating Kernel Applications to RTP Applications

For information about migrating VxWorks kernel applications to RTP applications, see *A. Kernel to RTP Application Migration*.

3.3.1 RTP Application Structure

VxWorks RTP applications have a simple structural requirement that is common to C programs on other operating systems—they must include a **main( )** routine.

The **main( )** routine can be used with the conventional **argc** and **argv** arguments, as well as two additional optional arguments, **envp** and **auxp**:

```
int main
    (
    int argc,      /* number of arguments */
    char * argv[], /* null-terminated array of argument strings */
    char * envp[], /* null-terminated array of environment variable strings */
    void * auxp    /* implementation specific auxiliary vector */
    );
```

The **argv[0]** argument is typically the relative path to the executable.

The **envp** and **auxp** arguments are usually not required by application code, but are used by the system as follows:

- The **envp** argument is used for passing VxWorks environment variables to the application. Environment variables are general properties of the running system, such as the default path—unlike **argv** arguments, which are passed to a particular invocation of the application, and are unique to that application.
- The **auxp** vector is used to pass system information to the new process, including page size, cache alignment size and so on.

NOTE: The **envp** and **auxp** arguments are usually not required by application code. They are not part of the ANSI C99 standard (although the standard allows for implementation-defined variants). Wind River recommends that you do not use them as support may be removed in a future release.

Environment variables are typically inherited from the calling environment, and can be set by a user. You should use the **getenv( )** routine to get the environment variables programmatically, and the **setenv( )** and **unsetenv( )** routines to change or remove them. (For more information about environment variables, see *2.2.8 RTPs and Environment Variables*, p.10.)

3.3.2 VxWorks Header Files

RTP applications often make use of VxWorks operating system facilities or utility libraries. This usually requires that the source code 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 for RTP applications are in the directory *installDir***/vxworks-6.***x***/target/usr/h** and its subdirectories (different directories are used for kernel applications).

CAUTION: Do not reference header files that are for kernel code (which are in and below *installDir***/vxworks-6.***x***/target/h**) in application code.

POSIX Header Files

Traditionally, VxWorks has provided many header files that are described by POSIX.1, although their content only partially complied with that standard. For user-mode applications the POSIX header files are more strictly compliant with the

POSIX.1 description, in both in their content and in their location. See *7.5 POSIX Header Files*, p.151 for more information.

VxWorks Header File: vxWorks.h

It is often useful to include header file **vxWorks.h** in all application modules in order to take advantage of architecture-specific VxWorks facilities. Many other VxWorks header files require these definitions. Include **vxWorks.h** with the following line:

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

Other VxWorks Header Files

Applications can include other VxWorks header files as needed to access VxWorks facilities. For example, an application 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/usr/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/usr/h**. No special flags are required to enable the compilers to find these headers. For more information about C++ development, see *5. 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**.

Compiler -I Flag

By default, the compiler searches for header files first in the directory of the source code and then in its internal subdirectories. In general, *installDir*/**vxworks-6.***x*/**target/usr/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/usr/h %WIND_BASE%/target/usr/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 <vxWorks.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, an application 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 your application. The only supported uses of VxWorks facilities are through the public definitions in the header file, and through the public APIs. Your adherence ensures that your application code is not affected by internal changes in the implementation of a VxWorks facility.

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

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

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

3.3.3 RTP Application APIs: System Calls and Library Routines

VxWorks provides an extensive set of APIs for developing RTP applications. As with other operating systems, these APIs include both system calls and library routines. Some library routines include system calls, and others execute entirely in user space. Note that the user-mode libraries provided for RTP applications are completely separate from kernel libraries.

Note that a few APIs operate on the process rather than the task level—for example, **kill( )** and **exit( )**.

VxWorks System Calls

Because kernel mode and user mode have different instruction sets and MMU settings, RTP applications—which run in user mode—cannot directly access kernel routines and data structures (as long as the MMU is on). System calls provide the means by which applications request that the kernel perform a service on behalf of the application, which usually involves operations on kernel or hardware resources.

System calls are transparent to the user, but operate as follows: For each system call, an architecture-specific trap operation is performed to change the CPU privilege level from user mode to kernel mode. Upon completion of the operation requested by the trap, the kernel returns from the trap, restoring the CPU to user mode. Because they involve a trap to the kernel, system calls have higher overhead than library routines that execute entirely in user mode.

Note that if VxWorks is configured without a component that provides a system call required by an application, **ENOSYS** is returned as an **errno** by the corresponding user-mode library API.

Also note that if a system call has trapped to the kernel and is waiting on a system resource when a signal is received, the system call may be aborted. In this case the errno **EINTR** may be returned to the caller of the API.

System calls are identified as such in the VxWorks API references.

The set of system calls provided by VxWorks can be extended by kernel developers. They can add their own facilities to the operating system, and make them available to processes by registering new system calls with the VxWorks system call infrastructure. For more information, see the *VxWorks Kernel Programmer's Guide: Kernel Customization*.

Monitoring System Calls

The VxWorks kernel shell provides facilities for monitoring system calls. For more information, see the *VxWorks Kernel Programmer's Guide: Target Tools*, the **syscall monitor** entry in the *VxWorks Kernel Shell Command Reference*, and the **sysCallMonitor( )** entry in the *VxWorks Kernel API Reference*.

VxWorks Libraries

VxWorks distributions include libraries of routines that provide APIs for RTP applications. Some of these routines execute entirely in the process in user mode. Others are wrapper routines that make one or more system calls, or that add additional functionality to one or more system calls. For example, **printf( )** is a wrapper that calls the system call **write( )**. The **printf( )** routine performs a lot of formatting and so on, but ultimately must call **write( )** to output the string to a file descriptor.

Library routines that do not include system calls execute in entirely user mode, and are therefore more efficient than system calls, which include the overhead of a trap to the kernel.

Dinkum C and C++ Libraries

Dinkum C and C++ libraries—including embedded (abridged) C++ libraries—are provided for VxWorks RTP application development. For more information about these libraries, see the Dinkum API references.

The VxWorks distribution also provides a C run-time shared library feature that is similar to that of the UNIX C run-time shared library. For information about this library, see *4.10 Using the VxWorks Run-time C Shared Library libc.so*, p.76.

For more information about C++ development, see *5. C++ Development*.

Custom Libraries

For information about creating custom user-mode libraries for applications, see *4. Static Libraries, Shared Libraries, and Plug-Ins*.

API Documentation

For detailed information about the routines available for use in applications, see the *VxWorks Application API Reference* and the Dinkumware library references.

3.3.4 Reducing Executable File Size With the strip Facility

For production systems, it may be useful to strip executables. The **strip***arch* utility can be used with the **--strip-unneeded** and **--strip-debug** (or **-d**) options for any RTP executables.

The **--strip-all** (or **-s**) option should only be used with absolutely-linked executables. For more information in this regard, see *Stripping Absolutely-Linked RTP Executables*, p.27 and *Caveat With Regard to Stripped Executables*, p.41). For information about absolutely-linked RTP executables and the overlapped virtual memory model, see *2.5 About VxWorks RTP Virtual Memory Models*, p.16 and *2.6 Using the Overlapped RTP Virtual Memory Model*, p.19.

3.3.5 RTP Applications and Multitasking

If an application is multi-threaded (has multiple tasks), the developer must ensure that the **main( )** routine task starts all the other tasks.

VxWorks can run one or more applications simultaneously. Each application can spawn multiple tasks, as well as other processes. Application tasks are scheduled by the kernel, independently of the process within which they execute—processes themselves are not scheduled. In one sense, processes can be viewed as containers for tasks.

In developing systems in which multiple applications will run, developers should therefore consider:

- the priorities of tasks running in all the different processes
- any task synchronization requirements *between* processes as well as *within* processes

For information about task priorities and synchronization, see *6.2 Tasks and Multitasking*, p.85 and *6.8 Intertask and Interprocess Communication*, p.107.

3.3.6 Checking for Required Kernel Support

VxWorks is a highly configurable operating system. Because RTP applications are built independently of the operating system, 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, important 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. If an API requires a facility that is not configured into the kernel, an **errno** value of **ENOSYS** is returned when the API is called.

The **syscallPresent( )** routine can also be used to determine whether or not a particular system call is present in the system.

3.3.7 Using Hook Routines

For information about using hook routines, which are called during the execution of **rtpSpawn( )** and **rtpDelete( )**, see the VxWorks API reference for **rtpHookLib** and *6.4.9 Tasking Extensions: Hook Routines*, p.100.

3.3.8 Developing C++ Applications

For information about developing C++ applications, see *5. C++ Development*.

3.3.9 Using POSIX Facilities

For information about POSIX APIs available with VxWorks, and a comparison of native VxWorks and POSIX APIs, see *7. POSIX Facilities*.

3.3.10 Building RTP Applications

RTP Applications can be built using Wind River Workbench or the command-line VxWorks development environment. For information about these facilities, see the *Wind River Workbench by Example* guide and the *VxWorks Command-Line Tools User's Guide*, respectively.

Note that applications that make use of share libraries or plug-ins must be built as dynamic executables. See *4.6 Common Development Facilities*, p. 59 for information about dynamic executables.

3.4 Developing Static Libraries, Shared Libraries and Plug-Ins

For information about developing libraries and plug-ins for use with RTP applications, see *4. Static Libraries, Shared Libraries, and Plug-Ins*.

3.5 Creating and Using Shared Data Regions

Shared data regions provide a means for RTP applications to share a common area of memory with each other. Processes otherwise provide for full separation and protection of all processes from one another.

The shared data region facility provides no inherent facility for mutual exclusion. Applications must use standard mutual exclusion mechanisms—such as public semaphores—to ensure controlled access to a shared data region resources (see *6.8 Intertask and Interprocess Communication*, p. 107).

For systems without an MMU enabled, shared data regions simply provide a standard programming model and separation of data for the applications, but without the protection provided by an MMU.

A shared data region is a single block of contiguous virtual memory. Any type of memory can be shared, such as RAM, memory-mapped I/O, flash, or VME.

Multiple shared data regions can be created with different characteristics and different users.

Common uses of a shared data region would include video data from buffers.

The **sdLib** shared data region library provides the facilities for the following activities:

- Creating a shared data region.
- Opening the region.
- Mapping the region to a process' memory context so that it can be accessed.
- Changing the protection attributes of a region that has been mapped.
- Un-mapping the region when a process no longer needs to access it.
- Deleting the region when no processes are attached to it.

Operations on shared data regions are not restricted to applications—kernel tasks may also perform these operations.

Shared data regions use memory resources from both the kernel's and the application's memory space. The kernel's heap is used to allocate the shared data object. The physical memory for the shared data region is allocated from the global physical page pool.

When a shared data region is created, it must be named. The name is global to the system, and provides the means by which applications identify regions to be shared.

Shared data regions can be created in systems with and without MMU support.

Also see *6.11 Shared Data Structures*, p. 109 and *7.19.3 Shared Memory Objects*, p. 199

3.5.1 Configuring VxWorks for Shared Data Regions

For applications to be able to use shared data region facilities, the **INCLUDE_SHARED_DATA** component must be included in VxWorks.

3.5.2 Creating Shared Data Regions

Shared data regions are created with **sdCreate( )**. They can be created by an application, or from a kernel task such as the shell. The region is automatically mapped into the creator's memory context. The **sdOpen( )** routine also creates and maps a region—if the region name used in the call does not exist in the system.

WARNING: If the shell is used to create shared data regions, the optional physical address parameter should not be used with architectures for which the **PHYS_ADDRESS** type is 64 bits. The shell passes the physical address parameter as 32 bits regardless. If it should actually be 64 bits, the arguments will not be aligned with the proper registers and unpredictable behavior will result. See the *VxWorks Architecture Supplement* for the processor in question for more information.

The creation routines take parameters that define the name of the region, its size and physical address, MMU attributes, and two options that govern the regions persistence and availability to other processes.

The MMU attribute options define access permissions and the cache option for the process' page manager:

- read-only
- read/write
- read/execute
- read/write/execute
- cache write-through, cache copy-back, or cache off

By default, the creator process always gets read and write permissions for the region, regardless of the permissions set with the creation call, which affect all client processes. The creator, can however, change its own permissions with **sdProtect( )**. See *Changing Shared Data Region Protection Attributes*, p. 40.

The **SD_LINGER** creation option provides for the persistence of the region after all processes have unmapped from it—the default behavior is for it to cease to exist, all of its resources being reclaimed by the system. The second option, **SD_PRIVATE**, restricts the accessibility of the region to the process that created it. This can be useful, for example, for restricting memory-mapped I/O to a single application.

3.5.3 Accessing Shared Data Regions

A shared data region is automatically opened and mapped to the process that created it, regardless of whether the **sdCreate( )** or **sdOpen( )** routine was used.

A client process must use the region's name with **sdOpen( )** to access the region. The region name can be hard-coded into the client process' application, or transmitted to the client using IPC mechanisms.

Mutual exclusion mechanisms should be used to ensure that only one application can access the same shared data region at a time. The **sdLib** library does not provide any mechanisms for doing so automatically. For more information about mutual exclusion, see *6.8 Intertask and Interprocess Communication*, p. 107.

For information about accessing shared data regions from interrupt service routines (ISRs), see the *VxWorks Kernel Programmer's Guide: Multitasking*.

Changing Shared Data Region Protection Attributes

The MMU attributes of a shared data region can be changed with **sdProtect( )**. The change can only be to a sub-set of the attributes defined when the region was created. For example, if a region was created with only read and write permissions, these can only be changed to read-only and no access, and not expanded to other permissions. In addition, the changes are made only for the caller's process; they do not affect the permissions of other processes.

A set of macros is provided with the library for common sets of MMU attribute combinations.

3.5.4 Deleting Shared Data Regions

Shared data regions can be deleted explicitly and automatically. However, deletion of regions is restricted by various conditions, including how the region was created, and if any processes are attached to it.

If a shared data region was created without the **SD_LINGER** option, the region is deleted if:

- Only one process is mapped to the region, and its application calls **sdUnmap( )**.
- Only one process is mapped to the region, and the process exits.

If a shared data region is created with the **SD_LINGER** option, it is never deleted implicitly. The region is only deleted if **sdDelete( )** is called on it after all clients have unmapped it.

3.6 Executing RTP Applications

Because a process is an instance of a program in execution, starting and terminating an application involves creating and deleting a process. A process must be spawned in order to initiate execution of an application; when the application exits, the process terminates. Processes may also be terminated explicitly.

Processes provide the execution environment for applications. They are started with **rtpSpawn( )**. The initial task for any application is created automatically in the create phase of the **rtpSpawn( )** call. This initial task provides the context within which **main( )** is called.

Caveat With Regard to Stripped Executables

Executables that have been stripped of their relocation information will not run on a system configured with the flat virtual memory model (the default). The launch will fail—silently if initiated from the shell's C interpreter. The error detection and reporting facility can be used to display the reason for failure, as follows (with output abbreviated for purposes of clarity):

```
-> edrShow
[…]
rtpLoadAndRun(): RTP 0x1415010 Init Task exiting. errno = 0xba006e […]

-> printErrno 0xba006e
errno = 0xba006e : S_loadRtpLib_NO_RELOCATION_SECTION.
```

Executables should only be stripped of their relocation information if they are built as absolutely-linked executables and run on a system that is properly configured for the overlapped virtual memory model.

For information about these topics, see *2.5 About VxWorks RTP Virtual Memory Models*, p.16 and *2.6 Using the Overlapped RTP Virtual Memory Model*, p.19; *3.3.4 Reducing Executable File Size With the strip Facility*, p.36; and *11. Error Detection and Reporting*.

Starting an RTP Application

An RTP application can be started and terminated interactively, programmatically, and automatically with various facilities that act on processes.

An application can be started by:

- a user from Workbench
- a user from the shell with **rtpSp** (for the C interpreter) or **rtp exec** (for the command interpreter)
- other applications or from the kernel with **rtpSpawn( )**
- one of the startup facilities that runs applications automatically at boot time

For more information, see *3.6.1 Running Applications Interactively*, p.42 and *3.6.2 Running Applications Automatically*, p.44.

Stopping an RTP Application

RTP applications terminate automatically when the program's **main( )** routine returns. They can also be terminated explicitly.

Automatic Termination

By default, a process is terminated when the **main( )** routine returns, because the C compiler automatically inserts an **exit( )** call at the end of **main( )**. This is undesirable behavior if **main( )** spawns other tasks, because terminating the process deletes all the tasks that were running in it. To prevent this from happening, any application that uses **main( )** to spawn tasks can call **taskExit( )** instead of **return( )** as the last statement in the **main( )** routine. When **main( )** includes **taskExit( )** as its last call, the process' initial task can exit without the kernel automatically terminating the process.

Explicit Termination

A process can explicitly be terminated when a task does either of the following:

- Calls **exit( )** to terminate the process in which it is are running, regardless of whether or not other tasks are running in the process.
- Calls the **kill( )** routine to terminate the specified process (using the process ID).

Terminating processes—either programmatically or by interactive user command—can be used as a means to update or replace application code. Once the process is stopped, the application code can be replaced, and the process started again using the new executable.

Storing Application Executables

Application executables can be stored in the VxWorks ROMFS file system on the target system, on the host development system, or on any other file system accessible to the target system (another workstation on a network, for example).

Various combinations of startup mechanisms and storage locations can be used for developing systems and for deployed products. For example, storing application executables on the host system and using the kernel shell to run them is ideal for the early phases of development because of the ease of application re-compilation and of starting applications. Final products, on the other hand, can be configured and built so that applications are bundled with the operating system, and started automatically when the system boots, all independently of humans, hosts, and hard drives.

⚠ **CAUTION:** The error **S_rtp_INVALID_FILE** is generated when the path and name of the RTP executable is not provided, or when the executable cannot be found using the indicated path. RTP executable files are accessed and loaded from the VxWorks target. Therefore, the path to the executable file must be valid from the point of view of the target itself. Correctly specifying the path may involve including the proper device name as part of the path. For example:

host:d:/my.vxe

3.6.1 Running Applications Interactively

Running applications interactively is obviously most desirable for the development environment, but it can also be used to run special applications on deployed systems that are otherwise not run as part of normal system operation (for diagnostic purposes, for example). In the latter case, it might be advantageous to store auxiliary applications in ROMFS; see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50.

Starting Applications

From the shell, applications can be started with shell command variants of the **rtpSpawn( )** routine.

Using the traditional C interpreter, the **rtpSp** command is used as follows:

```
rtpSp "host:c:/myInstallDir/vxworks-6.1/target/usr/root/PPC32diab/bin/myVxApp.vxe first second third"
```

In this example, a process is started to run the application **myVxApp.vxe**, which is stored on the host system in **c:\myInstallDir\vxworks-6.*x*\target\usr\root\PPC32diab\bin**. The application takes command-line arguments, and in this case they are first, second, and third. Additional arguments can also be used to specify the initial task priority, stack size, and other **rtpSpawn( )** options.

Note that some types of connections between the target and host require modifiers to the pathname (NFS is transparent; FTP requires **hostname:** before the path if it is not on the same system from which VxWorks was booted; the VxWorks simulator requires a **host:** prefix; and so on).

Using the shell's command interpreter, the application can be started in two different ways, either directly specifying the path and name of the executable file and the arguments (like with a UNIX shell):

```
host:c:/myInstallDir/vxworks-6.1/target/usr/root/PPC32diab/bin/myVxApp.vxe first second third
```

Or, the application can be started with the **rtp exec** command:

```
rtp exec host:c:/myInstallDir/vxworks-6.1/target/usr/root/PPC32diab/bin/myVxApp.vxe first second third
```

Note that you must use forward-slashes as path delimiters with the shell, even for files on Windows hosts. The shell does not work with back-slash delimiters.

Regardless of how the process is spawned, the application runs in exactly the same manner.

Note that you can switch from the C interpreter to the command interpreter with the **cmd** command; and from the command interpreter to the C interpreter with the **C** command. The command interpreter **rtp exec** command has options that provide more control over the execution of an application.

Terminating Applications

An application can be stopped by terminating the process in which it is running.

Using the shell's command interpreter, a process can be killed with the full **rtp delete** command, or with either of the command shell aliases **kill** and **rtpd**. It can also be killed with **CTRL+C** if it is running in the foreground (that is, it has not been started using an ampersand after the **rtp exec** command and the name of the executable—which is similar to UNIX shell command syntax for running applications in the background).

With the shell's C interpreter, a process can be terminated with **kill( )** or **rtpDelete( )**.

For a description of all the ways in which a process can be terminated, see *2.2.3 RTP Termination*, p.6.

And, of course, rebooting the system terminates all processes that are not configured to restart at boot time.

3.6.2 Running Applications Automatically

Running applications automatically—without user intervention—is required for many deployed systems. VxWorks applications can be started automatically in a variety of ways. In addition, application executables can be stored either on a host system—which can be useful during development even when a startup facility is in use—or they can be stored on the target itself.

The VxWorks application startup facility is designed to serve the needs of both the development environment and deployed systems.

For the development environment, the startup facility can be used interactively to specify a variety of applications to be started at boot time. The operating system does not need to be rebuilt to run different sets of applications, or to run the same applications with different arguments or process-spawn parameters (such as the priority of the initial task). That is, as long as VxWorks has been configured with the appropriate startup components, and with the components required by the applications themselves, the operating system can be completely independent and ignorant of the applications that it will run until the moment it boots and starts them. One might call this a blind-date scenario.

For deployed systems, VxWorks can be configured and built with statically defined sets of applications to run at boot time (including their arguments and process-spawn parameters). The applications can also be built into the system image using the ROMFS file system. And this scenario might be characterized as most matrimonial.

In this section, use of the startup facility is illustrated with applications that reside on the host system. For information about using ROMFS to bundle applications with the operating system, and for examples illustrating how applications in the ROMFS file system are identified for the startup facility, see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50.

Startup Facility Options

Various means can be used to identify applications to be started, as well as to provide their arguments and process-spawn parameters for the initial application task. Applications can be identified and started automatically at boot time using any of the following:

- an application startup configuration parameter
- a boot loader parameter
- a VxWorks shell script
- the **usrRtpAppInit( )** routine

The components that support this functionality are, respectively:

- **INCLUDE_RTP_APPL_INIT_STRING**

- **INCLUDE_RTP_APPL_BOOTLINE**
- **INCLUDE_RTP_APPL_INIT_CMD_SHELL_SCRIPT** (for the command interpreter; the C interpreter can also be used with other components)
- **INCLUDE_RTP_APPL_USER**

The boot loader parameter and the shell script methods can be used both interactively (without modifying the operating system) and statically. Therefore, they are equally useful for application development, and for deployed systems.

The startup configuration parameter and the **usrRtpAppInit( )** routine methods require that the operating system be re-configured and rebuilt if the developer wants to change the set of applications, application arguments, or process-spawn parameters.

There are no speed or initialization-order differences between the various means of automatic application startup. All of the startup facility components provide much the same performance.

Application Startup String Syntax

A common string syntax is used with both the startup facility configuration parameter and the boot loader parameter for identifying applications. The basic syntax is as follows:

#progPathName^arg1^arg2^arg3#progPathName...

This syntax involves only two special characters:

#

A pound sign identifies what immediately follows as the path and name of an application executable.

^

A caret delimits individual arguments (if any) to the application. A caret is not required after the final argument.

The carets are not required—spaces can be used instead—with the startup configuration parameter, but carets must be used with the boot loader parameter.

The following examples illustrate basic syntax usage:

```
#c:/apps/myVxApp.vxe
```

Starts **c:\apps\myVxApp.vxe**

```
#c:/apps/myVxApp.vxe^one^two^three
```

Starts **c:\apps\myVxApp.vxe** with the arguments **one**, **two**, **three**.

```
#c:/apps/myOtherVxApp.vxe
```

Starts **c:\apps\myOtherVxApp.vxe** without any arguments.

```
#c:/apps/myVxApp.vxe^one^two^three#c:/apps/myOtherVxApp.vxe
```

Starts both applications, the first one with its three arguments.

The startup facility also allows for specification of **rtpSpawn( )** routine parameters with additional syntax elements:

%p=*value*

Sets the priority of the initial task of the process. Priorities can be in the range of 0-255.

%**s=***value*
Sets the stack size for the initial task of the process (an integer parameter).

%**o=***value*
Sets the process options parameter.

%**t=***value*
Sets task options for the initial task of the process.

When using the boot loader parameter, the option values must be either decimal or hexadecimal numbers. When using the startup facility configuration parameter, the code is preprocessed before compilation, so symbolic constants may be used as well (for example, **VX_FP_TASK**).

The following string, for example, specifies starting **c:\apps\myVxApp.vxe** with the arguments **one**, **two**, **three**, and an initial task priority of 125; and also starting **c:\apps\myOtherVxApp.vxe** with the options value 0x10 (which is to stop the process before running in user mode):

```
#c:/apps/myVxApp.vxe p=125^one^two^three#c:/apps/myOtherVxApp.vxe %o=0x10
```

If the **rtpSpawn( )** options are not set, the following defaults apply: the initial task priority is 220; the initial task stack size is 64 Kb; the options value is zero; and the initial task option is **VX_FP_TASK**.

The maximum size of the string used in the assignment is 160 bytes, inclusive of names, parameters, and delimiters. No spaces can be used in the assignment, so application files should not be put in host directories for which the path includes spaces.

Specifying Applications with a Startup Configuration Parameter

Applications can be specified with the **RTP_APPL_INIT_STRING** parameter of the **INCLUDE_RTP_APPL_INIT_STRING** component.

The identification string must use the syntax described in *Application Startup String Syntax*, p.45. And the operating system must be rebuilt thereafter.

Specifying Applications with a Boot Loader Parameter

The VxWorks boot loader includes a parameter—the **s** parameter—that can be used to identify applications that should be started automatically at boot time, as well as to identify shell scripts to be executed.[1] (For information about the boot loader, see the *VxWorks Kernel Programmer's Guide: Boot Loader*.)

Applications can be specified both interactively and statically with the **s** parameter. In either case, the parameter is set to the path and name of one or more executables and their arguments (if any), as well as to the applications' process-spawn parameters (optionally). The special syntax described above is used to describe the applications (see *Application Startup String Syntax*, p.45).

This functionality is provided with the **INCLUDE_RTP_APPL_BOOTLINE** component.

[1] In versions of VxWorks 5.*x*, the boot loader **s** parameter was used solely to specify a shell script.

Note that the boot loader **s** parameter serves a dual purpose: to dispatch script file names to the shell, and to dispatch application startup strings to the startup facility. Script files used with the **s** parameter can only contain C interpreter commands; they cannot include startup facility syntax (also see *Specifying Applications with a VxWorks Shell Script*, p. 47).

If the boot parameter is used to identify a startup script to be run at boot time as well as applications, it must be listed before any applications. For example, to run the startup script file **myScript** and **myVxApp.vxe** (with three arguments), the following sequence would be required:

```
myScript#c:/apps/myVxApp.vxe^one^two^three
```

The assignment in the boot console window would look like this:

```
startup script (s)    : myScript#c:/apps/myVxApp.vxe^one^two^three
```

The interactively-defined boot-loader parameters are saved in the target's boot media, so that the application is started automatically with each reboot.

For the VxWorks simulator, the boot parameter assignments are saved in a special file on the host system, in the same directory as the image that was booted, for example, *installDir***/vxworks-6.***x***/target/proj/simpc_diab/default/nvram.vxWorks0**. The number appended to the file name is processor ID number—the default for the first instance of the simulator is zero.

For a hardware target, applications can be identified statically. The **DEFAULT_BOOT_LINE** parameter of the **INCLUDE_RTP_APPL_BOOTLINE** component can be set to an identification string using the same syntax as the interactive method. Of course, the operating system must be rebuilt thereafter.

Specifying Applications with a VxWorks Shell Script

Applications can be started automatically with a VxWorks shell script. Different methods must be used, however, depending on whether the shell script uses command interpreter or C interpreter commands.

If the shell script is written for the command interpreter, applications can be identified statically.

The **RTP_APPL_CMD_SCRIPT_FILE** parameter of the **INCLUDE_RTP_APPL_INIT_CMD_SHELL_SCRIPT** component can be set to the location of the shell script file.

A startup shell script for the command interpreter might, for example, contain the following line:

```
rtp exec c:/apps/myVxApp.vxe first second third
```

Note that for Windows hosts you must use either forward-slashes or double back-slashes instead of single back-slashes as path delimiters with the shell.

If a shell script is written for the C interpreter, it can be identified interactively using the boot loader **s** parameter— in a manner similar to applications—using a sub-set of the same string syntax. A shell script for the C interpreter can also be identified statically with the **DEFAULT_BOOT_LINE** parameter of the **INCLUDE_RTP_APPL_BOOTLINE** component. (See *Specifying Applications with a Boot Loader Parameter*, p. 46 and *Application Startup String Syntax*, p. 45.)

The operating system must be configured with the kernel shell and the C interpreter components for use with C interpreter shell scripts (see the *VxWorks Kernel Programmer's Guide: Target Tools*).

A startup shell script file for the C interpreter could contain the following line:

```
rtpSp "c:/apps/myVxApp.vxe first second third"
```

With the shell script file **c:\scripts\myVxScript**, the boot loader **s** parameter would be set interactively at the boot console as follows:

```
startup script (s)    : c:/scripts/myVxScript
```

Note that shell scripts can be stored in ROMFS for use in deployed systems (see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50).

Specifying Applications with usrRtpAppInit( )

The VxWorks application startup facility can be used in conjunction with the **usrRtpAppInit( )** initialization routine to start applications automatically when VxWorks boots. In order to use this method, VxWorks must be configured with the **INCLUDE_RTP_APPL_USER** component.

For each application you wish to start, add an **rtpSpawn( )** call and associated code to the **usrRtpAppInit( )** routine stub, which is located in *installDir***/vxworks-6.***x***/target/proj/***projDir***/usrRtpAppInit.c**.

The following example starts an application called **myVxApp**, with three arguments:

```
void usrRtpAppInit (void)
    {
    char * vxeName = "c:/vxApps/myVxApp/PPC32diab/myVxApp.vxe";
    char * argv[5];
    RTP_ID rtpId = NULL;

    /* set the application's arguments */

    argv[0] = vxeName;
    argv[1] = "first";
    argv[2] = "second";
    argv[3] = "third";
    argv[4] = NULL;

    /* Spawn the RTP. No environment variables are passed */

    if ((rtpId = rtpSpawn (vxeName, argv, NULL, 220, 0x10000, 0, 0)) == NULL)
        {
        printErr ("Impossible to start myVxApp application (errno = %#x)",
                  errno);
        }
    }
```

Note that in this example, the **myVxApp.vxe** application executable is stored on the host system in **c:\vxApps\myVxApp\PPC32diab**.

The executable could also be stored in ROMFS on the target system, in which case the assignment statement that identifies the executable would look like this:

```
char * vxeName = "/romfs/myVxApp.vxe";
```

For information about bundling applications with the system image in ROMFS, see *3.7 Bundling RTP Applications in a System using ROMFS*, p.50.

3.6.3 Spawning Tasks and Executing Routines in an RTP Application

The VxWorks kernel shell provides facilities for spawning tasks and for calling application routines in a real-time process. These facilities are particularly useful for debugging RTP applications.

For more information, see the *VxWorks Kernel Programmer's Guide: Target Tools* and the entries for the **task spawn** and **func call** commands in the *VxWorks Kernel Shell Command Reference*.

In addition, note that the kernel shell provides facilities for monitoring system calls. For more information, see the *VxWorks Kernel Programmer's Guide: Target Tools*, the **syscall monitor** entry in the *VxWorks Kernel Shell Command Reference*, and the **sysCallMonitor( )** entry in the *VxWorks Kernel API Reference*.

3.6.4 Applications and Symbol Registration

Symbol registration is the process of storing symbols in a symbol table that is associated with a given process. Symbol registration depends on how an application is started:

- When an application is started from the shell, symbols are registered automatically, as is most convenient for a development environment.
- When an application is started programmatically—that is, with a call to **rtpSpawn( )**—symbols are not registered by default. This saves on memory at startup time, which is useful for deployed systems.

The registration policy for a shared library is, by default, the same as the one for the application that loads the shared library.

The default symbol-registration policy for a given method of starting an application can be overridden, whether the application is started interactively or programmatically.

The shell's command interpreter provides the **rtp exec** options **–g** for global symbols, **-a** for all symbols (global and local), and **–z** for zero symbols. For example:

```
rtp exec -a /folk/pad/tmp/myVxApp/ppc/myVxApp.vxe one two three &
```

The **rtp symbols override** command has the options **–g** for global symbols, **-a** for all symbols (global and local), and **–c** to cancel the policy override.

The **rtpSpawn( )** options parameter **RTP_GLOBAL_SYMBOLS** (0x01) and **RTP_ALL_SYMBOLS** (0x03) can be used to load global symbols, or global and local symbols (respectively).

The shell's C interpreter command **rtpSp( )** provides the same options with the **rtpSpOptions** variable.

Symbols can also be registered and unregistered interactively from the shell, which is useful for applications that have been started without symbol registration. For example:

```
rtp symbols add –a –s 0x10000 –f /romfs/bin/myApp.vxe
rtp symbols remove –l –s 0x10000
rtp symbols help
```

Note that when the flat virtual memory model is in use symbols should not be stripped from executable files (**.vxe** files) because they are relocatable. It is possible to strip symbols from absolutely-linked executable files (intended to be used with the overlapped virtual memory model) because they are not relocated but it makes debugging them more difficult. The same apply to run-time shared library files (**.so** files)

3.7 Bundling RTP Applications in a System using ROMFS

The ROMFS facility provides the ability to bundle RTP applications—or any other files for that matter—with the operating system. No other file system is required to store applications; and no storage media is required beyond that used for the system image itself.

RTP applications do not need to be built in any special way for use with ROMFS. As always, they are built independently of the operating system and ROMFS itself. However, when they are added to a ROMFS directory on the host system and VxWorks is rebuilt, a single system image is that includes both the VxWorks and the application executables is created. ROMFS can be used to bundle applications in either a system image loaded by the boot loader, or in a self-loading image (for information about VxWorks image types, see the *VxWorks Kernel Programmer's Guide: Kernel Facilities and Kernel Configuration*).

When the system boots, the ROMFS file system and the application executables are loaded with the kernel. Applications and operating system can therefore be deployed as a single unit. And coupled with an automated startup facility (see *3.6 Executing RTP Applications*, p. 40), ROMFS provides the ability to create fully autonomous, multi-process systems.

This section provides information about using ROMFS to store process-based applications with the VxWorks operating system in a single system image. For general information about ROMFS, see *10.8 Read-Only Memory File System: ROMFS*, p. 273.

3.7.1 Configuring VxWorks with ROMFS

VxWorks must be configured with the **INCLUDE_ROMFS** component to provide ROMFS facilities.

3.7.2 Building a System With ROMFS and Applications

Configuring VxWorks with ROMFS and applications involves several simple steps. A ROMFS directory must be created in the BSP directory on the host system, application files must be copied into the directory, and then VxWorks must be rebuilt. For example:

```
cd c:\myInstallDir\vxworks-6.1\target\proj\wrSbc8260_diab
mkdir romfs
copy c:\myInstallDir\vxworks-6.1\target\usr\root\PPC32diab\bin\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 ROMFS_DIR** macro used to identify where it is in the build command. For example:

```
    make TOOL=diab ROMFS_DIR="c:\allMyVxAppExes"
```

Note that any files located in the **romfs** directory are included in the system image, regardless of whether or not they are application executables.

3.7.3 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 traditional **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, **myVxApp.vxe** copied to it, and the system rebuilt and booted, then using **ls** from the shell looks like this:

```
[vxWorks]# ls /romfs
/romfs/.
/romfs/..
/romfs/myVxApp.vxe
```

And **myVxApp.vxe** can also be accessed at run time as **/romfs/myVxApp.vxe** by any other applications running on the target, or by kernel modules (kernel-based applications).

3.7.4 Using ROMFS to Start Applications Automatically

ROMFS can be used with any of the application startup mechanisms simply by referencing the local copy of the application executables. See *3.6.2 Running Applications Automatically*, p. 44 for information about the various ways in which applications can be run automatically when VxWorks boots.

4
Static Libraries, Shared Libraries, and Plug-Ins

4.1 Introduction

Custom static libraries, shared libraries, and plug-ins can be created for use with RTP applications. This chapter describes their features, comparative advantages and uses, development procedures, and debugging methods. It also describes the C run-time shared library provided with the VxWorks distribution, which can be used with applications as an alternative to statically linking them to C libraries.

4.2 About Static Libraries, Shared Libraries, and Plug-ins

Static libraries are linked to an application at compile time. They are also referred to as *archives*. Shared libraries are dynamically linked to an application when the application is loaded. They are also referred to as dynamically-linked libraries, or

DLLs. Plug-ins are similar in most ways to shared libraries, except that they are loaded on demand (programmatically) by the application instead of automatically. Both shared libraries and plug-ins are referred to generically as *dynamic shared objects*.

Static libraries and shared libraries perform essentially the same function. The key differences in their utility are as follows:

- Only the elements of a static library that are required by an application (that is, specific **.o** object files within the archive) are linked with the application. The entire library does not necessarily become part of the system. If multiple applications (*n* number) in a system use the same library elements, however, those elements are duplicated (*n* times) in the system—in both the storage media and system memory.

- The dynamic linker loads the entire shared library when any part of it is required by an application. (As with a **.o** object file, a shared library **.so** file is an indivisible unit.) If multiple applications in a system need the shared library, however, they share a single copy. The library code is not duplicated in the system.

⚠ **CAUTION:** Applications that make use of shared libraries or plug-ins must be built as dynamic executables to include a dynamic linker in their image. The dynamic linker carries out the binding of the dynamic shared object and application at run time. For more information in this regard, see *4.8.9 Developing RTP Applications That Use Shared Libraries*, p.67 and *4.9.3 Developing RTP Applications That Use Plug-Ins*, p.73.

➡ **NOTE:** Shared libraries and plug-ins are not supported for the Coldfire architecture.

Advantages and Disadvantages of Shared Libraries and Plug-Ins

Both dynamic shared objects—Shared libraries and plug-ins—can provide advantages of footprint reduction, flexibility, and efficiency, as follows (*shared library* is used to refer to both here, except where *plug-in* is used specifically):

- The storage requirements of a system can be reduced because the applications that rely on a shared library are smaller than if they were each linked with a static library. Only one set of the required library routines is needed, and they are provided by the run-time library file itself. The extent to which shared libraries make efficient use of mass storage and memory depends primarily on how many applications are using how much of a shared library, and if the applications are running at the same time.

- Plug-ins provide flexibility in allowing for dynamic configuration of applications—they are loaded only when needed by an application (programmatically on demand).

- Shared libraries are efficient because their code requires fewer relocations than standard code when loaded into RAM. Moreover, lazy binding (also known as *lazy relocation* or *deferred binding*) allows for linking only those functions that are required.

At the same time, shared libraries use position-independent code (PIC), which is slightly larger than standard code, and PIC accesses to data are usually somewhat slower than non-PIC accesses because of the extra indirection through the global offset table (GOT). This has more impact on some architectures than on others. Usually the difference is on the order of a fraction of a percent, but if a time-sensitive code path in a shared library contains many references to global functions, global data or constant data, there may be a measurable performance penalty.

If *lazy binding* is used with shared libraries, it introduces non-deterministic behavior. (For information about lazy binding, see *4.8.8 Using Lazy Binding With Shared Libraries*, p.67 and *Using Lazy Binding With Plug-ins*, p.74.)

The startup cost of shared libraries makes up the largest efficiency cost (as is the case with UNIX). It is also greater because of more complex memory setup and more I/O (file accesses) than for static executables.

In summary, shared libraries are most useful when the following are true:

- Many programs require a few libraries.
- Many programs that use libraries run at the same time.
- Libraries are discrete functional units with little unused code.
- Library code represents a substantial amount of total code.

Conversely, it is not advisable to use shared libraries when only one application runs at a time, or when applications make use of only a small portion of the routines provided by the library.

Additional Considerations

There are a number of other considerations that may affect whether to use shared libraries (or plug-ins):

- Assembly code that refers to global functions or data must be converted by hand into PIC in order to port it to a shared library.
- The relocation process only affects the data section of a shared library. Read-only data identified with the **const** C keyword are therefore gathered with the data section and not with the text section to allow a relocation per executable. This means that read-only data used in shared libraries are not protected against erroneous write operations at run-time.
- Code that has not been compiled as PIC will not work in a shared library. Code that has been compiled as PIC does not work in an executable program, even if the executable program is dynamic. This is because function prologues in code compiled as PIC are edited by the dynamic linker in shared objects.
- All constructors in a shared library are executed together, hence a constructor with high priority in one shared library may be executed after a constructor with low priority in another shared library loaded later than the first one. All shared library constructors are executed at the priority level of the dynamic linker's constructor from the point of view of the executable program.
- Dynamic shared objects are not cached (they do not *linger*) if no currently executing program is using them. There is, therefore, extra processor overhead if a shared library is loaded and unloaded frequently.

- There is a limit on the number of concurrent shared libraries, which is 1024. This limit is imposed by the fact that the GOT table has a fixed size, so that indexing can be used to look up GOTs (which makes it fast).

⚠ **CAUTION:** There is no support for so-called *far PIC* on PowerPC. Some shared libraries require the global offset table to be larger than 16,384 entries; since this is greater than the span of a 16-bit displacement, specialized code must be used to support such libraries.

4.3 Additional Documentation

The following articles provide detailed discussions of dynamic shared objects (including recommendations for optimization) and the dynamic linker in the context of Linux development:

- Drepper, Ulrich. *How to Write Shared Libraries*. Red Hat, Inc. 2006.
- Jelinek, Jakub. *Prelink*. Red Hat, Inc. 2003.

4.4 Configuring VxWorks for Shared Libraries and Plug-ins

While shared libraries and plug-ins can only be used with RTP (user mode) applications (and not in the kernel), they do require additional kernel support for managing their use by different processes.

Shared library support is not provided by VxWorks by default. The operating system must be configured with the **INCLUDE_SHL** component.

Doing so automatically includes these components as well:

- **INCLUDE_RTP**, the main component for real-time process support
- **INCLUDE_SHARED_DATA** for storing shared library code
- **INCLUDE_RTP_HOOKS** for shared library initialization
- and various **INCLUDE_SC_***XYZ* components—for the relevant system calls

It can also be useful to include support for relevant show routines with these components:

- **INCLUDE_RTP_SHOW**
- **INCLUDE_SHL_SHOW**
- **INCLUDE_SHARED_DATA_SHOW**

Note that if you use the **INCLUDE_SHOW_ROUTINES** component, the three above are automatically added.

Configuration can be simplified through the use of component bundles. **BUNDLE_RTP_DEVELOP** and **BUNDLE_RTP_DEPLOY** provide support for shared libraries for the development systems and for deployed systems respectively (for more information, see *Component Bundles*, p.14).

For general information about configuring VxWorks for real-time processes, see & *2.3 Configuring VxWorks For Real-time Processes*, p.12.

4.5 Common Development Issues: Initialization and Termination

Development of static libraries, shared libraries, and plug-ins all share the issues of initialization and termination, which are covered below. For issues specific to development of each, see *4.7 Developing Static Libraries*, p.60, *4.8 Developing Shared Libraries*, p.60, and *4.9 Developing Plug-Ins*, p.72.

4.5.1 Library and Plug-in Initialization

A library or plug-in requires an initialization routine only if its operation requires that resources be created (such as semaphores, or a data area) before its routines are called.

If an initialization routine is required for the library (or plug-in), its prototype should follow this convention:

```
void fooLibInit (void);
```

The routine takes no arguments and returns nothing. It can be useful to use the same naming convention used for VxWorks libraries; *name***LibInit( )**, where *name* is the basename of the feature. For example, **fooLibInit( )** would be the initialization routine for **fooLib**.

The code that calls the initialization of application libraries is generated by the compiler. The **_WRS_CONSTRUCTOR** compiler macro must be used to identify the library's (or plug-in's) initialization routine (or routines), as well as the order in which they should be called. The macro takes two arguments, the name of the routine and a rank number. The routine itself makes up the body of the macro. The syntax is as follows:

```
_WRS_CONSTRUCTOR (fooLibInit, rankNumInteger)
    {
      /* body of the routine */
    }
```

The following example is of a routine that creates a mutex semaphore used to protect a scarce resource, which may be used in a transparent manner by various features of the application.

```
_WRS_CONSTRUCTOR (scarceResourceInit , 101)
    {
    /*
     * Note: a FIFO mutex is preferable to a priority-based mutex
     * since task priority should not play a role in accessing the scarce
     * resource.
     */

    if ((scarceResourceMutex = semMCreate (SEM_DELETE_SAFE | SEM_Q_FIFO |
                                     SEM_USER)) == NULL)
        EDR_USR_FATAL_INJECT (FALSE,
                    "Cannot enable task protection on scarce resource\n");
    }
```

(For information about using the error detection and reporting macro **EDR_USR_FATAL_INJECT**, see *11.7 Using Error Reporting APIs in Application Code*, p.286.)

The rank number is used by the compiler to order the initialization routines. (The rank number is referred to as a *priority* number in the compiler documentation.)

Rank numbers from 100 to 65,535 can be used—numbers below 100 are reserved for VxWorks libraries. Using a rank number below 100 does not have detrimental impact on the kernel, but may disturb or even prevent the initialization of the application environment (which involves creating resources such as the heap, semphores, and so on).

Initialization routines are called in numerical order (from lowest to highest). When assigning a rank number, consider whether or not the library (or plug-in) in question is dependent on any other application libraries that should be called before it. If so, make sure that its number is greater.

If initialization routines are assigned the same rank number, the order in which they are run is indeterminate within that rank (that is, indeterminate relative to each other).

4.5.2 C++ Initialization

Libraries or plug-ins written in C++ may require initialization of static constructors for any global objects that may be used, in addition to the initialization required for code written in C (described in *4.5.1 Library and Plug-in Initialization*, p.57).

By default, static constructors are called last, after the library's (or plug-in's) initialization routine. In addition, there is no guarantee that the library's static constructors will be called before any static constructors in the associated application's code. (Functionally, they both have the default rank of *last*, and there is no defined ordering within a rank.)

If you require that the initialization of static constructors be ordered, rank them explicitly with the **_WRS_CONSTRUCTOR** macro. However, well-written C++ should not need a specific initialization routine if the objects and methods defined by the library (or plug-in) are properly designed (using deferred initialization).

4.5.3 Handling Initialization Failures

Libraries and plug-ins should be designed to respond gracefully to initialization failures. In such cases, they should do the following:

- Check whether the **ENOSYS** errno has been set, and respond appropriately. For system calls, this errno indicates that the required support component has not been included in the kernel.
- Release all the resources that have been created or obtained by the initialization routine.
- Use the **EDR_USER_FATAL_INJECT** macro to report the error. If the system has been configured with the error detection and reporting facility, the error is

recorded in the error log (and the system otherwise responds to the error depending on how the facility has been configured). If the system has not been configured with the error detection and reporting facility, it attempts to print the message to a host console by way of a serial line. For example:

```
if (mutex = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE)) == NULL)
        {
        EDR_USR_FATAL_INJECT (FALSE, "myLib: cannot create mutex. Abort.");
        }
```

For more information, see *11.7 Using Error Reporting APIs in Application Code*, p.286.

4.5.4 Shared Library and Plug-in Termination

Shared libraries and plug-ins are removed from memory when the only (last) process making use of them exits. A plug-in can also be terminated explicitly when the only application making use of it calls **dlclose( )** on it.

Using Cleanup Routines

There is no library (or plug-in) *termination* routine facility comparable to that for initialization routines (particularly with regard to ranking). If there is a need to perform cleanup operations in addition to what occurs automatically with RTP deletion, (such as deleting kernel resources created by the library) then the **atexit( )** routine must be used. The call to **atexit( )** can be made at anytime during the life of the process, although it is preferably done by the library (or plug-in) initialization routine. Cleanup routines registered with **atexit( )** are called when **exit( )** is called. Note that if a process' task directly calls the POSIX **_exit( )** routine, none of the cleanup routines registered with **atexit( )** will be executed.

If the cleanup is specific to a task or a thread then **taskDeleteHookAdd( )** or **pthread_cleanup_push( )** should be used to register a cleanup handler (for a VxWorks task or pthread, respectively). These routines are executed in reverse order of their registration when a process is being terminated.

4.6 Common Development Facilities

There are three alternatives for developing static libraries, shared libraries, and plug-ins, as well as the applications that make use of them. The alternatives are as follows:

- Use Wind River Workbench. All of the build-related elements are created automatically as part of creating library and application projects. For information in this regard, see the *Wind River Workbench by Example* guide.
- Use the **make** build rules and macros provided with the VxWorks installation to create the appropriate makefiles, and execute the build from the command

line. For information in this regard, see the *VxWorks Command-Line Tools User's Guide*.

- Write makefiles and rules from scratch, or make use of a custom or proprietary build environment. For information in this regard, see the *VxWorks Command-Line Tools User's Guide*.

4.7 Developing Static Libraries

Static libraries (archives) are made up of routines and data that can be used by applications, just like shared libraries. When an application is linked against a static library at build time, however, the linker copies object code (in **.o** files) from the library into the executable—they are statically linked. With shared libraries, on the other hand, the linker does not perform this copy operation (instead it adds information about the name of the shared library and its run-time location into the application).

The VxWorks development environment provides simple mechanisms for building static libraries (archives), including a useful set of default makefile rules. Both Workbench and command line facilities can be used to build libraries. See *4.6 Common Development Facilities*, p.59.

4.7.1 Initialization and Termination

For information about initialization and termination of static libraries, see *4.5 Common Development Issues: Initialization and Termination*, p.57.

4.8 Developing Shared Libraries

Shared libraries are made up of routines and data that can be used by applications, just like static libraries. When an application is linked against a shared library at build time, however, the linker does not copy object code from the library into the executable—they are not statically linked. Instead it copies information about the name of the shared library (its *shared object name*) and its run-time location (if the appropriate compiler option is used) into the application. This information allows the dynamic linker to locate and load the shared library for the application automatically at run-time.

Once loaded into memory by the dynamic linker, shared libraries are held in sections of memory (shared data areas) that are accessible to all applications. Each application that uses the shared library gets its own copy of the private data, which is stored in its own memory space. When the last application that references a shared library exits, the library is removed from memory.

⚠ **CAUTION:** Applications that make use of shared libraries must be built as dynamic executables to include a dynamic linker in their image. The dynamic linker carries out the binding of the shared library and application at run time. For more information, see *4.8.9 Developing RTP Applications That Use Shared Libraries*, p.67.

NOTE: Shared libraries and plug-ins are not supported for the Coldfire architecture.

4.8.1 About Dynamic Linking

The dynamic linking feature in VxWorks is based on equivalent functionality in UNIX and related operating systems. It uses features of the UNIX-standard ELF binary executable file format, and it uses many features of the ELF ABI standards, although it is not completely ABI-compliant for technical reasons. The source code for the dynamic linker comes from NetBSD, with VxWorks-specific modifications. It provides **dlopen( )** for plug-ins, and other standard features.

Dynamic Linker

An application that is built as a dynamic executable contains a *dynamic linker* library that provides code to locate, read and edit dynamic shared objects at run-time (unlike UNIX, in which the dynamic linker is itself a shared library). The dynamic linker contains a constructor function that schedules its initialization at a very early point in the execution of process (during its instantiation phase). It reads a list of shared libraries and other information about the executable file and uses that information to make a list of shared libraries that it will load. As it reads each shared library, it looks for more of this dynamic information, so that eventually it has loaded all of the code and data that is required by the program and its libraries. The dynamic linker makes special arrangements to share code between processes, placing shared code in a shared memory region. The dynamic linker allocates its memory resources from shared data regions and additional pages of memory allocated on demand—and not from process memory—so that the use of process memory is predictable.

Position Independent Code: PIC

Dynamic shared objects are compiled in a special way, into position-independent code (PIC). This type of code is designed so that it requires relatively few changes to accommodate different load addresses. A table of indirections called a global offset table (GOT) is used to access all global functions and data. Each process that uses a given dynamic shared object has a private copy of the library's GOT, and that private GOT contains pointers to shared code and data, and to private data. When PIC must use the value of a variable, it fetches the pointer to that variable from the GOT and de-references it. This means that when code from a shared object is shared across processes, the same code can fetch different copies of the analogous variable. The dynamic linker is responsible for initializing and maintaining the GOT.

4.8.2 Configuring VxWorks for Shared Libraries

VxWorks must be configured with support for shared libraries. For information in this regard, see *4.4 Configuring VxWorks for Shared Libraries and Plug-ins*, p.56

4.8.3 Initialization and Termination

For information about initialization and termination of shared libraries, see *4.5 Common Development Issues: Initialization and Termination*, p.57.

4.8.4 About Shared Library Names and ELF Records

In order for the dynamic linker to determine that an RTP application requires a shared library, the application must be built in such a way that the executable includes the name of the shared library.

The name of a shared library—it's *shared object name*—must initially be defined when the shared library itself is built. This creates an ELF **SONAME** record with the shared object name in the library's binary file. A shared object name is therefore often referred to simply as an *soname*.

The shared object name is added to an application executable when the application is built as a dynamic object and linked against the shared library at build time. This creates an ELF **NEEDED** record, which includes the name originally defined in the library's **SONAME** record. One **NEEDED** record is created for each shared library against which the application is linked.

The application's **NEEDED** records are used at run-time by the dynamic linker to identify the shared libraries that it requires. The dynamic linker loads shared libraries in the order in which it encounters **NEEDED** records. It executes the constructors in each shared library in reverse order of loading. (For information about the order in which the dynamic linker searches for shared libraries, see *Specifying Shared Library Locations: Options and Search Order*, p.64)

Note that dynamic shared objects (libraries and plug-ins) may also have **NEEDED** records if they depend on other dynamic shared objects.

For information about the development process, see *4.8.5 Creating Shared Object Names for Shared Libraries*, p.62 and *4.8.9 Developing RTP Applications That Use Shared Libraries*, p.67. For examples of displaying ELF records (including **SONAME** and **NEEDED**), see *Using readelf to Examine Dynamic ELF Files*, p.69.

4.8.5 Creating Shared Object Names for Shared Libraries

Each shared library *must* be created with a *shared object name*, which functions as the run-time name of the library. The shared object name is used—together with other mechanisms—to locate the library at run-time, and it can also be used to identify different versions of a library.

For more information about shared object names, see *4.8.4 About Shared Library Names and ELF Records*, p.62. For information about identifying the runtime location of shared libraries, see *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64.

Note that a plug-in does not require a shared object name. For information about plug-ins, see *4.9 Developing Plug-Ins*, p.72.

Options for Defining Shared Object Names and Versions

Shared object names and versions can be defined with the following methods:

- Wind River Workbench uses the shared library file name for the shared object name by default. The **SHAREDLIB_VERSION** macro can be used to set a version number; it is not set by default.
- For the VxWorks command-line build environment, the **LIB_BASE_NAME** and **SL_VERSION** make macros are used for the name and version. By default, a version one instance of a dynamic shared library is created (that is, *libName***.so.1**).
- The compiler's **-soname** flag can also be used to set the shared object name.

When the library is built, the shared object name is stored in an ELF **SONAME** record. For information about using versions of shared libraries, see *4.8.6 Using Different Versions of Shared Libraries*, p.63.

Match Shared Object Names and Shared Library File Names

The shared object name must match the name of the shared library object file itself, less the version extension. That is, the base name and **.so** extension must match. In the following example, the **-soname** compiler option identifies the runtime name of the library as **libMyFoo.so.1**, which is also thereby defined as version one of the library created with the output file **libMyFoo.so**:

```
dplus -tPPCEH:rtp -Xansi -XO -DCPU=PPC32 -DTOOL_FAMILY=diab -DTOOL=diab -Xpic
-Xswitch-table-off -Wl, -Xshared -Wl, -Xdynamic -soname=libMyFoo.so.1
-L$(WIND_BASE)/target/usr/lib/ppc/PPC32/common/PIC -lstlstd
PPC32diab/foo1.sho PPC32diab/foo2.sho -o libMyFoo.so
```

If the **SONAME** information and file name do not match, the dynamic linker will not be able to locate the library.

For information about displaying shared object names, see *Using readelf to Examine Dynamic ELF Files*, p.69.

4.8.6 Using Different Versions of Shared Libraries

In addition to being used by the dynamic linker to locate shared libraries at run-time, shared object names (*sonames*) can be used to identify different versions of shared libraries for use by different applications.

For example, if you need to modify **libMyFoo** to support new applications, but in ways that would make it incompatible with old ones, you can merely change the version number when the library is recompiled and link the new programs against the new version. If the original version of the run-time shared library was **libMyFoo.so.1**, then you would build the new version with the soname **libMyFoo.so.2** and link new applications against that one (which would then add this soname to the ELF **NEEDED** records). You could then, for example, install **libMyFoo.so.1**, **libMyFoo.so.2**, and both the old and new applications in a common ROMFS directory on the target, and they would all behave properly.

For more information creating shared object names, see *4.8.5 Creating Shared Object Names for Shared Libraries*, p.62.

4.8.7 Locating and Loading Shared Libraries at Run-time

The dynamic linker must be able to locate a shared library at run-time. It uses the shared object name stored in the dynamic application to identify the shared library file, but it also needs information about the location of the file (on the target system, host, or network) as well. There are a variety of mechanisms for identifying the location of shared libraries at run-time. In addition, the dynamic linker can be instructed to load a set of libraries at startup time, rather than when the application that needs them is loaded (that is, to *preload* the shared libraries).

Specifying Shared Library Locations: Options and Search Order

In conjunction with shared libraries' shared object names (in ELF **SONAME** records), the dynamic linker can use environment variables, configuration files, compiled location information, and a default location to find the shared libraries required by its applications.

After determining that required libraries have not already been loaded, the dynamic linker searches for them in the directories identified with the following mechanisms, in the order listed:

1. The VxWorks environment variable **LD_LIBRARY_PATH**.

   For more information, see *Using the LD_LIBRARY_PATH Environment Variable*, p.64.

2. The configuration file **ld.so.conf**.

   For more information, see *Using the ld.so.conf Configuration File*, p.65.

3. The ELF **RPATH** record in the application's executable (created with the **–rpath** compiler option).

   For more information, see *Using the ELF RPATH Record*, p.65.

4. The same directory as the one in which the application file is located.

   For more information, see *Using the Application Directory*, p.66.

In addition, the VxWorks **LD_PRELOAD** environment variable can be used to identify a set of libraries to load at startup time, before loading any other shared libraries. For more information in this regard, see *Pre-loading Shared Libraries*, p.66.

Note that the dynamic linker loads shared libraries in the order in which it encounters **NEEDED** records. It executes the constructors in each shared library in reverse order of loading.

Using the LD_LIBRARY_PATH Environment Variable

The **LD_LIBRARY_PATH** environment variable can be used to specify shared library locations when an application is started. If set, it is the first mechanism used by the dynamic linker for locating libraries (see *Specifying Shared Library Locations: Options and Search Order*, p.64). The **LD_LIBRARY_PATH** environment variable is useful as an alternative to the other mechanisms, or as a means to override them. For information about environment variables, see *2.2.8 RTPs and Environment Variables*, p. 10.

Using the shell's command interpreter, for example, the syntax for using **LD_LIBRARY_PATH** (in this case within the **rtp exec** command) would be as follows:

```
rtp exec -e "LD_LIBRARY_PATH=libPath1;libPath2" exePathAndName arg1 arg2...
```

Note in particular that there are no spaces within the quote-enclosed string, that the paths are identified as a semicolon-separated list; and that the one space between the string and the executable argument to **rtp exec** is optional (the shell parser removes spaces between arguments in command-line mode).

If, for example, the shared libraries were stored in ROMFS on the target in the **lib** subdirectory, the command would look quite tidy:

```
rtp exec -e "LD_LIBRARY_PATH=/romfs/lib" /romfs/app/myVxApp.vxe one two three
```

In this next example, the command (which would be typed all on one line, of course), identifies the location of **libc.so.1** as well as a custom shared library on the host system:

```
rtp exec -e "LD_LIBRARY_PATH=host:c:/myInstallDir/vxworks-6.1/target/usr/root/SIMPENTIUMdiab/bin;
host:c:/wrwb_demo/RtpAppShrdLib/lib/SIMPENTIUMdiab"
host:c:/wrwb_demo/RtpAppShrdLib/app/SIMPENTIUM/bin/myVxApp.vxe one two three
```

Note that some types of connections between the target and host require modifiers to the pathname (NFS is transparent; FTP requires **hostname:** before the path if it is not on the same system from which VxWorks was booted; the VxWorks simulator requires a **host:** prefix; and so on).

Also note that even on Windows hosts you must use forward slashes (or double backslashes) as path delimiters. This is the case even when the executable is stored on the host system.

For general information about environment variables, see *2.2.8 RTPs and Environment Variables*, p. 10.

Using the ld.so.conf Configuration File

The second-priority mechanism for identifying the location of shared libraries is the configuration file **ld.so.conf** (see *Specifying Shared Library Locations: Options and Search Order*, p. 64).

The **ld.so.conf** file simply lists paths, one per line. The pound sign (#) can be used at the left margin for comment lines. By default the dynamic linker looks for **ld.so.conf** in the same directory as the one in which the application executable is located. The location of **ld.so.conf** can also specified with the VxWorks **LD_SO_CONF** environment variable. For information about environment variables, see *2.2.8 RTPs and Environment Variables*, p. 10.

Using the ELF RPATH Record

The third-priority mechanism for identifying the location of shared libraries is the ELF **RPATH** record in the application's executable (see *Specifying Shared Library Locations: Options and Search Order*, p. 64).

The ELF **RPATH** record is created if an application is built with the **–rpath** linker option to identify the run-time locations of shared libraries. For example, the following identifies the **lib** subdirectory in the ROMFS file system as the location for shared libraries:

```
-Wl,-rpath /romfs/lib
```

NOTE: The the usage for **-rpath** is different for the Wind River Compiler and the Wind River GNU compiler. For the latter, an equal sign (=) must be used between **-rpath** and the path name.

For more information about the **-rpath** option and about how to set it with the makefile system, see the *VxWorks Command-Line Tools User's Guide* and the Wind River compiler guides.

Using the Application Directory

If none of the other mechanisms for locating a shared library have worked (see *Specifying Shared Library Locations: Options and Search Order*, p.64), the dynamic linker checks the directory in which the application itself is located.

By default, the VxWorks makefile system creates both the application executable and run-time shared library files in the same directory, which facilitates running the application during the development process. Workbench manages projects differently, and creates applications and shared libraries in different directories.

Pre-loading Shared Libraries

By default, shared libraries are loaded when an application that makes use of them is loaded. There are two mechanisms that can be used to load libraries in advance of the normal loading procedure: using the **LD_PRELOAD** environment variable and using a dummy RTP application.

Pre-Loading Shared Libraries with LD_PRELOAD

The VxWorks **LD_PRELOAD** environment variable can be used to identify a set of libraries to load at startup time, before any other shared libraries are loaded (that is, to *preload* them). The variable can be set to a semicolon-separated list of absolute paths to the shared libraries that are to be pre-loaded. For example:

```
/romfs/lib/libMyFoo.so.1;c:/proj/lib/libMyBar.so.1;/home/moimoi/proj/libMoreStuff.so.1
```

A typical use of pre-loaded shared libraries is to override definitions in shared libraries that are loaded in the normal manner. The definition of symbols by pre-loaded libraries take precedence over symbols defined by an application or any other shared library.

Note that even pre-loaded shared libraries are removed from memory immediately if no RTP application makes use of them. To keep them in memory the dummy RTP method must be used (for information in this regard, see *Pre-Loading Shared Libraries with a Dummy RTP Application*, p.67).

For information about environment variables, see *2.2.8 RTPs and Environment Variables*, p.10.

Pre-Loading Shared Libraries with a Dummy RTP Application

Startup speed can be increased by using a *dummy* RTP to load all required libraries in advance, so that the other RTP applications spend less time at startup. The dummy RTP should be built to require all the necessary libraries, should be designed to remain in the system at least until the last user of the loaded libraries has started—the reference counting mechanism in the kernel ensures that the libraries are not unloaded.

4.8.8 Using Lazy Binding With Shared Libraries

The lazy binding (also known as *lazy relocation* or *deferred binding*) option postpones the binding of shared library symbols until the application actually uses them, instead of when the library is loaded.

By default, the dynamic linker computes the addresses of all routines and data to which a dynamic shared object refers when it loads the object. The dynamic linker can save some work in computing routine addresses by delaying the computation until a routine is called for the first time. If a routine is never called, the dynamic linker does not need to compute its address. This feature can improve performance for large libraries when an application uses only a subset of the routines in the library. However, it can cause non-real-time latencies, so it is not enabled by default.

To have the dynamic linker defer binding symbols until they are first used by an application, use the following compiler option (with either compiler):

- **-Xbind-lazy**

Note that you can use VxWorks environment variables when you start an application to select or disable lazy binding, regardless of whether or not the compiler option was used with the shared libraries. Both **LD_BIND_NOW** and **LD_BIND_LAZY** override the compiler, and operate in the following manner:

- If **LD_BIND_LAZY** is set (even to nothing) binding is lazy.
- if **LD_BIND_NOW** is set to an non-empty string, binding is immediate. If it is not set, or set to nothing, binding is lazy.
- If both are set, **LD_BIND_LAZY** takes precedence.

The actual value of the variable does not matter.

For a discussion of lazy binding and plug-ins, see *Using Lazy Binding With Plug-ins*, p.74.

4.8.9 Developing RTP Applications That Use Shared Libraries

RTP applications that make use of shared libraries are sometimes referred to as dynamic applications. To create an application that can make use of a shared library, you must compile the application as a dynamic executable and link it to the required shared libraries.

Use the following compiler option to build the application as a dynamic executable:

- **-Xdynamic** for the Wind River Compiler.

- **-non-static** for the Wind River GNU compiler.

Use the **-l** linker option to link the application to each shared library that it needs.

Building the application in this manner embeds the dynamic linker within the application and creates an ELF **NEEDED** record for each shared library that is linked against (based on the shared libraries' **SONAME** records; for more information see *4.8.4 About Shared Library Names and ELF Records*, p.62).

For information about the various options available for locating a shared library at run-time—including using a compiler option—see *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64.

For general information about developing RTP applications, see *3.3 Developing RTP Applications*, p.30. For information about building applications that use shared libraries, see *4.6 Common Development Facilities*, p.59.

4.8.10 Getting Runtime Information About Shared Libraries

This section illustrates how to get information about shared libraries from the shell, using command interpreter commands.

The two commands below illustrate different ways start the **tmPthreadLib.vxe** application in the background, so that the shell is available for other commands:

```
[vxWorks *]# tmPthreadLib.vxe 2 &
Launching process 'tmPthreadLib.vxe' ...
Process 'tmPthreadLib.vxe' (process Id = 0x8109a0) launched.
Attachment number for process 'tmPthreadLib.vxe' is %1.
```

and

```
[vxWorks *]# rtp exec -e "LD_LIBRARY_PATH=/romfs/lib" tmPthreadLib.vxe 2 &
Launching process 'tmPthreadLib.vxe' ...
Process 'tmPthreadLib.vxe' (process Id = 0x807c90) launched.
Attachment number for process 'tmPthreadLib.vxe' is %1.
```

The **rtp** command can then be used to display information about processes. In this case it shows information about the process started with the first of the two commands above.

```
[vxWorks *]# rtp

       NAME              ID          STATE      ENTRY ADDR  OPTIONS   TASK CNT
-------------------- ---------- --------------- ---------- ---------- --------
./tmPthreadLib.vxe     0x8109a0 STATE_NORMAL 0x10002360         0x1        1
```

The **shl** command displays information about shared libraries. The **REF CNT** column provides information about the number of clients per library. The < symbol to the left of the shared library name indicates that the full path is not displayed. In this case, **libc.so.1** is not in the same place as **threadLibTest.so.1**; it is in the same directory as the executable.

```
[vxWorks *]# shl

      SHL NAME           ID      TEXT ADDR  TEXT SIZE  DATA SIZE  REF CNT
-------------------- ---------- ---------- ---------- ---------- -------
< threadLibTest.so.1    0x30000 0x10031000     0x979c     0x6334       1
./libc.so.1             0x40000 0x10043000    0x5fe24    0x2550c       1
```

Note that the **shl info** command will provide the full path.

4.8.11 Debugging Problems With Shared Library Use

This section describes common problems with the interaction between shared libraries and the applications that use them, as well as a tool for examining ELF files.

Shared Library Not Found

Failures related to the inability of the application to locate **libc.so.1** or some other run-time share library would manifest themselves from the shell as follows:

```
[vxWorks *]# tmPthreadLib.vxe 2 &
Launching process 'tmPthreadLib.vxe' ...
Process 'tmPthreadLib.vxe' (process Id = 0x811000) launched.
Attachment number for process 'tmPthreadLib.vxe' is %1.
Shared object "libc.so.1" not found
```

When a shared library cannot be found, make sure that its location has been correctly identified or that it resides in the same location as the executable (*4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64). If the shared libraries are not stored on the target, also make sure that they are accessible from the target.

Incorrectly Started Application

If an application is started with the incorrect assignment of **argv[0]**, or no assignment at all, the behavior of any associated shared libraries can be impaired. The dynamic linker uses **argv[0]** to uniquely identify the executable, and if it is incorrectly defined, the linker may not be able to match the executable correctly with shared libraries. For example, if an application is started more than once without **argv[0]** being specified, a shared library may be reloaded each time; or if the paths are missing for executables with the same filename but different locations, the wrong shared library may be loaded for one or more of the executables.

This issue applies only to applications started with **rtpSpawn( )**, which involves specification of **argv[0]**. It does not apply to applications started from the shell with **rtpSp( )** (for the C interpreter) or with **rtp exec** (for the command interpreter).

Note that shared library symbols are not visible if an application is started in *stopped* mode. Until execution of **_start( )** (the entry point of an application provided by the compiler) calls the dynamic linker, shared library symbols are not yet registered. (For information about symbol registration, see *3.6.4 Applications and Symbol Registration*, p.49.)

Using readelf to Examine Dynamic ELF Files

The **readelf** tool can be used to extract dynamic records from an executable or a dynamic shared object, such as a shared object name and path. This can be particularly useful to determine whether the build has created the required ELF records—in particular the **SONAME** record in the shared library and the **NEEDED** record in the dynamic RTP application that uses the library (and optionally the **RPATH** record in the application).

Use the version of **readelf** that is appropriate for the target architecture; for example, use **readelfppc** on a PowerPC file. The various versions of the tool are provided in *installDir***/gnu/3.3.2-vxworks6***x***/***hostType***/bin**.

The **-d** flag causes **readelf** to list dynamic records by tag type, such as **NEEDED**, **SONAME**, and **RPATH**. (For information about how these records are used, see *4.8.4 About Shared Library Names and ELF Records*, p.62 and *Using the ELF RPATH Record*, p.65.

readelf Example for RTP Application

The following example shows information about an RTP application that has been linked against both **MySharedLibrary.so.1** and **libc.so.1** (as indicated by the **NEEDED** records). The **RPATH** record indicates that the dynamic linker should look for the libraries in **/romfs/lib** at runtime.

```
C:\WindRiver\gnu\4.1.2-vxworks-6.6\x86-win32\bin>readelfpentium -d \
C:\workspace3\MyRTP\SIMPENTIUMdiab_RTP\MyRTP\Debug\MyRTP.vxe

Dynamic section at offset 0x8554 contains 19 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [MySharedLibrary.so.1]
 0x00000001 (NEEDED)                     Shared library: [libc.so.1]
 0x0000000f (RPATH)                      Library rpath: [/romfs/lib]
 0x00000004 (HASH)                       0xc0
 0x00000006 (SYMTAB)                     0x424
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000005 (STRTAB)                     0x9a4
 0x0000000a (STRSZ)                      1268 (bytes)
 0x00000017 (JMPREL)                     0x15f0
 0x00000002 (PLTRELSZ)                   160 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000016 (TEXTREL)                    0x0
 0x00000003 (PLTGOT)                     0x18680
 0x00000011 (REL)                        0xe98
 0x00000012 (RELSZ)                      1880 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000018 (BIND_NOW)
 0x00000000 (NULL)                       0x0
```

readelf Example for Shared Library

The next example shows information about the **MySharedLibrary.so.1** shared library, against which the RTP application was linked.

```
C:\WindRiver\gnu\4.1.2-vxworks-6.6\x86-win32\bin>readelfpentium -d \
C:\workspace3\MySharedLibrary\SIMPENTIUMdiab_RTP\MySharedLibrary\Debug\mySharedLibrary.so.1

Dynamic section at offset 0x6c8 contains 17 entries:
  Tag        Type                         Name/Value
 0x0000000e (SONAME)                     Library soname: [MySharedLibrary.so.1]
 0x00000004 (HASH)                       0xa0
 0x00000006 (SYMTAB)                     0x218
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000005 (STRTAB)                     0x3e8
 0x0000000a (STRSZ)                      396 (bytes)
 0x00000017 (JMPREL)                     0x5bc
 0x00000002 (PLTRELSZ)                   8 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000016 (TEXTREL)                    0x0
 0x0000000c (INIT)                       0x6b8
 0x0000000d (FINI)                       0x6c0
 0x00000003 (PLTGOT)                     0x1774
 0x00000011 (REL)                        0x574
 0x00000012 (RELSZ)                      72 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
```

```
0x00000000 (NULL)                     0x0
```

Note that the information in each of an application's **NEEDED** records (the shared library name), is derived from corresponding shared library **SONAME** record when the application is linked against the library at build time.

For information about shared object names, shared library versions, and locating shared libraries at run-time, see *4.8.5 Creating Shared Object Names for Shared Libraries*, p.62, *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64, and *4.8.6 Using Different Versions of Shared Libraries*, p.63.

4.8.12 Working With Shared Libraries From a Windows Host

Loading shared libraries from a Windows host system with FTP (the default method) can be excessively slow. As an alternative, shared libraries can be included in the VxWorks system image with the ROMFS file system, or NFS can be used to provide the target system with access to shared libraries on the host.

While ROMFS is useful for deployed systems, using it for development means long edit-compile-debug cycles, as you must rebuild the system image and reboot the target whenever you want to use modified code. During development, therefore, it is better to maintain shared libraries on the host file system and have the target load them from the network. The NFS file system provides for much faster loading than ftp or the Target Server File System.

Using NFS

To use of NFS, you can either install an NFS server on Windows or make use of remote access to a UNIX machine that runs an NFS server. If you have remote access, you can use the UNIX machine to boot your target and export its file system.

Installing NFS on Windows

If you choose to install an NFS server, you can use the one that Microsoft provides free of charge as part of its Windows Services for UNIX (SFU) package. It can be downloaded from **http://www.microsoft.com/**. The full SFU 3.5 package is a 223MB self-extracting executable to download, but if you only install the NFS Server, it takes about 20MB on your hard disk.

To install the Microsoft NFS server, run the SFU **setup.exe** and select **NFS Server** only. The setup program prompts you to install **NFS User Synchronization** as well, which you should do. The corresponding Windows services are installed and started automatically.

To configure the Windows NFS server for use with a VxWorks target:

1. In Windows Explorer, select your Workspace and use the context menu to select **Share...**
2. Select the **NFS Sharing** tab.
3. Enter **Share this folder**, Share name = **Workspace**

4. Enable **Allow anonymous access**. This provides the VxWorks target with read-only access to the share without having to set up user mappings or access permissions.

Configuring VxWorks With NFS

Before you can use NFS to load shared libraries, VxWorks also must be reconfigured with NFS facilities.

Adding the **INCLUDE_NFS_MOUNT_ALL** component provides all the necessary features. Make sure the target the target connection is disconnected before you rebuild your kernel image.

Testing the NFS Connection

When you reboot the target it automatically mounts all NFS shares exported by the host. To verify that VxWorks can access your NFS mount, use the **devs** and **ls "/Workspace"** commands from the kernel shell.

4.9 Developing Plug-Ins

Plug-ins are a type of dynamic shared object similar to shared libraries. Plug-ins can be used to add functionality or to modify the operation of the program by loading replacement plug-ins rather than replacing the entire application.

Plug-ins and shared libraries differ primarily in how they are used with RTP applications. Plug-ins are loaded on-demand (programmatically) by an application rather than being loaded automatically by the dynamic linker when the application is loaded. The development requirements for an application’s use of plug-ins is therefore different from the development requirements for an application’s use of shared libraries. A shared library can, however, be used as a plug-in if the application that requires it is not linked against it at build time.

For general information about building plug-ins, see *4.6 Common Development Facilities*, p.59.

NOTE: A shared library can be used as a plug-in if the application that requires it is not linked against it at build time.

CAUTION: Applications that make use of plug-ins must be built as dynamic executables to include a dynamic linker in their image. The dynamic linker carries out the binding of the dynamic shared object and application at run time. For more information, see *4.9.3 Developing RTP Applications That Use Plug-Ins*, p.73.

NOTE: Shared libraries and plug-ins are not supported for the Coldfire architecture.

4.9.1 Configuring VxWorks for Plug-Ins

VxWorks must be configured with support for plug-ins. For information in this regard, see *4.4 Configuring VxWorks for Shared Libraries and Plug-ins*, p.56.

4.9.2 Initialization and Termination

For information about initialization and termination of plug-ins, see *4.5 Common Development Issues: Initialization and Termination*, p.57.

4.9.3 Developing RTP Applications That Use Plug-Ins

The key differences between developing an RTP application that uses a plug-in and an RTP application that uses a shared library are as follows:

- An application that uses a plug-in must make an API call to load the plug-in, whereas the dynamic linker automatically loads a shared library for an application that requires it.
- An application that uses a plug-in must not be linked against the plug-in at build time, whereas an application that uses a shared library must be linked against the library. Not linking an application against a plug-in means that an ELF **NEEDED** record is not created for the shared object in the application, and the dynamic linker will not attempt to load the shared object when the associated application is loaded at run-time. For information about the role of ELF NEEDED records in applications that use shared libraries, see *4.8.4 About Shared Library Names and ELF Records*, p.62.

Code Requirements

The code requirements for developing an RTP application that makes use of plug-ins are as follows:

- Include the **dlfcn.h** header file.
- Use **dlopen( )** to load the plug-in and to access its functions and data.
- Use **dlsym( )** to resolve a symbol (defined in the shared object) to its address.
- Use **dlclose( )** when the plug-in is no longer needed. The **rtld** library, which provides the APIs for these calls, is automatically linked into a dynamic executable.

For an examples illustrating implementation of these requirements, see *Example of Dynamic Linker API Use*, p.74 and *Example Application Using a Plug-In*, p.75. For general information about developing RTP applications, see *3.3 Developing RTP Applications*, p.30.

Build Requirements

The requirements for building an RTP application that makes use of plug-ins are as follows:

- Compile the application as a dynamic executable (that is, with the Wind River Compiler **-Xdynamic** option or the GNU **-non-static** option), which links the application with the dynamic linker. Static executables cannot load plug-ins because they do not have the loader embedded in them (which provides **dlopen( )** and so on).
- Do *not* link the application against the plug-in when you build the application. If it is linked against the plug-in, the dynamic linker will attempt to load it when the application is started (as with a shared library)—and succeed if the shared object name and run-time path are defined appropriately for this action (that is, as for shared libraries).

For general information about building applications that use plug-ins, see *4.6 Common Development Facilities*, p.59.

Locating Plug-Ins at Run-time

An RTP application can explicitly identify the location of a plug-in for the dynamic linker with the **dlopen( )** call that is used to load the plug-in. It does so by providing the full path to the plug-in (as illustrated in *Example Application Using a Plug-In*, p.75).

If only the plug-in name—and not the full path—is used in the **dlopen( )** call, the dynamic linker relies on same mechanisms as are used to find shared libraries. For information in this regard, see *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64.

Using Lazy Binding With Plug-ins

The second parameter to **dlopen( )** defines whether or not lazy binding is employed for undefined symbols. If **RTLD_NOW** is used, all undefined symbols are resolved before the call returns (or if they are not resolved the call fails). If **RTLD_LAZY** is used, symbols are resolved as they are referenced from the dynamic application and the share object code is executed. **RTLD_GLOBAL** may optionally be OR'd with either **RTLD_NOW** or **RTLD_LAZY**, in which case the external symbols defined in the shared object are made available to any dynamic shared objects that are subsequently loaded.

For a discussion of lazy binding and shared libraries, see *4.8.8 Using Lazy Binding With Shared Libraries*, p.67.

Example of Dynamic Linker API Use

The following code fragment illustrates basic use of the APIs required to work with a plug-in:

```
void *handle;
void * addr;
void (*funcptr)();

handle = dlopen("/romfs/lib/myLib.so", RTLD_NOW);

addr = dlsym(handle, "bar");

funcptr = (void (*)())addr;
```

```
funcptr();

dlclose(handle);
```

Example Application Using a Plug-In

Assume, for example, that you have a networking application and you want to be able to add support for new datagram protocols. You can put the code for datagram protocols into plug-ins, and load them on demand, using a separate configuration protocol. The application might look like the following:

```
#include <dlfcn.h>
[...]

const char plugin_path[] = "/romfs/plug-ins";

void *attach(const char *name)
{
    void *handle;
    char *path;
    size_t n;

    n = sizeof plugin_path + 1 + strlen(name) + 3;
    if ((path = malloc(n)) == -1) {
        fprintf(stderr, "can't allocate memory: %s",
            strerror(errno));
        return NULL;
}
    sprintf(path, "%s/%s.so", plugin_path, name);

    if ((handle = dlopen(path, RTLD_NOW)) == NULL)
        fprintf(stderr, "%s: %s", path, dlerror());

    free(path);

    return handle;
}

void detach(PROTO handle)
{
    dlclose(handle);
}

[...]

int
send_packet(PROTO handle, struct in_addr addr, const char *data, size_t len)
{
    int (*proto_send_packet)(struct in_addr, const char *, size_t);

/* find the packet sending routine within the plugin and use it to send the
packet as requested */

    if ((proto_send_packet = dlsym(handle, "send_packet")) == NULL) {
        fprintf(stderr, "send_packet: %s", dlerror());
        return -1;
    }

    return (*proto_send_packet)(addr, data, len);
}
```

Assume you implement a new protocol named *reliable*. You would compile the code as PIC, then link it using the **-Xdynamic -Xshared** flags (with the Wind River compiler) into a shared object named **reliable.so** (the comparable GNU flags would be **-non-static** and **-shared**). You install **reliable.so** as **/romfs/plug-ins/reliable.so** on the target.

When a configuration request packet arrives on a socket, the application would take the name of the protocol (**reliable**) and call **attach( )** with it. The **attach( )** routine uses **dlopen( )** to load the shared object named **/romfs/plug-ins/reliable.so**. Subsequently, when a packet must be sent to a particular address using the new protocol, the application would call **send_packet( )** with the return value from **attach( )**, the packet address, and data parameters. The **send_packet( )** routine looks up the protocol-specific **send_packet( )** routine inside the plug-in and calls it with the address and data parameters. To unload a protocol module, the application calls **detach( )**.

Routines for Managing Plug-Ins

The routines used by an application to manage plug-ins are described in Table 4-1.

Table 4-1 **Plug-In Management Routines**

Routine	Description
dlopen( )	Load the plug-in.
dlsym( )	Look up a function or data element in the plug-in.
dlclose( )	Unload a plug-in (if there are no other references to it).
dlerror( )	Return the error string after an error in **dlopen( )**, **dlclose( )** or **dlsym( )**.

For more information about these APIs, see the **rtld** library entry in the *VxWorks Application API Reference*.

4.9.4 Debugging Plug-Ins

Debugging plug-ins is similar to debugging shared libraries. For information in this regard, see *4.8.11 Debugging Problems With Shared Library Use*, p.69.

4.10 Using the VxWorks Run-time C Shared Library libc.so

The VxWorks distribution provides a C run-time shared library that is similar to that of the UNIX C run-time shared library. The VxWorks shared library **libc.so** provides all of the basic facilities that an application might require. It includes all of the user-side libraries provided by its static library equivalent (**libc.a**), with the exception of the following:

- **aioPxLib** (see *9.7 Asynchronous Input/Output*, p.234)
- **memEdrLib** (see *8.4 Memory Error Detection*, p.209)
- message channel libraries (see *6.17 Message Channels*, p.133)
- networking libraries (see the *Wind River Network Stack Programmer's Guide*)

All dynamic executables require **libc.so.1** at run time.

When generating a dynamic executable, the GNU and Wind River toolchains automatically use the corresponding build-time shared object, **libc.so**, which is located in *installDir*/**vxworks-6.***x*/**target/usr/lib/***arch*/*cpu*/**common** (where *arch* is a the architecture such as **ppc**, **pentium**, or **mips)**. If required, another location can be referred to by using the linker's **-L** option.

The run-time version of the library is **libc.so.1**, which is located in the directory *installDir*/**vxworks-6.***x*/**target/usr/root/***cpuTool*/**bin**, where *cpu* is the name of the target CPU (such as PPC32, or PENTIUM4) and *Tool* is the name of a toolchain—including modifiers indicating the endianness and the floating point attributes applied when generating the code (for example **diab**, **sfdiable**, **gnu**, or **gnule**). For example: *installDir*/**vxworks-6.***x*/**target/usr/root/SIMPENTIUMdiab/bin/libc.so.1**

For a development environment, various mechanisms can be used for providing the dynamic linker with information about the location of the **libc.so.1** file.

For deployed systems, the **libc.so.1** file can be copied manually to whatever location is appropriate. The most convenient way to make the dynamic linker aware of the location of **libc.so.1** is to store the file in the same location as the dynamic application, or to use the **-Wl,-rpath** compiler flag when the application is built. For more information, see *4.8.7 Locating and Loading Shared Libraries at Run-time*, p.64.

NOTE: Note that the default C shared library is intended to facilitate development, but may not be suitable for production systems because of its size.

5
C++ Development

5.1 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 for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the *VxWorks Kernel Programmer's Guide*.

5.2 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.

5.3 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.

5.3.1 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.

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.

-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.

-fno-implicit-templates
This is the option for explicit instantiation. Using this strategy explicitly instantiates any templates that you require.

5.3.2 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.

5.4 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;
        }
```

5.5 C++ Demo Example

For a sample C++ application, see *installDir***/vxworks-6.***x***/target/usr/apps/samples/cplusplus/factory**.

6
Multitasking

6.1 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.

VxWorks provides watchdog timers, but they can only be used in the kernel (see *VxWorks Kernel Programmer's Guide: Multitasking*. However, process-based applications can use POSIX timers (see *7.9 POSIX Clocks and Timers*, p.155).

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 *7. POSIX Facilities*.

NOTE: This chapter provides information about facilities available for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the *VxWorks Kernel 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 and migration, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

6.2 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 *7.13 POSIX Threads*, p. 160).

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. On a uniprocessor system 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
- error status (errno)
- debugging and performance monitoring values

Note that—consistent with the POSIX standard—all tasks in a process share the same environment variables (unlike kernel tasks, which each have their own set of environment variables).

For more information about virtual memory contexts, see the *VxWorks Kernel Programmer's Guide: 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 *7.13 POSIX Threads*, p. 160.

6.2.1 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 activity such as certain function calls made by the application (for example, when attempting to take a semaphore that is not available) and the use of development tools such as the debugger.

The highest priority task that is in the *ready* state is the task that executes. When tasks are created with **taskSpawn( )**, they immediately enter the ready state. For information about the ready state, see *Scheduling and the Ready Queue*, p.91.

When tasks are created with **taskCreate( )**, or **taskOpen( )** with the **VX_TASK_NOACTIVATE** options parameter, they are instantiated in the *suspended* state. They can then be activated with **taskActivate( )**, which causes them to enter the ready state. The activation phase is fast, enabling applications to create tasks and activate them in a timely manner.

Tasks States and State Symbols

Table 6-1 describes the task states and the *state symbols* that you see when working with development tools.

Note that task states are additive; a task may be in more than one state at a time. Transitions may take place with regard to one of multiple states. For example, a task may transition from pended to pended and stopped. And if it then becomes unpended, its state simply becomes stopped.

Table 6-1 **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 (such as a semaphore).
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 (also used by the error detection and reporting facilities).
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 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.

Table 6-1 **Task State Symbols** (cont'd)

State Symbol	Description
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 debugging facilities when a breakpoint is hit. It is also used by the error detection and reporting facilities (for more information, see *11. Error Detection and Reporting*). Example 6-1 shows output from the **i( )** shell command, displaying task state information.

Example 6-1 **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
```

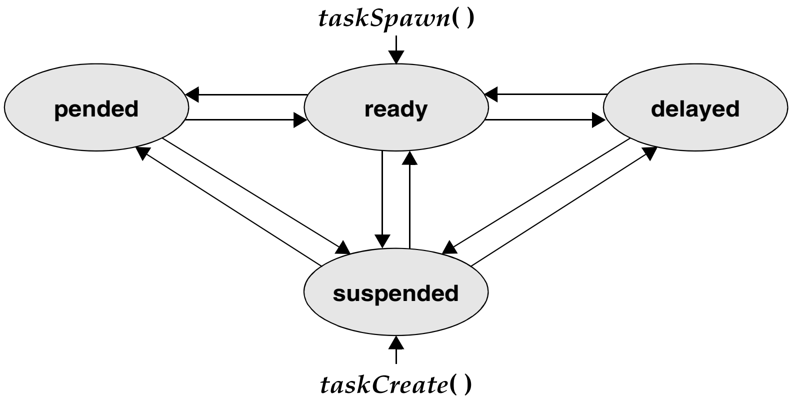
Illustration of Basic Task State Transitions

Figure 6-1 provides a simplified illustration of task state transitions. For the purpose of clarity, it does not show the additive states discussed in *Tasks States and State Symbols*, p.86, nor does it show the **STOP** state used by debugging facilities.

The routines listed are examples of those that would cause the associated transition. For example, a task that called **taskDelay( )** would move from the ready state to the delayed state.

Note that **taskSpawn( )** causes a task to enter the ready state when it is created, whereas **taskCreate( )** causes a task to enter the suspended state when it is created (using **taskOpen( )** with the **VX_TASK_NOACTIVATE** option also achieves the latter purpose).

Figure 6-1 **Basic Task State Transitions**



	→	ready	taskSpawn( )
	→	suspended	taskCreate( )
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( )

6.3 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 *6.3.3 VxWorks Traditional Scheduler*, p.90.
- The VxWorks POSIX threads scheduler, which is designed (and required) for running pthreads in processes (RTPs). See *7.15 POSIX and VxWorks Scheduling*, p.171.)
- A custom scheduler framework, which allows you to develop your own scheduler. See the *VxWorks Kernel Programmer's Guide: Kernel Customization*.

6.3.1 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 *6.4.1 Task Creation and Activation*, p.93 and *6.3.2 Task Scheduling Control*, p.89).

All application tasks should be in the priority range from 100 to 255.

6.3.2 Task Scheduling Control

The routines that control task scheduling are listed in Table 6-2.

Table 6-2 **Task Scheduling Control Routines**

Routine	Description
taskPrioritySet( )	Changes the priority of a task.
taskRtpLock( )	Disables task context switching within a process (as long as the task is not blocked or has not voluntarily give up the CPU). Prevents any other task in the process from preempting the calling task.
taskRtpUnLock( )	Enables task context switching within a process.

Task Priority

Tasks are assigned a priority when they are created (see *6.4.1 Task Creation and Activation*, p.93). 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.

Note that if a task's priority is changed with **taskPrioritySet( )**, it is placed at the end of the ready queue priority list for its new priority. For information about the ready queue, see *Scheduling and the Ready Queue*, p.91.

Preemption Locks

The scheduler can be explicitly disabled and enabled on a per-task basis—within a process—with the routines **taskRtpLock ( )** and **taskRtpUnLock ( )**. When a task disables the scheduler by calling **taskRtpLock( )**, no priority-based preemption can take place by other tasks running in the same process while that task is running. Using a semaphore is, however, preferable to **taskRtpLock( )** as a means of mutual exclusion, because preemption lock-outs add preemptive latency to the process.

If the task that has disabled the scheduler with **taskRtpLock( )** 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.

If mutual exclusion between tasks in different processes is required, use a public semaphore. For information about global objects, see *6.9 Inter-Process Communication With Public Objects*, p. 107.

NOTE: The **taskRtpLock ( )** and **taskRtpUnLock( )** routines are provided for the UP configuration of VxWorks, but not the SMP configuration. Use semaphores or another mechanism supported for SMP instead of **taskRtpLock ( )** and **taskRtpUnLock( )**. For more information, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

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 *6.12 Mutual Exclusion*, p. 110.

6.3.3 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.

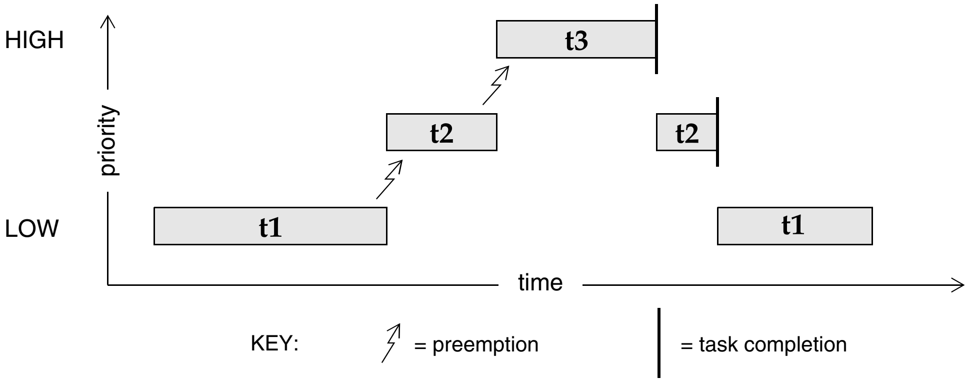
For information about the POSIX thread scheduler and custom schedulers, see *7.15 POSIX and VxWorks Scheduling*, p. 171 and *VxWorks Kernel Programmer's Guide: Kernel Customization*, respectively.

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 6-2, 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 (for more information, see *Round-Robin Scheduling*, p. 92).

Figure 6-2 **Priority Preemption**



Scheduling and the Ready Queue

The VxWorks scheduler maintains a FIFO ready queue mechanism that includes lists of all the tasks that are ready to run (that is, in the ready state) at each priority level in the system. When the CPU is available for given priority level, the task that is at the front of the list for that priority level executes.

A task's position in the ready queue may change, depending on the operation performed on it, as follows:

- If a task is preempted, the scheduler runs the higher priority task, but the preempted task retains its position at the front of its priority list.
- If a task is pended, delayed, suspended, or stopped, it is removed from the ready queue altogether. When it is subsequently ready to run again, it is placed at the end of its ready queue priority list. (For information about task states and the operations that cause transitions between them, see *6.2.1 Task States and Transitions*, p. 85).
- If a task's priority is changed with **taskPrioritySet( )**, it is placed at the end of its new priority list.
- If a task's priority is temporarily raised based on the mutual-exclusion semaphore priority-inheritance policy (using the **SEM_INVERSION_SAFE** option), it returns to the end of its original priority list after it has executed at the elevated priority. (For more information about the mutual exclusion semaphores and priority inheritance, see *Priority Inheritance Policy*, p. 117.)

The **taskRotate( )**routine can be used to shift a task from the front to the end of its priority list. For more information about this routine, see *6.3.2 Task Scheduling Control*, p. 89 and the VxWorks API reference entry.

Note that with the optional POSIX pthread scheduler, pthreads and tasks are handled differently when their priority is programmatically lowered. If the priority of a *pthread* is lowered, it moves to the front of its new priority list. However, if the priority of a *task* is lowered, it moves to the end of its new priority list. For more information in this regard, see *Differences in Re-Queuing Pthreads and Tasks With Lowered Priorities*, p. 177.

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, and you want to share the CPU amongst these tasks.

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 6-3 for an illustration of this activity. When the time slice expires, the task moves to last place in the ready queue list for that priority (for information about the ready queue, see *Scheduling and the Ready Queue*, p.91).

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.

It may be useful to use round-robin scheduling in systems that execute the same application in more than one process. In this case, multiple tasks would be executing the same code, and it is possible that a task might not relinquish the CPU to a task of the same priority running in another process (running the same binary). Note that round-robin scheduling is global, and controls all tasks in the system (kernel and processes); it is not possible to set round-robin scheduling for selected processes.

Enabling Round-Robin Scheduling

Round-robin scheduling is enabled by calling **kernelTimeSlice( )**, which takes a parameter for a time slice, or interval. It is disabled by using zero as the argument to **kernelTimeSlice( )**.

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 end of the ready queue priority list for its priority level. New tasks joining a given priority group are placed at the end of the priority list for that priority with their run-time counter initialized to zero.

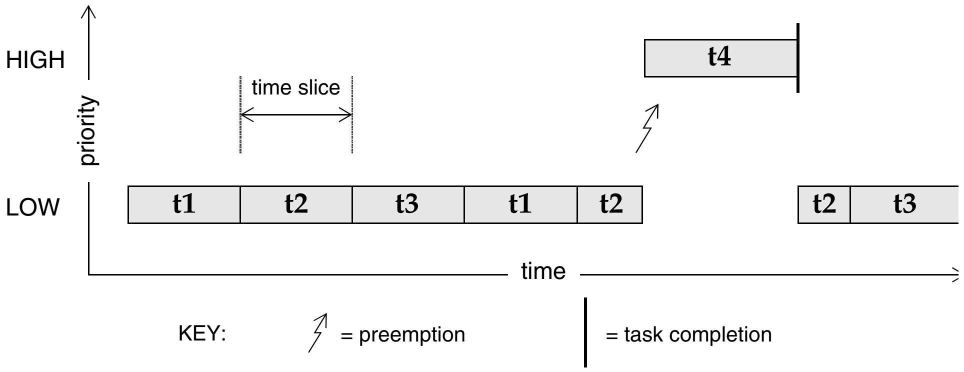
Enabling round-robin scheduling does not affect the performance of task context switches, nor is additional memory allocated.

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 resumes 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 end of the ready queue list for 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 6-3 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 6-3 **Round-Robin Scheduling**



6.4 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 by Example* guide, the *Wind River Workbench Host Shell User's Guide*, and *VxWorks Kernel Programmer's Guide: Target Tools*.

6.4.1 Task Creation and Activation

The routines listed in Table 6-3 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 *6.3.2 Task Scheduling Control*, p.89.

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 6-3 **Task Creation Routines**

Routine	Description
taskSpawn( )	Spawns (creates and activates) a new task.
taskCreate( )	Creates, but not activates a new task.
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 *6.4.2 Task Names and IDs*, p.94). 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.

6.4.2 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.

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 *6.4.3 Inter-Process Communication With Public Tasks*, p.95.

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 6-4 manage task IDs and names.

Table 6-4 **Task Name and ID Routines**

Routine	Description
taskName( )	Gets the task name associated with a task ID (restricted to the context—process or kernel—in which it is called).
taskNameGet( )	Gets the task name associated with a task ID anywhere in the entire system (kernel and any processes).
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.

Note that for use within a process, it is preferable to use **taskName( )** rather than **taskNameGet( )** from a process, as the former does not incur the overhead of a system call.

6.4.3 Inter-Process Communication With Public Tasks

VxWorks tasks 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 are accessible throughout the system.

For detailed information, see *6.9 Inter-Process Communication With Public Objects*, p.107 and the **taskLib** entry in the VxWorks Application API references.

6.4.4 Task Creation Options

When a task is spawned, you can pass in one or more option parameters, which are listed in Table 6-5. The result is determined by performing a logical OR operation on the specified options.

Table 6-5 **Task Options**

Name	Description
VX_ALTIVEC_TASK	Execute with Altivec coprocessor support.
VX_DSP_TASK	Execute with DSP coprocessor support.

Table 6-5 **Task Options** (cont'd)

Name	Description
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 *6.4.5 Task Stack*, p.96).
VX_PRIVATE_ENV	Executes a task with a private environment.
VX_TASK_NOACTIVATE	Used with **taskOpen( )** so that the task is not activated.

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).

6.4.5 Task Stack

The size of each task's stack is defined when the task is created (see *6.4.1 Task Creation and Activation*, p.93).

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.97).

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.

User-mode (RTP) tasks have overflow and underflow guard zones on their *execution stacks* by default. This protection is provided by the **INCLUDE_RTP** component, which is required for process support. It is particularly important in providing protection from system calls overflowing the calling task's stack. Configuring VxWorks with the **INCLUDE_PROTECT_TASK_STACK** component adds overflow (but not underflow) protection for user-mode task *exception stacks*.

Note that RTP task guard zones for *execution* stacks do not use any physical memory—they are virtual memory entities in user space. The guard zones for RTP task *exception* stacks, however, are in kernel memory space and mapped to physical memory.

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 *6.4.4 Task Creation Options*, p.95). 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_USER_EXEC_STACK_OVERFLOW_SIZE** for user task execution stack overflow size.
- **TASK_USER_EXEC_STACK_UNDERFLOW_SIZE** for user task execution stack underflow size.
- **TASK_USER_EXC_STACK_OVERFLOW_SIZE** for user task exception stack overflow 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.

Note that for POSIX threads in processes, the size of the execution stack guard zone can be set on an individual basis before the thread is created (using the pthread attributes object, **pthread_attr_t**). For more information, see *7.13.1 POSIX Thread Stack Guard Zones*, p.161.

6.4.6 Task Information

The routines listed in Table 6-6 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 6-6 **Task Information Routines**

Routine	Description
taskInfoGet( )	Gets information about a task.
taskPriorityGet( )	Examines the priority of a task.
taskIsSuspended( )	Checks whether a task is suspended.
taskIsReady( )	Checks whether a task is ready to run.
taskIsPended( )	Checks whether a task is pended.
taskIsDelayed( )	Checks whether or not a task is delayed.

For information about task-specific variables and their use, see *6.7.3 Task-Specific Variables*, p.105.

6.4.7 Task Deletion and Deletion Safety

Tasks can be dynamically deleted from the system. VxWorks includes the routines listed in Table 6-7 to delete tasks and to protect tasks from unexpected deletion.

Table 6-7 **Task-Deletion Routines**

Routine	Description
exit( )	Terminates the specified process (and therefore all tasks in it) and frees the process' memory resources.
taskExit( )	Terminates the calling task (in a process) and frees the stack and any other memory resources, including the task control block.[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.

Table 6-7 **Task-Deletion Routines** (cont'd)

Routine	Description
taskSafe( )	Protects the calling task from deletion by any other task in the same process. A task in a different process can still delete that task by terminating the process itself with **kill( )**.
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.

A process implicitly calls **exit( )**, thus terminating all tasks within it, if the process' **main( )** routine returns. For more information see *2.2.3 RTP Termination*, p.6.

Tasks implicitly call **taskExit( )** 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 *6.13.4 Mutual-Exclusion Semaphores*, p.116.

6.4.8 Task Execution Control

The routines listed in Table 6-8 provide direct control over a task's execution.

Table 6-8 **Task Execution Control Routines**

Routine	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( )**, as follows:

```
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 *7.9 POSIX Clocks and Timers*, p.155.

Note that calling **taskDelay( )** removes the calling task from the ready queue. When the task is ready to run again, it is placed at the end of the ready queue priority list for its priority. This behavior can be used to 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( )**. The **nanosleep( )** routine treats zero as an error. For information about the ready queue, see *Scheduling and the Ready Queue*, p.91.

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.

6.4.9 Tasking Extensions: Hook Routines

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 6-9; for more information, see the VxWorks API reference for **taskHookLib**.

Table 6-9 **Task Create, Switch, and Delete Hooks**

Routine	Description
taskCreateHookAdd( )	Adds a routine to be called at every task create.
taskCreateHookDelete( )	Deletes a previously added task create routine.
taskDeleteHookAdd( )	Adds a routine to be called at every task delete.
taskDeleteHookDelete( )	Deletes a previously added task delete routine.

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 6-10 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 6-10 **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( )**, **taskIsPended( )**, **taskIsDelayed( )**, **taskTcb( )**
vxLib	**vxTas( )**

NOTE: For information about POSIX extensions, see *7. POSIX Facilities*.

6.5 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, as well as about migration, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

6.5.1 A Separate errno Value for Each Task

In processes, there is no single global **errno** variable. Instead, standard application accesses to **errno** directly manipulate the per-task **errno** field in the TCB (assuming, of course, that **errno.h** has been included).

6.5.2 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.

6.5.3 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.

6.6 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*).

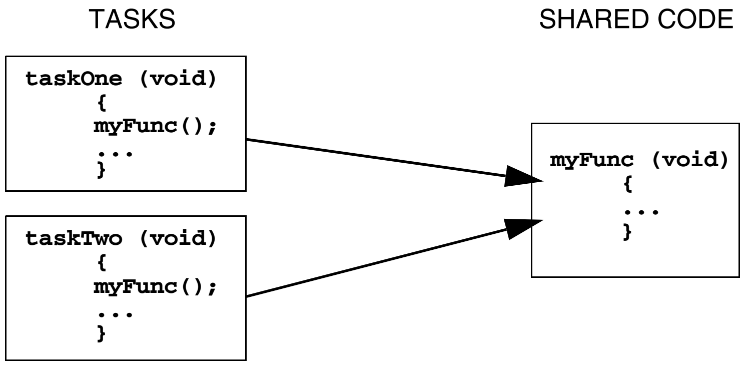
Tasks can also attach their own handlers for certain hardware exceptions through the *signal* facility. If a task has supplied a signal handler for an exception, the default exception handling described above is not performed. A user-defined signal handler is useful for recovering from catastrophic events. 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 *6.19 Signals*, p.134 and in the VxWorks API reference for **sigLib**.

6.7 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 6-4.

Figure 6-4 **Shared Code**



```
taskOne (void)
    {
    myFunc();
    ...
    }
```

```
taskTwo (void)
    {
    myFunc();
    ...
    }
```

```
myFunc (void)
    {
    ...
    }
```

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.

The majority of VxWorks routines use the following reentrancy techniques:

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

Wind River recommends applying these same techniques when writing application code that can be called from several task contexts simultaneously.

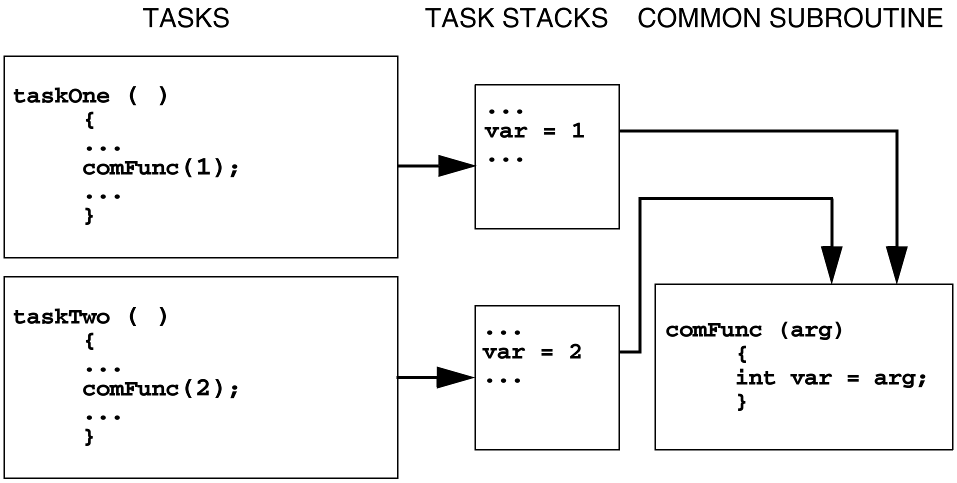
NOTE: Initialization routines should be callable multiple times, even if logically they should only be called once. As a rule, routines should avoid **static** variables that keep state information. Initialization routines are an exception; using a **static** variable that returns the success or failure of the original initialization routine call is appropriate.

6.7.1 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 6-5.

Figure 6-5 **Stack Variables and Shared Code**



```
taskOne ( )
     {
     ...
     comFunc(1);
     ...
     }
```

```
taskTwo ( )
     {
     ...
     comFunc(2);
     ...
     }
```

```
...
var = 1
...
```

```
...
var = 2
...
```

```
comFunc (arg)
     {
     int var = arg;
     }
```

6.7.2 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 *6.13.4 Mutual-Exclusion Semaphores*, p.116.

6.7.3 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 their 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 not compatible with VxWorks SMP. They are now obsolete and will be removed from a future release of VxWorks. In addition to being incompatible with VxWorks SMP, the **taskVarLib** and **tlsOldLib** facilities increase task context switch times. For information about migration, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

Also note that each task has a VxWorks events register, which receives events sent from other tasks, ISRs, semaphores, or message queues. See *6.16 VxWorks Events*, p.128 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.

tlsOldLib and Task Variables

VxWorks provides the task-local storage (TLS) facility and the routines provided by **tlsOldLib** (formerly **tlsLib**) to maintain information on a per-task basis in RTPs.

NOTE: Wind River does not recommend using the **tlsOldLib** facility, which is maintained primarily for backwards-compatibility. Use thread-local (**__thread**) storage class variables instead.

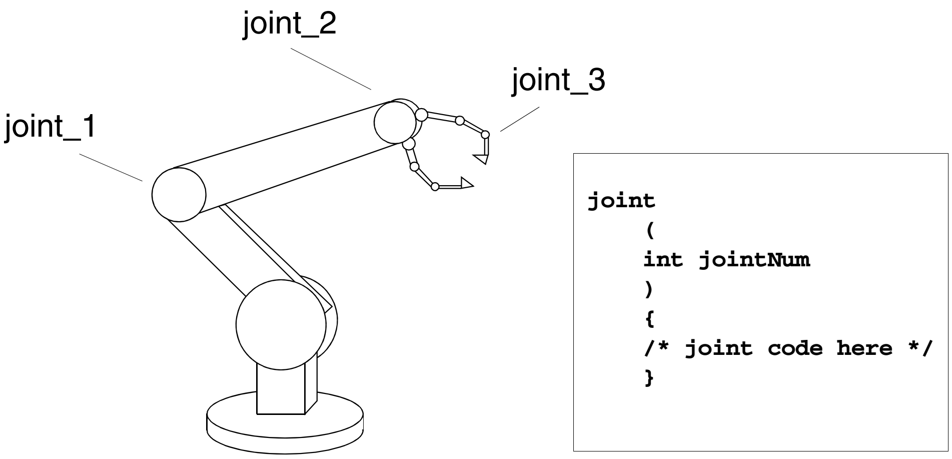
6.7.4 Multiple Tasks with the Same Main Routine

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 *6.7.3 Task-Specific Variables*, p.105 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 6-6, 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 6-6 **Multiple Tasks Utilizing Same Code**



```
joint
    (
    int jointNum
    )
    {
    /* joint code here */
    }
```

6.8 Intertask and Interprocess Communication

The complement to the multitasking routines described in *6.2 Tasks and Multitasking*, p.85 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, and about migration, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

In addition, the VxMP component provides for intertask communication between multiple CPUs that share memory. See the *VxWorks Kernel Programmer's Guide*.

6.9 Inter-Process Communication With Public 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 *6.4.2 Task Names and IDs*, p.94.

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 *6.4.2 Task Names and IDs*, p.94.

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.

For detailed information about the APIs used to create public user-mode (RTP) objects, see the **msgQLib**, **semLib**, **taskLib**, and **timerLib** entries in the *VxWorks Application API Reference*.

6.10 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 **objHandleShow( )** show routine can be used to display information about ownership relations between objects in a process.

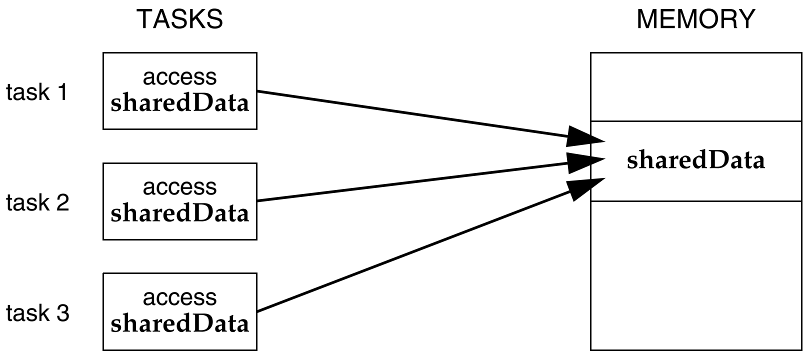
6.11 Shared Data Structures

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 6-7.

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 *3.5 Creating and Using Shared Data Regions*, p.38.

Figure 6-7 **Shared Data Structures**



6.12 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 *7.14 POSIX Thread Mutexes and Condition Variables*, p.167.

6.13 Semaphores

VxWorks semaphores are highly optimized, providing a very fast intertask communication mechanism. 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 *6.12 Mutual Exclusion*, p.110.
- 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 the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

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 *6.13.3 Binary Semaphores*, p.114.

mutual exclusion
A special binary semaphore optimized for problems inherent in mutual exclusion: priority inversion, deletion safety, and recursion. For more information, see *6.13.4 Mutual-Exclusion Semaphores*, p.116.

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 *6.13.5 Counting Semaphores*, p.119.

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 *6.13.6 Read/Write Semaphores*, p.120.

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 *7.16 POSIX Semaphores*, p.179.

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 across processors an asymmetric multiprocessor (AMP) system, but only in the VxWorks kernel (and not in UP or SMP systems). For more information, see *VxWorks Kernel Programmer's Guide: Shared Memory Objects*.

6.13.1 Inter-Process Communication With Public Semaphores

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 are accessible throughout the system.

For detailed information, see *6.9 Inter-Process Communication With Public Objects*, p.107 and the **semLib** entry in the *VxWorks Application API Reference*.

6.13.2 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 to this rule are as follows:

- The creation routines, which are specific to each semaphore type.
- The give and take routines for read/write semaphores, which support read and write modes for each operation.
- The scalable and inline variants of the give and take routines for binary and mutex semaphores, which provide optimized alternatives to the standard routines.

Table 6-11 lists the semaphore control routines. Table 6-12 lists the options that can be used with the scalable and inline routines. For general information about the scalable and inline routines, see *Scalable and Inline Semaphore Take and Give Routines*, p.113.

Table 6-11 **Semaphore Control Routines**

Routine	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 (all types).
semTake( )	Takes a binary, mutual-exclusion, or counting semaphore, or a read/write semaphore in write mode.

Table 6-11 **Semaphore Control Routines** (cont'd)

Routine	Description
semRTake( )	Takes a read/write semaphore in read mode.
semWTake( )	Takes a read/write semaphore in write mode.
semBTakeScalable( )	Takes a binary semaphore (with scalable functionality).
semMTakeScalable( )	Takes a mutual-exclusion semaphore (with scalable functionality).
semBTake_inline( )	Takes a binary semaphore (with scalable functionality).
semMTake_inline( )	Takes a mutual-exclusion semaphore (with scalable functionality). Implemented as an inline function.
semGive( )	Gives a binary, mutual-exclusion, or counting semaphore.
semRWGive( )	Gives a read/write semaphore.
semMGiveForce( )	Gives a mutual-exclusion semaphore without restrictions. Intended for debugging purposes only.
semRWGiveForce( )	Gives a read-write semaphore without restrictions. Intended for debugging purposes only.
semBGiveScalable( )	Gives a binary semaphore (with scalable functionality).
semMGiveScalable( )	Gives a mutual-exclusion semaphore (with scalable functionality).
semBGive_inline( )	Gives a binary semaphore (with scalable functionality). Implemented as an inline function.
semMGive_inline( )	Gives a mutual-exclusion semaphore (with scalable functionality). Implemented as an inline function.
semFlush( )	Unblocks all tasks that are waiting for a binary or counting 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.

Options for Scalable and Inline Semaphore Routines

Table 6-12 lists the options that can be used with the scalable and inline routines. For general information about scalable and inline routines, see *Scalable and Inline Semaphore Take and Give Routines*, p.113.

Table 6-12 **Scalable and Inline Semaphore Options**

Routine	Description
SEM_NO_ID_VALIDATE	No object validation is performed on a semaphore.
SEM_NO_ERROR_CHECK	Error checking code is not executed. This includes tests for interrupt restriction, task validation of owners selected from the pend queue, and ownership validation for mutex semaphores.
SEM_NO_EVENT_SEND	Do not send VxWorks events, even if a task has registered to receive event notification on this semaphore.
SEM_NO_SYSTEM_VIEWER	Do not send System Viewer events. This applies only when semaphores are uncontested. If it is necessary to pend on a take call or to unpend a task on a give call, System Viewer events are sent. This differs from calls to the **semLib** APIs which send events for all invocations as well as second events when pending (or unpending).
SEM_NO_RECURSE	Do not perform recursion checks (applies only to mutex semaphores only) whether semaphores are contested or not. It is important that this is used consistently during any single thread of execution.

⚠ **CAUTION:** The options listed in Table 6-12 must not be used when semaphores are created. Errors are generated if they are used.

Scalable and Inline Semaphore Take and Give Routines

In addition to the standard semaphore give and take routines, VxWorks provides scalable and inline of variants for use with binary and mutex semaphores. These routines provide the following advantages:

- Performance improvements for both UP and SMP configurations of VxWorks based on either—or both—scalable options and inline use.
- Additional performance improvements for SMP over standard routines even if just inline variants are used, because they provide optimizations for uncontested take and give operations in an SMP system.

The scalable routines are designed for use with lightly contested resources when performance is of greater significance than features of the standard routines that provide for their robustness (such as various forms of error checking). Several options are available for de-selecting operational features that would normally be conducted for any given take or give operation.

The inline variants of the take and give routines provide the same options as the scalable routines, but also avoid the overhead associated with a function call. As with any inline code, repeated use adds to system footprint. If an application makes numerous calls to a routine for which there is an inline variant, either a wrapper should be created for the inline routine, or the scalable variant should be used instead.

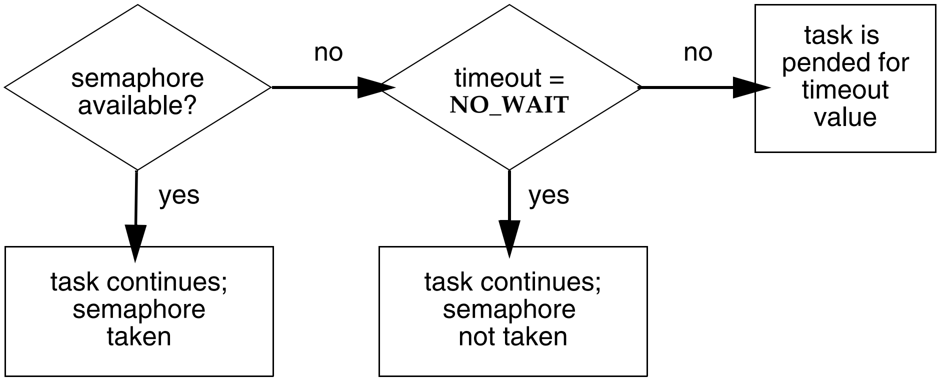
The scalable and inline of variants are listed in Table 6-11; and the options for the scalable and inline routines are listed in Table 6-12). Note that in order to use these routines, you must include the **semLibInline.h** header file.

6.13.3 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 *6.13.4 Mutual-Exclusion Semaphores*, p.116 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.

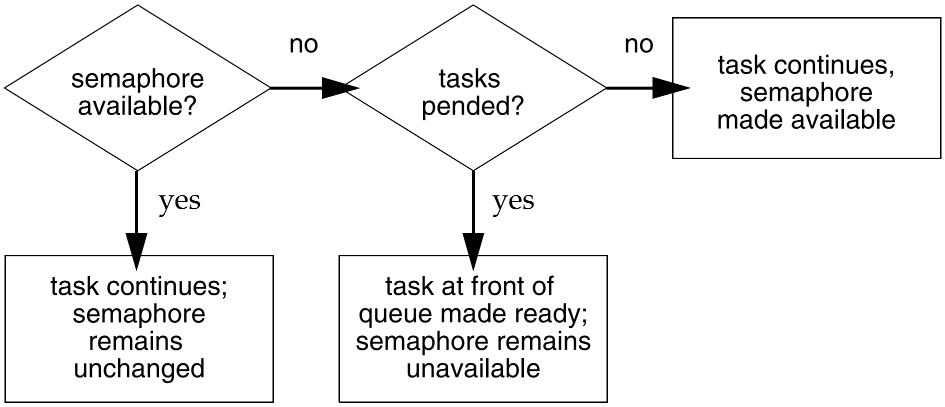
A binary semaphore can be viewed as a flag that is available (full) or unavailable (empty). When a task takes a binary semaphore, with **semTake( )**, the outcome depends on whether the semaphore is available (full) or unavailable (empty) at the time of the call; see Figure 6-8. If the semaphore is available (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 6-8 **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 6-9. 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 6-9 **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.

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.

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.

6.13.4 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.
- The **semFlush( )** operation is illegal.

User-Level Mutex Semaphores

Note that mutex semaphores can be created as *user-level* (user-mode) objects. They are faster than *kernel-level* semaphores as long as they are uncontested, which means the following:

- The mutex semaphore is available during a **semTake( )** operation.
- There is no task waiting for the semaphore during a **semGive( )** operation.

The uncontested case should be the most common, given the intended use of a mutex semaphore.

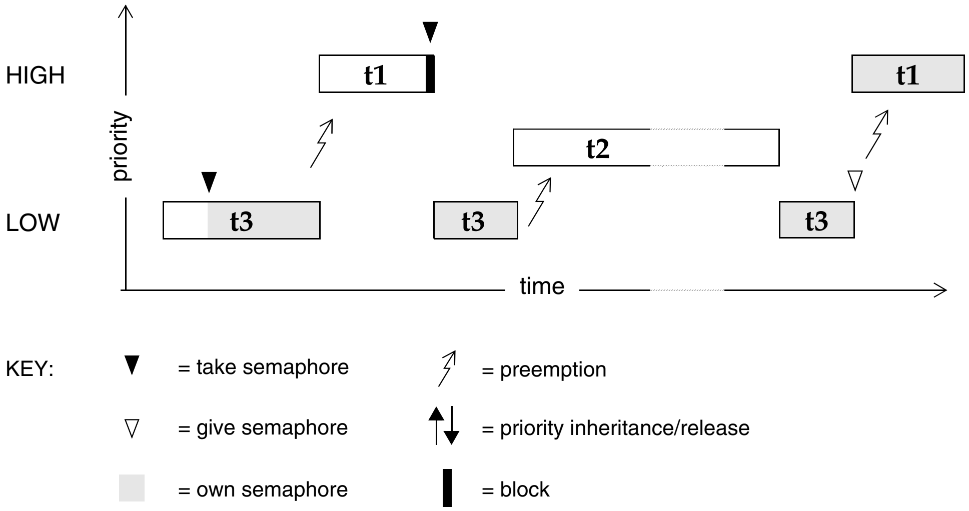
By default, using the **semMCreate()** routine in a process creates a user-level mutex semaphore. However, a kernel-level semaphore can be created when **semMCreate( )** is used with the **SEM_KERNEL** option. The **semOpen()** routine can only be used to create kernel-level semaphores in a process. Note that user-level semaphores can only be created as private objects, and not public ones.

NOTE: User-level semaphores are not supported for the symmetric multiprocessing (SMP) configuration of VxWorks. With the SMP configuration, a **semMCreate( )** call in a process creates a kernel-level mutex semaphore. For information about VxWorks SMP and about migration, see *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

Priority Inversion and Priority Inheritance

Figure 6-10 illustrates a situation called priority inversion.

Figure 6-10 **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 6-10: **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.

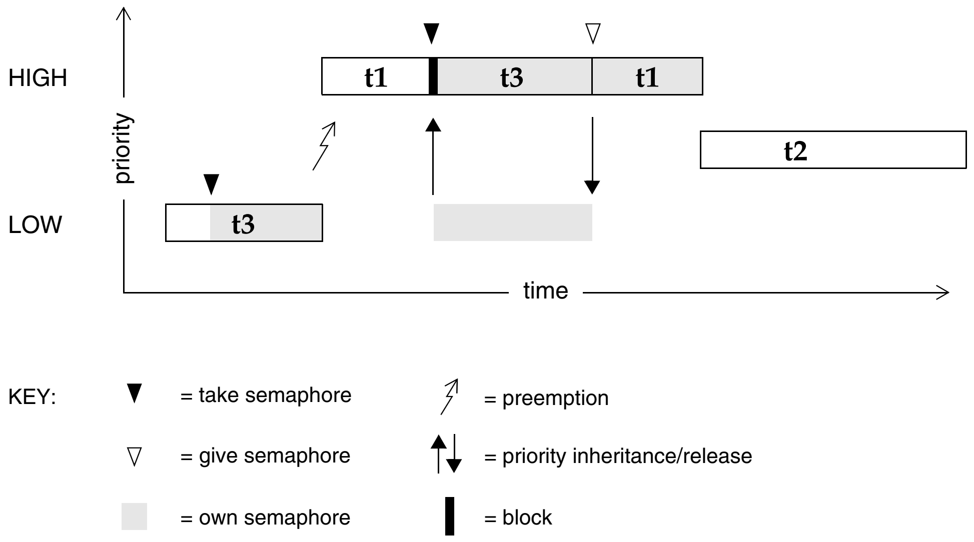
Priority Inheritance Policy

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 that is blocked on that resource.

Once the task's 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**).

Note that after the inheriting task has finished executing at the elevated priority level, it returns to the end of the ready queue priority list for its original priority. (For more information about the ready queue, see *Scheduling and the Ready Queue*, p.91.)

Figure 6-11 **Priority Inheritance**



In Figure 6-11, 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 6-2 **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");
    }
```

6.13.5 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 6-13 shows an example time sequence of tasks taking and giving a counting semaphore that was initialized to a count of 3.

Table 6-13 **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.

6.13.6 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 the VxWorks Kernel Programmer's Guide: 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 that owns it is in the same mode.

For more information about read/write semaphore APIs, see Table 6-11 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.

6.13.7 Special Semaphore Options

The uniform VxWorks semaphore interface includes four special options: timeout, queueing, interruptible by signals, and use with VxWorks events. These options are not available for either the read/write semaphores described in *6.13.6 Read/Write Semaphores*, p.120, or the POSIX-compliant semaphores described in *7.16 POSIX Semaphores*, p.179.

Semaphore Timeout

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.

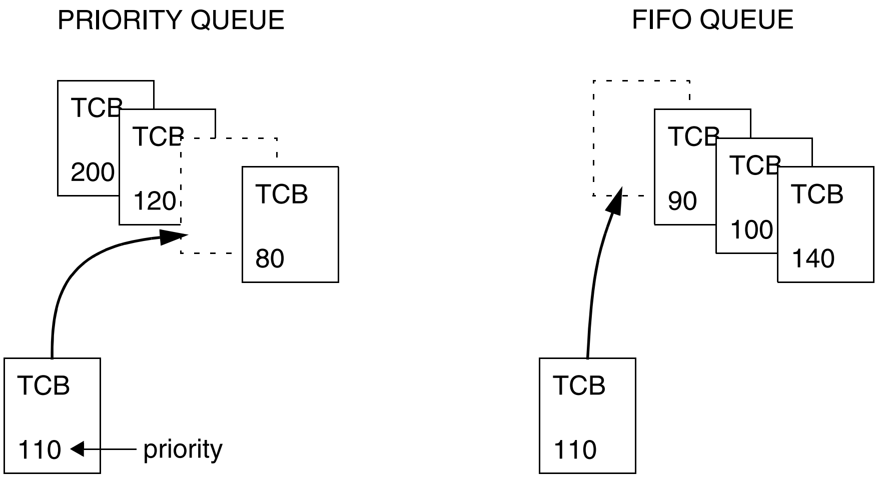
This behavior is controlled by a parameter to **semTake( )** and the take routines for read/write semaphores that specifies the amount of time in ticks that the task is willing to wait in the pended state. If the task succeeds in taking the semaphore within the allotted time, the take routine returns **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*.

Semaphores and Queueing

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 6-12.

Figure 6-12 **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.

Semaphores Interruptible by Signals

By default, a task pending on a semaphore is not interruptible by signals. A signal is only delivered to a task when it is no longer pending (having timed out or acquired the semaphore).

This behavior can be changed for binary and mutex semaphores by using the **SEM_INTERRUPTIBLE** option when they are created. A task can then receive a signal while pending on a semaphore, and the associated signal handler is executed. However, the **semTake( )** call then returns **ERROR** with errno set to **EINTR** to indicate that a signal occurred while pending on the semaphore (the task does not return to pending).

Semaphores and VxWorks Events

Semaphores can send VxWorks events to a specified task when they becomes free. For more information, see *6.16 VxWorks Events*, p.128.

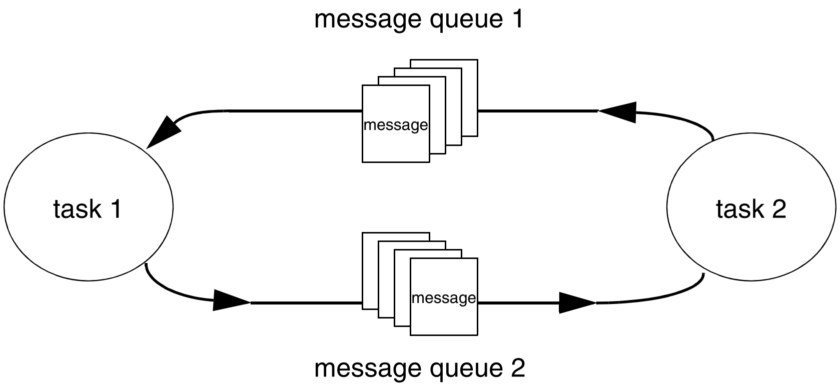
6.14 Message Queues

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 *6.17 Message Channels*, p.133.

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 6-13 **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 6-13.

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 *7.16.1 Comparison of POSIX and VxWorks Semaphores*, p.180 for a discussion of the differences between the two message-queue designs.

6.14.1 Inter-Process Communication With Public Message Queues

VxWorks message queues 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 are accessible throughout the system.

For detailed information, see *6.9 Inter-Process Communication With Public Objects*, p.107 and the **msgQLib** entry in the *VxWorks Application API Reference*.

6.14.2 VxWorks Message Queue Routines

VxWorks message queues are created, used, and deleted with the routines shown in Table 6-14. 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 6-14 **VxWorks Message Queue Control**

Routine	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.

Message Queue Timeout

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.

Message Queue 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 6-3 **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);
    }
```

Message Queues Interruptible by Signals

By default, a task pending on a message queue is not interruptible by signals. A signal is only delivered to a task when it is no longer pending (having timed out or acquired the message queue).

This behavior can be changed by using the **MSG_Q_INTERRUPTIBLE** option when creating a message queue. A task can then receive a signal while pending on a message queue, and the associated signal handler is executed. However, the **msgQSend( ) or msgQReceive( )** call then returns **ERROR** with errno set to **EINTR** to indicate that a signal occurred while pending on the message queue (the task does not return to pending).

Message Queues and Queuing Options

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( )**.

6.14.3 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):

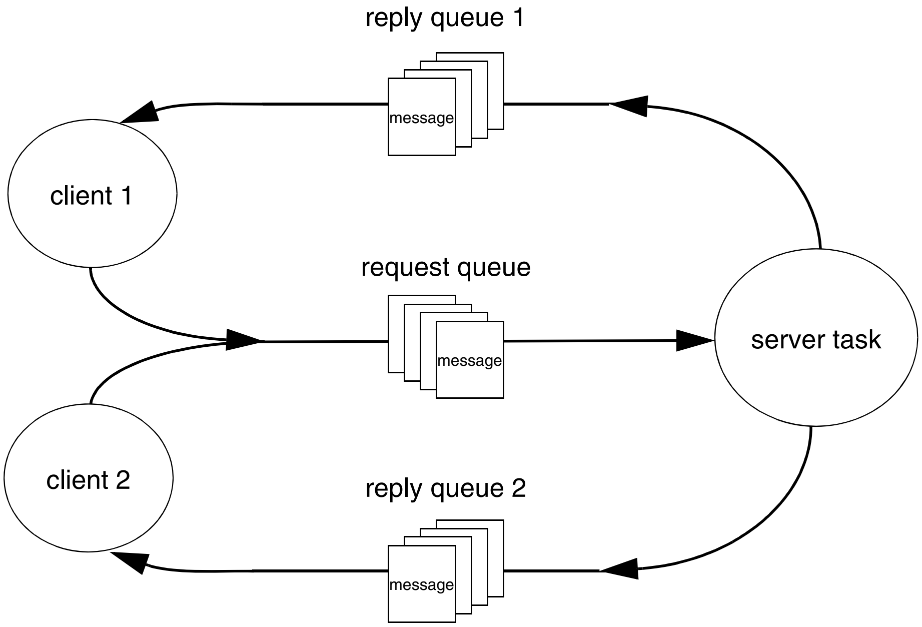
```
-> show myMsgQId
Message Queue Id  : 0x3adaf0
Task Queuing      : FIFO
Message Byte Len  : 4
Messages Max      : 30
Messages Queued   : 14
Receivers Blocked : 0
Send timeouts     : 0
Receive timeouts    : 0
```

6.14.4 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 *6.15 Pipes*, p.128) are a natural way to implement this functionality.

For example, client-server communications might be implemented as shown in Figure 6-14. 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 6-14 **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.

6.14.5 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 *6.16 VxWorks Events*, p.128.

6.15 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.

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 *9.4.9 Pending on Multiple File Descriptors with select( )*, p.229.

Pipes allow you to implement a client-server model of intertask communications; see *6.14.4 Servers and Clients with Message Queues*, p.127.

6.16 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, 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

[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.

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.

6.16.1 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( )**.

6.16.2 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.

6.16.3 Accessing Event Flags

When events are sent to a task, they are stored in the task's events register (see *6.16.5 Task Events Register*, p.132), 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.

6.16.4 Events Routines

The routines used for working with events are listed in Table 6-15.

Table 6-15 **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**.

6.16.5 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 (*6.16.3 Accessing Event Flags*, p.131), a task cannot access the contents of its events registry directly.

Table 6-16 describes the routines that affect the contents of the events register.

Table 6-16 **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.

Table 6-16 **Routines That Modify the Task Events Register** (cont'd)

Routine	Effect on the Task 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.

6.16.6 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( )**

6.17 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 detailed information about message channels, including a comparison of message channels and message queues, see the *VxWorks Kernel Programmer's Guide: Message Channels*.

6.18 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 Programmer's Guide.* For information about TIPC networking, see *Wind River TIPC Programmer's Guide*.

6.19 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 user -space (process-based) task can raise a signal for any of the following:

- itself
- any other task in its process
- any public task in the system
- its own process
- any other process in the system

A user-space task cannot raise a signal for a kernel task (even if it is a public task; for information about public tasks, see *6.4.2 Task Names and IDs*, p.94.). By default, signals sent to a task in a process results in the termination of the process.

Unlike kernel signal generation and delivery, which runs in the context of the task or ISR that generates the signal, user-space signal generation is performed by the sender task, but the signal delivery actions take place in the context of the receiving task.

For processes, signal handling routines apply to the entire process, and are not specific to any one task in the process.

Signal handling is done on an process-wide basis. That is, if a signal handler is registered by a task, and that task is waiting for the signal, then a signal to the process is handled by that task. Otherwise, any task that does not have the signal blocked will handle the signal. If there is no task waiting for a given signal, the signal remains pended in the process until a task that can receive the signal becomes available.

Each task has a signal mask associated with it. The signal mask determines which signals the task accepts. When a task is created, its signal mask is inherited from the task that created it. If the parent is a kernel task (that is, if the process is spawned from the kernel), the signal mask is initialized with all signals unblocked. It also inherits default actions associated with each signal. Both can later be changed with **sigprocmask( )**.

VxWorks provides a software signal facility that includes POSIX 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.

The following non-POSIX APIs are also provided for signals: **taskSigqueue( )**, **taskKill( )**, **rtpKill( )**, and **taskRaise( )**. These APIs are provided to facilitate porting VxWorks kernel applications to RTP applications.

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 the kernel, see *VxWorks Kernel 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 *6.16 VxWorks Events*, p.128.

6.19.1 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 real-time processes (and not in the kernel). It therefore requires that VxWorks also be configured with the **INCLUDE_POSIX_PTHREAD_SCHEDULER**, **INCLUDE_POSIX_TIMERS**, and

INCLUDE_POSIX_CLOCKS components. To be fully compliant with the PSE52 profile (Realtime Controller System Profile), the **BUNDLE_RTP_POSIX_PSE52** component bundle should be used.

The maximum number of queued signals in a process is set with the configuration parameter **RTP_SIGNAL_QUEUE_SIZE**. The default value, 32, is set in concordance with the POSIX 1003.1 standard (**_POSIX_SIGQUEUE_MAX**). Changing the value to a lower number may cause problems for applications relying on POSIX guidelines.

Process-based (RTP) applications are automatically linked with the **sigLib** library when they are compiled. Initialization of the library is automatic when the process starts.

6.19.2 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.

VxWorks also provides a POSIX and BSD-like **kill( )** routine, which sends a signal to a task.

For a list and description of the basic set of POSIX signal routines provided by VxWorks for use with processes, see Table 6-17.

Table 6-17 **Basic POSIX 1003.1b Signal Calls**

Routine	Description
signal( )	Specifies the handler associated with a signal.
kill( )	Sends a signal to a process.
raise( )	Sends a signal to the caller's process.
sigaction( )	Examines or sets the signal handler for a signal.
sigsuspend( )	Suspends a task until a signal is delivered.
sigpending( )	Retrieves a set of pending signals blocked from delivery.

sigemptyset( )
sigfillset( )
sigaddset( )
sigdelset( )
sigismember( )
	Manipulates a signal mask.

sigprocmask( )	Sets the mask of blocked signals.
sigprocmask( )	Adds to a set of blocked signals.
sigaltstack( )	Set or get a signal's alternate stack context.

For more information about signal routines, see the VxWorks API reference for **sigLib**.

6.19.3 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 6-18. For detailed information on signals, see the API reference for **sigLib**.

Table 6-18 **POSIX 1003.1b Queued Signal Routines**

Routine	Description
sigqueue( )	Sends a queued signal to a process.
sigwaitinfo( )	Waits for a signal.
sigtimedwait( )	Waits for a signal with a timeout.

Additional non-POSIX VxWorks queued signal routines are described in Table 6-19. These routines are provided for assisting in porting VxWorks 5.*x* kernel applications to processes. The POSIX routines described in Table 6-18 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 6-19 **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 6-4 **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 an RTP application, VxWorks must be configured with **BUNDLE_NET_SHELL** and **BUNDLE_RTP_POSIX_PSE52**.

To send a queued signal to a process:

1. Build the application as an RTP application.

2. Spawn the application. For example:

```
-> rtpSp "signal_ex.vxe"
value = 10531472 = 0xa0b290
-> Task 0x10000 installed signal handler for signal # 30.  Ready for signal.
```

3. Determine the RTP ID; for example:

```
-> rtpShow

       NAME              ID          STATE      ENTRY ADDR  OPTIONS   TASK CNT
-------------------- ---------- --------------- ---------- ---------- --------
signal_ex.vxe          0xa0b290 STATE_NORMAL    0x10000298        0x1        1

value = 1 = 0x1
```

4. From a kernel task (note that the kernel shell runs as a kernel task), use the **sig( )** helper routine to send a queued signal to an RTP, where the *id* parameter is the RTP ID. For example, using the RTP ID of 0xa0b290 found in the previous step:

```
-> sig 0xa0b290, 50
value = 0 = 0x0
-> Task 0x10000 got signal # 30  signal value 50
```

To send a queued signal to a task in a process:

1. Build the code as an RTP application.

2. Spawn the application.

3. From a kernel task (such as the kernel shell), use the **sig( )** helper routine to send a queued signal to an RTP task, where the *id* parameter is the RTP task ID.

For information on using the code in the kernel (as a kernel application), see the *VxWorks Application Programmer's Guide: Multitasking*.

6.19.4 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.

As noted above, the **SIGEV_THREAD** option is only supported in processes, and it requires that VxWorks 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).

6.19.5 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 6-20. 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 6-20 **Routines Callable by Signal Handlers**

Library	Routines
bLib	All routines
errnoLib	**errnoGet( )**, **errnoSet( )**
eventLib	**eventSend( )**
logLib	**logMsg( )**
lstLib	All routines except **lstFree( )**
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( )**, **taskIsPended( )**, **taskIsDelayed( )**, **taskTcb( )**
tickLib	**tickAnnounce( )**, **tickSet( )**, **tickGet( )**

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 parent process 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.

6.20 Timers

VxWorks provides watchdog timers, but they can only be used in the kernel (see *VxWorks Kernel Programmer's Guide: Multitasking*. However, real-time process (RTP) applications can use POSIX timers. For information in this regard, see *7.9 POSIX Clocks and Timers*, p.155).

6.20.1 Inter-Process Communication With Public Timers

VxWorks timers 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 are accessible throughout the system.

For detailed information, see *6.9 Inter-Process Communication With Public Objects*, p. 107 and the **timerLib** entry in the VxWorks Application API references.

7
POSIX Facilities

7.1 Introduction

The VxWorks user space environment provides the execution environment of choice for POSIX applications. The compliance to the POSIX standard is very high for all the APIs and behaviors expected by the PSE52 profile (Realtime Controller System Profile), which is described by IEEE Std 1003.13-2003 (POSIX.13). Various APIs which operate on a task in kernel mode, operate on a process in user mode—such as **kill( )**, **exit( )**, and so on.

Support for a conforming application is achieved in a process (RTP) in the following way:

- By configuring the VxWorks kernel with the appropriate functionality, such as a POSIX-compliant file system and scheduler.
- By using only POSIX PSE52 profile APIs in the application.

This implementation of the POSIX.13 PSE52 profile is based on IEEE Std 1003.1-2001 (POSIX.1). As defined by the PSE52 profile, it is restricted to individual processes.

VxWorks does provide a process model that allows multiple processes, and provides many POSIX.1 APIs associated with facilities such as networking and inter-process communication that are not part of the PSE52 profile.

Note that the only VxWorks file system that is POSIX-compliant is the Highly Reliable File System (for more information, see *10.4 Highly Reliable File System: HRFS*, p. 249).

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 *7.12 POSIX Page-Locking Interface*, p. 160).
- 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 *7.13 POSIX Threads*, p. 160 and *7.15 POSIX and VxWorks Scheduling*, p. 171).

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 *7.16.1 Comparison of POSIX and VxWorks Semaphores*, p. 180, 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 for real-time processes (RTPs). For information about facilities available in the VxWorks kernel, see the corresponding chapter in the *VxWorks Kernel Programmer's Guide*.

7.2 Configuring VxWorks with POSIX Facilities

Process-based (RTP) applications are automatically linked with the appropriate user-level POSIX libraries when they are compiled. The libraries are automatically initialized at run time.

Some user-level POSIX features also require support from the kernel. If VxWorks is not configured with the POSIX components required by an application using these features, the calls that lack support return an **ENOSYS** error at run-time.

Note that support for POSIX threads (pthreads) in processes requires that the kernel be configured with the POSIX thread scheduler component **INCLUDE_POSIX_PTHREAD_SCHEDULER**.

⚠ **CAUTION:** The set of components used for POSIX support in user space is not the same as the set of components used for POSIX support in kernel space. For information about the components for kernel space, see Table 7-1 and the *VxWorks Kernel Programmer's Guide: POSIX Facilities* for the appropriate component bundle.

7.2.1 POSIX PSE52 Support

The components required for POSIX PSE52 support within processes are provided with the **BUNDLE_RTP_POSIX_PSE52** 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.

If an application requires a PSE52 file system, then VxWorks must be configured with the following components as well:

- **INCLUDE_HRFS** for the highly reliable file system. See *10.4 Highly Reliable File System: HRFS*, p. 249.

- The appropriate device driver component —for example **INCLUDE_ATA** or **INCLUDE_XBD_RAMDRV**—and **INCLUDE_XBD_BLK_DEV** if the device driver requires it (that is, if it is not XBD-compatible). See the *VxWorks Kernel Programmer's Guide: I/O System*.

In addition, a **/tmp** directory must be created at run-time and must appear on the virtual root file system, which is provided with the **BUNDLE_RTP_POSIX_PSE52** component bundle (see *10.4.3 HRFS and POSIX PSE52*, p.251).

NOTE: Configuring VxWorks with support for POSIX PSE52 conformance (using **BUNDLE_RTP_POSIX_PSE52**) provides the **/dev/null** device required by the PSE52 profile. Note that the **devs** shell command lists **/dev/null** with other devices, but the **ls** command does not list it 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 VRFS, see *10.3 Virtual Root File System: VRFS*, p.248. For information about null devices, see *9.8.6 Null Devices*, p.242.

VxWorks PSE52 support in processes includes the following units of functionality (POSIX.13):

- **POSIX_C_LANG_JUMP**
- **POSIX_C_LANG_MATH**
- **POSIX_C_LANG_SUPPORT**
- **POSIX_DEVICE_IO**
- **POSIX_FD_MGMT**
- **POSIX_FILE_LOCKING**
- **POSIX_FILE_SYSTEM**
- **POSIX_SIGNALS**
- **POSIX_SINGLE_PROCESS**
- **POSIX_THREADS_BASE**
- **XSI_THREAD_MUTEX_EXT**
- **XSI_THREADS_EXT**

It also provides the following options (POSIX.1):

- **_POSIX_CLOCK_SELECTION**
- **_POSIX_FSYNC**
- **_POSIX_MAPPED_FILES**
- **_POSIX_MEMLOCK**
- **_POSIX_MEMLOCK_RANGE**
- **_POSIX_MESSAGE_PASSING**
- **_POSIX_MONOTONIC_CLOCK**
- **_POSIX_NO_TRUNC**
- **_POSIX_REALTIME_SIGNALS**
- **_POSIX_SHARED_MEMORY_OBJECTS**
- **_POSIX_SYNCHRONIZED_IO**
- **_POSIX_THREAD_ATTR_STACKADDR**
- **_POSIX_THREAD_ATTR_STACKSIZE**
- **_POSIX_THREAD_CPUTIME**
- **_POSIX_THREAD_PRIO_INHERIT**
- **_POSIX_THREAD_PRIO_PROTECT**
- **_POSIX_THREAD_PRIORITY_SCHEDULING**
- **_POSIX_THREAD_SAFE_FUNCTIONS**
- **_POSIX_THREAD_SPORADIC_SERVER**
- **_POSIX_THREADS**
- **_POSIX_TIMEOUTS**
- **_POSIX_TIMERS**

For information about POSIX namespace isolation for PSE52-conforming applications, see *7.5 POSIX Header Files*, p.151.

⚠ **CAUTION:** If a PSE52-conforming application relies on C99 constructs, the Wind River Compiler must be used. The version of the GNU compiler provided with this release does not support the C99 language constructs.

7.2.2 VxWorks Components for POSIX Facilities

Table 7-1 provides an overview of the individual VxWorks components that must be configured in the kernel to provide support for the specified POSIX facilities.

Table 7-1 **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
POSIX PSE52 support	N/A	BUNDLE_RTP_POSIX_PSE52

Networking facilities are described in the *Wind River Network Stack Programmer's Guide*.

7.3 General POSIX Support

Many POSIX-compliant libraries are provided for VxWorks. These libraries are listed in Table 7-2; see the API references for these libraries for detailed information.

Table 7-2 **POSIX Libraries**

Functionality	Library
Asynchronous I/O	**aioPxLib**
Buffer manipulation	**bLib**
Clock facility	**clockLib**
Directory handling	**dirLib**
Environment handling	C Library
Environment information	**sysconf** and **uname**
File duplication	**ioLib**
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
Network/Socket APIs	network libraries
String manipulation	C Library
Trace facility	**pxTraceLib**
Wide character support	C library

The following sections of this chapter describe the POSIX APIs available to user-mode applications 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.

7.4 Standard C Library: libc

The standard C library for user space (RTP applications) complies with the POSIX PSE52 profile. For information about the APIs provided in the C library, see the *VxWorks Application API Reference* and the *Dinkum C Library Reference Manual*.

7.5 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 user development environment are listed in Table 7-3. The *PSE52* column indicates compliance with the POSIX PSE52 profile.

Table 7-3 **POSIX Header Files**

Header File	PSE52	Description
aio.h	*	asynchronous input and output
assert.h	*	verify program assertion
complex.h	*	complex arithmetic
ctype.h	*	character types
dirent.h	*	format of directory entries
dlfcn.h	*	dynamic linking
errno.h	*	system error numbers
fcntl.h	*	file control options
fenv.h	*	floating-point environment
float.h	*	floating types
inttypes.h	*	fixed size integer types
iso646.h	*	alternative spellings
limits.h	*	implementation-defined constants
locale.h	*	category macros
math.h	*	mathematical declarations
mqueue.h	*	message queues
pthread.h	*	pthreads
sched.h	*	execution scheduling
search.h	*	search tables
semaphore.h	*	semaphores
setjmp.h	*	stack environment declarations
signal.h	*	signals
stdbool.h	*	boolean type and values
stddef.h	*	standard type definitions
stdint.h	*	integer types
stdio.h	*	standard buffered input/output
stdlib.h	*	standard library definitions
string.h	*	string operations
strings.h	*	string operations
sys/mman.h	*	memory management declarations

Table 7-3 **POSIX Header Files** (cont'd)

Header File	PSE52	Description
sys/resource.h	*	definitions for XSI resource operations
sys/select.h		select types
sys/stat.h		data returned by the **stat( )** function
sys/types.h		data types
sys/un.h		definitions for UNIX domain sockets
sys/utsname.h		system name structure
sys/wait.h		declarations for waiting
tgmath.h		type-generic macros
time.h		time types
trace.h		trace facility
unistd.h		standard symbolic constants and types
utime.h		access and modification times structure
wchar.h		wide-character handling
wctype.h		wide-character classification and mapping utilities

7.6 POSIX Namespace

The POSIX.1 standard guarantees that the namespace of strictly conforming POSIX applications is not polluted by POSIX implementation symbols.

For VxWorks, this namespace isolation can be enforced for all symbols that can be included by way of POSIX PSE52 header files, but does not extend to VxWorks library symbols (for more information, see *Limitations to Namespace Isolation*, p.153).

POSIX namespace isolation can, of course, only be enforced for applications that only include header files that are defined by the POSIX.1 standard. It cannot be enforced for applications that include native VxWorks header files.

Limitations to Namespace Isolation

For VxWorks, POSIX features are implemented using native VxWorks features. When VxWorks system libraries are linked with POSIX applications, therefore, some native VxWorks public functions and public variable identifiers become part of the application's namespace. Since native VxWorks public functions and public variable identifiers do not abide by POSIX naming conventions (in particular they do not generally start with an underscore) they may pollute a POSIX application's own namespace. In other words, the identifiers used by VxWorks native public functions and public variables must be considered as reserved even by POSIX applications.

POSIX Header Files

The POSIX header files that have been implemented in compliance with the POSIX PSE52 profile are identified in Table 7-3.

These header files are also used by traditional VxWorks applications, and may be included in native VxWorks header files (that is, header files that are not described by standards like POSIX or ANSI).

Using POSIX Feature-Test Macros

POSIX provides feature-test macros used during the compilation process to ensure application namespace isolation. These macros direct the compiler's pre-processor to exclude all symbols that are not part of the POSIX.1 authorized symbols in the POSIX header files.

The supported POSIX feature test macros and their values are as follows:

- **_POSIX_C_SOURCE** set to 200112L
- **_XOPEN_SOURCE** set to 600
- **_POSIX_AEP_RT_CONTROLLER_C_SOURCE** set to 200312L

Wind River supports use of these macros for use with POSIX PSE52 header files (see *7.5 POSIX Header Files*, p.151).

The macros must be defined *before* any other header file inclusions. Defining any of the three feature test macros with the appropriate value is sufficient to enforce POSIX namespace isolation with regard to header file symbols.

Note that the macros must be used with the values provided here to ensure the appropriate results. Also note that the POSIX.1 and POSIX.13 standards could evolve so that using different supported values for those macros, or defining these macros in various combinations, might yield different results with regard to which symbols are made visible to the application.

The **_POSIX_SOURCE** feature test macro is *not* supported since it has been superseded by **_POSIX_C_SOURCE**.

⚠ **CAUTION:** Only POSIX header files may be included in a POSIX application that is compiled with the POSIX feature test macros. Inclusion of any other header files may pollute the application's namespace, and may also result in compilation failures.

⚠ **CAUTION:** If a PSE52-conforming application relies on C99 constructs, the Wind River Compiler must be used. The version of the GNU compiler provided with this release does not support the C99 language constructs.

⚠ **CAUTION:** The C99 **_Bool** type is not compatible with the VxWorks **BOOL** type. The C99 **_Bool** type is intrinsically defined by the Wind River (**diab**) and GNU compilers, and has a 8-bit size. The VxWorks **BOOL** type is a macro (declared in **b_BOOL.h**), defined as an integer—that is, with a 32-bit size.

7.7 POSIX Process Privileges

POSIX describes the concept of *appropriate privileges* associated with a process with regard to functions and the options offered by these functions. The associated mechanism is implementation-defined. Currently VxWorks' POSIX implementation does not provide this kind of mechanism. Therefore, privileges are granted either to all callers of the function calls that may need appropriate privileges, or privileges are denied to all callers without exception. In the latter case, the **EPERM** errno is set or returned, as appropriate, by these functions.

Examples of functions that can return the **EPERM** errno include: **clock_settime( )**, and several pthreads APIs such as **pthead_cond_wait( )**.

7.8 POSIX Process Support

VxWorks provides support for a user-mode process model with real-time processes (RTPs). Table 7-4 provides a partial list of the POSIX APIs provided for manipulating processes. They take a **_pid_** argument (also known as an **RTP_ID** in VxWorks).

Table 7-4 **POSIX Process Routines**

Routine	Description
atexit( )	Register a handler to be called at **exit( )**.
_exit( )	Terminate the calling process (system call).
exit( )	Terminate a process, calling **atexit( )** handlers.
getpid( )	Get the process ID of the current process.
getppid( )	Get the process ID of the parent's process ID.
kill( )	Send a signal to a process.
raise( )	Send a signal to the caller's process.
wait( )	Wait for any child process to die.
waitpid( )	Wait for a specific child process to die.

Basic VxWorks process facilities, including these routines, are provided with the **INCLUDE_RTP** component.

7.9 POSIX Clocks and Timers

VxWorks provides POSIX.1 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.

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**.

In addition to system wide clocks, VxWorks supports thread CPU-time clocks for individual POSIX threads. A thread CPU-time clock measures the execution time of a specific pthread, including the time that the system spends executing system services on behalf of the pthread (that is, the time it runs in both user and kernel mode). The initial execution time is set to zero when the pthread is created.

Thread CPU-time clocks are implemented only for pthreads that run in processes (and not in the kernel). These clocks are accessible only through POSIX APIs.

Thread CPU-time clocks provide a means for detecting and managing situations in which a pthread overruns its assigned maximum execution time. Bounding the execution times of the pthreads in an application increases predictability and reliability.

Each POSIX thread has its own CPU-time clock to measure its execution time of pthread, which can be identified by using the **pthread_getcpuclockid( )** routine. The **CLOCK_THREAD_CPUTIME_ID** *clock_id* parameter refers to the calling thread's CPU-time clock.

See Table 7-5 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 7-5 **POSIX Clock Routines**

Routine	Description
clock_getres( )	Get the clock resolution (**CLOCK_REALTIME, CLOCK_MONOTONIC**, and thread CPU-time).
clock_setres( )	Set the clock resolution. Obsolete VxWorks-specific POSIX extension.
clock_gettime( )	Get the current clock time (**CLOCK_REALTIME, CLOCK_MONOTONIC**, and thread CPU-time).
clock_settime( )	Set the thread CPU-time clock (fails for **CLOCK_REALTIME** or **CLOCK_MONOTONIC**).
clock_nanosleep( )	Suspend the current pthread (or task) for a specified amount of time.
pthread_getcpuclockid( )	Get the clock ID for the CPU-time clock (for use with **clock_gettime( )** and **clock_settime( )**).

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.

Process-based (RTP) applications are automatically linked with the **clockLib** library when they are compiled. The library is automatically initialized when the process starts.

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 *6.19 Signals*, p.134).

See Table 7-6 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 7-6 **POSIX Timer Routines**

Routine	Description
timer_create( )	Allocate a timer using the specified clock for a timing base (**CLOCK_REALTIME, CLOCK_MONOTONIC**, or thread CPU-time of the calling thread).
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.
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 7-1 **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.

Process-based (RTP) applications are automatically linked with the **timerLib** library when they are compiled. The library is automatically initialized when the process starts.

7.10 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 *9.7 Asynchronous Input/Output*, p.234.

7.11 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.

7.12 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 except to validate the parameters passed, and if the addresses are passed, they are validated to ensure they are resident in memory. The routines do not perform any other 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 7-7.

Table 7-7 **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.

Process-based (RTP) applications are automatically linked with the **mmanPxLib** library when they are compiled.

7.13 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 *7.15 POSIX and VxWorks Scheduling*, p.171.

⚠ **CAUTION:** POSIX thread APIs must not be invoked in the context of custom system call handlers—either directly by system call handler itself or by any routine that the system call handler may use. Use of POSIX thread APIs in the context of system call handlers produces unspecified behavior and execution failure. For information about custom system calls, see the *VxWorks Kernel Programmer's Guide*.

7.13.1 POSIX Thread Stack Guard Zones

Execution-stack guard zones can be used with POSIX threads. A **SIGSEGV** signal is then generated when a stack overflow or underflow occurs.

Execution stack overflow protection can be provided for individual pthreads with a call to the **pthread_attr_setguardsize( )** routine before the pthread is created. By default the size of a pthread guard zone is one architecture-specific page (**VM_PAGESIZE**). It can be set to any value with the **pthread_attr_setguardsize( )** routine, but the actual size is always rounded to a multiple of a number of memory pages.

The size of the guard area must be set in a pthread attribute *before* a thread is created that uses the attribute. Once a thread is created, the size of its stack guard cannot be changed. If the size attribute is changed after the thread is created, it does not affect the thread that has already been created, but it does apply to the next thread created that uses the attribute. If a pthread's stack is created with guard protection, the **pthread_attr_getguardsize( )** routine can be used to get the size of the guard zone.

Note that the default overflow guard zone size for *pthreads* is independent of the size of the execution stack overflow guard zone for *tasks* (with is defined globally for tasks, and provided for them with the **INCLUDE_RTP** component).

Also note that pthreads are provided with execution stack *underflow* guard zones by default (they are provided for both pthreads and tasks by the INCLUDE_RTP component).

For information about task guard zones, see *6.4.5 Task Stack*, p.96.

7.13.2 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.

7.13.3 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( )**

7.13.4 Specifying Attributes when Creating Pthreads

The following examples create a pthread using the default attributes and use explicit attributes.

Example 7-2 **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 7-3 **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 7-4 **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);
```

7.13.5 POSIX Thread Creation and Management

VxWorks provides many POSIX thread routines. Table 7-8 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 7-8 **POSIX Thread Routines**

Routine	Description
pthread_create( )	Create a pthread.
pthread_cancel( )	Cancel the execution of a pthread

Table 7-8 **POSIX Thread Routines** (cont'd)

Routine	Description
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.
pthread_setschedprio( )	Dynamically set scheduling priority 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.
pthread_getconcurrency( )	Get level of concurrency. In VxWorks this routine does nothing.
pthread_setconcurrency( )	Set level of concurrency In VxWorks this routine does nothing.

7.13.6 POSIX Thread Attribute Access

The POSIX attribute-access routines are described in Table 7-9. The VxWorks-specific POSIX extension routines are described in section *7.13.3 VxWorks-Specific Pthread Attributes*, p.161.

Table 7-9 **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_getstack( )	Get the values of the stack address and size attributes.
pthread_attr_setstack( )	Set the values of the stack address and size attributes.
pthread_attr_getguardsize( )	Get the stack guard zone size attribute.
pthread_attr_setguardsize( )	Set the stack guard zone size attribute.

Table 7-9 **POSIX Thread Attribute-Access Routines** (cont'd)

Routine	Description
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.
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.

7.13.7 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 a process.

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.

7.13.8 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 7-10.

Table 7-10 **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( )**, **waitpid( )**
timerLib	**sleep( )**, **nanosleep( )**

[1] Asynchronous cancellation is actually implemented with a special signal, **SIGCNCL**, which users should be careful not to block or to ignore.

Routines that can be used with cancellation points of pthreads are listed in Table 7-11.

Table 7-11 **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.

7.14 POSIX Thread Mutexes and Condition Variables

Pthread mutexes (mutual exclusion variables) and condition variables comply with the POSIX.1-2001 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 three attributes: priority ceiling, protocol and type.

The routines used to manage these attributes are described below. For more information about these and other mutex routines, see the VxWorks API reference for **pthreadLib**.

7.14.1 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 7-12 and Table 7-13 (respectively).

Table 7-12 **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.

Table 7-12 **POSIX Routines Acting on a Mutex Object** (cont'd)

Routine	Description
pthread_mutex_trylock( )	Check and lock a mutex if available.
pthread_mutex_unlock( )	Unlock a mutex.
pthread_mutex_timedlock( )	Lock a mutex with timeout.

Table 7-13 **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.
pthread_mutexattr_gettype( )	Get a mutex **type** attribute.
pthread_mutexattr_settype( )	Set a mutex **type** attribute.

Type Mutex Attribute

The **type** mutex attribute controls the behavior of the mutex object when a pthread attempts to lock and unlock a mutex. The possible values are **PTHREAD_MUTEX_NORMAL**, **PTHREAD_MUTEX_ERRORCHECK**, **PTHREAD_MUTEX_RECURSIVE**, and **PTHREAD_MUTEX_DEFAULT** which is mapped to the **PTHREAD_MUTEX_NORMAL** type on VxWorks.

Mutexes of types **PTHREAD_MUTEX_NORMAL** and **PTHREAD_MUTEX_DEFAULT** do not detect deadlocks, so a pthread will deadlock if it attempts to lock a mutex it has locked already. A pthread attempting to unlock a mutex of this type that is locked by another pthread, or is already unlocked, will get the **EPERM** error.

Mutexes of type **PTHREAD_MUTEX_ERRORCHECK** do detect a deadlock situation, so a pthread will get the **EDEADLK** error if it attempts to lock a mutex that it has locked already. A pthread attempting to unlock a mutex of this type that is locked by another pthread, or is already unlocked, will get the **EPERM** error.

Mutexes of type **PTHREAD_MUTEX_RECURSIVE** allow a pthread to re-lock a mutex that it has already locked. The same number of unlocks as locks is required to release the mutex. A pthread attempting to unlock a mutex of this type that has been locked by another pthread, or is already unlocked, will get the **EPERM** error.

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 *6.13.4 Mutual-Exclusion Semaphores*, p.116).

- Attribute Name: **protocol**
- Possible Values: **PTHREAD_PRIO_NONE**, **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT**
- Access Routines: **pthread_mutexattr_getprotocol( )** and **pthread_mutexattr_setprotocol( )**

By default, a mutex is created with the **PTHREAD_PRIO_NONE** protocol, which ensures that owning the mutex does not modify the priority and scheduling characteristics of a pthread. This may, however, not be appropriate in situations in which a low priority pthread can lock a mutex that is also required by higher priority threads. The **PTHREAD_PRIO_INHERIT** and **PTHREAD_PRIO_PROTECT** protocols can be used to address this issue.

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 *7.15.2 POSIX and VxWorks Priority Numbering*, p.173.

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.

7.14.2 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 7-12 and Table 7-13 (respectively).

Table 7-14 **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 7-15 **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.
pthread_condattr_getclock( )	Get the clock selection condition variable attribute.
pthread_condattr_setclock( )	Set the clock selection condition variable attribute.

Clock Selection

The only attribute supported for the condition variable object is the clock ID. By default the **pthread_cond_timedwait( )** routine uses the **CLOCK_REALTIME** clock. The clock type can be changed with the **pthread_condattr_setclock( )** routine. The accepted clock IDs are **CLOCK_REALTIME** (the default) and **CLOCK_MONOTONIC**. For information about POSIX clocks, see *7.9 POSIX Clocks and Timers*, p.155.

7.15 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, and is in any case not advisable.

Table 7-16 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. 177.)
- 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 7-16 is discussed in detail in subsequent sections.

Table 7-16 **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.

7.15.1 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 *7.15.2 POSIX and VxWorks Priority Numbering*, p.173.
- 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.

7.15.2 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.

7.15.3 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.

7.15.4 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 *7.15.3 Default Scheduling Policy*, p.173), and concurrent policies cannot be applied to individual pthreads. For general information about the traditional scheduler and how it works with tasks, see *6.3 Task Scheduling*, p.88.

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.176.
- **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 *7.15.5 POSIX Threads Scheduler*, p.174.

7.15.5 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 *7.15.2 POSIX and VxWorks Priority Numbering*, p.173.

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 *6.3 Task Scheduling*, p.88 for information about the traditional scheduler and how it works with VxWorks tasks, and *7.15.4 VxWorks Traditional Scheduler*, p.173 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.177).

- **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 where pthreads and tasks are placed in the ready queue when their priorities are lowered.
- 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 remains at the *front* of the ready queue priority list for that priority (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 repositions pthreads that have had their priority lowered differently in the ready queue than 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 *front* of the ready queue list for that priority.
- When the priority of a *task* is lowered—with the **taskPrioritySet( )** routine—the POSIX thread scheduler places it at the *end* of the ready queue list for that priority, 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 of the same priority in the ready queue). For example, if a task and a pthread each have their priority lowered to effectively the same level, the pthread will be at the *front* of the list for that priority and the task will be at the end of the list. The pthread will run before any other pthreads (or tasks) at this level, and the task after any other tasks (or pthreads).

Note that 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.

For information about the ready queue, see *Scheduling and the Ready Queue*, p.91.

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).

7.15.6 POSIX Scheduling Routines

The POSIX 1003.1b scheduling routines provided by the **schedPxLib** library for VxWorks are described in Table 7-17.

Table 7-17 **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 that the POSIX priority numbering scheme is the inverse of the VxWorks scheme. For more information see *7.15.2 POSIX and VxWorks Priority Numbering*, p.173.

To include the **schedPxLib** library in the system, configure VxWorks with the **INCLUDE_POSIX_SCHED** component.

Process-based (RTP) applications are automatically linked with the **schedPxLib** library when they are compiled.

7.15.7 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.

User tasks and pthreads 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.

Note that a non-null result does not imply that the POSIX thread calling this routine is being scheduled with the **SCHED_RR** policy. To make this determination, a pthread must use the **pthread_getschedparam( )** routine.

Example 7-5 **Getting the POSIX Round-Robin Time Slice**

```
/*
 * The following example gets the length of the time slice,
 * and then displays the time slice.
 */

/* includes */
```

```
#include <time.h>
#include <sched.h>

STATUS rrgetintervalTest (void)
{
struct timespec slice;

if (sched_rr_get_interval (0, &slice) == ERROR)
{
printf ("get-interval test failed\n");
return (ERROR);
}
printf ("time slice is %ld seconds and %ld nanoseconds\n",
slice.tv_sec, slice.tv_nsec);
return (OK);
}
```

7.16 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 7-18.

Table 7-18 **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.

[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.

Table 7-18 **POSIX Semaphore Routines** (cont'd)

Routine	Description
sem_getvalue( )	Get the value of a semaphore.
sem_timedwait( )	Lock a semaphore with a timeout.

Note that the behavior of the user space **sem_open( )** routine complies with the POSIX specification, and thus differs from the kernel version of the routine. The kernel version of **sem_open( )** returns a reference copy for the same named semaphore when called multiple times, provided that **sem_unlink( )** is not called. The user space version of **sem_open( )** returns the same ID for the same named semaphore when called multiple times.

To include the POSIX **semPxLib** library semaphore routines in the system, configure VxWorks with the **INCLUDE_POSIX_SEM** component.

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.

Process-based (RTP) applications are automatically linked with the **semPxLib** library when they are compiled. The library is automatically initialized when the process starts.

7.16.1 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 *6. 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.

7.16.2 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.

When used in a user application, unnamed semaphores can be accessed only by the tasks belonging to the process executing the application.

Only public named semaphores (that is, named semaphores whose name begins with a forward slash) can be shared between processes or between processes and the kernel. For information about public objects, see *6.9 Inter-Process Communication With Public Objects*, p.107.

Example 7-6 **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 */
     }
```

7.16.3 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 7-19.

Table 7-19 **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]

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.

[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).

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 *6.9 Inter-Process Communication With Public Objects*, p.107.

Example 7-7 **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;
        }
    }
```

7.17 POSIX Message Queues

The POSIX message queue routines, provided by **mqPxLib**, are shown in Table 7-20.

Table 7-20 **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.

Table 7-21 describes the permissions that are allowed.

Table 7-21 **Message Queue Permissions**

Permission When Created	Access Permitted in Same Process	Access Forbidden
O_RDONLY	**O_RDONLY**	**O_RDWR**, **O_WRONLY**
O_WRONLY	**O_WRONLY**	**O_RDWR**, **O_RDONLY**
O_RDWR	**O_RDWR**, **O_WRONLY**, **O_RDONLY**	

Process-based (RTP) applications are automatically linked with the **mqPxLib** library when they are compiled. Initialization of the library is automatic as well, when the process is started.

For information about the VxWorks message queue library, see the **msgQLib** API reference.

7.17.1 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 7-22.

Table 7-22 **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)

7.17.2 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 7-8 **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);
    }
```

7.17.3 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 *6.9 Inter-Process Communication With Public Objects*, p.107.

Example 7-9 **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");
    }
```

7.17.4 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 7-10 **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);
    }
    }
```

7.18 POSIX Signals

VxWorks provides POSIX signal routines in user space. They described in *6.19 Signals*, p.134.

7.19 POSIX Memory Management

VxWorks provides the following POSIX memory management features in user-space (for real-time processes):

- Dynamic memory allocation—with the **calloc( ) malloc( ) realloc( )** and **free( )** routines. See *8.3 Heap and Memory Partition Management*, p.208.
- POSIX shared memory objects (the **_POSIX_SHARED_MEMORY_OBJECTS** option). See *7.19.3 Shared Memory Objects*, p.199.
- POSIX memory-mapped files (the **_POSIX_MAPPED_FILES** option). See *7.19.4 Memory Mapped Files*, p.200.
- POSIX memory protection (the **_POSIX_MEMORY_PROTECTION** options). See *7.19.5 Memory Protection*, p.200.
- POSIX memory locking (the **_POSIX_MEMLOCK** and **_POSIX_MEMLOCK_RANGE** options). See *7.19.6 Memory Locking*, p.200.

In addition, VxWorks also supports anonymous memory mapping, which is not a POSIX standard, but an implementation-specific extension of it.

7.19.1 POSIX Memory Management APIs

The POSIX memory management APIs provided by VxWorks facilitate porting applications from other operating systems. Note, however, that the VxWorks implementation of POSIX memory management facilities is based on requirements for real-time operating systems, which include determinism, small-footprint, and scalability. Features that are commonly found in general-purpose operating systems, such as demand-paging and copy-on-write, are therefore not supported in VxWorks. This ensures deterministic memory access and execution time, but also means that system memory limits the amount of virtual memory that processes can map. In other words, memory mapped in processes are always memory resident. Similarly, some APIs that are not relevant to real-time operating systems, such as those for memory locking are formally available to facilitate porting, but do not perform any function.

The **mmanLib** and **shmLib** routines are listed in Table 7-23.

Table 7-23 **POSIX Memory Management Routines**

Routine	Description
mmap( )	Establishes a mapping between a process' address space and a file, or a shared memory object, or directly with system RAM. In VxWorks the **MAP_FIXED** *flags* parameter option is not supported.
munmap( )	Un-maps pages of memory.
msync( )	Synchronizes a mapped file with a physical storage.
mprotect( )	Sets protection of memory mapping.
mlockall( )	Locks all pages used by a process into memory. In VxWorks this routine does nothing.
munlockall( )	Unlocks all pages used by a process. In VxWorks this routine does nothing.
mlock( )	Locks specified pages into memory. In VxWorks this routine does nothing.
munlock( )	Unlocks specified pages. In VxWorks this routine does nothing.
shm_open( )	Opens a shared memory object.
shm_unlink( )	Removes a shared memory object.

The restrictions on the VxWorks implementation of **mmanLib** are as follows:

- Mapping at fixed a address (**MAP_FIXED**) is not supported.
- The **munmap( )**, **mprotect( )**, **msync( )**, **mlock( )** routines only succeed for memory pages obtained with **mmap( )**.
- The **msync( )** routine must be called explicitly to ensure that data for memory mapped files is synchronized with the media.
- If the file size changes after a mapping is created, the change is not reflected in the mapping. It is, therefore, important to make sure that files are not changed with the standard file operations—**ftruncate( )**, **write( )**, and so on—while mappings are in effect. For example, the second **ftruncate( )** call in the following (abbreviated) example would have no effect on the mapping created with the **mmap( )** call:

```
fd = shm_open (...)
ftruncate (fd, size1);      /* set size */
addr1 = mmap (..., fd);     /* map it */
```

```
    ftruncate (fd, size2);     /* change size */
```

- Mappings that extend an existing shared mapping may not always succeed. This is because VxWorks uses memory models that do not ensure that adjacent virtual memory is available for a specific process.

7.19.2 Anonymous Memory Mapping

Anonymous memory-mapping is a VxWorks extension of the POSIX memory mapping APIs. It provides a simple method for a process to request (map) and release (un-map) additional pages of memory, without associating the mapping to a named file system object. This feature is always available when basic process (RTP) support included in VxWorks. Only private mappings are supported with the anonymous option.

The following example shows the use of **mmap( )** with the anonymous flag, as well as **mprotect( )**, and **unmap( )**.

```
#include <sys/mman.h>
#include <sys/sysctl.h>

/****************************************************************************
* main - User application entry function
*
* This application illustrates the usage of the mmap(),mprotect(),and
* munmap()
* API. It does not perform any other useful function.
*
* EXIT STATUS: 0 if all calls succeeded, otherwise 1.
*/

int main ()
    {
    int    mib[3]; /*MIB array for sysctl() */
    size_t pgSize;     /*variable to store page size */
    size_t sz = sizeof (pgSize);    /* size of pgSize variable */
    size_t bufSize;            /* size of buffer */
    char * pBuf;            /* buffer  */

    /* get "hw.mmu.pageSize" info */

    mib[0] = CTL_HW;
    mib[1] = HW_MMU;
    mib[2] = HW_MMU_PAGESIZE;

    if (sysctl (mib, 3, &pgSize, &sz, NULL, 0) != 0)
    exit (1);

    /* buffer size is 4 pages */

    bufSize = 4 * pgSize;

    /* request mapped memory for a buffer */

    pBuf = mmap (NULL, bufSize, (PROT_READ | PROT_WRITE),
         (MAP_PRIVATE | MAP_ANON), MAP_ANON_FD, 0);

/* check for error */

    if (pBuf == MAP_FAILED)
         exit (1);

    /* write protect the first page */

    if (mprotect (pBuf, pgSize, PROT_READ) != 0)
        {
    /*
```

```
 * no need to call unmap before exiting as all memory mapped for
 * a process is automatically released.
 */

exit (1);
}

/*
 * Unmap the buffer; the unmap() has to be called for the entire buffer.
 * Note that unmapping before exit() is not necesary; it is shown here
 * only for illustration purpose.
 */

if (munmap (pBuf, bufSize) != 0)
exit (1);

printf ("execution succeded\n");
exit (0);
}
```

For more information about the **mmanLib** routines, and an additional code example, see the VxWorks API reference for **mmanLib**.

7.19.3 Shared Memory Objects

VxWorks provides support for POSIX shared memory objects. With this type of mapping, the file descriptor used to call **mmap( )** is obtained with **shm_open( )**. Both shared and private mappings are supported. Support for this type of mapping is provided by two kernel components: a pseudo-file system called **shmFs** that provides the functionality for **shm_open( )** and **shm_unlink( )**, and the **mmap( )** extension for mapped files. These facilities are provided in the **INCLUDE_POSIX_SHM** and **INLUDE_POSIX_MAPPED_FILES** components, respectively.

The shared memory file system provides the name space for shared memory objects. It is a storage-less file system, which means that read, and write operations are not supported. The contents of shared memory objects can be managed exclusively by way of their memory mapped images.

Note that the functionality provided by shared memory objects functionality overlaps somewhat with the VxWorks native shared data region support. Table 7-24 summarizes their most important similarities and distinctions. For information about shared memory regions, see *3.5 Creating and Using Shared Data Regions*, p.38.

Table 7-24 **Shared Memory and Shared Data Regions**

Feature	**Shared Memory**	**Shared Data Regions**
Standard, API portability	POSIX	proprietary
Name space	file system	named object
Identifier	file descriptor	object ID
Ability to map at fixed physical address	no	yes
Ability to map at fixed virtual address	no	no
Available in the kernel	no	yes

Table 7-24 **Shared Memory and Shared Data Regions** (cont'd)

Feature	**Shared Memory**	**Shared Data Regions**
Unit of Operation[a]	MMU page	entire shared data region
Size management	dynamic[b]	set at creation

[a] *Unit of operation* means the portion of shared memory or a shared data region that can be operated on (mapped, unmapped, or protected). For shared memory the unit of operation is the MMU page (which means that shared memory can be partially mapped, unmapped, and protected on page basis). For shared data regions, all operations are on the entire region.

[b] If the file size changes after a mapping is created, the change is not reflected in the mapping. It is, therefore, important to make sure that files are not changed with the standard file operations—**ftruncate( )**, **write( )**, and so on—while mappings are in effect.

For more information, see the **shmLib** and **mmanLib** API references. For a code example, see the **shmLib** API.

7.19.4 Memory Mapped Files

With memory-mapped files, the file descriptor used to call **mmap( )** is obtained by opening a regular file in a POSIX-compliant file system. Both shared and private mappings are supported. This type of mapping type is available when VxWorks is configured with the **INLUDE_POSIX_MAPPED_FILES** component.

There is no automatic synchronization for memory mapped files, and there is no unified buffering for **mmap( )** and the file system. This means that he application must use **msync( )** to synchronize a mapped image with the file's storage media. The only exception to this rule is when memory is unmapped explicitly with **munmap( )**, or unmapped implicitly when the process exits. In that case, the synchronization is performed automatically during the un-mapping process.

For more information, including a code example, see the **mmanLib** API reference.

7.19.5 Memory Protection

Memory protection of mappings established with **mmap( )** can be changed using the **mprotect( )** routine. It works with both anonymous memory mapped files and with shared memory objects.

For more information see the **mprotect( )** API reference.

7.19.6 Memory Locking

In VxWorks, memory mappings are always memory resident. Demand paging and copy-on write are not performed. This ensures deterministic memory access for mapped files, but it also means that physical memory is always associated with mappings, until it is unmapped. The memory locking APIs provided with

VxWorks only perform address space validation, and have no effect on the mappings.

For more information see the **mmanLib** API reference.

7.20 POSIX Trace

VxWorks provide support for the POSIX trace facility in user-space for real-time processes. The trace facility allows a process to select a set of trace event types, to initiate a trace stream of events as they occur, and to retrieve the record of trace events.

The trace facility can be used to for debugging application code during development, for monitoring deployed systems, and for performance measurement. When used for debugging, it can be used at run-time as well as for postmortem analysis. For run-time debugging, the trace facility should be used with the filtering mechanism to provide focus on specific information an to avoid swamping the trace stream and the system itself. For postmortem analysis, collection of comprehensive data is desirable, and tracing can be used at selected intervals.

Note that the VxWorks kernel shell provides native system monitoring facilities similar to the BSD **ktrace** and Solaris **truss** utilities. For more information in this regard, see the *VxWorks Kernel Programmer's Guide: Target Tools*.

Trace Events, Streams, and Logs

A *trace event* is a recorded unit of the trace facility—it is the datum generated. Trace events include a trace event type identifier, a time-stamp, and other information. Standard POSIX trace events also have defined payloads; application-defined trace events may define a payload. Trace events are initiated either by a user application or by the trace system itself. The event structure is defined in *installDir***/vxworks-6.***x***/target/usr/h/base/b_struct_posix_trace_event_info.h**.

A *trace stream* is sequence of trace events, which are recorded in memory in a stream-buffer, and may be written to a file as well. The stream buffer is a ring-based buffer mechanism. Depending on the options selected when the stream is created, the buffer may stop logging when it is full, wrap (overwrite old data), or write the event data to a log file. When the trace facility is used without a trace log, the trace event data is lost when the tracing process is deleted or exits (note that the tracing process and the traced process may or may not be the same).

A *trace log* is a file to which a trace stream is written. A trace stream can be written to a log automatically (for example, when the **posix_trace_shutdown( )** routine is called) or on demand. The trace log file format is not specified by the POSIX trace specification. The trace log is write-only while it is being created, and read-only thereafter.

Trace events are registered by name, and then recorded with a numeric identifier. Some events are defined in the POSIX standard. These have fixed numeric IDs and names. The symbolic values, and the textual names, are in the standard (for example **POSIX_TRACE_START** and **posix_trace_start**, **POSIX_TRACE_OVERFLOW** and **posix_trace_overflow**, and so on). These events types are included in any log you create. Other event type ID and name mappings can be created with the **posix_trace_eventid_open( )** routine.

The **posix_trace_eventtypelist_getnext( )** can be used to retrieve an ID, and then **posix_trace_eventid_get_name( )** can be used to get the corresponding name.

There is a per-process limit on the number of events that are obtained, which may be obtained with the following call:

```
sysconf(TRACE_USER_EVENT_MAX);
```

Attempts to create event mappings beyond the limit return the special event ID **POSIX_TRACE_UNNAMED_USEREVENT**.

Trace Operation

The trace operation involves three logically distinct roles, each of which has its own APIs associated with it. These roles are the following:

- The *trace controller*, which governs the operation of recording trace events into the trace stream. This operation includes initializing the attributes of a trace stream, creating the stream, starting and stopping the stream, filtering the stream, and shutting the trace stream down, as well as other trace stream management and information retrieval functions.
- The *trace event generator*, which during execution of an instrumented application injects a trace record into a trace stream whenever a trace point is reached, as long as that type of trace has not been filtered out.
- The *trace analyzer*, which retrieves trace event data. It can either retrieve the traced events from an active trace stream or from a trace log file, extract stream attributes, look up event type names, iterate over event type identifiers, and so on.

While logically distinct, these roles can be performed by a the same process, or by different processes. A process may trace itself or another process; and may have several traces active concurrently.A process may also open and read a trace log, provided the log was generated on the same CPU architecture as the process that is going to read it.

A trace *event filter* can be used to define specific types of trace events to collect, and to reduce the overall amount of data that is collected. All traces have an event filter, which records all events by default.

Trace APIs

The basic set of trace routines is the following:

- **posix_trace_create( )**
- **posix_trace_eventid_open( )**
- **posix_trace_start( )**
- **posix_trace_stop( )**
- **posix_trace_getnext_event( )**
- **posix_trace_shutdown( )**

More of the routines used for the controller, generator, and analyzer roles are described in Table 7-25.

Table 7-25 **POSIX Trace Routines**

Routine	Description	Role
posix_trace_attr_init( )	Initializes the trace stream attributes object.	Controller
posix_trace_attr_getgenversion( )	Gets the generation version information.	Controller
posix_trace_attr_getinherited( )	Gets the inheritance policy.	Controller
posix_trace_attr_getmaxusereventsize( )	Calculates the maximum memory size required to store a single user trace event.	Controller
posix_trace_create( )	Creates an active trace stream.	Controller
posix_trace_clear( )	Re-initializes the trace stream.	Controller
posix_trace_trid_eventid_open( )	Associates a user trace event name with a trace event type identifier	Controller
posix_trace_eventtypelist_getnext_id( )	Returns the next event type.	Controller
posix_trace_eventset_empty( )	Excludes event types.	Controller
posix_trace_set_filter( )	Gets trace filter from a specified trace.	Controller
posix_trace_start( )	Starts a trace stream.	Controller
posix_trace_get_attr( )	Get a trace attributes.	Controller
posix_trace_eventid_open( )	Associates and event name with an event-type.	Generator
posix_trace_event( )	Adds event to trace stream.	Generator
posix_trace_attr_getgenversion( )	Gets version information.	Analyzer
posix_trace_attr_getinherited( )	Gets the inheritance policy.	Analyzer
posix_trace_attr_getmaxusereventsize( )	Gets the maximum total log size, in bytes.	Analyzer
posix_trace_trid_eventid_open( )	Associates a user trace event name with a trace event type identifier.	Analyzer
posix_trace_eventtypelist_getnext_id( )	Gets next trace event type identifier.	Analyzer
posix_trace_open( )	Opens a trace log file.	Analyzer
posix_trace_get_attr( )	Gets the attributes of the current trace stream.	Analyzer
posix_trace_getnext_event( )	Gets next trace event.	Analyzer

For more information about these and other APIs, see the **pxTraceLib** API reference entries.

To include the trace facility in your system, configure VxWorks with the **INCLUDE_POSIX_TRACE** component. This component is included automatically when the **BUNDLE_RTP_POSIX_PSE52** bundle is used.

Trace Code and Record Example

The section provides an example of code that uses the trace facility and a trace record that it produces.

Trace Code Example

```
LOCAL int eventShow
    (
    trace_id_t                                  trid,
    const struct posix_trace_event_info *       pEvent,
    const void *                                pData,
    int                                         dataSize
    )
    {
    int                                 i;
    int                                 result;
    char                                eventName [TRACE_EVENT_NAME_MAX];
    char *                              truncationMsg;
    char                                payloadData [32];
    const char *                        dataString = (const char *)pData;

    result = posix_trace_eventid_get_name (trid,
                                           pEvent->posix_event_id, eventName);
    if (result != 0)
        sprintf (eventName, "error: %d", result);

    vxTestMsg (V_GENERAL, "EventId:           %d (%s)",
               pEvent->posix_event_id,
               eventName);
    vxTestMsg (V_GENERAL, "PID:               0x%x",
               pEvent->posix_pid);
    vxTestMsg (V_GENERAL, "Prog address:      0x%x",
               pEvent->posix_prog_address);

    switch (pEvent->posix_truncation_status)
        {
        case POSIX_TRACE_NOT_TRUNCATED:
            truncationMsg = "POSIX_TRACE_NOT_TRUNCATED";
            break;

        case POSIX_TRACE_TRUNCATED_RECORD:
            truncationMsg = "POSIX_TRACE_TRUNCATED_RECORD";
            break;

        case POSIX_TRACE_TRUNCATED_READ:
            truncationMsg = "POSIX_TRACE_TRUNCATED_READ";
            break;

        default:
            truncationMsg = "Unknown";
        }

    vxTestMsg (V_GENERAL, "Truncation:        %d (%s)",
               pEvent->posix_truncation_status, truncationMsg);
    vxTestMsg (V_GENERAL, "Time:              (%d, %d)",
            pEvent->posix_timestamp.tv_sec, pEvent->posix_timestamp.tv_nsec);
    vxTestMsg (V_GENERAL, "ThreadId:          0x%x", pEvent->posix_thread_id);

    vxTestMsg (V_GENERAL, "Payload size:      %d", dataSize);

    for (i=0; i < (dataSize < 16 ? dataSize : 16); i++)
        {
        char    ch = dataString [i];
        if (isspace(ch) || isgraph(ch))
            payloadData [i] = ch;
        else
            payloadData [i] = '.';
        }
    payloadData [i] = '\0';
    vxTestMsg (V_GENERAL, "Payload data: %s", payloadData);
    return (0);
    }
```

Trace Event Record

```
EventId:           1025 (event1)
PID:               0x20001
Prog address:      0xa0002e90
Truncation:        0 (POSIX_TRACE_NOT_TRUNCATED)
Time:              (16, 1367900)
ThreadId:          0xa0025410
Payload size:      0
Payload data:

EventId:           1025 (event1)
PID:               0x20001
Prog address:      0xa0002e90
Truncation:        0 (POSIX_TRACE_NOT_TRUNCATED)
Time:              (16, 1368171)
ThreadId:          0xa0025410
Payload size:      0
Payload data:
```

8
Memory Management
in Processes

8.1 Introduction

This chapter describes the memory management facilities available to applications that execute as real-time processes. Each process has its own heap, and can allocate and free buffers from its heap with the routines provided in the **memLib** and **memPartLib** libraries. See *8.3 Heap and Memory Partition Management*, p.208.

In addition, run-time error detection facilities provide the ability to debug memory-related errors in application code. See *8.4 Memory Error Detection*, p.209.

User applications can also make use of the following facilities for shared memory and memory mapping:

- Native VxWorks shared data regions (see *3.5 Creating and Using Shared Data Regions*, p.38).
- POSIX memory management facilities (see *7.19 POSIX Memory Management*, p.196).

NOTE: This chapter provides information about facilities available for real-time processes. For information about the memory management facilities available in the VxWorks kernel, diagrams of memory for VxWorks configurations that includes processes, and instructions on how to configure VxWorks with process support—but without MMU support, see the *VxWorks Kernel Programmer's Guide: Memory Management*.

⚠ **CAUTION:** VxWorks SMP does not support MMU-less configurations. For information about VxWorks SMP (which supports RTPs), see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

8.2 Configuring VxWorks With Memory Management

When VxWorks is configured with real-time process support, it includes the basic memory management features required for process-based applications (see *2.3 Configuring VxWorks For Real-time Processes*, p.12).

For the error detection features provided by heap and memory partition instrumentation, the **INCLUDE_EDR_RTP_SHOW** component is also required (*8.4.2 Compiler Instrumentation*, p.214).

8.3 Heap and Memory Partition Management

VxWorks provides support for heap and memory partition management in processes. By default, the heap is implemented as a memory partition within the process.

The heap is automatically created during the process initialization phase. The initial size of the heap, and the automatic incrementation size, are configurable with the environment variables **HEAP_INITIAL_SIZE**, **HEAP_INCR_SIZE** and **HEAP_MAX_SIZE**. These environment variables only have effect if they are set when the application is started. The application cannot change it's own values. For more information on these environment variables, see the VxWorks API reference for **memLib**. For information about working with environment variables, see *2.2.8 RTPs and Environment Variables*, p.10, *3.3.1 RTP Application Structure*, p.31, and the API reference for **rtpSpawn( )**.

Memory partitions are contiguous areas of memory that are used for dynamic memory allocation by applications. Applications can create their own partitions and allocate and free memory from these partitions.

The heap and any partition created in a process are private to that process, which means that only that process is allowed to allocate memory to it, or free from it.

For more information, see the VxWorks API references for **memPartLib** and **memLib**.

Alternative Heap Manager

The VxWorks process heap implementation can be replaced by a custom version simply by linking the replacement library into the application (the replacement library must precede **vxlib.a** in the link order).

The memory used as heap can be obtained with either one of the following:

- A statically created array variable; for example:

  ```
  char heapMem[HEAP_SIZE];
  ```

  This solution is simple, but it creates a fixed sized heap.

- Using the dynamic memory mapping function, **mmap( )**. With **mmap( )**, it is possible to implement automatic or non-automatic growth of the heap. However, it is important to keep in mind that subsequent calls to **mmap( )** are not guaranteed to provide memory blocks that are adjacent. For more information about **mmap( )** see *7.19.1 POSIX Memory Management APIs*, p.196.

In case of applications that are dynamically linked with **libc.so** (which by default contains **memLib.o**), the default heap provided by **memLib** is automatically created. To avoid creation of the default heap, it is necessary to create a custom **libc.so** file that does not hold **memLib.o**. For information about creating a custom shared library that provides this functionality, please contact Wind River Support.

To ensure that process initialization code has access to the replacement heap manager early enough, the user-supplied heap manager must either:

- Initialize the heap automatically the very first time **malloc( )** or any other heap routine is called.
- Have its initialization routine linked with the application, and declared as an automatic constructor using the **_WRS_CONSTRUCTOR** macro with an initialization order lower than 6 (for information about this macro, see *4.5.1 Library and Plug-in Initialization*, p.57).

8.4 Memory Error Detection

Support for memory error detection is provided by the following optional instrumentation libraries:

- The **memEdrLib** library, which performs error checks of operations in the user heap and memory partitions in a process. This library can be linked to an executable compiled with either the Wind River Compiler or the GNU compiler.
- The Run-Time Error Checking (RTEC) facility, which checks for additional errors, such as buffer overruns and underruns, static and automatic variable reference checks. This feature is only provided by the Wind River Compiler.

Errors detected by these facilities are reported with the logging facility of the error detection and reporting component; which must be included in the VxWorks configuration to provide this functionality.

For information about configuring VxWorks for error detection and reporting, as well as information about additional facilities useful for debugging software faults, see *11. Error Detection and Reporting*.

NOTE: The memory error detection facilities described in this section are not included in any shared library provided by Wind River with this release. They can only be statically linked with application code.

8.4.1 Heap and Partition Memory Instrumentation

To supplement the error detection features built into **memLib** and **memPartLib** (such as valid block checking), the memory partition debugging library, **memEdrLib**, can be used to perform automatic, programmatic, and interactive error checks on **memLib** and **memPartLib** operations. This is performed by installing instrumentation hooks for **memPartLib**.

The instrumentation helps detect common programming errors such as double-freeing an allocated block, freeing or reallocating an invalid pointer, and memory leaks. In addition, with compiler-assisted code instrumentation, it helps 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.

VxWorks Kernel Configuration

A VxWorks kernel configured for process support (with the **INCLUDE_RTP** component) is sufficient for providing processes with heap instrumentation. The optional **INCLUDE_MEM_EDR_RTP_SHOW** component can be used to provide show routines for the heap and memory partition instrumentation. Note that the kernel's heap instrumentation component (**INCLUDE_MEM_EDR**) is not required.

Linking

In order to enable heap and memory partition instrumentation of a process, the executable must be linked with the **memEdrLib** library support included. This can be accomplished by using the following linker option (with either the Wind River or GNU toolchain):

-Wl,-umemEdrEnable

Note that the syntax is slightly different for the Coldfire architecture, for which underscores are pre-pended to symbol names, as follows:

-Wl,-u_memEdrEnable

For example, the following makefile based on the provided user-side build environment can be used:

```
EXE = heapErr.vxe
OBJS = main.o
LD_EXEC_FLAGS += -Wl,-umemEdrEnable
include $(WIND_USR)/make/rules.rtp
```

Alternatively, adding the following lines to the application code can also be used:

```
extern int memEdrEnable;
memEdrEnable = TRUE;
```

The location of these lines in the code is not important.

Environment Variables

When executing an application, the following environment variables may be set to override the defaults. The variables have to be set when the process is started. For information about working with environment variables, see *2.2.8 RTPs and Environment Variables*, p.10, *3.3.1 RTP Application Structure*, p.31, and the API reference for **rtpSpawn( )**.

MEDR_EXTENDED_ENABLE
Set to **TRUE** to enable saving trace information for each allocated block, but at the cost of increased memory used to store entries in the allocation database. Without this feature enabled the database entry for each allocated block is 32 bytes, with this feature enabled it is 64 bytes. 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. Default setting is **FALSE**.

MEDR_FREE_QUEUE_LEN
Maximum allowed length of the free queue. When a memory block is freed by an application, instead of immediately returning it to the memory pool, it is kept in a queue. When the queue reaches the maximum length allowed, the blocks are returned to the memory pool in a FIFO order. Queuing is disabled when this parameter is 0. Default setting 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, under-runs, and some heap memory corruption. Default setting is **FALSE**.

MEDR_POOL_SIZE
Set the size of the memory pool used to maintain the memory block database. Default setting in processes is 64 K. The database uses 32 bytes per memory block without extended information enabled, and 64 bytes per block with extended information enabled (call stack trace).

MEDR_SHOW_ENABLE
Enable heap instrumentation show support in the process. This is needed in addition to configuring VxWorks with the **INCLUDE_MEM_EDR_RTP_SHOW** component. When enabled, the kernel routines communicate with a dedicated task in the process with message queues. The default setting is **FALSE**.

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 a block address within an already allocated block from the same partition. This would indicate corruption in the partition data structures.
- Allocation returns a block address which is in the task's stack space. This would indicate corruption in the partition data structures.
- Allocation returns a block address that is in the kernel's static data section. This would indicate corruption in the partition data structures.
- Freeing a pointer which is in the task’s stack space.
- Freeing a memory that was already freed and is still in the free queue.
- Freeing memory which is in the kernel’s static data section.
- Freeing memory in a different partition than where it was allocated.
- Freeing a partial memory block.
- Freeing 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 8-1 are available for use with the shell's C and command interpreters to display information.

Table 8-1 **Shell Commands**

C Interpreter	Command Interpreter	Description
edrShow( )	**edr show**	Displays error records.
memEdrRtpPartShow( )	**mem rtp part list**	Displays a summary of the instrumentation information for memory partitions located in a given process.
memEdrRtpBlockShow( )	**mem rtp block list**	Displays information about allocated blocks in a given process. Blocks can be selected using a combination of various querying criteria: partition ID, block address, allocating task ID, block type.
memEdrRtpBlockMark( )	**mem rtp block mark** **mem rtp 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. Can be used to monitor memory leaks by displaying information of unmarked blocks with **memEdrRtpBlockShow( )** and **mem rtp block list**.

Code Example

The following application code can be used to generate various errors that can be monitored from the shell (line numbers are included for reference purposes). Its use is illustrated in *Shell Session Example*, p.213.

```
#include <vxWorks.h>
#include <stdlib.h>
#include <taskLib.h>
int main ()

    {
    char * pChar;

    taskSuspend(0);        /* stop here first */

    pChar = malloc (24);
    free (pChar + 2);      /* free partial block */
    free (pChar);
    free (pChar);          /* double-free block */
    pChar = malloc (32);   /* leaked memory */
    taskSuspend (0);       /* stop again to keep RTP alive */
    }
```

Shell Session Example

First set up the environment variables in the shell task. These variables will be inherited by processes created with **rtpSp( )**. The first environment variable enables trace information to be saved for each allocation, the second one enables the show command support inside the process.

```
-> putenv "MEDR_EXTENDED_ENABLE=TRUE"
value = 0 = 0x0
-> putenv "MEDR_SHOW_ENABLE=TRUE"
value = 0 = 0x0
```

Spawn the process using the executable produced from the example code:

```
-> rtp = rtpSp ("heapErr.vxe")
rtp = 0x223ced0: value = 36464240 = 0x22c6670
```

At this point, the initial process task (**iheapErr**), which is executing **main( )**, is stopped at the first **taskSuspend( )** call (line 9 of the source code). Now mark all allocated blocks in the process which resulted from the process initialization phase:

```
-> memEdrRtpBlockMark rtp
value = 27 = 0x1b
```

Next, clear all entries in the error log. This step is optional, and is used to limit the number of events displayed by the **edrShow( )** command that will follow:

```
    -> edrClear
    value = 0 = 0x0
```

Resume the initial task **iheapErr** to continue execution of the application code:

```
-> tr iheapErr
value = 0 = 0x0
```

After resuming the process will continue execution until the second **taskSuspend( )** call (line 17). Now list all blocks in the process that are unmarked. These are blocks that have been allocated since **memEdrRtpBlockMark( )** was called, but have not been freed. Such blocks are possible memory leaks:

```
-> memEdrRtpBlockShow rtp, 0, 0, 0, 5, 1

  Addr    Type    Size    Part ID  Task ID  Task Name       Trace
-------- ------ -------- -------- -------- ------------ ------------
30053970  alloc       32 30010698  22c8750    iheapErr  main()
                            malloc()
                            0x30004ae4()
value = 0 = 0x0
```

Display the error log. The first error corresponds to line 12 in the test code, while the second error corresponds to line 14.

```
-> 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:       4
Generation count: 6

==[1/2]==============================================================
Severity/Facility:   NON-FATAL/RTP
Boot Cycle:          4
OS Version:          6.0.0
Time:                THU JAN 01 00:09:56 1970 (ticks = 35761)
Task:                "iheapErr" (0x022c8750)
RTP:                 "heapErr.vxe" (0x022c6670)
RTP Address Space:   0x30000000 -> 0x30057000

freeing part of allocated memory block
       PARTITION: 0x30010698
       PTR=0x30053942
       BLOCK: allocated at 0x30053940, 24 bytes

<<<<<Traceback>>>>>

0x300001b4 _start       +0x4c : main ()
0x300001e4 main         +0x2c : free ()
0x30007280 memPartFree  +0x5c : 0x30004a10 ()
0x30004ac8 memEdrItemGet+0x6e8: 0x30004514 ()
0x30003cb8 memEdrErrorLog+0x138: saveRegs ()

==[2/2]==============================================================
Severity/Facility:   NON-FATAL/RTP
Boot Cycle:          4
OS Version:          6.0.0
Time:                THU JAN 01 00:09:56 1970 (ticks = 35761)
Task:                "iheapErr" (0x022c8750)
RTP:                 "heapErr.vxe" (0x022c6670)
RTP Address Space:   0x30000000 -> 0x30057000

freeing memory in free list
    PARTITION: 0x30010698
    PTR=0x30053940
    BLOCK: free block at 0x30053940, 24 bytes

<<<<<Traceback>>>>>

0x300001b4 _start       +0x4c : main ()
0x300001f4 main         +0x3c : free ()
0x30007280 memPartFree  +0x5c : 0x30004a10 ()
0x30004ac8 memEdrItemGet+0x6e8: 0x30004514 ()
0x30003cb8 memEdrErrorLog+0x138: saveRegs ()
value = 0 = 0x0
```

Finally, resume **iheapErr** again to allow it to complete and to be deleted:

```
-> tr iheapErr
value = 0 = 0x0
```

8.4.2 Compiler Instrumentation

Additional errors are detected if the application is compiled using the Run-Time Error Checking (RTEC) feature of the Wind River Compiler. The following flag should be used:

-Xrtc=*option*

NOTE: The Run-Time Error Checking (RTEC) 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 8-2 lists the **-Xrtc** options that are supported. Using the **-Xrtc** flag without specifying any option implements them all. An individual option (or bit-wise OR'd combinations of options) can be enabled using the following syntax:

-Xrtc=*option*

Table 8-2 **-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.

For information beyond what is provided in this section, see the *Wind River Compiler User's Guide: Run-Time Error Checker*.

Configuring VxWorks for RTEC Support

Support for this feature in the kernel is enabled by configuring VxWorks with the basic error detection and reporting facilities. See *11.2 Configuring Error Detection and Reporting Facilities*, p.280.

Shell Commands

The compiler provided instrumentation automatically logs errors detected in applications using the error detection and reporting facility. For information about using shell commands with error logs, see *11.4 Displaying and Clearing Error Records*, p.283.

Code Example

This application code generates various errors that can be recorded and displayed, if built with the Wind River Compiler and its **-Xrtc** option (line numbers are included for reference purposes). Its use is illustrated in *Shell Session Example*, p.216.

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

```
int main ()
    {
    char   name[] = "very_long_name";
    char * pChar;
    int    state[] = { 0, 1, 2, 3 };
    int    ix = 0;

    pChar = (char *) 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 */
                          /* of automatic variable */
    free (pChar);

    *pChar = '\0'; /* reference a free block */
    }
```

Shell Session Example

In the following shell session example, the C interpreter is used to execute **edrClear( )**, which clears the error log of any existing error records. Then the application is started with **rtpSp( )**. Finally, the errors are displayed with **edrShow( )**.

First, clear the error log. This step is only performed to limit the number of events that are later displayed, when the events are listed:

```
-> edrClear
value = 0 = 0x0
```

Start the process using the executable created from the sample code listed above:

```
-> rtpSp "refErr.vxe"
value = 36283472 = 0x229a450
```

Next, list the error log. As shown below, three errors are detected by the compiler instrumentation:

```
-> 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:       4
Generation count: 8
```

The first one is caused by the code on line 13. A string of length 14 is copied into a allocated buffer of size 13:

```
==[1/3]==============================================================
Severity/Facility:   NON-FATAL/RTP
Boot Cycle:          4
OS Version:          6.0.0
Time:                THU JAN 01 01:55:42 1970 (ticks = 416523)
Task:                "irefErr"   (0x0229c500)
RTP:                 "refErr.vxe" (0x0229a450)
RTP Address Space:   0x30000000 -> 0x30058000
Injection Point:     main.c:13

memory block bounds-check violation
    PTR=0x30054940  OFFSET=0  SIZE=14
    BLOCK: allocated at 0x30054940, 13 bytes
    <<<<<Traceback>>>>>

    0x300001b4 _start       +0x4c : main ()
    0x300002ac main         +0xf4 : __rtc_chk_at ()
```

The second error refers to line 17. 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/RTP
    Boot Cycle:          4
    OS Version:          6.0.0
    Time:                THU JAN 01 01:55:42 1970 (ticks = 416523)
    Task:                "irefErr"   (0x0229c500)
    RTP:                 "refErr.vxe" (0x0229a450)
    RTP Address Space:   0x30000000 -> 0x30058000
    Injection Point:     main.c:17

    memory block bounds-check violation
        PTR=0x30022f34  OFFSET=16  SIZE=4
        BLOCK: automatic at 0x30022f34, 16 bytes

    <<<<<Traceback>>>>>

    0x300001b4 _start       +0x4c : main ()
    0x300002dc main         +0x124: __rtc_chk_at ()
```

The last error is caused by the code on line 21. A memory block that has been freed is being modified:

```
==[3/3]==============================================================
    Severity/Facility:   NON-FATAL/RTP
    Boot Cycle:          4
    OS Version:          6.0.0
    Time:                THU JAN 01 01:55:42 1970 (ticks = 416523)
    Task:                "irefErr"   (0x0229c500)
    RTP:                 "refErr.vxe" (0x0229a450)
    RTP Address Space:   0x30000000 -> 0x30058000
    Injection Point:     main.c:21

    pointer to free memory block
        PTR=0x30054940  OFFSET=0  SIZE=1
        BLOCK: free block at 0x30054940, 13 bytes

    <<<<<Traceback>>>>>

    0x300001b4 _start       +0x4c : main ()
    0x30000330 main         +0x178: __rtc_chk_at ()
    value = 0 = 0x0
```

9
I/O System

9.1 Introduction

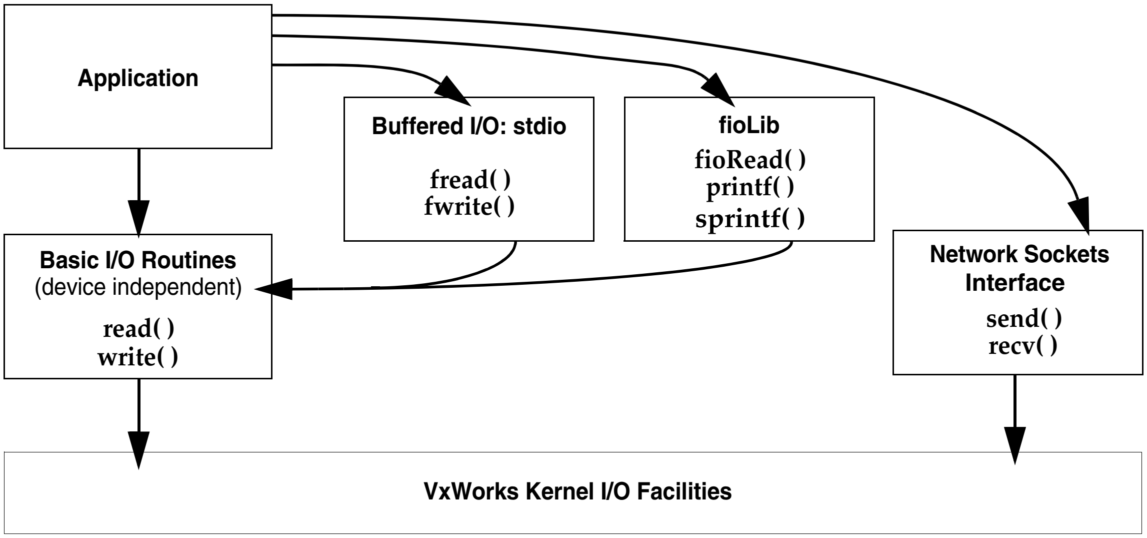
The VxWorks I/O system is designed to present a simple, uniform, device-independent interface to any kind of device, including:

- 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.

The diagram in Figure 9-1 illustrates the relationships between the different elements of the VxWorks I/O system available to real-time processes (RTPs). All of these elements are discussed in this chapter.

Figure 9-1 **Overview of the VxWorks I/O System for Processes**



NOTE: This chapter provides information about facilities available for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the *VxWorks Kernel Programmer's Guide*.

9.2 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_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.

9.3 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 *9.8 Devices in VxWorks*, p.236.

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.

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 *10.3 Virtual Root File System: VRFS*, p.248.

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.

9.4 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 9-1.

Table 9-1 **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.

Table 9-1 **Basic I/O Routines** (cont'd)

Routine	Description
write( )	Writes to a previously created or opened file.
ioctl( )	Performs special control functions on files.

9.4.1 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 size of the file descriptor table, which defines the maximum number of files that can be open simultaneously in a process, is inherited from the spawning environment. If the process is spawned by a kernel task, the size of the kernel file descriptor table is used for the initial size of the table for the new process.

The size of the file descriptor table for each process can be changed programmatically. The **rtpIoTableSizeGet( )** routine reads the current size of the table, and the **rtpIoTableSizeSet( )** routine changes it.

By default, file descriptors are reclaimed only when the file is closed for the last time. However, the **dup( )** and **dup2( )** routines can be used to duplicate a file descriptor. For more information, see *9.4.3 Standard I/O Redirection*, p.224.

9.4.2 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 real time processes 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. You can use these descriptors to manipulate the input and output for all tasks in a process at once by changing the files associated with these descriptors.

These standard file descriptors are used to make an application independent of its actual I/O assignments. If a process sends its output to standard output (where the file descriptor is 1), then its output can be redirected to any file of any device, without altering the application's source code.

9.4.3 Standard I/O Redirection

If a process is spawned by a kernel task, the process inherits the standard I/O file descriptor assignments of the spawning kernel task. These may be the same as the global standard I/O file descriptors for the kernel, or they may be different, task-specific standard I/O file descriptors. (For more information about kernel standard I/O assignments, see the *VxWorks Kernel Programmer's Guide: I/O System*.)

If a process is spawned by another process, it inherits the standard I/O file descriptor assignments of the spawning process.

After a process has been spawned, its standard I/O file descriptors can be changed to any file descriptor that it owns.

The POSIX **dup( )** and **dup2( )** routines are used for redirecting standard I/O to a different file and then restoring them, if necessary. (Note that this is a very different process from standard I/O redirection in the kernel).

The first routine is used to save the original file descriptors so they can be restored later. The second routine assigns a new descriptor for standard I/O, and can also be used to restore the original. Every duplicated file descriptor should be closed explicitly when it is no longer in use. The following example illustrates how the routines are used.

First use the **dup( )** routine to duplicate and save the standard I/O file descriptors, as follows:

```
/* Temporary fd variables */
int  oldFd0;
int  oldFd1;
int  oldFd2;
int  newFd;

/* Save the original standard file descriptor. */
oldFd0 = dup(0);
oldFd1 = dup(1);
oldFd2 = dup(2);
```

Then use **dup2( )** to change the standard I/O files:

```
/* Open new file for stdin/out/err */
newFd = open ("newstandardoutputfile", O_RDWR, 0);
```

```
/* Set newFd to fd 0, 1, 2 */
dup2 (newFd, 0);
dup2 (newFd, 1);
dup2 (newFd, 2);
```

If the process' standard I/O must be redirected again, the preceding step can be repeated with a another new file descriptor.

If the original standard I/O file descriptors must be restored, the following procedure can be performed:

```
/* When complete, restore the original standard IO */
dup2 (oldFd0, 0);
dup2 (oldFd1, 1);
dup2 (oldFd2, 2);

/* Close them after they are duplicated to fd 0, 1, 2 */
close (oldFd0);
close (oldFd1);
close (oldFd2);
```

This redirection only affect the process in which it is done. It does not affect the standard I/O of any other process or the kernel. Note, however, that any new processes spawned by this process inherit the current standard I/O file descriptors of the spawning process (whatever they may be) as their initial standard I/O setting.

For more information, see the VxWorks API references for **dup( )** and **dup2( )**.

9.4.4 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 processes, the mode parameter is optional.

The file-access options that can be used with the *flags* parameter to **open( )** are listed in Table 9-2.

Table 9-2 **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.
O_DSYNC	Write on the file descriptor complete as defined by synchronized I/O data integrity completion.

Table 9-2 **File Access Options** (cont'd)

Flag	Description
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.

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( )**.

For processes, files descriptors are closed automatically only when a process terminates. It is, therefore, recommended that tasks running in processes explicitly close all file descriptors when they are no longer needed. As stated previously (*9.4.1 File Descriptors*, p.223), there is a limit to the number of files that can be open at one time. Note that a process owns the files, so that when a process is destroyed, its file descriptors are automatically closed.

9.4.5 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.

9.4.6 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.

The **read( )** and **write( )**routines are POSIX-compliant.

9.4.7 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.

9.4.8 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 **ioctl( )** calls made in processes, the *arg* parameter is optional. Both of the following are legitimate calls:

```
fd = ioctl (fd, func, arg);
fd = ioctl (fd, func);
```

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 *9.8 Devices in VxWorks*, p.236 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.

The **ioctl( )** routine is POSIX-compliant.

VxWorks also provides the **posix_devctl( )** routine for special devices. For more information, see the API reference entry.

9.4.9 Pending on Multiple File Descriptors with select( )

The VxWorks **select( )** routine provides a UNIX- and Windows-compatible method for pending on multiple file descriptors (allowing tasks to wait for multiple devices to become active).

To use **select( )**, configure VxWorks with the **INCLUDE_IO_BASIC** component.

The task-level support provided by the select facility 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 9-3.

Table 9-3 **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 more information, see the API reference entry for **select( )**.

Example 9-1 **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 <vxWorksCommon.h>
#include <unistd.h>
#include <fcntl.h>
#include <strings.h>
#include <sys/select.h>
#include <stdio.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:
 * -> pipeDevCreate ("/pipe/highPriority", 5, 1024)
 * -> pipeDevCreate ("/pipe/normalPriority", 5, 1024)
 * -> fdHi = open ("/pipe/highPriority", 1, 0)
 * -> fdNorm = open ("/pipe/normalPriority", 1, 0)
 * -> rtpSp "selServer.vxe"
 * -> 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 */
    while (1) {
        /* 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);
            }
        }
    }
}

int
main(int argc, char *argv[])
{
    printf("selServer starts..\n");
    selServer();

    return (0);
}
```

9.4.10 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 9-4.

Table 9-4 **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**.

9.5 Standard I/O

VxWorks provides a standard I/O package (**stdio.h**) with full ANSI C support that is compatible with the UNIX and Windows standard I/O packages.

For user mode (RTP) applications, the standard I/O routines are provided by the Dinkum C libraries.

9.5.1 About Standard I/O and Buffering

The use of the buffered I/O routines provided with standard I/O can provide a performance advantage over the non-buffered basic I/O routines when an application performs many small read or write operations. (For information about non-buffered I/O, see *9.4 Basic I/O*, p.222).

Although the VxWorks I/O system is efficient, some overhead is associated with each low-level (basic I/O) 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, as follows:

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

To make this type of I/O more efficient and flexible, the standard I/O facility 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 standard I/O routines and macros. To access a file using standard I/O, a file is opened with **fopen( )** instead of **open( )**, as follows:

```
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( )**, as follows:

```
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 **FILE** buffer is deallocated when **fclose( )** is called.

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 standard I/O 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.

9.5.2 About Standard Input, Standard Output, and Standard Error

As discussed in *9.4 Basic I/O*, p.222, 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.

9.6 Other Formatted I/O

The **fioLib** library provides additional formatted output and scanning routines (non-ANSI).

For example, 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.

9.7 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.

NOTE: The asynchronous I/O facilities are not included in any RTP shared library provided by Wind River for use with this release. They can only be statically linked with application code. For information about creating a custom shared library that provides this functionality, please contact Wind River Support.

9.7.1 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 9-5.

Table 9-5 **Asynchronous Input/Output Routines**

Function	Description
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.

9.7.2 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.

9.7.3 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( )**.

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.

9.8 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. See Table 9-6 for a list of supported devices.

Table 9-6 **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

Table 9-6 **Devices Provided with VxWorks** (cont'd)

Device	Driver Description
romfs	ROMFS device
–	Other hardware-specific device

See the *VxWorks Kernel Programmer's Guide: I/O System* and the *VxWorks Device Driver Developer's Guide* for more detailed information about I/O device drivers.

9.8.1 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 about I/O control functions, see *VxWorks Kernel Programmer's Guide: I/O System*.

Table 9-7 is a summary of the available options. The listed names are defined in the header file **ioLib.h**. For more detailed information, see the API reference entry for **tyLib**.

Table 9-7 **Tty Options**

Library	Description
OPT_LINE	Selects *line mode*. (See *Raw Mode and Line Mode*, p.238.)
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**.

Table 9-7 **Tty Options** (cont'd)

Library	Description
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.237).

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.238.

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 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 9-8.

Table 9-8 **Tty Special Characters**

Character	Description	Modifier
CTRL+H	backspace (character delete)	**tyBackspaceSet( )**
CTRL+U	line delete	**tyDeleteLineSet( )**
CTRL+D	EOF (end of file)	**tyEOFSet( )**
CTRL+C	kernel shell abort	**tyAbortSet( )**
CTRL+X	trap to boot ROMs	**tyMonitorTrapSet( )**
CTRL+S	output suspend	N/A
CTRL+Q	output resume	N/A

9.8.2 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.

Named pipes can be created in processes. However, unless they are specifically deleted by the application they will persist beyond the life of the process in which they were created. Applications should allow for the possibility that the named pipe already exists, from a previous invocation, when the application is started.

[1] It will not be possible to restart VxWorks if un-handled external interrupts occur during the boot countdown.

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.

I/O Control Functions

Pipe devices respond to the **ioctl( )** functions summarized in Table 9-9. 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 9-9 **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.

9.8.3 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.

For information about implementing **memDrv**, see the *VxWorks Kernel Programmer's Guide: I/O System*.

I/O Control Functions

The memory device responds to the **ioctl( )** functions summarized in Table 9-10. The functions listed are defined in the header file **ioLib.h**.

Table 9-10 **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( )**.

9.8.4 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 the *VxWorks Kernel Programmer's Guide: Network File System*.

I/O Control Functions for NFS Clients

NFS client devices respond to the **ioctl( )** functions summarized in Table 9-11. The functions listed are defined in **ioLib.h**. For more information, see the reference entries for **nfsDrv**, **ioLib**, and **ioctl( )**.

Table 9-11 **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.

9.8.5 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 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.

NOTE: Within processes, there are limitations on RSH and FTP usage: directories cannot be created and the contents of file systems cannot be listed.

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( )**.

9.8.6 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 *7.2.1 POSIX PSE52 Support*, p.147. For information about VRFS, see *10.3 Virtual Root File System: VRFS*, p.248.

9.8.7 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 Programmer's Guide*.

9.8.8 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 *10.5 MS-DOS-Compatible File System: dosFs*, p.257.

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 the *VxWorks Kernel Programmer's Guide: Local File Systems*.

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.

To commit a file system from a process, call:

```
ioctl(fd, CBIO_TRANS_COMMIT, 0);
```

where *fd* is any file descriptor that is open.

TRFS Code Example

The following code example illustrates setting a transaction point.

```
void transTrfs
    (
    void
    )
    {
    int fd;
    /* This assumes a TRFS with DosFs on "/trfs" */

    fd = open ("/trfs/test.1", O_RDWR | O_CREAT, 0);

    ... /* Perform file operations here */
    ioctl (fd, CBIO_TRANS_COMMIT, 0);

    ... /* Perform more file operations here */
    ioctl (fd, CBIO_TRANS_COMMIT, 0);

    close (fd);
    }
```

10 Local File Systems

10.1 Introduction

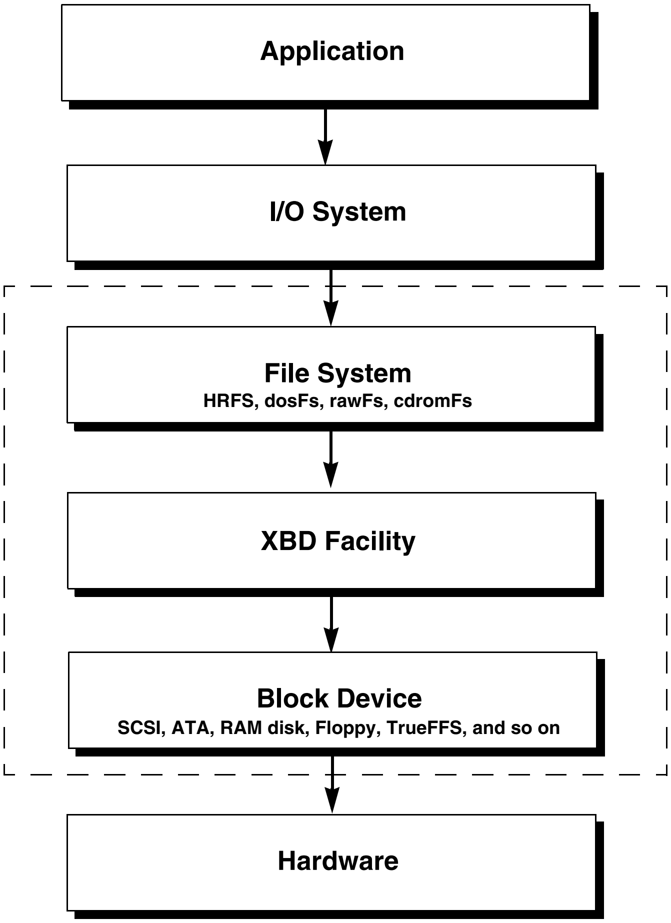
VxWorks provides a variety of file systems that are suitable for different types of applications. The file systems can be used simultaneously, and in most cases in multiple instances, for a single VxWorks system.

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 10-1. 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 10-1 **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 the *VxWorks Kernel Programmer's Guide: I/O System*.

This chapter discusses the following VxWorks file systems and how they are 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 *10.3 Virtual Root File System: VRFS*, p. 248.

- **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 *10.4 Highly Reliable File System: HRFS*, p. 249.

- **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 *10.5 MS-DOS-Compatible File System: dosFs*, p. 257.

- **rawFS**

  Provides a simple raw file system that treats an entire disk as a single large file. See *10.6 Raw File System: rawFs*, p. 268.

- **cdromFs**

  Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system. See *10.7 CD-ROM File System: cdromFs*, p. 270.

- **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 *10.8 Read-Only Memory File System: ROMFS*, p. 273.

- **TSFS**

  Uses the host target server to provide the target with access to files on the host system. See *10.9 Target Server File System: TSFS*, p. 274.

For information about the file system monitor, see the *VxWorks Kernel Programmer's Guide: Local File Systems*. For information about the XBD facility, see the *VxWorks Kernel Programmer's Guide: I/O System*.

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 *10.5 MS-DOS-Compatible File System: dosFs*, p. 257 and the *VxWorks Kernel Programmer's Guide: Flash File System Support with TrueFFS*.

NOTE: This chapter provides information about facilities available for real-time processes. For information about creating file systems, and file system facilities available in the kernel, see the *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.2 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.

For detailed information about how the file system monitor works, see the *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.3 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 *9.8.6 Null Devices*, p. 242. For information about POSIX PSE52, see *7.2.1 POSIX PSE52 Support*, p. 147.

⚠ **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**.

10.4 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 the *VxWorks Kernel Programmer's Guide: Flash File System Support with TrueFFS*.

10.4.1 Configuring VxWorks for HRFS

To include HRFS support in VxWorks, configure the kernel with the appropriate required and optional components.

Required Components

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 there are no facilities for disk modifications. The configuration parameters for these components are as follows:

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. For information about using this parameter in configuration for performance, see *10.4.5 Optimizing HRFS Performance*, p. 252.

HRFS_DEFAULT_MAX_FILES
Defines how many files can be open simultaneously on an HRFS volume. The minimum is 1. The default setting is 10. Note that this is not the same as the maximum number of file descriptors.

The **INCLUDE_HRFS_DEFAULT_WRITE_MODE** component is included automatically, providing the default write-mode, which performs a commit with each write. (The alternative is *INCLUDE_HRFS_HISPEED_WRITE_MODE*, p.250.)

In addition to either **INCLUDE_HRFS** or **INCLUDE_HRFS_READONLY**, HRFS requires the appropriate component for your block device; for example, **INCLUDE_ATA**.

And 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**.

For more information about XBD, see the *VxWorks Kernel Programmer's Guide: I/O System*.

Optional HRFS Components

The optional components for HRFS are as follows:

INCLUDE_HRFS_HISPEED_WRITE_MODE
The high-speed write-mode, which performs a commit for the current transaction when either of the following are true: cache usage is over 40%, or five seconds have elapsed (if the current transaction is still active). This component provides an alternative to the default **INCLUDE_HRFS_DEFAULT_WRITE_MODE** component, which performs a commit with each write.

For information about using **INCLUDE_HRFS_HISPEED_WRITE_MODE** in configuration for performance, see *10.4.5 Optimizing HRFS Performance*, p.252. For information about transactions and commit policies, see *10.4.6 Transactional Operations and Commit Policies*, p.252.

INCLUDE_HRFS_FORMAT
Formatter for HRFS media.

INCLUDE_HRFS_CHKDSK
File system consistency checker.

INCLUDE_HRFS_ACCESS_TIMESTAMP
Access time stamp for HRFS volumes. Note that this 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 the *VxWorks Kernel Programmer's Guide: I/O System*.

10.4.2 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. For information about using this parameter in configuration for performance, see *10.4.5 Optimizing HRFS Performance*, p. 252.

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.

10.4.3 HRFS and POSIX PSE52

If an application requires a PSE52 file system, then VxWorks must be configured with the following components:

- The **INCLUDE_HRFS** component for the highly reliable file system.
- The **INCLUDE_HRFS_ACCESS_TIMESTAMP** component for updating file and directory access timestamps on an HRFS formatted volume must be included. This component is part of the **BUNDLE_RTP_POSIX_PSE52** bundle.

⚠ **WARNING:** The use of the **INCLUDE_HRFS_ACCESS_TIMESTAMP** component can cause significant degradation in file system performance. For more information, see *10.4.7 File Access Time Stamps*, p. 254.

- The appropriate device driver component (for example **INCLUDE_ATA** or **INCLUDE_XBD_RAMDRV)** and **INCLUDE_XBD_BLK_DEV** if the device driver requires it. See the *VxWorks Kernel Programmer's Guide: I/O System*.
- The **INCLUDE_VRFS** component for the VRFS file system, which is included in the **BUNDLE_RTP_POSIX_PSE52** component bundle

In addition, a **/tmp** directory must be created at run-time. It must appear on the VRFS file system, and must be formatted as an HRFS file system. The following examples illustrate creation of a **/tmp** RAM disk device and formatting the directory as an HRFS file system:

```
xbdRamDiskDevCreate (512, 0x10000, 0, "/tmp")
hrfsDiskFormat "/tmp", 1000.
```

For more information, see *10.3 Virtual Root File System: VRFS*, p. 248 and *7.2.1 POSIX PSE52 Support*, p. 147.

10.4.4 Creating an HRFS File System

For information about creating an HRFS file system, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.4.5 **Optimizing HRFS Performance**

For information about optimizing HRFS performance (by using specific configuration and formatting procedures), see the *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.4.6 **Transactional Operations and Commit Policies**

HRFS is a transactional file system. Transaction or commit points are set to make disk changes permanent. *Commit policies* define specific conditions under which commit points are set.

HRFS can be configured to automatically commit with each write (the default commit policy), or to perform commits based on cache usage and elapsed time (the high-speed commit policy). Some disk operations trigger commits regardless of the configuration, and under certain circumstances, HRFS rollbacks undo disk changes made since the last commit, in order to protect the integrity of the file system.

In addition, applications can perform commits independently of the configured commit policy.

Automatic Commit Policy

By default, HRFS automatically performs a commit with each write operation. This functionality is provided by the **INCLUDE_HRFS_DEFAULT_WRITE_MODE** component.

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. The following routines, for example, cause modifications to disk and result in a commit when the automatic commit policy is used:

- **write( )**
- **remove( )**
- **delete( )**
- **mkdir( )**
- **rmdir( )**
- **link( )**
- **unlink( )**
- **truncate( )**
- **ftruncate( )**
- **ioctl( )** when used with a control function that requires modifying the disk.

High-Speed Commit Policy

The high-speed commit policy performs a commit for the current transaction when either of the following are true:

- When cache usage is over 40%.
- When five seconds have elapsed, if the current transaction is still active.

This functionality is provided by the **INCLUDE_HRFS_HISPEED_WRITE_MODE** component. For information about using this component in configuration for performance, see the *VxWorks Kernel Programmer's Guide: Local File Systems*.

Mandatory Commits

Regardless of commit policy, 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.

Note that mandatory commits are a subset of automatic commits—they do not, for example, include **write( )** and **truncate( )**.

Rollbacks

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.

Programmatically Initiating Commits

Applications can be coded to initiate commit operations as required with the **commit( )** routine. The application can decide when critical data has been written and needs to be committed. This is fast in terms of performance, but has the potential for greater data loss. The **commit( )** routine is provided by the **INCLUDE_DISK_UTILS** component.

10.4.7 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.

10.4.8 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.

10.4.9 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**.

10.4.10 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 *9.4 Basic I/O*, p.222, 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.

10.4.11 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 10-1.

Table 10-1 **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.
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.**

10.4.12 Crash Recovery and Volume Consistency

For detailed information about crash recovery and volume consistence, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.4.13 File Management and Full Devices

When an HRFS device is full, attempts to delete or write to files with standard commands will fail (it is, however, unlikely that a device will become full). This behavior is not a defect, but is inherent in highly reliable file systems that use copy-on-write (COW) to implement their transaction mechanism.

During a write operation, the file system writes modified data to a temporary block during a transaction. For example, in order to write block A, block B is first allocated, and the data is written to B. When the transaction is complete, block B is re-mapped to replace block A, and block A is freed. When HRFS removes a file, it first renames the file. It then allocates a new block, and copies the modified directory entry block to it.

If the device is full, the allocation operation fails. It is, however, unlikely that a device will become full, because when a write operation fails due to a lack of free blocks, HRFS rolls back the failed transaction, and the blocks allocated for the transaction are freed. It is then possible to delete files. If a transaction does happen to write the device full, there is no way to free blocks for further operations. Wind River therefore recommends that you do not allow an HRFS device to become full. Leaving a margin of five percent of device space free is a useful guideline.

10.5 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 the *VxWorks Kernel Programmer's Guide: Flash File System Support with 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 *9.8.8 Transaction-Based Reliable File System Facility: TRFS*, p.242.

10.5.1 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 the *VxWorks Kernel Programmer's Guide: I/O System* 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 the following:

INCLUDE_DOSFS_PRTMSG_LEVEL	Message output level. To suppress messages, set the **DOSFS_PRINTMSG_LEVEL** configuration parameter to 0 (zero). The default (with or without this component) is to output messages.
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 the *VxWorks Kernel Programmer's Guide: I/O System*.

10.5.2 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_FAT_CACHE_SIZE
The size of the FAT group cache (for each dosFs volume). The default 16 KB.

DOSFS_DEFAULT_DATA_DIR_CACHE_SIZE
The disk-data and directory-group cache size (for each dosFs volume).

DOSFS_CACHE_BACKGROUND_FLUSH_TASK_ENABLE
The cache-flushing mode. Set to **FALSE** (default), cache-flushing takes place in the context of the user's read/write task. Set to **TRUE**, a separate high-priority task is spawned to carry out cache flushing. Use of a separate task speeds up write operations (especially when dosFs is configured with a larger cache size), but using a separate high-priority task might not be suitable for systems with strict real-time requirements.

For information about the use of some of these component parameters in optimizing dosFs performance, see *10.5.4 Optimizing dosFs Performance*, p.260.

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.

10.5.3 Creating a dosFs File System

For information about creating a dosFs file system, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.5.4 Optimizing dosFs Performance

For information about optimizing dosFs performance (by using specific component-parameter settings, cache tuning, and **write( )** usage), see the *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.5.5 Working with Volumes and Disks

This section discusses accessing volume configuration information and synchronizing volumes. For information about **ioctl( )** support functions, see *10.5.10 I/O Control Functions Supported by dosFsLib*, p.266.

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( )**.

10.5.6 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**.

10.5.7 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 *9.4 Basic I/O*, p.222, 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 10-2.

Table 10-2 **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 10-1 **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**.

10.5.8 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 *10.5.10 I/O Control Functions Supported by dosFsLib*, p.266.

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 10-2 **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);
    }
```

10.5.9 Crash Recovery and Volume Consistency

For information about crash recovery and volume consistence, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.5.10 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 10-3.

Table 10-3 **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).

Table 10-3 **I/O Control Functions Supported by dosFsLib** (cont'd)

Function	Decimal Value	Description
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.
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.
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.**

10.5.11 Booting from a Local dosFs File System Using SCSI

For information about booting from a local dosFs file system using SCSI, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.6 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.

10.6.1 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 the *VxWorks Kernel Programmer's Guide: I/O System* 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 *10.6.4 rawFs File I/O*, p.269.

10.6.2 Creating a rawFs File System

For information about creating a rawFs file system, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.6.3 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.

10.6.4 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 *9.4 Basic I/O*, p.222.

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.

10.6.5 I/O Control Functions Supported by rawFsLib

The rawFs file system supports the **ioctl( )** functions shown in Table 10-4. 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 10-4 **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.

Table 10-4 **I/O Control Functions Supported by rawFsLib** (cont'd)

Function	Decimal Value	Description
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.

10.7 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[1]
- Multi-session CD-R or CD-RW[2]

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.

[1] 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.

[2] 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).

After you have created the CD file system device (*10.7.2 Creating and Using cdromFs*, p.271), 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:

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**.

10.7.1 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 the *VxWorks Kernel Programmer's Guide: I/O System* 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.

10.7.2 Creating and Using cdromFs

For information about creating and using a CD block device, see *VxWorks Kernel Programmer's Guide: Local File Systems*.

10.7.3 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 10-5 describes the **ioctl( )** functions that **cdromFsLib** supports. For more information, see the API references for **cdromFsLib** and for **ioctl( )** in **ioLib**.

Table 10-5 **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).

10.7.4 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( )**.

10.8 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).

10.8.1 Configuring VxWorks with ROMFS

VxWorks must be configured with the **INCLUDE_ROMFS** component to provide ROMFS facilities.

10.8.2 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.

10.8.3 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.

10.8.4 Using ROMFS to Start Applications Automatically

ROMFS can be used with various startup mechanisms to start real-time process (RTP) applications automatically when VxWorks boots.

See *3.7.4 Using ROMFS to Start Applications Automatically*, p.51 for more information.

10.9 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 *9.8.5 Non-NFS Network Devices*, p.241), 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.276). 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 *9.8.5 Non-NFS Network Devices*, p.241). 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 *VxWorks Kernel Programmer's Guide: Kernel*.

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.

For more information about error detection and reporting routines in addition to that provided in this chapter, see the API reference for **edrLib**. Also see the

VxWorks Kernel Programmer's Guide: Error Detection and Reporting for information about facilities available only in the kernel.

For information about related facilities, see *8.4 Memory Error Detection*, p.209.

NOTE: This chapter provides information about facilities available for real-time processes. For information about facilities available in the VxWorks kernel, see the corresponding chapter in the *VxWorks Kernel 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**

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 and core dumps. 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 all architectures (see the *VxWorks Architecture Supplement*), and it is not write-protected for the symmetric multiprocessing (SMP) configuration of VxWorks. For general information about VxWorks SMP and for information about migration, see the *VxWorks Kernel Programmer's Guide: VxWorks SMP*.

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 (which is provided by the **INCLUDE_EDR_PM** component). 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 (which is needed to maintain internal persistent-memory data structures).

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. For information in this regard, see the *VxWorks Kernel Programmer's Guide: Boot Loader*.

⚠ **WARNING:** If the boot loader is not properly configured, 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 the *VxWorks Kernel Programmer's Guide: Memory Management*, 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.284 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. 284).

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.
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.285). 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.285). 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.

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.

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 the *VxWorks Kernel Programmer's Guide: Boot Loader*.

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.

11.6 Other Error Handling Options for Processes

By default, any faults generated by a process are handled by the error detection and reporting facility.

A process can, however, handle its own faults by installing an appropriate signal handler in the process. If a signal handler is installed (for example, **SIGSEGV** or

SIGBUS), the signal handler is run instead of an error record being created and an error handler being called. The signal handler may pass control to the facility if it chooses to by using the **edrErrorInject( )** system call.

For more information about signals, see *6.19 Signals*, p.134.

11.7 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/usr/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.

11.8 Sample Error Record

The following is an example of a record generated by a failed process task:

```
==[1/1]==============================================================
Severity/Facility:    FATAL/RTP
Boot Cycle:           1
OS Version:           6.0.0
Time:                 THU JAN 01 05:21:16 1970 (ticks = 1156617)
Task:                 "tInitTask" (0x006f4010)
RTP:                  "edrdemo.vxe" (0x00634048)
RTP Address Space:    0x10226000 -> 0x10254000
Injection Point:      rtpSigLib.c:4893

Default Signal Handling : Abnormal termination of RTP edrdemo.vxe (0x634048)
```

```
<<<<<Memory Map>>>>>

0x00100000 -> 0x002a48dc: kernel
0x10226000 -> 0x10254000: RTP

<<<<<Registers>>>>>

r0         = 0x10226210   sp         = 0x10242f70   r2         = 0x10238e30
r3         = 0x00000037   r4         = 0x102440e8   r5         = 0x10244128
r6         = 0x00000000   r7         = 0x10231314   r8         = 0x00000000
r9         = 0x10226275   r10        = 0x0000000c   r11        = 0x0000000c
r12        = 0x00000000   r13        = 0x10239470   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        = 0x00000002   r28        = 0x10242f9c   r29        = 0x10242fa8
r30        = 0x10242fac   r31        = 0x50000000   msr        = 0x0000f032
lr         = 0x10226210   ctr        = 0x0024046c   pc         = 0x10226214
cr         = 0x80000080   xer        = 0x20000000   pgTblPtr   = 0x00740000
scSrTblPtr = 0x0064ad04   srTblPtr   = 0x0064acc4

<<<<<Disassembly>>>>>

 0x102261f4  48003559    bl          0x1022974c # strtoul
 0x102261f8  3be30000    addi        r31,r3,0x0 # 0
 0x102261fc  3c601022    lis         r3,0x1022 # 4130
 0x10226200  38636244    addi        r3,r3,0x6244 # 25156
 0x10226204  389f0000    addi        r4,r31,0x0 # 0
 0x10226208  4cc63182    crxor       crb6,crb6,crb6
 0x1022620c  48002249    bl          0x10228454 # printf
 0x10226210  39800000    li          r12,0x0 # 0
*0x10226214  999f0000    stb         r12,0(r31)
 0x10226218  48000014    b           0x1022622c # 0x1022622c
 0x1022621c  3c601022    lis         r3,0x1022 # 4130
 0x10226220  38636278    addi        r3,r3,0x6278 # 25208
 0x10226224  4cc63182    crxor       crb6,crb6,crb6
 0x10226228  4800222d    bl          0x10228454 # printf
 0x1022622c  80010014    lwz         r0,20(r1)
 0x10226230  83e1000c    lwz         r31,12(r1)

<<<<<Traceback>>>>>

0x102261cc _start       +0x4c : main ()
```

12
RTP Core Dumps

12.1 Introduction

The VxWorks RTP core dump facility allows for generation and storage of RTP core dumps, for excluding selected memory regions from core dumps, and for retrieving core dumps from the target system. Core dump files can be analyzed with the Workbench debugger to determine the reason for a failure or to analyze the state of a system at a particular moment in its execution.

NOTE: VxWorks can be configured with both kernel core dump and RTP core dump facilities. For information about the VxWorks kernel core dump facility, see the *VxWorks Kernel Programmer's Guide*.

12.2 About VxWorks RTP Core Dumps

A VxWorks RTP core dump consists of a copy of the contents of the address space of an RTP at a specific time, generally when the program has terminated abnormally.

Generation

An RTP core dump is generated when an RTP gets an exception or when it receives any unmasked signal except for **SIGSTOP**, **SIGCONT**, and **SIGCHLD**. They can also be generate on demand (interactively or programmatically).

The VxWorks RTP core dump facility enables core dump generation for all RTPs that are running in a system. You *cannot* selectively enabled or disabled individual RTPs for core dumps. Multiple core dump events are handled sequentially. That is, if a core dump event is triggered while generation of another core dump is in process, the second is not processed until the first is complete.

Kernel core dumps take precedence over RTP core dumps. If a kernel core dump is triggered while an RTP core dump is in process, the RTP core dump is aborted and the kernel core dump is processed.

For information about generating core dumps, see *12.8 Generating RTP Core Dumps*, p.294.

Data

An RTP core dump includes the following data:

- The RTP's memory space.
- The RTP's tasks.
- Any shared libraries to which the RTP is attached.
- Any shared data regions to which the RTP is attached.

An RTP core dump does not include information about any kernel tasks or kernel objects (such as semaphores and so on), nor does it include any information about any other RTPs.

The RTP core dump facility provides the ability to exclude (filter) regions of memory from core dumps.

For information about core dump analysis, see *12.11 Analyzing RTP Core Dumps*, p.295.

Size

RTP core dumps can be quite large due to the fact that they must include the text section of the RTP. (Without the text section, it cannot be used with the Workbench debugger.) In addition, core dumps will be larger if the RTP is attached to a shared library at the time the core dump is triggered. This is because the shared library is included in the runtime image of the RTP and is therefore included in the core dump.

The total size of the RTP core dump is determined as follows:

RTP text + RTP data + shared library + shared data + some kernel data

The element of kernel data is for internal purposes only, and is not visible when the debugger is used to analyze the core dump.

Filtering

The RTP core dump facility provides a routine for filtering (excluding) specific memory regions from RTP core dumps to reduce their size, or simply to eliminate data that is not of interest. For more information, see *12.5 Eliminating Areas of Memory Captured by Core Dumps*, p.293.

File Locations and Names

The location to which core dump files are written is specified by the configuration of the RTP core dump facility. The location can also be changed at runtime.

RTP core dump files are named using the following syntax:

rtpcore[_*suffix*_]*index*[**.z**]

For more information about file names and locations, see *12.6 Setting Core Dump File Names*, p.293 and *12.7 Setting Core Dump File Locations*, p.294.

File Compression

The RTP core dump facility provides options for file compression (zlib, RLE, or none). For more information, see *12.4 Enabling File Compression*, p.292.

12.3 Configuring VxWorks With RTP Core Dump Support

The VxWorks RTP core dump facility provides components for creating and managing core dumps, and optional components for file compression.

Basic Configuration

INCLUDE_CORE_DUMP_RTP

This component provides basic RTP core dump support. IT also includes facilities for excluding (filtering) regions of memory from core dumps and for adding a suffix element to the default core dump file name.

For more information on excluding regions of memory, see *12.5 Eliminating Areas of Memory Captured by Core Dumps*, p.293. For more information on customizing core file names, see *12.6 Setting Core Dump File Names*, p.293)

INCLUDE_CORE_DUMP_RTP_FS

This component provides facilities for writing RTP core dumps to a local or remote file system. The following parameter identifies the location:

CORE_DUMP_RTP_PATH
Set this parameter to the file system and path to the directory into which core dump files are written. For example, **/myLocalFs/rtpcore** for a local file system,

or **remote:/rtpcore** for a remote file system. The path must be accessible from the target.

The **INCLUDE_CORE_DUMP_RTP_FS** component also provides routines for getting and setting the storage directory at runtime (for more information, see *12.7 Setting Core Dump File Locations*, p.294), as well as for producing an information file for each RTP core dump file (for more information, see *12.9 Getting Information about RTP Core Dumps*, p.294).

Core Dump Compression

INCLUDE_CORE_DUMP_RTP_COMPRESS_RLE

This component provides core dump compression support (based on run-length encoding) that compresses the core dump image. This is the default for RTPs.

For more information see *12.4 Enabling File Compression*, p.292.

INCLUDE_CORE_DUMP_RTP_COMPRESS_ZLIB

This component provides a zlib-based alternative to the default, RLE-based, compression method.

CORE_DUMP_RTP_ZLIB_COMPRESSION_LEVEL
This parameter specifies the amount of compression. The default is 6 and the range is 1 through 9. A higher number means that it is more fully compressed—but it takes longer to compress the image.

For more information, see *12.4 Enabling File Compression*, p.292.

12.4 Enabling File Compression

Using compression saves disk space and reduces the upload time from the target to the host. RTP core dump files can be compressed using either zlib or a facility based on the run-lenght encoding (RLE) algorithm. zlib has better compression ratios than RLE, and also allows you to configure the degree of compression. The RLE facility, however, provides faster compression.

The **INCLUDE_CORE_DUMP_RTP_COMPRESS_RLE** component provides compression support based on run-length encoding. It is the default compression method.

The **INCLUDE_CORE_DUMP_RTP_COMPRESS_ZLIB** component provides a zlib-based compression method. Its **CORE_DUMP_RTP_ZLIB_COMPRESSION_LEVEL** parameter specifies the amount of compression. The default is 6 and the range is 1 through 9. A higher number means that it is more fully compressed—but it takes longer to compress the image.

⚠ **CAUTION:** On some targets compression can take a long time, which may lead you to believe that the target has hung. If this occurs, Wind River recommends that you remove the compression component or reconfigure the level of compression.

12.5 Eliminating Areas of Memory Captured by Core Dumps

To reduce the size of an RTP core dump, or simply to eliminate data that you are not interested in, you can exclude (filter) specific memory regions from RTP core dumps. Filters can be installed or removed by the RTP application itself with the **coreDumpMemFilterAdd( )** and **coreDumpMemFilterDelete( )** routines. Filters are deleted automatically when a RTP application exits.

⚠ **CAUTION:** Do not use memory filters to exclude the text sections of an RTP or any shared libraries that are attached to it. If the text sections are not present, the Workbench debugging tools cannot be used with the core dump.

Memory filters are provided by the basic **INCLUDE_CORE_DUMP_RTP** component. For more information, see the **coreDumpLib** API reference entry.

12.6 Setting Core Dump File Names

Core dump files are named automatically using the following syntax:

rtpcore[_*suffix*_]*index*[**.z**]

The syntax is used as follows:

- The index is set to **1** for the first core dump generated during a given session (that is, between reboots), and incremented for each subsequent core dump during that session. The index is reset to zero each time the system is rebooted.
- Core dump files remaining from a previous session are overwritten by new core dump files with the same file name.
- The suffix element can be used to provide unique names for core dump files for a given session, so they are not overwritten during a subsequent session.
- The **.z** extension is added for compressed files.

In order to preserve core dump files between sessions, you can use the **coreDumpRtpNameSuffixSet( )** routine to add a file-name suffix based on unique string (such as the time and date of booting VxWorks). This call must be made from the *kernel* context. You can make the call from the shell before launching the RTP, or from **usrAppInit( )**.

For information about **coreDumpRtpNameSuffixSet( )**, see the *VxWorks Application API Reference*. For information about **usrAppInit( ),** see the *VxWorks Kernel API Reference* and the *VxWorks Kernel Programmer's Guide: Kernel Applications*.

12.7 Setting Core Dump File Locations

The location to which core dump files are written is defined statically at system configuration time (for information, see *12.3 Configuring VxWorks With RTP Core Dump Support, p.291*).

You can, however, use the **coreDumpRtpFsPathGet( )** and **coreDumpRtpFsPathSet( )** routines to get and set the location dynamically. These calls must be made from the *kernel* context. You can make them from the shell before launching the RTP, or from **usrAppInit( )**.

For information about **coreDumpRtpFsPathGet( )** and **coreDumpRtpNameSuffixSet( )**, see the *VxWorks Application API Reference*. For information about **usrAppInit( ),** see the *VxWorks Kernel API Reference* and the *VxWorks Kernel Programmer's Guide: Kernel Applications*.

12.8 Generating RTP Core Dumps

RTP core dumps are generated automatically when an RTP generates an exception or receives a signal that is not masked; the exceptions being **SIGSTOP** signals, **SIGCONT** and **SIGCHLD**.

You can generate RTP core dumps interactively and programmatically (from a kernel task or another RTP) by calling **rtpKill( )**. In addition, an RTP application can generate an RTP core dump of itself when, for example, detecting a fatal error that should be debugged:

```
if (fatalError)
    {
    /* Fatal error detected: generate RTP Core Dump */

    kill ((int) getpid(), SIGABRT);
    }
```

At the end of core dump generation, the RTP is either killed or left stopped depending on the configuration of the error detection and reporting facility.

For information about signals, see *6.19 Signals*, p.134. For information about the error detection and reporting facilities, see *11. Error Detection and Reporting*. For information about **rtpKill( )**, see the VxWorks API reference entry.

12.9 Getting Information about RTP Core Dumps

The RTP core dump facility produces an information file for each RTP core dump file that it produces. The file is written to the same directory on a local or remote file system as the core dump file. The name of the information file is the same as that of the core dump file, but with a **.txt** file extension.

The file provides basic information about the core dump. For example:

```
RTP Core Dump Name:     /myLocalFs/rtpcore/rtpcore_device128_3.z
Size:                   1186905
Time:                   WED JUL 22 11:15:17 2009 (ticks = 5411)
Valid:                  Yes
Errno:                  N/A
Task:                   "tReader1" (0xc3aa78)
Process:                "/romfs/reader.vxe" (0xc35818)
Description:            RTP Core Dump
Exception number:       0x300
Program counter:        0xa0002214
Frame pointer:          0x0
Stack pointer:          0xa001ded0
```

The **Valid** field is used to indicate whether or not RTP core dump generation was successful. If not, the **Errno** field provides a hint as to why it failed.

This reporting facility is provided by the **INCLUDE_CORE_DUMP_RTP_FS** component.

12.10 Retrieving RTP Core Dump Files

If you have configured the core dump facility to write RTP core dump files to a remote file system, you can manage them with the file system facilities provided by the host.

If you have configured the core dump facility to write RTP core dump files to a file system local to the target, you can move them to the host using the standard VxWorks file system commands (**ls**, **cp**, **mv**, and so on). If your core dump files are accessible to the Workbench debugger (on a file system accessible to the host system), you do not need to move them from the location to which they were written. For information about the VxWorks commands, see the VxWorks API reference entry for **usrLib**.

For information about configuration, see *12.3 Configuring VxWorks With RTP Core Dump Support, p.291*.

12.11 Analyzing RTP Core Dumps

You can use the Workbench debugger for analyzing RTP core dumps. The host shell cannot be used to debug RTP core dumps, unlike kernel core dumps. For information about using the debugger to work with core dumps, see the *Workbench by Example* guide.

NOTE: To successfully debug an RTP core dump, you need the following:

- The generated RTP core dump.
- The VxWorks image that was running on the target at the time the core dump was generated.
- The RTP image (**.vxe** file) that generated the core dump.
- Any shared libraries (**.so** files) that were attached to the RTP when the core dump was generated.

A
Kernel to RTP Application Migration

A.1 Introduction

This chapter assumes that you have decided to migrate your application out of the kernel and into a real-time process (RTP). For information about processes and RTP applications, see *2. Real-Time Processes*.

A.2 Migrating Kernel Applications to Processes

When migrating an application from the kernel to a real-time process, issues not relevant to a kernel-based application must be considered. The process environment offers protection and is thus innately different from the kernel environment where the application originated. This section highlights issues that are not present in kernel mode, or that are different in user mode.

To run a 5.5 application in a 6.*x* real-time process, the software startup code must be changed, and the application must be built with different libraries. Furthermore, certain kernel-only APIs are not available as system calls, which may prevent certain types of software from being migrated out of the kernel. In particular, software that must execute with supervisor privilege (ISRs, drivers, and so on) or software that cannot communicate using standard APIs (interprocess communication, file descriptors, or sockets) cannot be migrated out of the kernel without more substantial changes.

A.2.1 Reducing Library Size

For production processes, it may be desirable to create stripped images of included libraries. Shared libraries and dynamic executables can be stripped with the command:

```
strip*arch* --strip-unneeded
```

A.2.2 Limiting Process Scope

One of the key aspects of running an application within a process is that code running within the process can only access memory owned by the process. It is not possible for a process to access memory directly within the memory context of another process, or to access memory owned by the kernel.

When migrating applications, it is important to bear these restrictions in mind. The approaches discussed in the following sections can be helpful.

Communicating Between Applications

Although real-time processes are designed to isolate and protect applications, many alternatives exist for communication between processes or between processes and kernel applications.

If large amounts of data need to be shared, either between applications or between an application and the kernel, consider using a shared-memory or shared-data region. The applications that map a given shared-data region into their memory context can specify different access permissions; for instance, the application that creates and initializes the shared data can open it with read and write permissions, while all the consumer applications may open the shared data region with read-only permissions. This provides some level of protection to the shared data. For more information, see the reference entry for **sdLib**.

If your data needs to be more strictly protected and separated from other applications or from the kernel, use inter-process communication mechanisms to pass data between processes or between a process and the kernel. Common options are public versions of:

- semaphores
- message queues
- message channels
- sockets (especially the **AF_LOCAL** domain sockets)
- pipes

Communicating Between an Application and the Kernel

While some applications which are closely coupled with the kernel are not suitable to run in a process, this is not necessarily always the case. Consider the whole range of solutions for communicating between applications before reaching a conclusion. In addition to standard inter-process communication methods, the following options are available.

- You might architect code that must remain in the kernel as a VxWorks driver. Then open the driver from user mode and use the **read**/**write**/**ioctl( )** model to communicate with it.
- You might implement a **sysctl( )** method.
- You might add an additional system call. For more information, see the *VxWorks Application Programmer's Guide: Kernel*.

⚠ **WARNING:** This is the riskiest option as the possibility exists of breaking the protection of either the kernel or your process.

A.2.3 Using C++ Initialization and Finalization Code

For kernel code, the default method for handling C++ program startup and termination is unchanged from earlier VxWorks releases. However, the method has changed for process code. For details, see the *Wind River Compiler User's Guide: Use in an Embedded Environment*. The key points are:

- **.init$***nn* and **.fini$***nn* code sections are replaced by **.ctors** and **.dtors** sections.
- **.ctors** and **.dtors** sections contain pointers to initialization and finalization functions.
- Functions to be referenced in **.ctors** and **.dtors** can exist in any program module and are identified with **__attribute__((constructor))** and **__attribute__((destructor))**, respectively, instead of the old **_STI__***nn*_ and **_STD__***nn*_ prefixes. The priority of initialization and finalization functions can be specified through optional arguments to the **constructor** and **destructor** attributes. Example:

  ```
  __attribute__((constructor(75))) void hardware_init()
  {
  ...  // hardware initialization code
  }
  ```

  Wind River recommends that initialization and finalization functions be specified with an explicit priority. If no priority is specified, functions are assigned the lowest (last) priority by default; this default can be changed with **-Xinit-section-default-pri**. Unless the default is changed, C++ global class object constructors are also assigned the lowest (last) priority.
- Linker command (**.dld**) files for legacy projects must be modified to define **.ctors** and **.dtors** sections. For an example, see **bubble.dld** and the *Wind River Compiler User's Guide: Linker Command Language*.
- Old-style **.init$***nn* and **.fini$***nn* sections are still supported, as are **_STI__***nn*_ and **_STD__***nn*_ function prefixes, through the **-Xinit-section=2** option.

A.2.4 Eliminating Hardware Access

Process code executes in user mode. Any supervisor-level access attempt is illegal, and is trapped. Access to hardware devices usually falls into this category. The following are prohibited:

- Do not call **intLock( )** or **intUnlock( )**. Process tasks must not lock out interrupts. Process tasks can lock out preemption using **taskRtpLock( )** and

taskRtpUnlock( ), but only for tasks within the same process. In any case, a process cannot preempt another running process.

- Do not access devices directly from user mode even if the device is accessible. (Access to devices is sometimes possible depending on the memory map and the mappings for the address area for the device.) Instead of direct access, use the standard I/O library APIs: **open( )**, **close( )**, **read( )**, and so forth.

  An appropriate user-mode alternative is to access a memory-mapped device directly by creating a shared-data region that maps the physical location of the device into the process memory space. A private shared-data region can be created if access to the device must be limited to a single process. For more information, see the *VxWorks Application Programmer's Guide: Shared Data*.

- Do not use processor-specific features and instructions in application code. This hampers portability.

A.2.5 Eliminating Interrupt Contexts In Processes

A real-time process cannot contain an interrupt context. This separation protects the kernel from actions taken within a process. It also means that when you migrate an application that previously ran in the kernel to a process, you must look for routines that require an interrupt context. Modify the code that contains them so that an interrupt is not required.

The following sections identify areas where APIs have different behavior or where different APIs are used in processes in order to eliminate interrupt contexts; POSIX signals do not generate interrupts.

POSIX Signals

Signal handling in processes follows POSIX semantics, not VxWorks kernel semantics. If your existing application used VxWorks signals, you must confirm that the new behavior is what the application requires. For more information, see *A.2.9 POSIX Signal Differences*, p.304.

Watchdogs

The **wdLib** routines cannot be used in user mode. Replace them with POSIX timers from **timerLib** as shown in Table A-1.

Table A-1 **Corresponding wdLib and timerLib Routines**

wdCreate( ) Routines	timer_create( ) Routines
wdCreate( )	**timer_create( )**
wdStart( )	**timer_connect( )** + **timer_settime( )**
wdCancel( )	**timer_cancel( )**
wdDelete( )	**timer_delete( )**

There are slight differences in the behavior of the two timers, as shown in Table A-2.

Table A-2 **Differences Between Watchdogs and POSIX Timers**

VxWorks wdLib	POSIX timerLib
A routine executes in an interrupt context when the watchdog timer expires.	A signal handler executes as a response to the timer expiring.
The handler executes when the timer expires, right in the context of the system clock tick handler.	A signal handler executes in the context of a task; the handler cannot run until the scheduler switches in the task (which is, of course, based on the task priority). Thus, there may be a delay, even though the timeout has expired.

Drivers

Hardware interface services are provided by the kernel in response to API kernel calls. From a process you should access drivers through **ioctl( )**, system calls, message queues, or shared data. For more information, see *A.2.4 Eliminating Hardware Access*, p.299.

A.2.6 Redirecting I/O

I/O redirection is possible for the whole process, but not for individual tasks in the process.

The routines **ioTaskStdGet( )** and **ioTaskStdSet( )** are not available in user mode. You can use **dup( )** and **dup2( )** instead, but these routines change the file descriptors for the entire process.

The POSIX **dup( )** and **dup2( )** routines have been introduced to VxWorks for manipulation of file descriptor numbers. They are used for redirecting standard I/O to a different file and then restoring it to its previous value when the operations are complete.

Example A-1 **VxWorks 5.5 Method of I/O Redirection**

```
/* temporary data values */
int  oldFd0;
int  oldFd1;
int  oldFd2;
int  newFd;

/* Get the standard file descriptor numbers */
oldFd0 = ioGlobalStdGet(0);
oldFd1 = ioGlobalStdGet(1);
oldFd2 = ioGlobalStdGet(2);


/* open new file to be stdin/out/err */
newFd = open ("newstandardoutputfile",O_RDWR,0);

/* redirect standard IO to new file */
```

```
ioGlobalStdSet (0, newFd);
ioGlobalStdSet (1, newFd);
ioGlobalStdSet (2, newFd);

/* Do operations using new standard file for input/output/error */

/* When complete, restore the standard IO to normal */

ioGlobalStdSet (0, oldFd0);
ioGlobalStdSet (1, oldFd1);
ioGlobalStdSet (2, oldFd2);
```

Example A-2 **VxWorks 6.x Method of I/O Redirection**

The process shown in Example A-1 is easily emulated using **dup( )** and **dup2( )**. Use the **dup( )** command to duplicate and save the standard file descriptors upon entry. The **dup2( )** command is used to change the standard I/O files and then later used to restore the standard files that were saved. The biggest difference is the need to close the duplicates that are created at the start.

```
/* temporary data values */
int  oldFd0;
int  oldFd1;
int  oldFd2;
int  newFd;

/* Get the standard file descriptor numbers */
oldFd0 = dup(0);
oldFd1 = dup(1);
oldFd2 = dup(2);

/* open new file to be stdin/out/err */
newFd = open ("newstandardoutputfile",O_RDWR,0);

/* redirect standard IO to new file */
dup2 (newFd, 0);
dup2 (newFd, 1);
dup2 (newFd, 2);

/*
Do operations using new standard file for input/output/error
*/

/* When complete, restore the standard IO to normal */
dup2 (oldFd0, 0);
dup2 (oldFd1, 1);
dup2 (oldFd2, 2);

/* close the dupes */
close (oldFd0);
close (oldFd1);
close (oldFd2);
```

A.2.7 Process and Task API Differences

This section highlights API changes that affect applications in real-time processes.

Task Naming

Initial process tasks are named differently from kernel tasks.

Differences in Scope Between Kernel and User Modes

Applications running in a process are running in a different environment from the kernel. Some APIs display a different scope in user mode than in kernel mode, typically to match POSIX semantics.

exit( )
In user mode, this routine terminates the current process. In kernel mode, **exit( )** terminates only the current task. The user-mode behavior of **exit( )** matches the POSIX standard. The API **taskExit( )** can be used in a process instead of **exit( )** if you want to kill only the current task.

kill( )
In user mode, this routine sends a signal to a process. In kernel mode, **kill( )** sends a signal only to a specific task. The user-mode behavior of **kill( )** matches the POSIX standard. The API **taskKill( )** can be used in a process instead of **kill( )** if you want to send a signal only to a particular task within the process.

raise( )
In user mode, this routine sends a signal to the calling process. In kernel mode, **raise( )** sends a signal only to the calling task. The user-mode behavior of **raise( )** matches the POSIX standard. The API **taskRaise( )** can be used in a process instead of **raise( )** if you wish to send a signal to the calling task. In addition, if you wish to send a signal to the calling process, the API **rtpRaise( )** can be used in a process instead of **raise( )**.

sigqueue( )
In user mode, this routine sends a queued signal to a process. In kernel mode, **sigqueue( )** sends a queued signal only to a specific task. The user-mode behavior of **sigqueue( )** matches the POSIX standard. The API **taskSigqueue( )** can be used in a process instead of **sigqueue( )** if you wish to send a queued signal to a particular task within the process. The API **rtpSigqueue( )** can be used in a process instead of **sigqueue( )** if you wish to send a queued signal to a particular process.

Task Locking and Unlocking

Task locking is available in a process, but is restricted to the tasks of the process where the locking or unlocking calls are made. In other words, you cannot provoke a system-wide task lock from within an application. This also means that, while the process task that disables the task context switching is ensured not to be preempted by other tasks in this same process, it probably will be preempted by other tasks from the kernel or from other applications.

The API available for task locking and unlocking in user mode is different from the one available in the kernel. In an application, task locking can be obtained by calling the **taskRtpLock( )** API. Task unlocking can be done by calling the **taskRtpUnlock( )** API.

Private and Public Objects

The traditional means for inter-task communication used in VxWorks 5.5, such as semaphores and message queues, have been extended such that they can be defined as *private* or *public*, as well as named. Private objects are visible only to tasks within a process, whereas public objects—which must be named—are visible to tasks throughout the system. Public objects can therefore be used for inter-process communication. For more information about public and private objects and about naming, see *6.9 Inter-Process Communication With Public Objects*, p. 107.

A.2.8 Semaphore Differences

In a real-time process, semaphores are available using the traditional VxWorks API (**semTake( )**, **semGive( )**, and so forth). They are known as user-mode semaphores because they are optimized to generate a system call only when necessary. The scope of a semaphore object created by a VxWorks application is, however, restricted to the process it was created in. In other words, two different applications cannot be synchronized using user-level semaphores. If mutual exclusion or synchronization is required between applications, then a public semaphore must be used. A public semaphore can be created using **semOpen( )** by assigning a name starting with **/** (forward slash).

There are restrictions on the type of information regarding semaphores available in user-mode. In particular, the semaphore owner and list of tasks pending on a semaphore is not provided by the **semInfoGet( )** API. If this information is required, its management must be implemented within the user-mode library itself.

A.2.9 POSIX Signal Differences

There are significant differences between signal handling in a kernel environment (in other words, in the equivalent of a VxWorks 5.5 environment) and in the process environment. The VxWorks 5.5 kernel signal behavior still holds true for kernel tasks, but in the process environment the behavior is both new for VxWorks 6.*x*, and different from signal behavior in the kernel environment in some respects.

The signal model in user mode is designed to follow the POSIX process model. (For more information, see the POSIX 1003.1-2004 specification at **http://www.opengroup.org**.)

Signal Generation

A kernel task or an ISR can send signals to any task in the system, including both kernel and process tasks.

A process task can send signals to itself, to any task within its process, to its process, to another process, and to any public tasks in another process. Process tasks cannot send signals to kernel tasks. For more information, see *Private and Public Objects*, p. 303.

Signal Delivery

The process of delivering a signal involves setting up the signal context so that the action associated with the signal is executed, and setting up the return path so that when the signal handler returns, the target task gets back to its original execution context.

Kernel signal generation and delivery code runs in the context of the task or ISR that generates the signal.

Process signal generation is performed by the sender task, but the signal delivery actions take place in the context of the receiving task.

Scope Of Signal Handlers

The kernel is an entity with a single address space. Tasks within the kernel share that address space, but are really different applications that coexist in that one address space. Hence, each kernel task can individually install a different handler for any given signal.

The signal model in user mode follows the POSIX process model. A process executes an application. Tasks that belong to the process are equivalent to threads within a process. Therefore, process tasks are not allowed to register signal handlers individually. A signal handler is effective for all tasks in a given process.

Default Handling Of Signals

By default, signals sent to kernel tasks are ignored (in other words, **SIG_DFL** in kernel mode means “ignore the signals” or **SIG_IGN**).

However, by default, signals sent to process tasks result in process termination (in other words, **SIG_DFL** for process tasks means “terminate the process”).

Default Signal Mask for New Tasks

Kernel tasks, when created, have all signals unmasked. Process tasks inherit the signal mask of the task that created them. Thus, if a kernel task created a process, the initial task of the process has all signals unblocked.

Signals Sent to Blocked Tasks

Kernel tasks that receive signals while blocked are immediately unblocked and run the signal handler. After the handler returns, the task goes back to blocking on the original object.

Signals sent to a blocked process task are delivered only if the task is blocked on an interruptible object. In this case, the blocking system call returns **ERROR** with **errno** set to **EINTR**. After the signal handler returns, it is the responsibility of the task to re-issue the interrupted call if it wishes.

Signals sent to process tasks blocked on non-interruptible objects are queued. The signal is delivered whenever the task unblocks.

For more information, see *Semaphores Interruptible by Signals*, p.123 and *Message Queues Interruptible by Signals*, p.126.

Signal API Behavior

Table A-3 shows signal APIs that behave differently in the kernel than in a process.

Table A-3 **Differences in Signal API Behavior**

API	Kernel Behavior	Process Behavior
kill( )	sends a signal to a task	sends a signal to a process
raise( )	sends a signal to the current task	sends a signal to the current task's process
sigqueue( )	sends a queued signal to a task	sends a queued signal to a process

A.2.10 Networking Issues

The context of a real-time process creates certain differences in network support from an application running in the kernel.

Socket APIs

In the process of porting network applications to processes, Wind River has exposed both standard socket APIs and routing socket APIs at the process level. If you have an application that limits (or can be made to limit) its interaction with the network stack to standard or routing socket API calls, that application is a good candidate for porting to a process.

routeAdd( )

routeAdd( ) is not supported in user mode. In order to make or monitor changes to the routing table from user mode, **routeAdd( )** must be replaced by a routing socket. For more information, see the *Wind River Network Stack Programmer's Guide*.

A.2.11 Header File Differences

The most likely issue to arise when you try to move existing code into user mode is that a header file you used previously is unavailable or no longer contains something you need, and hence your code fails to compile. This may occur commonly when transitioning code and suggests that the feature you are trying to use is not available in user mode.

It may be tempting to find the missing header file on the kernel side of the VxWorks tree and use that, but this is unlikely to help. Wind River supplies specific header files for user-mode development in a distinct user-mode part of the directory tree. These header files only supply features that have been designed and tested to work under user-mode protections.

If a header file does not exist or exposes less functionality than is available in kernel mode, this is because those features are not available from user mode. Usually these features cannot be implemented in user mode due to the nature of the protection model. For example, layer 2 networking facilities typically access hardware I/O drivers directly; however, this is not allowed within the protected user-mode environment.

There is a newly created system for alerting customers to some of the differences between kernel and user modes. The **_WRS_DEPRECATED** macro is used to tag an API as being deprecated. The Wind River Compiler allows for a message to be applied as well. If the compiler encounters an API tagged as deprecated it issues an immediate warning with the optional message. Many routines, like **ioGlobalStdSet( )**, that are not available in user mode, generates the following message when using the Wind River Compiler:

```
fileline ioGlobalStdSet  is deprecated  not available in RTP.
```

A.2.12 Object IDs as Pointers to Memory

In user mode, object IDs such as **SEM_ID** are not pointers to memory. Instead, they are handles typically comprised of a small integer reference number and a generation ID. It is not possible to access the internal structures of objects in user mode.

A.3 Differences in Kernel and RTP APIs

In order to support real-time processes, a set of user APIs was introduced in VxWorks 6.0. These routines are based on the VxWorks 5.*x* routines, but they have specific changes required to support processes.

- The APIs that make system calls (marked *system*, *system call*, or *syscall* in the reference entry) cannot complete their work without assistance from facilities provided only by the kernel.
- In processes, the routines use POSIX semantics rather than VxWorks semantics.

While additional changes to APIs have occurred since VxWorks 6.0 as the product has developed, it is helpful to compare the earliest differences to highlight the distinction between running an application in the kernel as opposed to a process.

A.3.1 APIs Not Present in User Mode

Some APIs are not present in user mode because their action is not compatible with a protected environment.

intLock( ), **intUnlock( )**, **taskLock( )**, **taskUnlock( )**

It is not possible to lock and unlock interrupts from user mode; thus **intLock( )** and **intUnlock( )** are not present in user mode. Similarly, from a process, it is not possible to globally lock and unlock the scheduler as is done in the kernel by **taskLock( )** and **taskUnlock( )**. You can disable context switching within a process using **taskRtpLock( )** and **taskRtpUnlock( )**; the calling task becomes the only task in the process that is allowed to execute, unless the task explicitly gives up the CPU by making itself no longer ready. However, tasks in other processes may preempt a task locked with **taskRtpLock( )**. If exclusion between tasks in different processes is required, use a public semaphore in place of **taskLock( )**.

taskInit( )
The **taskInit( )** routine is not available in user mode. Instead, use **taskCreate( )**.

taskOptionsSet( )
There are no user-changeable task options available in user mode; thus **taskOptionsSet( )** is not present. Also, not all task options are available; in particular, **VX_UNBREAKABLE** and **VX_SUPERVISOR** are unavailable in user mode.

taskSwitchHookAdd( ), **taskSwitchHookDelete( )**
Adding and deleting task switch hooks in user mode is not supported. Thus, the routines **taskSwitchHookAdd( )** and **taskSwitchHookDelete( )** do not exist in user mode. However, task delete and create hooks are supported in user mode; therefore the routines **taskCreateHookAdd( )**, **taskCreateHookDelete( )**, **taskDeleteHookAdd( )**, and **taskDeleteHookDelete( )** do exist in user mode. For more information, see *A.3.3 APIs that Work Differently in Processes*, p.308.

NOTE: There is no hardware, BSP, or driver access from user-mode. For a list of all APIs that are present in user-mode, see the reference entries.

A.3.2 APIs Added for User Mode Only

Some new user-mode APIs are available in processes only. The largest group of these is the Dinkumware C and C++ libraries. For more information, see the reference entries for these libraries.

A.3.3 APIs that Work Differently in Processes

This section highlights a few APIs that work differently in processes. For more information, see *A.2 Migrating Kernel Applications to Processes*, p.297.

Task Hook Routines

The **taskCreateHookAdd( )**, **taskDeleteHookAdd( )** routines work differently in processes. The kernel versions of these routines are unchanged from VxWorks 5.5. However, the user-mode versions are slightly different:

- They pass an integer task ID as an argument.
- They return **STATUS** instead of **void**.

For more information, see the reference entries for the user versions of **taskCreateHookAdd( )** and **taskDeleteHookAdd( )**.

String and Time Routines and Migration Macro

Several VxWorks routines have different signatures depending on whether they are the kernel or the user-mode (RTP) variants. These routines are the following:

- **strerror_r( )**

- **asctime_r( )**
- **ctime_r( )**
- **gmtime_r( )**
- **localtime_r( )**

The **_VXWORKS_COMPATIBILITY_MODE** macro can be used to facilitate migrating kernel code that uses these routines to a user-mode application. The macro resolves the signature differences. It should be set before the inclusion of **string.h** for **strerror_r( )**; and before **time.h** for **asctime_r( )**, **ctime_r( )**,**gmtime_r( )**, and **localtime_r( )**.

Note that the macro enables the use of the signatures used in the kernel, which are non-POSIX. The default prototypes on the user side are POSIX compliant.

A.3.4 Kernel Calls Require Kernel Facilities

It is possible to call a user-mode API, even if the kernel component that implements that service is not compiled into the system. In this case, an error is returned, with **errno** set to **ENOSYS**. The solution is to add the appropriate component to the kernel and rebuild VxWorks.

Note that all user-mode APIs that are system calls have all arguments and memory addresses validated before the call is allowed to complete.

A.3.5 Other API Differences

The following are differences between using user-mode APIs in processes and using kernel-mode APIs:

- There is no way to get the task list for all tasks in a process.
- Show routines are not available from user mode.

Index